

Taming Multi-Dimensional Skew in Sparse Matrix Multiplication with Contention-Aware Scheduling

Wentao Huang
National University of Singapore
huangwentao@u.nus.edu

Qiming Huang
National University of Singapore
huang_qiming@u.nus.edu

Yunhong Ji
Renmin University of China
jiyunhong@ruc.edu.cn

Mian Lu
Phancy Group Co., Ltd.
lumian@4paradigm.com

Yuqiang Chen
Phancy Group Co., Ltd.
chenyuqiang@4paradigm.com

Bingsheng He
National University of Singapore
dcsheb@nus.edu.sg

Kian-Lee Tan
National University of Singapore
dcstankl@nus.edu.sg

ABSTRACT

Sparse general matrix-matrix multiplication (SpGEMM) has emerged as a key kernel in graph analytics, relational queries, and machine learning, yet skewed workloads expose two co-existing challenges. First, *multi-dimensional skew*: imbalance in computation cost, output density, and data locality cannot be captured by any single balancing metric. Second, *cache contention*: dense substructures induce severe on-chip interference. To address both challenges, we propose adaptive morsel scheduling (AMS), a dynamic scheduling framework that decouples skew dimensions via fixed-size output morsels and integrates contention-aware load balancing (CALB) and data-conscious simultaneous multithreading activator (SMTact), alongside hybrid accumulation mechanism (HYB) that selects the best accumulator per morsel. Across matrix squaring, relational queries, and graph analytics, AMS significantly outperforms the state of the art on skewed workloads and remains comparable on non-skewed ones, with gains and bandwidth improvements holding across hardware systems.

PVLDB Reference Format:
Wentao Huang, Qiming Huang, Yunhong Ji, Mian Lu, Yuqiang Chen, Bingsheng He, and Kian-Lee Tan. Taming Multi-Dimensional Skew in Sparse Matrix Multiplication with Contention-Aware Scheduling. PVLDB, 20(3): 835 - 848, 2026.
doi:10.14778/3860016.3860044

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/fukien/contention-aware-spgemm.

1 INTRODUCTION

Matrix multiplication is a fundamental computational kernel, and its sparse variant, sparse general matrix-matrix multiplication (SpGEMM), is especially important in practice. The reason is that sparse matrices are a natural model for real-world graph data, which makes SpGEMM a core building block for graph analytics and for accelerating deep learning [53, 63, 66, 71, 86]. More recently, SpGEMM has also found its way into relational query processing, notably join-aggregate and join-project queries [27, 42–44, 50, 56, 83–85]. The idea is to represent relations as sparse matrices, so that the join is fused with the subsequent aggregation or projection to avoid the prohibitive costs of materializing excessive intermediate results. In all such cases, application performance is dictated by the efficiency of the underlying parallel SpGEMM execution. Reflecting its critical role, numerous studies have investigated SpGEMM optimization, reaching a broad consensus that it is a fundamentally bandwidth-intensive kernel whose throughput is ultimately governed by available memory bandwidth [36, 69].

Being bandwidth-bound, however, does not mean that running SpGEMM on a high-bandwidth platform translates into higher throughput. The primary obstacle is the *load imbalance* caused by the skew patterns widely observed in real-world workloads [12, 23, 55, 60, 77, 94, 102]. For instance, in a typical OLAP ClickHouse benchmark [79], the top 5% of users generate 24.4% (47.8 million) of all clicks, while the median number of clicks per user is only 2. Similarly, in a common PageRank workload [98], the top 1.8% of pages receive more incoming links than the remaining 98.2% combined. From a matrix perspective, such skewed key frequencies and power-law graph degrees translate into highly irregular nonzero structures, resulting in significant load imbalance. Running SpGEMM on these workloads therefore overloads a small subset of cores while the rest remain underutilized [31, 36, 69]. This not only limits compute efficiency but also leaves much of the available memory bandwidth unused, rendering high-bandwidth platforms far less effective than their specifications suggest.

Moreover, the challenge of addressing load imbalance in skewed workloads is further exacerbated by the "memory scaling wall," where memory technology advancements lag behind those of processing units [74, 91]. This trend results in decreasing memory bandwidth and capacity per processing core [49, 97], which further compounds the load imbalance problem for two reasons.

First, decreasing bandwidth-per-core has an acutely negative impact on SpGEMM workloads with load imbalance, where a few straggling threads drag down performance. Since these threads largely rely on core-level bandwidth, their overall performance is inevitably reduced due to the insufficient bandwidth-per-core caused by the "memory scaling wall."

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 20, No. 3 ISSN 2150-8097.
doi:10.14778/3860016.3860044

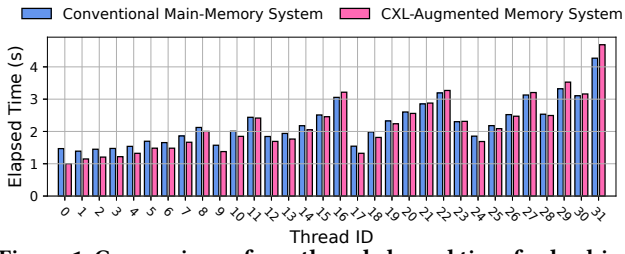


Figure 1: Comparison of per-thread elapsed time for hashing-based SpGEMM [69] on a G500 skewed matrix (scale=17, *edgefactor*=16) [19] between a conventional main-memory system (28.50 GB/s bandwidth) and a CXL-augmented system (46.50 GB/s bandwidth) (see § 4 for experimental setups).

