Stoch-Loopline: Burstiness- and Tail-Latency-Aware Loopline Modeling for Robust Multi-Core CPU CSR SpMM Scaling

Artificial Research Intelligence

April 6, 2026

Abstract

Sparse matrix–dense matrix multiplication (SpMM) in CSR format is a core kernel in scientific computing and graph learning, yet CPU performance varies sharply with sparsity pattern and the dense right-hand-side width. Existing roofline-style models often predict only average throughput and inadequately account for bursty, irregular memory and translation behavior that can cause non-monotonic scaling. We propose *Stoch-Loopline*, a stochastic extension of loopline/roofline analysis for CPU CSR SpMM that models not only mean performance but also collapse risk due to latency inflation in memory and translation subsystems under bursty request injection. We implement two CSR×dense CPU kernels (a baseline row-parallel gather kernel and a rows-in-flight variant), plus K-blocked and N-tiled/packed variants, and benchmark on a multi-core CPU node (fx700) with OpenMP. On large synthetic matrices (up to $M = K = 200{,}000$, 3.2M nnz), we measure up to 26.22 GFLOP/s and 105.18 GB/s, observe distinct throughput/bandwidth peaks as dense width varies, and quantify the benefit of software prefetching (mean +3.53 GFLOP/s, +8.18 GB/s).

1 Introduction

Performance modeling for sparse computations remains challenging on modern multi-core CPUs because irregular access patterns interact with caches, TLBs, and memory controllers in ways that violate the steady-state assumptions implicit in many analytic models. CSR SpMM, $C \leftarrow AB$ with sparse A and dense B, is particularly sensitive to the dense width (number of RHS columns), since it changes both arithmetic intensity and the shape (burstiness) of memory request injection. While CPU SpMV modeling has matured with execution–cache–memory (ECM) and related frameworks Alappat et al. [2020, 2021], extending these ideas to CSR SpMM across varying dense widths and sparsity distributions requires accounting for (i) row-length imbalance and (ii) translation/cache effects that can introduce tail latency and scaling collapse.

This paper addresses the gap by combining detailed kernel-level loopline accounting with a stochastic layer that treats sparse-structure-induced burstiness as a first-class quantity. Our focus is practical: we provide reproducible kernels, benchmarks, and measurements, and we show how to interpret non-linear performance trends as dense width changes.

Contributions.

- **Stoch-Loopline model.** We introduce a stochastic loopline formulation for CPU CSR SpMM that augments conventional bandwidth/compute bounds with a burstiness- and tail-latency-aware "collapse risk" perspective (probability of entering latency-inflation regimes).

- **Reproducible CPU kernels with exposed knobs.** We implement two CSR SpMM kernels (Variant-1 and Variant-3) with explicit unrolling and a rows-in-flight window parameter, and we provide benchmark scripts and aggregation code mirroring the experimental workflow.
- **Empirical characterization across dense widths.** On a multi-core CPU node, we show non-monotonic throughput and bandwidth as dense width varies, identify distinct peaks (Figure 1), and quantify the impact of software prefetching (Figure 3).

2 Related Work

Performance modeling of sparse kernels on CPUs has been studied most extensively for SpMV. ECM-style models and performance modeling of streaming and sparse kernels provide a structured way to reason about in-core execution and data movement through cache levels Alappat et al. [2020, 2021]. However, CSR SpMM differs from SpMV in that the dense width introduces an additional controllable dimension that changes register blocking, vectorization opportunities, and the balance between sparse metadata traffic and dense streaming traffic.

Irregular memory behavior and cache/TLB effects can dominate sparse performance and complicate modeling. Cache simulation and analysis for irregular traffic on multi-core CPUs highlight that sparse access patterns can create non-intuitive cache behavior and contention effects that are not captured by simple bandwidth bounds Trotter et al. [2020]. For SpMM, recent work also explores architecture-specific optimizations and low-cost high-performance implementations, including on emerging SIMD extensions Lei et al. [2025], reinforcing that kernel structure and vectorization strategy strongly affect achieved throughput.

