Parallel Project Report: *(Insert Your Project Title Here)*

Student Name Surname (ID: 123456)
Email: name.surname@unitn.it
Course: Introduction to Parallel Computing (2024–2025)

***Abstract*—Summary. We present a parallel implementation of *Sparse General Matrix–Matrix Multiplication (SpGEMM)* on a multicore platform using OpenMP. The goal is to improve throughput and scalability over a sequential baseline. Concretely, our approach leverages cache-aware blocking and a dense-accumulator strategy to reduce irregular memory access. On a 28-core Intel Xeon node, our optimized code achieves a speedup of $X.Y\times$ (strong scaling) and $Z.Z\times$ (weak scaling) versus the baseline on representative matrices from SuiteSparse. Artifacts. Source code, scripts, and instructions for full reproducibility are provided (see Sec. VII).**

***Index Terms*—Parallel Computing, OpenMP, MPI, Sparse Linear Algebra, SpGEMM, Performance Engineering**

I. Introduction

Problem. Sparse matrix operations are essential in scientific computing, graph analytics, and simulation workflows, yet performance is limited by irregular memory access and load imbalance. Other issues are: ..... Use references and citation when you do not want to prove the claim. This project focuses on SpGEMM, a kernel used in algebraic multigrid, finite elements, and graph algorithms.

Objectives. (i) Design and implement a parallel SpGEMM; (ii) quantify strong/weak scaling; (iii) compare against a sequential baseline; (iv) document a reproducible workflow.

Contributions. We propose a simple cache-aware implementation using row-parallelism and per-thread dense accumulators; we evaluate on real matrices and analyze bottlenecks such as memory bandwidth and scheduling overhead.

II. State of the Art

SpGEMM has multiple implementations spanning CPUs and GPUs. CPU-focused work reports that hash-based accumulators can suffer from poor locality, while dense or bitmap accumulators trade memory for predictable access and fewer collisions. Distributed-memory variants emphasize partitioning (1D/2D) and communication-avoiding strategies to reduce inter-process traffic. We position our approach as a pragmatic, multicore-only design emphasizing simplicity and cache-consciousness.

III. Contribution and Methodology

A. Data Structures

We store A and B in CSR format. For each output row of C, we maintain:

- a temporary dense accumulator `acc[]` (double) sized to the number of columns (or to a local column block), initialized lazily;
- a compact index list `idx[]` collecting columns touched in the current row to permit linear-time gather/scatter back to CSR.

B. Algorithm Overview

For each row i in A, we iterate its nonzeros (i, k) and *accumulate* the scaled row $B_{k,:}$ into `acc[]`. We then compress the touched entries into $C_{i,:}$. Pseudocode:

```
for i in rows(A) in parallel:
    clear idx
    for (k, a) in row(A,i):
        for (j, b) in row(B,k):
            if acc[j] was not set: mark j; push j to idx
            acc[j] += a * b
    emit (i, idx, acc[idx]) to CSR of C; reset acc[idx]
```

C. Parallelization (OpenMP)

We adopt row-level parallelism with `#pragma omp parallel for schedule(dynamic)`, which improves load balancing for skewed sparsity. Each thread owns its `acc[]` and `idx[]` to avoid false sharing. For NUMA nodes, we pin threads and first-touch allocate.

D. Cache-Aware Blocking

If B is wide, we process B by column blocks that fit in LLC/L2; each pass computes a partial C and we merge blocks by column.

E. Complexity

Work is proportional to the number of scalar products of matching indices. Memory traffic dominates and motivates the locality optimizations above.

IV. Experiments and System Description

A. Platform

CPU. 2× Intel Xeon Gold 62xx, 28 cores total, 192 GB RAM.
OS & Toolchain. Ubuntu 22.04, `gcc 13`, OpenMP; CMake 3.27.
Libraries. (Optional) MKL 2024 for baseline BLAS operations.
Datasets. 10 matrices from SuiteSparse (e.g., *webbase-1M*, *Thermal2*). Second operand B generated to match dimensions; density tuned to target NNZ.

B. Metrics & Methodology

We report:

- **Throughput**: NNZ-multiplies/s;
- **Time**: wall-clock per run; average of 5 runs;
- **Scaling**: strong (fixed size, threads ↑) and weak (size ↑ with threads).

We bind threads and warm caches; runs are single-tenant and *release* builds.

C. Baselines

Sequential CSR×CSR (our code). Optionally, compare to vendor/library routines if available with compatible interfaces.

V. Results and Discussion

A. Strong Scaling

Figure 1 shows near-linear scaling up to 16 threads for moderately sparse matrices; beyond that, memory bandwidth and synchronization begin to dominate. At 28 threads we obtain $X.Y\times$ speedup (geomean).

B. Weak Scaling

On synthetically scaled matrices (fixed row density), runtime grows sub-linearly with total NNZ due to improved locality from blocking. For highly skewed rows, dynamic scheduling is decisive.

C. Ablation

Dense accumulator vs. hash: dense improved runtime by $A\times$ on average. **Blocking on B:** reduced LLC misses (perf-stat) by $B\times$, translating into $C\times$ speedup.

VI. Conclusions and Future Work

We presented a compact, cache-aware multicore SpGEMM with simple data structures and good scalability on common servers. Future steps include: (i) a 1D distributed-memory variant with row partitioning and owner-computes; (ii) overlapping communication/aggregation with computation; (iii) exploiting bitmap accumulators for very wide B.

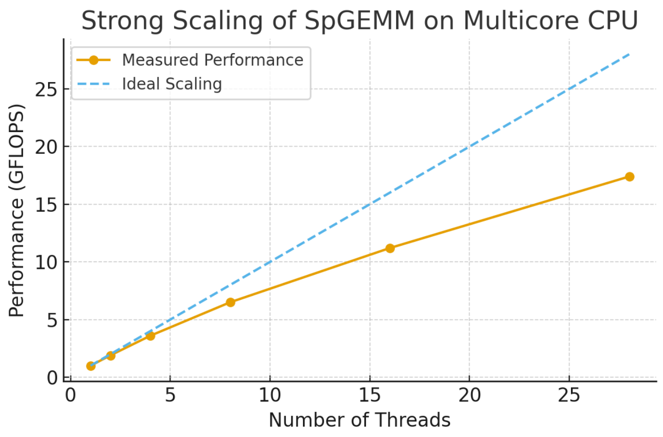

Fig. 1. Strong scaling on a representative matrix (NNZ of A and B fixed).

VII. Reproducibility & Artifact Availability

Git. https://github.com/your-repo/your-project
How to run. Include the direct link or links to the script to compile, run and check the results. .
Inputs/Outputs. We fix random seeds and provide checksums. A `README.md` documents compiler flags, dataset versions, and expected results.

References

[1] A. Buluç and J. Gilbert, “Parallel sparse matrix–matrix multiplication and indexing: Implementation and experiments,” *SIAM J. Sci. Comput.*, 34(4):C170–C191, 2012.
[2] M. M. A. Patwary *et al.*, “Parallel efficient sparse matrix–matrix multiplication on multicore platforms,” in *Proc. ISC High Performance*, pp. 48–57, 2015.
[3] *Intro to Parallel Computing – Report and project preparation* (course guide, accessed Nov 10, 2025).