Second, the prevalent approach to compensating for decreasing bandwidth-per-core, namely the adoption of heterogeneous memory expansion technologies like CXL [48, 59, 97], can inadvertently amplify this problem. Since CXL-attached memory typically exhibits higher access latency than local DRAM, straggling threads that run disproportionately long are more exposed to CXL's latency penalty, further widening the gap between fast and slow threads. Figure 1 illustrates this using a hashing-based SpGEMM algorithm [69] on a skewed workload, where per-thread runtimes vary due to load imbalance. [1] Because total runtime is governed by the slowest thread, the disproportionate latency on stragglers negates the aggregate bandwidth gain, yielding worse overall performance than pure DRAM. This counterintuitive result is also reported in existing CXL studies [64, 65, 87, 90]. As a result, addressing load imbalance is becoming more imperative than ever.

However, addressing load imbalance in SpGEMM is not trivial. While numerous strategies have been proposed [31, 36, 72], we argue that they fall short because of two co-existing root causes: *multi-dimensional skew* and *cache contention*, which not only hinder load balancing but also degrade overall performance.

First, existing methods fail to accommodate the *multi-dimensional skew* inherent in sparse matrices. Current methods rely on static schemes that balance along a single metric (e.g., flop count) to minimize scheduling overhead. However, skew manifests across multiple dimensions, including computation cost, materialization overhead, and data locality, making it impossible for any single static metric to achieve balanced execution across all threads.

Second, in skewed workloads, existing approaches suffer from significant cache contention, induced by dense substructures that are co-scheduled together. In SpGEMM, irregular sparsity forces practitioners to cache row-wise intermediates to reduce random access costs. However, dense substructures within skewed matrices incur excessive irregular random entry access, which creates heavy cache pressure and even induces significant cache contention, which has long been overlooked as a key performance factor. This cache contention is particularly pronounced under simultaneous multithreading (SMT), which aims to increase computation parallelism but worsens the interference.

In this paper, we address these challenges through a novel approach, *adaptive morsel scheduling (AMS)*. In contrast to de facto methods, AMS revisits *dynamic scheduling*, a strategy conducive to load balancing that is often dismissed due to concerns regarding scheduling overhead and poor performance [36, 69]. Through analysis (§ 2.3), we find that task granularity is the primary performance factor: a granularity that is too fine is unlikely to preserve spatial locality and thereby suffers from excessive cache thrashing. Hence, we employ a morsel-wise dynamic scheduling strategy, where a "morsel" [2] is defined as a basic scheduling unit of several consecutive matrix rows. AMS ensures a morsel is large enough to maintain spatial locality, thereby greatly mitigating cache thrashing and, more importantly, achieving load balance on skewed workloads through dynamic scheduling, effectively preventing computational idling and improving memory bandwidth utilization.

To alleviate cache contention, AMS introduces two lightweight and generalizable techniques: *contention-aware load balancing (CALB)*, which mitigates cache interference by adaptively scheduling dense morsels, and *data-conscious simultaneous multithreading activator (SMTact)*, which selectively enables SMT based on workload contention. These optimizations are simple in design, yet novel, effective in practice, and easy to generalize to other sparse matrix kernels. Importantly, their benefits carry over to heterogeneous bandwidth-boosted systems such as CXL-based platforms, where latency heterogeneity would otherwise degrade performance.

Furthermore, the morsel-wise scheduling strategy enables a *hybrid accumulation mechanism (HYB)* for SpGEMM. Since morsels differ in their intrinsic characteristics, HYB applies the accumulator best suited to each one, yielding more efficient execution and better overall SpGEMM performance.

The contributions of this paper are summarized as follows:

(1) We examine and characterize two co-existing root causes of performance degradation in skewed SpGEMM workloads: *multi-dimensional skew*, where no single static metric can balance all threads, and *cache contention*, where dense substructures induce severe on-chip interference that existing methods overlook.
(2) We develop a novel morsel-wise dynamic scheduling framework, adaptive morsel scheduling (AMS), that systematically addresses both *multi-dimensional skew* and *cache contention* in SpGEMM. AMS decouples skew dimensions through fixed-output-size morsels, integrates two key techniques, contention-aware load balancing (CALB) and data-conscious simultaneous multithreading activator (SMTact), to alleviate on-chip cache interference, and enables hybrid accumulation mechanism (HYB) for morsel-wise accumulation adaptively. Our approach is straightforward in design, yet novel, effective in practice, and readily generalizable.
(3) We evaluate AMS across a range of workloads, including matrix squaring, relational queries, and graph analytics, and show that it performs on par with state-of-the-art methods on non-skewed workloads while significantly outperforming them on skewed ones, addressing both computational idling and bandwidth underuse.

The rest of the paper is organized as follows. We first present the background and motivation in § 2, followed by a description

[1] We use the following notation for R-MAT graphs [19]: **ER** and **G500** denote non-skewed Erdős–Rényi and skewed Graph500 graphs, respectively; *scale* sets the number of rows to 2^{scale}, and *edgefactor* sets the average nonzeros per row.

[2] We adopt the term from Leis et al. [61], where it denotes a fixed-size fragment of input data, whereas in our work it denotes a group of consecutive rows with a roughly fixed output size.

Algorithm 1 Parallel Gustavson's Algorithm for SpGEMM

Input: CSR matrices $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$
Output: CSR matrix $C \in \mathbb{R}^{m \times n}$
parallel for $i \leftarrow 0$ to $m-1$ **do** ▷ Parallel over rows of A
  Initialize empty `accumulator` for row i
  for all $(k, A_{ik}) \in$ row i of A **do**
    for all $(j, B_{kj}) \in$ row k of B **do**
      `accumulator[`j`]` $+= A_{ik} \cdot B_{kj}$ ▷ Insert or update
    end for
  end for
  Store `accumulator` as row i of C
end parallel for

of AMS in § 3. We then show the experimental results in § 4 and discuss related work in § 5. Finally, we conclude the paper in § 6.