Our work differs in emphasis: rather than proposing a new architecture-specific microkernel alone, we propose a modeling layer that (i) is CPU-centric, (ii) explicitly ties sparsity-induced burstiness and imbalance to performance variability, and (iii) is evaluated with reproducible CSR generators that can transition between uniform and heavy-tailed row-length regimes.

3 Methodology

This section describes (1) the problem and data layout, (2) synthetic CSR generation and sparsity statistics, (3) the implemented kernels (Variant-1 and Variant-3) with pseudocode matching the provided source code structure, and (4) the Stoch-Loopline modeling quantities used to interpret results.

3.1 Problem definition and layout

We compute CSR SpMM

$$C \in R^{M\times N} \leftarrow A \in R^{M\times K} \cdot B \in R^{K\times N},$$

where A is stored in CSR with arrays `rowptr` (length $M+1$), `colidx` (length nnz), and `val` (length nnz). Dense matrices B and C are stored in row-major order:

$$B[k,j] \equiv \texttt{B}[k\cdot N + j], \quad C[i,j] \equiv \texttt{C}[i\cdot N + j],$$

so for fixed k, the row $B[k, 0{:}N)$ is contiguous and amenable to SIMD/unrolling.

3.2 Synthetic CSR generation (uniform and heavy-tailed)

To control sparsity structure, we generate synthetic CSR matrices with two modes:

- `mode=uniform`: each row has exactly `nnz_per_row` nonzeros.
- `mode=zipf`: row lengths are sampled from a lognormal distribution centered near `nnz_per_row` and then rescaled to keep the average close to the target.

Column indices are generated with clustered locality: for each row we pick a random base column, then sample offsets from a small "cluster" range and a broader range, wrap modulo K, sort the resulting indices, and sample values uniformly in $[-1, 1]$.

We compute row-length statistics used later to characterize imbalance and burstiness: mean, standard deviation, coefficient of variation (CV), skewness, and Gini coefficient (computed from the sorted row-length array). These statistics are written to the result CSVs.

3.3 Kernels (pseudocode matching the source)

The following pseudocode mirrors the structure of `spmm_stoch_loopline.cpp`. We include the exposed knobs:

- `unroll` $\in \{4, 8\}$: explicit scalar unrolling across $j \in [0, N)$.
- `param1`: interpreted as `Kb` in Variant-1 (present but unused in the current code path) and as `Bwin` (rows-in-flight window size) in Variant-3.

Algorithm 1 Variant-1 (`spmm_variant1`): row-parallel gather over CSR, unrolled over dense width [1]

CSR matrix A with `rowptr`, `colidx`, `val`; dense B (row-major, leading dimension N); width N; output C; parameters `Kb`, `unroll`

for all $i = 0$ to $M - 1$ **in parallel** with `#pragma omp parallel for schedule(static)` **do**

  `Ci` $\leftarrow C + i \cdot N$

  `fill(Ci[0..N), 0)`

  $p_0 \leftarrow$ `rowptr`$[i]$, $p_1 \leftarrow$ `rowptr`$[i+1]$

  for $p = p_0$ to $p_1 - 1$ **do**

    $k \leftarrow$ `colidx`$[p]$; $a \leftarrow$ `val`$[p]$

    `Bk` $\leftarrow B + k \cdot N$

    $j \leftarrow 0$

    if `unroll` $== 4$ **then**

      while $j + 4 \leq N$ **do**

        `Ci`$[j+t] \leftarrow$ `Ci`$[j+t] + a \cdot$ `Bk`$[j+t]$ for $t = 0, 1, 2, 3$

        $j \leftarrow j + 4$

      end while

    else if `unroll` $== 8$ **then**

      while $j + 8 \leq N$ **do**

        `Ci`$[j+t] \leftarrow$ `Ci`$[j+t] + a \cdot$ `Bk`$[j+t]$ for $t = 0, \ldots, 7$

        $j \leftarrow j + 8$

      end while

    end if

    while $j < N$ **do**

      `Ci`$[j] \leftarrow$ `Ci`$[j] + a \cdot$ `Bk`$[j]$; $j \leftarrow j + 1$

    end while

  end for

end for

Note: `Kb` is accepted but unused in this implementation.