2 BACKGROUND AND MOTIVATION

We first review state-of-the-art SpGEMM algorithms, then analyze how multi-dimensional skew and cache contention degrade performance under skewed workloads, and finally revisit dynamic scheduling as a promising solution, motivating our AMS approach.

2.1 The SpGEMM

SpGEMM computes the product $C = AB$, where A, B, and C are all sparse matrices. This form has proven effective across diverse large-scale workloads, including scientific computing, graph analytics, and more recently relational query processing [42, 68, 69, 84]. To reduce memory footprint at scale, sparse matrix formats are widely used, with compressed sparse row (CSR) being the most common representation in existing works [10, 18, 34, 52, 76]. However, due to the irregular and input-dependent sparsity patterns of sparse matrices, predicting the number and locations of nonzeros in the output matrix is often intricate, making memory pre-allocation challenging. Combined with irregular memory access patterns, these challenges make SpGEMM a fundamentally memory-intensive workload, with performance critically dependent on sustained memory bandwidth across cores.

To address these challenges, SpGEMM is typically divided into two phases: a symbolic phase, which determines the nonzero structure of the output, and a numeric phase, which performs the actual computation based on that structure. In addition, the irregular nature of sparsity makes it difficult to efficiently gather and accumulate scattered contributions to each output entry. To tackle this, prior work has introduced a variety of accumulator designs to aggregate partial results during computation, including sorted variants such as heaps [68, 69], SPA [33, 39], and expand-sort-compress (ESC) [11, 25], as well as non-sorted variants based on hash tables [41, 68, 69].

Building on these data representations, SpGEMM algorithms are commonly implemented using one of three formulations: inner-product [73], outer-product [2, 36], or the row-row paradigm (Gustavson's algorithm) [39]. Among these, Gustavson's algorithm operates in a row-wise manner, allowing each row of the left-hand side matrix (A) to be computed independently, enabling straightforward thread-level parallelization without the need for inter-thread synchronization (see Algorithm 1). It is this high degree of parallelism,

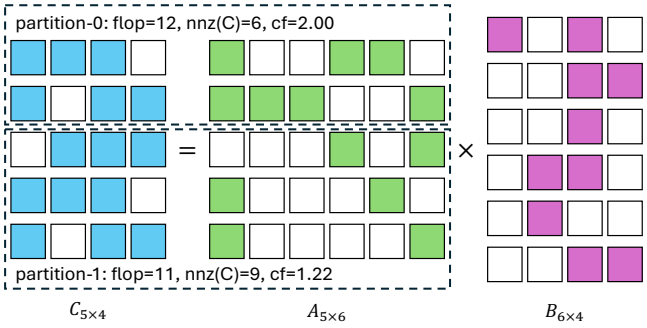


Figure 2: An illustration of $C = AB$ in SpGEMM (colored cells denote nonzero elements), where the two dashed boxes represent partitions with comparable flop counts but differing numbers of output nonzeros and compression factors.

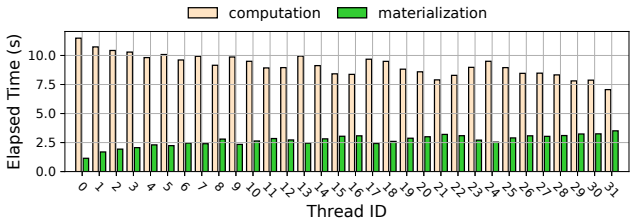


Figure 3: Thread-level computation time vs. materialization time for hashing-based SpGEMM on a G500 skewed matrix (scale=19, *edgefactor*=8).

combined with its efficient memory access patterns, that has made Gustavson's algorithm a dominant formulation for SpGEMM across a wide range of computing platforms.

Following Gustavson's algorithm, several load-balancing techniques have been proposed [2, 4, 8, 16, 20, 92], with static scheduling based on a flop-count scheme emerging as the dominant strategy [68, 69]. This method partitions the rows of matrix A into contiguous regions, each assigned approximately the same number of floating-point operations (i.e., additions plus multiplications). To achieve this, a lightweight preprocessing pass scans matrices A and B to estimate the flop count for each output row of C without performing actual calculations. Based on these estimates, rows are grouped into partitions such that each thread receives a workload with a nearly equal number of floating-point operations. This approach enables static, flop-count-balanced scheduling across threads while avoiding runtime synchronization.

2.2 The Flop-Count-Based Scheme Under Skew

Despite its simplicity and widespread adoption [18, 69], the flop-count-based scheme is inherently limited and fails to achieve effective workload balance under skewed patterns. We identify two root causes, the inherent inability to account for multi-dimensional skew and the performance penalty of overlooking cache contention, both of which hurt load balancing and degrade bandwidth utilization.

(1) Ignoring multi-dimensional skew. Skewed SpGEMM workloads can exhibit imbalance across multiple dimensions, such as submatrix density and row-wise nonzero (*nnz*) distribution. However, the flop-count-based scheme equalizes only the total number