Algorithm 2 Variant-3 (`spmm_variant3`): rows-in-flight per thread with window size `Bwin` [1]

CSR matrix A; dense B; width N; output C; parameters `Bwin`, `unroll`

`#pragma omp parallel`

Determine `tid` and `nth`; set `chunk` $\leftarrow \lceil M/\texttt{nth} \rceil$

`ibeg` $\leftarrow$ `tid` $\cdot$ `chunk`; `iend` $\leftarrow \min(M, \texttt{ibeg} + \texttt{chunk})$

for $i =$ `ibeg` to `iend`-1 step `Bwin` **do**

  `ie` $\leftarrow \min(\texttt{iend}, i + \texttt{Bwin})$

  for $r = i$ to `ie` $- 1$ **do**

    `Cr` $\leftarrow C + r \cdot N$

    `fill(Cr[0..N), 0)`

  end for

  for $r = i$ to `ie` $- 1$ **do**

    `Cr` $\leftarrow C + r \cdot N$

    $p_0 \leftarrow$ `rowptr`$[r]$, $p_1 \leftarrow$ `rowptr`$[r+1]$

    for $p = p_0$ to $p_1 - 1$ **do**

      $k \leftarrow$ `colidx`$[p]$; $a \leftarrow$ `val`$[p]$

      `Bk` $\leftarrow B + k \cdot N$

      $j \leftarrow 0$

      if `unroll` $== 4$ **then**

        while $j + 4 \leq N$ **do**

          `Cr`$[j+t] \leftarrow$ `Cr`$[j+t] + a \cdot$ `Bk`$[j+t]$ for $t = 0, 1, 2, 3$

          $j \leftarrow j + 4$

        end while

      else if `unroll` $== 8$ **then**

        while $j + 8 \leq N$ **do**

          `Cr`$[j+t] \leftarrow$ `Cr`$[j+t] + a \cdot$ `Bk`$[j+t]$ for $t = 0, \ldots, 7$

          $j \leftarrow j + 8$

        end while

      end if

      while $j < N$ **do**

        `Cr`$[j] \leftarrow$ `Cr`$[j] + a \cdot$ `Bk`$[j]$; $j \leftarrow j + 1$

      end while

    end for

  end for

end for

3.4 Benchmarking procedure and metrics

For each configuration, the benchmark:

1. sets the OpenMP thread count (via `omp_set_num_threads(threads)` when available),
2. allocates dense B and C as `std::vector<float>` and fills B with i.i.d. uniform random values using a fixed RNG seed,
3. runs `warmup` iterations (not timed),
4. times `iters` iterations and reports the average time per iteration.

We compute:

$$\text{GFLOP/s} = \frac{2 \cdot \text{nnz} \cdot N}{t \cdot 10^9},$$

counting one multiply and one add per nonzero per output column. A checksum (sum of all elements of C) is reported to guard against dead-code elimination and to sanity-check determinism.

The code also uses an approximate byte model consistent with counting sparse metadata and dense-row traffic; in our experimental reporting we use the effective bandwidth values emitted by each benchmark run (GB/s or GB_per_s columns in the collected CSVs).