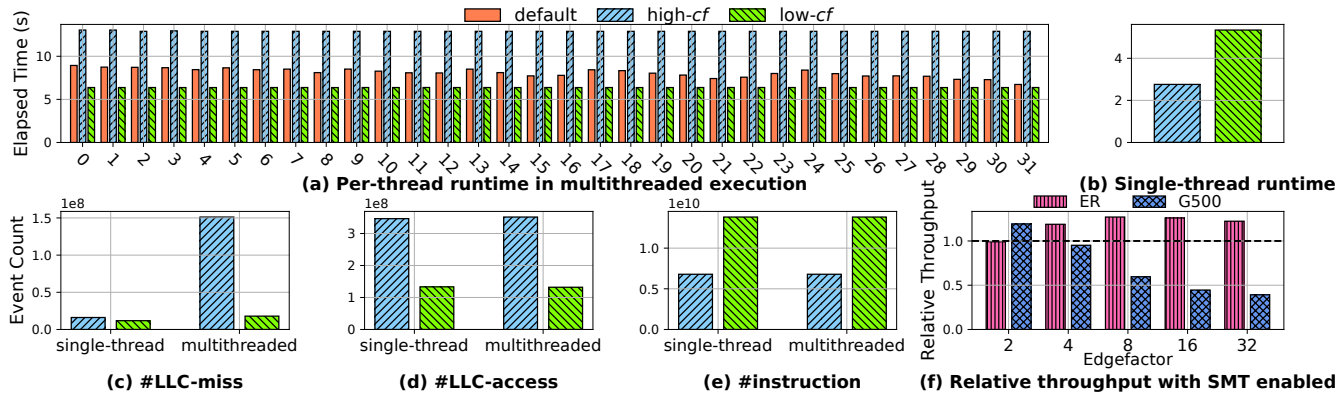


Figure 4: (a) Per-thread computation time of HASH-based SpGEMM on a G500 skewed matrix (scale=19, *edgefactor*=8) under three settings: all threads execute their default partitions; all execute a high-cf partition (thread-0's); all execute a low-cf partition (thread-31's). (b) Computation time of the low-cf and high-cf partitions under single-threaded execution. (c)–(e) LLC misses, LLC accesses, and instruction counts for the low-cf and high-cf partitions under both single-threaded and multithreaded execution. (f) Normalized throughput of HASH-based SpGEMM with SMT enabled on ER and G500 matrices (scale=18) across varying *edgefactor* values. The black dashed line marks the relative throughput without SMT.

of floating-point operations, ignoring other forms of skew. As a result, one thread may receive a few dense rows (with high *nnz*), while another is assigned many sparse rows. Although these assignments may have similar total flop counts, their structural differences lead to divergent runtime behaviors. Dense rows, in particular, induce "clustering" effects during processing, which manifest as temporal bursts of memory and cache pressure. These effects significantly degrade performance, and the extent of impact varies by accumulation strategy [25, 33, 68]. Therefore, the scheme is inherently unable to account for these skew dimensions.

Additionally, thread-level partitions often exhibit different compression factors (cf) [3] , which leads to divergent trends in computation and materialization times, making it challenging to effectively balance these two workload components. Consider Figure 2: using Gustavson's algorithm, partition-0 (the upper two rows of A) has a flop count of 12, producing 6 nonzeros; partition-1 (the bottom three rows) has a flop count of 11 but produces 9 nonzeros [4]. Despite both involving comparable amounts of computation, the second has a larger output, inducing more materialization cost. Empirical data on a skewed workload confirms this disparity, as Figure 3 shows: materialization time varies across threads, with computation time also exhibiting thread-level variation, but in the opposite trend. Moreover, these complexities are further compounded by hardware variability: performance sensitivity to the relative costs of computation versus memory access differs considerably across hardware platforms. This architectural variance makes it exceedingly difficult for purely flop-count-based schemes to establish a load balance that is both universally effective and readily generalizable across diverse machines. Thus, such schemes are inherently limited in delivering robust load balancing.

(2) Overlooking cache contention. Existing methods also overlook the impact of cache contention during parallel execution. In Gustavson's algorithm, each thread maintains an exclusive row-level accumulator whose size scales with the nonzero density of its assigned rows. Reading Figure 2 row-wise, each row of A gathers the B rows indexed by its nonzeros into that accumulator, so cf measures the nonzeros of B streamed per accumulator entry; a high-cf row thus places significantly greater pressure on shared caches than a low-cf row. As high-cf partitions generally contain more high-cf rows and Gustavson's algorithm is typically parallelized row-wise, they are more prone to inter-thread cache interference than low-cf partitions.

Figure 4(a) shows this effect using the same skewed matrix as in Figure 3. When all threads compute their assigned default partitions, the runtime of a high-cf partition (originally assigned to thread-0 in the flop-count-based scheme) increases substantially: from 2.76 seconds in single-threaded execution (see Figure 4(b)) to 8.93 seconds under default multithreaded partitioning, and further to 13.01 seconds if all threads execute this high-cf partition. Note that no sharing benefit arises in this case, as each thread operates on its own exclusive accumulator; instead, these accumulators compete for the same shared cache, amplifying interference as more threads process high-cf regions. In contrast, a low-cf partition (originally assigned to thread-31) sees only a modest runtime increase, from 5.36 to 6.73 seconds.

Further profiling reveals that cache contention is the root cause of this disparity: Figure 4(c)–(e) shows a sharp rise in last-level cache (LLC) misses for the high-cf region, while instruction count and LLC accesses remain relatively stable, confirming cache contention as the primary bottleneck.

The problem is further exacerbated under simultaneous multithreading (SMT): Figure 4(f) shows that non-skewed workloads (e.g., ER matrices with typically low cf) benefit from SMT, whereas skewed workloads (e.g., G500 matrices with high-cf subregions) experience increasing throughput degradation as density grows, since higher density amplifies the skew and thus intensifies SMT-induced contention. This underscores the importance of contention-aware

[3]The compression factor (cf) is defined as the flop count divided by the *nnz* of the result matrix C, i.e., $cf = \frac{flop}{nnz(C)}$ [36].

[4]Only multiplications are counted, as the number of additions is identical.