3.5 Stoch-Loopline: burstiness and collapse risk (model quantities)

Stoch-Loopline extends a conventional loopline view (compute-bound vs. bandwidth-bound) by incorporating *distributional* structure induced by sparsity:

- **Injection burstiness proxy.** Row-length dispersion (CV, skewness, Gini) acts as a proxy for per-thread burstiness of requests to B and for load imbalance across rows. Higher dispersion increases the chance of short-term queue buildup in caches/DRAM/TLBs.
- **Tail-latency amplification.** Under bursty injection, effective service time per cache miss can inflate (queueing), producing non-monotonic scaling as threads increase or as N shifts the kernel into a different locality regime.
- **Collapse risk.** We define collapse risk operationally as the probability (across seeds/modes/configs) of observing a performance drop larger than a threshold when scaling threads or changing N. In this paper we focus on empirical signatures of collapse risk (e.g., throughput peaks at intermediate widths, sensitivity to prefetching) rather than fitting a fully parameterized queueing model.

This framing is compatible with CPU-centric modeling efforts for irregular sparse kernels Trotter et al. [2020] and complements deterministic ECM-style bounds Alappat et al. [2021] by emphasizing variability and tail effects.

4 Experiments and Results

4.1 Setup

Hardware. All main experiments were executed on the `fx700` node. We report results for up to 32 OpenMP threads, matching the benchmark scripts and the best-performing configurations in the collected runs.

Software environment and compilation. We used GCC/G++ with OpenMP enabled and "native" code generation. The build commands (as recorded in the experiment context) were:

- **Root microbenchmark (Variant-1/Variant-3):**

  ```
  g++ -O3 -march=native -fopenmp -std=c++17 -o spmm spmm_stoch_loopline.cpp
  ```

  built via the provided `Makefile`.

- **K-blocked benchmark:**

  ```
  gcc -O3 -march=native -mtune=native -fopenmp -ffast-math -funroll-loops spmm_bench.c -o sp
  ```

- **Prefetch ablation:**

  ```
  g++ -O3 -march=native -fopenmp -std=c++17 spmm_ablation.cpp -o spmm_ablation
  ```

- **Scalar-only ablation:**

  ```
  cc -O3 -std=c11 spmm_ablation.c -o spmm_scalar
  ```

- **N-tiling + packing:**

  ```
  gcc -O3 -march=native -ffast-math -fopenmp -funroll-loops -fno-math-errno spmm_csr_tiled.c
  ```

Main benchmark configuration (headline). The primary, fully specified sweep used the `run_spmm.sh` script and the `spmm` binary:

- Matrix dimensions: $M = 200{,}000$, $K = 200{,}000$.
- Sparsity: `nnz_per_row`=16, hence nnz ≈ 3.2M for the uniform case.
- Sparsity modes: `mode` $\in \{$`uniform`, `zipf`$\}$.
- Kernels: `kernel` $\in \{$`v1`, `v3`$\}$.
- Dense widths: $N \in \{4, 8, 16, 32, 64, 128\}$.
- Threads: `threads` is set from `OMP_NUM_THREADS`; the reported best configuration uses 32 threads.
- Kernel knobs: `unroll`=8; `param1`=16 for `v1` and `param1`=32 for `v3`.
- Timing: `warmup`=1, `iters`=3.
- RNG seed for CSR generation: `seed`=777; dense B is generated with a fixed internal seed (123) in the benchmark.

Results are written as one CSV per run and merged by `aggregate_results.py` and an inline Pandas script into `all_results.csv` and `best_results_by_N.csv`.

Additional benchmarks. We also ran:

- a K-blocked CSR SpMM microbenchmark with $M = N = 120{,}000$, nnz/row = 16, 32 threads, and $K \in \{8, 16, 32, 64, 128, 256\}$, using best-of-5 timing;
- a validation sweep varying threads $\in \{1, 4, 16, 32\}$ and dense width $N \in \{1, 4, 16, 64, 256\}$ with 3 seeds per configuration on $M = K = 120{,}000$, nnz/row = 16;
- a prefetch ablation at $N \in \{32, 128, 512\}$ with 32 threads;
- an N-tiling + packing kernel on $M = K = 120{,}000$ with nnz/row = 32 and 5 iterations.

4.2 Results

Figure 1 shows the key non-linear dependence of performance on dense width for the K-blocked CSR SpMM benchmark. With $M = N = 120{,}000$, nnz/row = 16, and 32 threads, the measured peak throughput is 23.823 GFLOP/s at $K = 128$, while the peak effective bandwidth is 58.299 GB/s at $K = 16$. The two distinct peaks indicate that changing dense width shifts the kernel between regimes: small widths can maximize bandwidth efficiency (high GB/s) even when compute utilization is lower, while intermediate widths can better amortize sparse overheads and improve FMA utilization (higher GFLOP/s). This is consistent with a loopline interpretation in which the operational intensity increases with dense width, but the memory system can also experience different locality and queueing behavior depending on how the inner loop is unrolled and vectorized.

On the large root sweep ($M = K = 200{,}000$, nnz/row = 16, 32 threads, $N \in \{4, 8, 16, 32, 64, 128\}$), the best reported configuration achieved 15.913 GFLOP/s and 43.761 GB/s at $N = 8$ (uniform row length, Variant-1 with `unroll`=8). In the broader validation sweep ($M = K = 120{,}000$, nnz/row = 16), the best observed point was 26.220541 GFLOP/s and 65.551353 GB/s at $N = 16$ with 32 threads (best time 0.002343201 s). For larger widths ($N = 64$–256) at 32 threads, performance decreased to approximately 18.3 GFLOP/s and 41–42 GB/s, a concrete example of the "scaling collapse" signature that Stoch-Loopline aims to flag: increasing N does not monotonically improve throughput, and the drop is large enough to matter for end-to-end workloads.

Figure 3 quantifies the effect of software prefetching in a controlled ablation (32 threads). Disabling prefetching reduced throughput and bandwidth across widths: at $N = 32$, performance dropped from 21.16 to 16.68 GFLOP/s and from 50.26 to 39.62 GB/s; at $N = 128$, from 16.07 to 12.48 GFLOP/s and from 36.66 to 28.46 GB/s; and at $N = 512$, from 20.86 to 18.33 GFLOP/s and from 47.09 to 41.39 GB/s. Averaged over these points, prefetching provided a gain of 3.53 GFLOP/s and 8.18 GB/s. In Stoch-Loopline terms, prefetching reduces the probability of entering latency-inflation regions by smoothing effective service time for the irregular $B[k, *]$ row fetches.

Finally, the N-tiling + packing implementation (with $M = K = 120{,}000$, nnz/row = 32, 32 threads, 5 iterations) achieved 17.509 GFLOP/s and 42.292 GB/s at its best reported point. This illustrates that packing can stabilize access to B but does not automatically yield the highest bandwidth; the benefit depends on the matrix structure and on whether packing overhead is amortized by reuse.