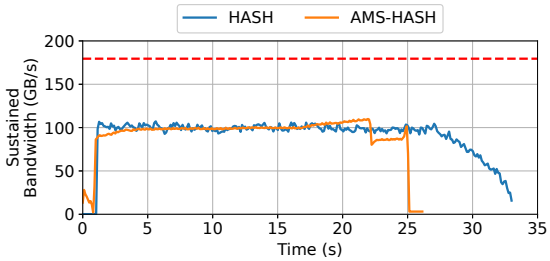
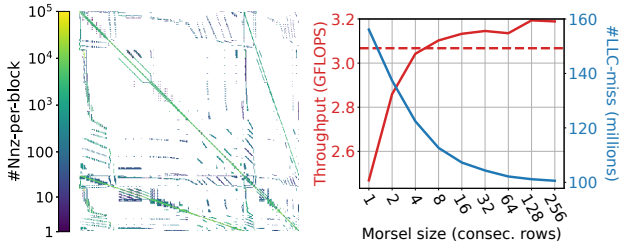


Figure 5: Comparison of sustained bandwidth of the numeric phase between flop-count-based hashing-based SpGEMM and the AMS variant on a G500 skewed workload (scale=19, *edge-factor*=16); the horizontal red dashed line denotes system memory bandwidth (179.50 GB/s).



(a) NNZ-pattern **(b) Throughput & LLC-miss**

Figure 6: (a) Nonzero pattern of the `vas_stokes_1M` matrix. (b) Throughput and LLC misses of hashing-based SpGEMM on `vas_stokes_1M` using dynamic scheduling with varying morsel sizes (measured in consecutive rows per morsel); the horizontal red dashed line indicates the throughput of the flop-count-based scheme.

scheduling; neglecting this factor not only compromises load balance but also degrades overall SpGEMM efficiency.

Ignoring multi-dimensional skew and overlooking cache contention together compound into a macro-level penalty on system-wide bandwidth utilization. As Figure 5 illustrates on a skewed workload, bandwidth resources are only fully utilized during the initial burst of parallel execution; as straggler cores emerge, system-wide bandwidth consumption starts to drop, leaving expensive memory resources idle. This renders prevailing bandwidth expansion technologies, such as CXL, far less effective, as the slowest thread governs total runtime, and its prolonged latency [87] further widens the gap despite the increased aggregate bandwidth available. In contrast, a contention-aware load-balancing approach (our AMS method) sustains markedly higher bandwidth execution.

2.3 Revisiting Dynamic Scheduling

Given these limitations, we revisit *dynamic scheduling* [69, 72]: rows of A are placed into a shared task pool, and threads repeatedly fetch and process one row at a time, achieving nearly 100% load balance. While simple and theoretically well-balanced, this approach has been shown to underperform compared to flop-count-based scheduling and has largely been dismissed in recent literature, which attributes the performance penalty primarily to the synchronization overhead associated with dynamic task dispatch [36, 69]. However, we contend that it is task granularity, rather than synchronization overhead, that plays the dominant role in determining performance. When task size is carefully managed, dynamic scheduling can match, if not outperform, flop-count-based schemes.

Take a real-world SpGEMM workload, `vas_stokes_1M` [26, 81], for example. The matrix exhibits clear spatial locality across rows, particularly within blocks of consecutive rows (Figure 6(a)). We benchmark the performance of hashing-based SpGEMM using dynamic scheduling with varying task granularities (i.e., the number of consecutive rows per task). Figure 6(b) shows that throughput improves with increasing granularity, plateauing around 128 rows, and surpasses its flop-count-based counterpart due to better load balance. Further analysis of LLC miss counts reveals that larger task sizes result in fewer misses, suggesting better preservation of spatial locality. It is worth noting that many real-world matrices exhibit strong spatial locality in their nonzero patterns, and prior work has introduced several offline reordering algorithms to enhance locality for downstream operations [4, 20, 92]. Based on these observations, we maintain that dynamic scheduling remains a practical and efficient strategy as long as spatial locality is effectively preserved. We therefore build on this insight for the design of our AMS approach.

3 METHODOLOGY

This section presents our proposed framework, adaptive morsel scheduling (AMS). We begin with the core concept of morsel-wise dynamic scheduling, our strategy for addressing multi-dimensional skew and achieving efficient load balancing in SpGEMM. Building upon this foundation, we present two key techniques, contention-aware load balancing (CALB) and data-conscious simultaneous multithreading activator (SMTact), designed specifically to alleviate on-chip cache contention. Furthermore, we explain how the morsel-wise framework enables our hybrid accumulation mechanism (HYB) approach for optimizing SpGEMM execution. Finally, we distill the transferable design pattern behind AMS and discuss its implications for relational and graph processing, for bandwidth-expanded memory, and for other sparse matrix kernels.

Before detailing our techniques, we briefly digress to outline three key design principles that an efficient SpGEMM solution should satisfy, all of which are met by our proposed AMS.

(1) **Simple and lightweight.** SpGEMM has been extensively optimized over decades [31], and any added complexity can degrade its performance. This is especially critical when SpGEMM is repeatedly invoked as a core kernel in real-world systems [18, 37, 52, 63, 69, 93]. Thus, any proposed method must be simple and lightweight to avoid introducing significant execution penalties.
(2) **Non-disruptive to non-skewed workloads.** While skewed patterns are common in real-world workloads, non-skewed distributions (e.g., uniform distribution, symmetric distribution) also arise frequently. A design that benefits skewed cases but penalizes non-skewed ones forces practitioners to make case-dependent trade-offs, increasing system complexity and operational overhead.
(3) **Generalizable across algorithms and platforms.** An effective SpGEMM solution should generalize across algorithmic variants and hardware platforms. In practice, SpGEMM is used in

diverse contexts with varying requirements, e.g., numerical computing often demands sorted output [25], while relational and graph queries typically accept non-sorted results [4, 20, 34]. Also, it should adapt to different hardware architectures, where sensitivities to compute and memory bottlenecks vary. A practical solution must therefore accommodate both, supporting sorted and non-sorted pipelines while maintaining robust performance across platforms.

3.1 Morsel-Wise Dynamic Scheduling

We previously demonstrated that, by using morsels composed of a sufficient number of consecutive rows, dynamic scheduling effectively preserves spatial locality in sparse matrices, significantly reducing LLC misses with negligible synchronization overhead (§ 2.3). We thus adopt this line of design for SpGEMM execution.

While scheduling work in morsels of a fixed row count helps reduce cache thrashing, we argue that this approach leaves room for further improvement, as it does not address the *multi-dimensional skew* identified in § 2.2. Specifically, it has two limitations: (1) Fixed-row-count morsels do not account for differences in flop count or result *nnz*, making it difficult to balance computation and materialization costs. This limitation is particularly pronounced in skewed workloads, where high-cf submatrices incur high computational expense while others involve substantial write-out overhead. Due to such disparities in two dimensions, modeling inter-thread cache contention becomes challenging. (2) This fixed-row strategy also fails to control the size of the resulting output: a single morsel containing a few dense rows can produce more data than the CPU cache can accommodate, making it challenging to construct a cache-aware analysis model [41, 67, 78].

To address these issues, we propose scheduling morsels based on a fixed result size, i.e., a target $nnz(C)$ per morsel. Recall that SpGEMM consists of two phases, where the symbolic phase determines the per-row nonzero structure. We thus leverage this information to form morsels of consecutive rows in an adaptive manner, such that each morsel contains approximately the same number of output nonzeros. By fixing the materialization cost per morsel, this approach normalizes one dimension of skew, allowing us to focus on the remaining computational cost dimension. It also decouples cf from the output size that flop-count-based partitioning leaves uncontrolled: with $nnz(C)$ fixed, cf varies solely with the flop count. Additionally, to facilitate on-chip cache modeling, we limit the morsel size to half of the L2 cache capacity [5].

This strategy is simple and efficient, forms the foundation for our subsequent designs, and introduces no additional algorithmic phase, fully aligning with our design principle of being **simple and lightweight**.

3.2 Contention-Aware Load Balancing

We now elaborate on how we alleviate inter-thread on-chip cache contention, which can degrade SpGEMM performance even under balanced workload distribution. Since we form morsels based on a fixed result size, we can easily compute the compression factor (cf) for each morsel. Recall from § 2.2 that contention originates at the row level: a high-cf row streams more nonzeros of B per accumulator entry (see Figure 2), and is thus contention-prone regardless of how rows are grouped. With $nnz(C)$ fixed per morsel, a high-cf morsel likewise carries more computation against an identical footprint, and remains more prone to contention than a low-cf one. Therefore, we aim to co-execute high-cf and low-cf morsels concurrently, alleviating cache contention and reducing susceptibility to inter-thread interference.

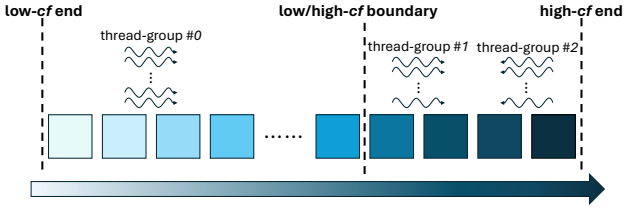


Figure 7: Thread scheduling with contention-aware load balancing (CALB). Darker colors indicate higher compression factors (cf).

To that end, we first sort all morsels in ascending order of cf and empirically set a boundary between low-cf and high-cf regions, typically between the 80th and 95th percentile. This is motivated by the observation that most skewed workloads follow a power-law distribution [4, 20, 92, 102], where a small fraction of dense rows contribute to the majority of the workload. We then divide the available threads into three groups: thread group #0 starts from the low-cf end and progresses toward the high-cf end; group #1 begins at the low/high-cf boundary and proceeds toward the high-cf end; group #2 starts at the high-cf end and moves backward toward the boundary (see Figure 7). The majority of threads are allocated to group #0 to handle low-cf tasks, while smaller subsets are assigned to groups #1 and #2 for high-cf regions. This arrangement ensures that only a limited number of high-cf morsels are processed concurrently. Dividing the remaining threads into groups #1 and #2 further helps mitigate cache contention by avoiding simultaneous execution of multiple contention-prone morsels at the high-cf end. Meanwhile, co-running low-cf morsels, which are less sensitive to cache interference, alongside high-cf morsels helps absorb contention effects and improves overall execution efficiency.

While this scheme alleviates cache contention, it does not guarantee load balance. Some thread groups may finish early and become idle, leading to underutilization of computational and memory bandwidth resources. To address this, we allow early-finished threads to assist other active groups. Specifically, if group #0 reaches the boundary early, it merges with group #1 to continue processing toward the high-cf end. Likewise, early-finished threads from groups #1 and #2 join group #0 to help complete tasks in the low-cf region. This cooperative mechanism keeps all threads active, ensuring efficient use of computation and memory bandwidth.

The CALB scheme is conceptually simple, yet novel and highly effective: to the best of our knowledge, it is the first SpGEMM load-balancing technique that explicitly considers on-chip cache contention as a scheduling factor and thereby delivers tangible performance benefits. It is also platform-agnostic and integrates seamlessly into any SpGEMM algorithm based on Gustavson's formulation, adhering to the design principle of being **generalizable across algorithms and platforms**.

[5] We empirically find that morsel sizes ranging from $\frac{1}{8}$ to $\frac{1}{2}$ of the L2 cache capacity offer the best trade-off between performance and cache contention.

3.3 Data-Conscious SMT Activation

While SpGEMM is generally considered bandwidth-bound, its computational phase still incurs a non-negligible overhead. This overhead can typically be mitigated by leveraging simultaneous multithreading (SMT), which increases the logical core count to effectively improve computational throughput. However, as shown in Figure 4(f), this performance benefit from SMT is primarily observed in non-skewed workloads; in contrast, skewed workloads degrade significantly when SMT is enabled.