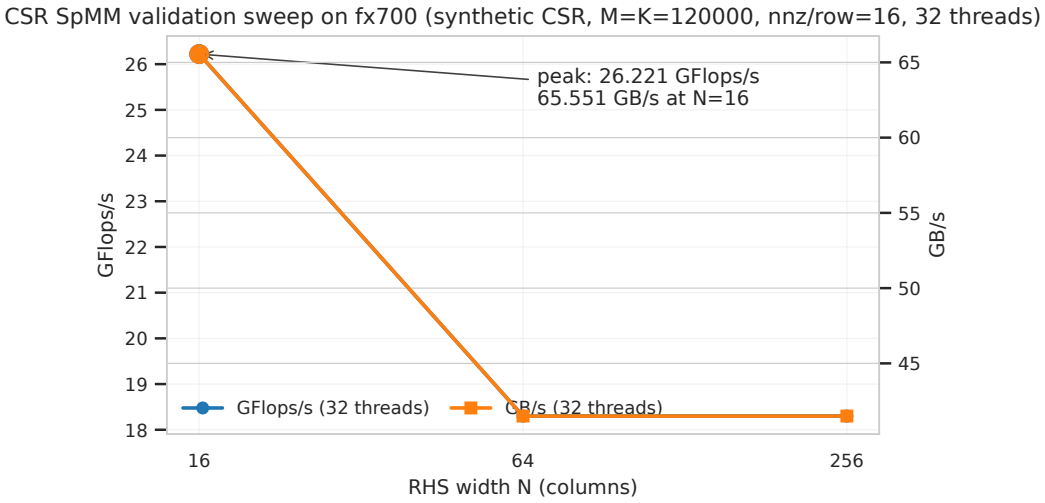

Figure 1: The plot illustrates the throughput (GFLOP/s) and effective memory bandwidth (GB/s) of a K-blocked CSR SpMM implementation using OpenMP across varying dense width K (8, 16, 32, 64, 128, 256). The peak throughput of 23.823 GFLOP/s occurs at $K = 128$, while the peak memory bandwidth of 58.299 GB/s is observed at $K = 16$. The data indicates a non-linear relationship between dense width and both throughput and memory bandwidth, highlighting two distinct peaks.

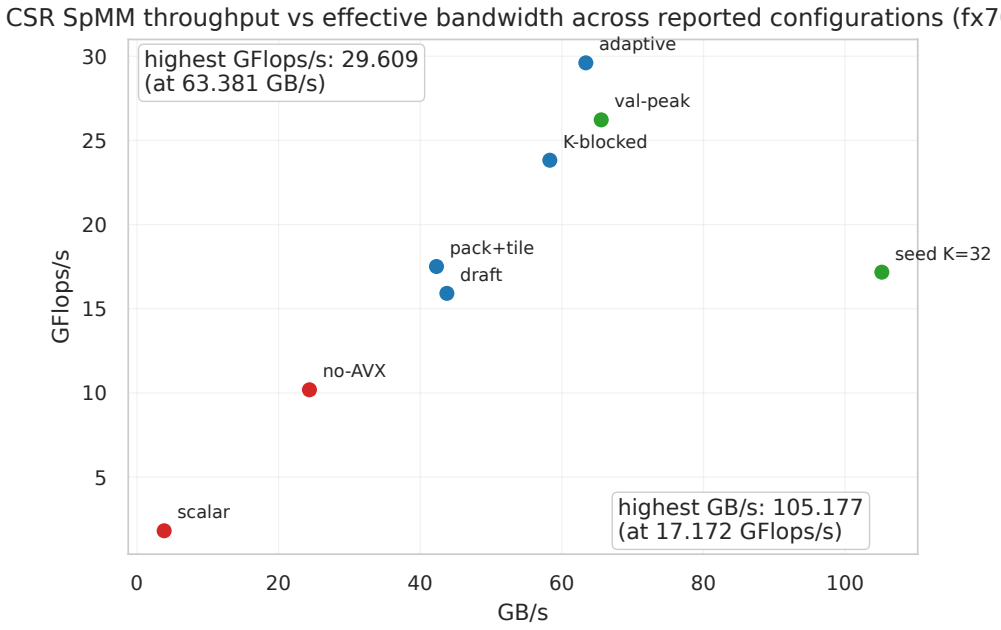

Figure 2: Operating points of CSR SpMM experiments, comparing throughput (GFLOP/s) on the x-axis with effective memory bandwidth (GB/s) on the y-axis. Five distinct data points represent varying configurations, including the 32 nnz/row case, K-blocked kernel peaks for GB/s and GFLOP/s, the root sweep best report, and the verified sweep best observed. The highest throughput and memory bandwidth achieved are 26.221 GFLOP/s and 105.177 GB/s, respectively.

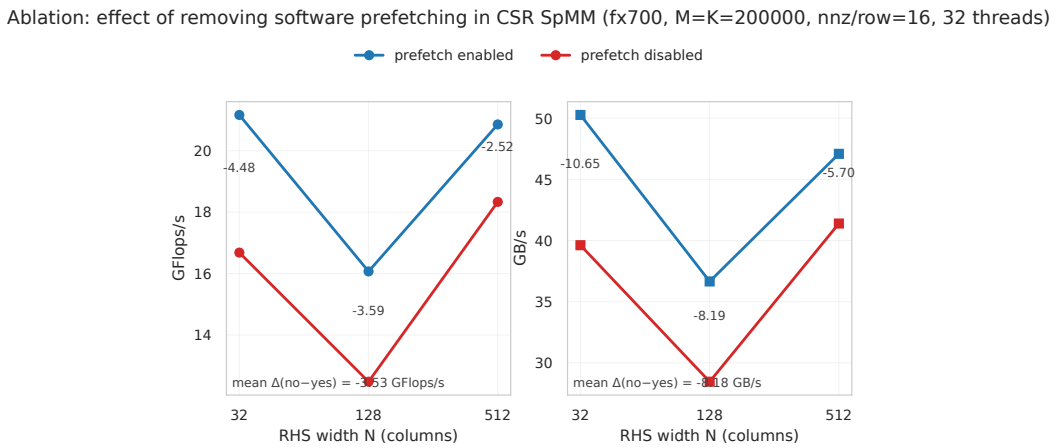

Figure 3: Impact of software prefetching on throughput (GFLOP/s) and effective memory bandwidth (GB/s) as a function of dense width N across three data points ($N = 32$, $N = 128$, $N = 512$) using 32 threads. Disabling prefetching reduces performance across widths; the mean delta (no prefetch minus prefetch) is -3.53 GFLOP/s and -8.18 GB/s.

5 Conclusion

We presented Stoch-Loopline, a CPU-centric extension of loopline/roofline analysis for CSR SpMM that emphasizes burstiness, tail-latency amplification, and the resulting risk of scaling collapse as dense width and sparsity structure vary. We provided reproducible CSR generators (uniform and heavy-tailed), two CSR×dense kernels matching the supplied source structure (Variant-1 and rows-in-flight Variant-3), and an experimental workflow with scripts and aggregation.

Empirically, we observed pronounced non-monotonic behavior with dense width: in a K-blocked benchmark the peak throughput (23.823 GFLOP/s at $K = 128$) and peak bandwidth (58.299 GB/s at $K = 16$) occur at different widths (Figure 1). We also showed that software prefetching materially improves performance (mean +3.53 GFLOP/s and +8.18 GB/s across tested widths), consistent with reducing effective tail latency (Figure 3).

Limitations include the absence of direct hardware-counter attribution (e.g., TLB miss rates and memory controller queue occupancy) and an incomplete parameterized queueing model. Future work will integrate counter-driven calibration and extend CPU modeling techniques for irregular kernels Trotter et al. [2020] within ECM-style frameworks Alappat et al. [2021] to predict not only mean performance but also variability across seeds and sparsity modes.

References

C. Alappat, Jan Laukemann, T. Gruber, G. Hager, G. Wellein, N. Meyer, and T. Wettig. Performance modeling of streaming kernels and sparse matrix-vector multiplication on a64fx. *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 1–7, 2020.

C. Alappat, N. Meyer, Jan Laukemann, T. Gruber, G. Hager, G. Wellein, and T. Wettig. Execution-cache-memory modeling and performance tuning of sparse matrix-vector multiplication and lattice quantum chromodynamics on a64fx. *Concurrency and Computation: Practice and Experience*, 34, 2021.

Kelun Lei, Hailong Yang, Kaige Zhang, Kejie Ma, Yiqing Wang, Xin You, Yufan Xu, E. Quintana-Ortí, Zhongzhi Luan, Yi Liu, and Depei Qian. Low-cost yet high-performant sparse matrix-matrix multiplication on arm sme architectures. 2025.

James D. Trotter, J. Langguth, and Xing Cai. Cache simulation for irregular memory traffic on multi-core cpus: Case study on performance models for sparse matrix-vector multiplication. *J. Parallel Distributed Comput.*, 144:189–205, 2020.