We attribute this disparity to the increased on-chip cache pressure from SMT. Although SMT increases the logical thread count, it does not expand the physical cache. Consequently, high-cf morsels suffer from intensified cache contention, an issue that is particularly exacerbated in skewed workloads where cf values are disproportionately large for a few morsels. These specific morsels are highly vulnerable to SMT-induced interference, which in turn leads to substantial performance degradation.

To mitigate this issue while preserving the computational benefits of SMT for non-skewed workloads, we adopt a selective activation strategy of SMT, namely data-conscious simultaneous multithreading activator (SMTact). Specifically, we set a threshold based on the highest cf observed. SMT is disabled if the workload contains any morsels exceeding this threshold, and enabled otherwise. This strategy ensures that SMT is applied only when the workload is less prone to cache contention [6]. The approach is simple yet effective, as it balances the computational benefits of SMT against the risk of elevated cache contention, aligning with our second design principle of being **non-disruptive to non-skewed workloads**.

3.4 Hybrid Accumulation Mechanism

Due to the inherently sparse and irregular nature of SpGEMM, computing a single output row generates multiple intermediate sparse products. These products must then be merged together, a process that requires appropriate accumulators to mitigate two challenges: severe irregular memory access patterns and the high cost of managing collisions for identical column indices. Recognizing these challenges, numerous works have focused on developing efficient accumulators [25, 33, 69]. However, the effectiveness of any given accumulation strategy varies dramatically based on the characteristics of the underlying matrix morsel. This makes a static, one-size-fits-all accumulator inherently suboptimal.

To address this, we leverage the morsel-wise framework of AMS to introduce a hybrid accumulation mechanism (HYB) that dynamically selects the most suitable accumulator for each morsel. Specifically, we make this choice based on the aforementioned morsel-wise cf value, selectively choosing between two widely used accumulators: hash tables (HASH) [68, 69], which use linear probing and multiplicative hashing to reduce collisions, and expand-sort-compress (ESC) [25, 36], which relies on highly optimized radix sorting to alleviate common logarithmic sorting overheads.

The choice between these two accumulators is highly dependent on the morsel-wise cf. As ESC generates sorted output, it executes more instructions than HASH. However, this instruction overhead is not necessarily detrimental, particularly for low-cf morsels. For such morsels, the number of intermediate products is similar to the final number of nonzeros, leading to a moderate cost for merging and sorting. In contrast, HASH becomes more populated for low-cf morsels, and the overhead of hash computation and probing becomes more pronounced.

For high-cf morsels, however, HASH is the better choice, as a massive number of intermediates are hashed into a few slots, greatly reducing potential cache misses. Conversely, ESC must pay a significant cache miss cost to merge a large volume of intermediate results into a small final output. Microbenchmark results in Figure 8 corroborate this finding: the ratio of ESC's LLC misses to HASH's is substantially higher on skewed G500 workloads with high cf than on non-skewed ER workloads.

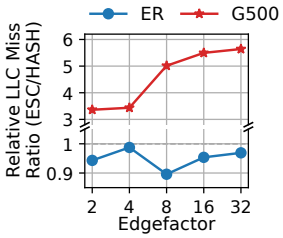


Figure 8: The LLC miss ratio of ESC normalized to HASH (i.e., LLC Misses of ESC / LLC Misses of HASH), evaluated on ER and G500 datasets at scale 18 with varying *edgefactor*.

Therefore, we empirically identify a cf threshold, using ESC for morsels with a cf below this value and opting for HASH for morsels above it [7]. This morsel-wise selection also offers an architectural benefit: selecting the accumulator on a per-morsel basis, rather than per-row, reduces the branch-misprediction overhead that would arise from finer-grained, per-row selection. Consequently, this adaptive approach ensures each morsel is processed efficiently, leading to better performance for the overall SpGEMM execution.

It is worth noting that by including HASH, our HYB approach may produce a non-sorted matrix, which is acceptable for applications like relational queries and graph analytics that do not require sorted output [50, 68]. However, if sorted output is necessary, HYB can be easily adapted: instead of choosing between HASH and ESC, it can be configured to choose the efficient strategy from a set of sorted accumulators, such as ESC, HEAP [69], SPA [33], or even a hash-then-sort approach [22], thereby ensuring sorted output.

3.5 Implications and Broader Applicability

AMS highlights an important interaction between load balance and cache efficiency. Static partitioning can preserve locality, but it may leave cores idle under skew and therefore underutilize the available memory bandwidth. Dynamic scheduling mitigates this imbalance by keeping cores busy, but it can increase cache contention when the combined working sets of concurrently executed morsels exceed the shared-cache capacity. Effective scheduling must account for both effects: it should maintain sufficient concurrency to utilize the memory system while limiting avoidable cache traffic. This interaction is particularly relevant to bandwidth-expanded memory [97], where additional bandwidth becomes useful only when enough cores issue concurrent requests. The combination of balanced execution and reduced cache interference consequently allows AMS to utilize bandwidth beyond the local-DRAM peak (see § 4.4).

[6] We also experiment with using quantiles of the cf distribution instead of the maximum, and observe comparable performance with an appropriately chosen threshold.

[7] We find that a cf of 1.5–1.7 provides a robust threshold for selecting the appropriate accumulator.

Table 1: Summary of all evaluated SpGEMM methods

Method	Category	Sorted	Description
Algorithmic baselines (built upon by our AMS methods)			
`HASH [69]`	Baseline	✗	Hash table accumulator with flop-count-based scheduler.
`ESC [25]`	Baseline	✓	Radix-sort (expand-sort-compress) accumulator with flop-count-based scheduler.
Industry solutions			
`MKL [52]`	Competitor	✗	Intel oneMKL SpGEMM — non-sorted variant.
`MKLS [52]`	Competitor	✓	Intel oneMKL SpGEMM — sorted variant.
Proposed AMS-based methods			
`AMS-HASH`	Proposed	✗	AMS approach using a HASH accumulator.
`AMS-ESC`	Proposed	✓	AMS approach using an ESC accumulator.
`AMS-HYB`	Proposed	✗	AMS with per-morsel adaptive HASH/ESC selection.

The same design principle can apply beyond SpGEMM when lightweight metadata is available to estimate each morsel's processing cost and demand on the shared cache. For SpGEMM, the symbolic phase provides this information directly. Other operators can obtain approximate metadata through a preliminary statistics pass [6, 21] or sampling [95, 96]; exact estimates are unnecessary because CALB uses only the relative ordering of morsels by cache demand and SMTact only tests whether a threshold is crossed. This creates opportunities to apply contention-aware morsel scheduling to relational and graph workloads with skewed, cache-resident state [3, 27, 34, 50, 86]. For example, a morsel-driven engine [29, 61] could use partition-size estimates to approximate cache demand when scheduling hash-aggregation or hash-join tasks over skewed keys. The principle can also be adapted to sparse algebra and graph kernels, using a preliminary metadata-collection pass when the required estimates are not readily available.

4 EXPERIMENTAL EVALUATION

This section evaluates AMS in addressing multi-dimensional skew and on-chip cache contention. We begin with the experimental setup, followed by matrix squaring benchmarking across diverse workloads, a dissection of CALB and SMTact to validate their contention alleviation, and evaluations on a bandwidth-expanded CXL system, relational join-project queries, and meta-path network construction to demonstrate its broad applicability.

4.1 Experimental Setup

We conduct experiments on a single-socket Intel Xeon Gold 6430 system (Linux kernel 6.11.0) with 32 physical cores and 64 SMT threads. Each core includes a 48 KB L1 data cache, a 32 KB L1 instruction cache, and a 2 MB L2 cache, while all cores share a 60 MB last-level cache (LLC) and 192 GB of DDR5 DRAM providing 179.50 GB/s of bandwidth. AMS is further validated on a CXL-augmented memory system, with setup details deferred to § 4.4.

Following prior settings [16, 18, 68], we primarily analyze matrix squaring on R-MAT graphs [19, 32], namely the non-skewed ER and the skewed G500 workloads. We sweep *scale* (i.e., 2^{scale} rows) over 19–21 for ER and 17–19 for G500, with *edgefactor* over 2–32 in both; a higher *edgefactor* amplifies multi-dimensional skew and cache contention in G500 (§ 2.2). We further report 20 representative SuiteSparse matrices [26] to validate the robustness of AMS. Beyond SpGEMM, we evaluate relational join-project queries and PathSim meta-path construction [86], with setups detailed in § 4.5 and § 4.6. We also evaluate join-aggregate queries [50], breadth-first search [28], triangle counting [88], and other sparse kernels [62, 99, 102] and report the results in our full report [45].

We integrate our AMS framework into two prevalent Gustavson-based SpGEMM algorithms, the non-sorted HASH [68, 69] and the sorted ESC [11, 25], referred to as AMS-HASH and AMS-ESC. We also implement HYB as AMS-HYB, which adaptively selects between the two accumulators. We empirically set the morsel size to half the L2 cache and split threads 60%-25%-15% across groups #0–#2 for optimal CALB performance.

We compare against the same HASH and ESC algorithms driven by a state-of-the-art flop-count-based scheduler and the de facto industry solution, Intel MKL [52], in both its sorted (MKLS) and non-sorted (MKL) variants (see Table 1 for an overview). All implementations are compiled using `GCC-11.5.0` with the `-O3` optimization flag and parallelized with OpenMP [24]. Baselines are manually tuned for optimal performance, including enabling SMT for non-skewed workloads and disabling it for skewed ones; our AMS-based methods are tuned likewise, though a detailed sensitivity study (e.g., thread group sizes, the low/high-cf boundary, and the SMTact threshold) is omitted due to space limits. As in prior works, throughput in GFLOPS (giga floating-point operations per second) serves as the primary metric, and all results are averaged over 10 runs.

4.2 Matrix Squaring Performance Evaluation

Figure 9 presents the matrix squaring performance on R-MAT and SuiteSparse workloads, and shows our AMS-based algorithms consistently achieve high performance across this broad spectrum. Specifically, the proposed AMS-HYB is generally the superior solution: it maintains performance comparable to AMS-HASH in skewed workloads (G500) while remaining as competitive as AMS-ESC in non-skewed workloads (ER) with high *edgefactor* values. This outcome validates both the effectiveness of our HYB design and the soundness of our cf-based criterion for adaptively selecting the best morsel-wise accumulator for efficient multiplication.

Beyond the superior performance of AMS-HYB, the AMS framework itself significantly boosts both its non-sorted (AMS-HASH) and sorted (AMS-ESC) variants. These methods perform comparably to their respective baselines on ER matrices but deliver a substantial performance advantage on the skewed G500 matrices. This demonstrates that AMS's dynamic scheduling, when paired with appropriately sized morsels, effectively addresses multi-dimensional skew on skewed workloads without sacrificing performance on non-skewed ones. The strong results on ER matrices (Figure 9(a)–(c)) also validate the effectiveness of SMTact, which selectively enables SMT in low-contention scenarios to enhance computational throughput, a critical factor in non-skewed workloads.

Comparing the impact of AMS across different algorithms, we observe that the performance improvement is more pronounced in HASH than in ESC. This is because ESC relies on radix sort, which has a complexity of $O(\omega \cdot n)$ and correlates directly with the flop count. Although HASH-based methods also exhibit $O(n)$ complexity in hash table lookups, their efficiency is more susceptible to

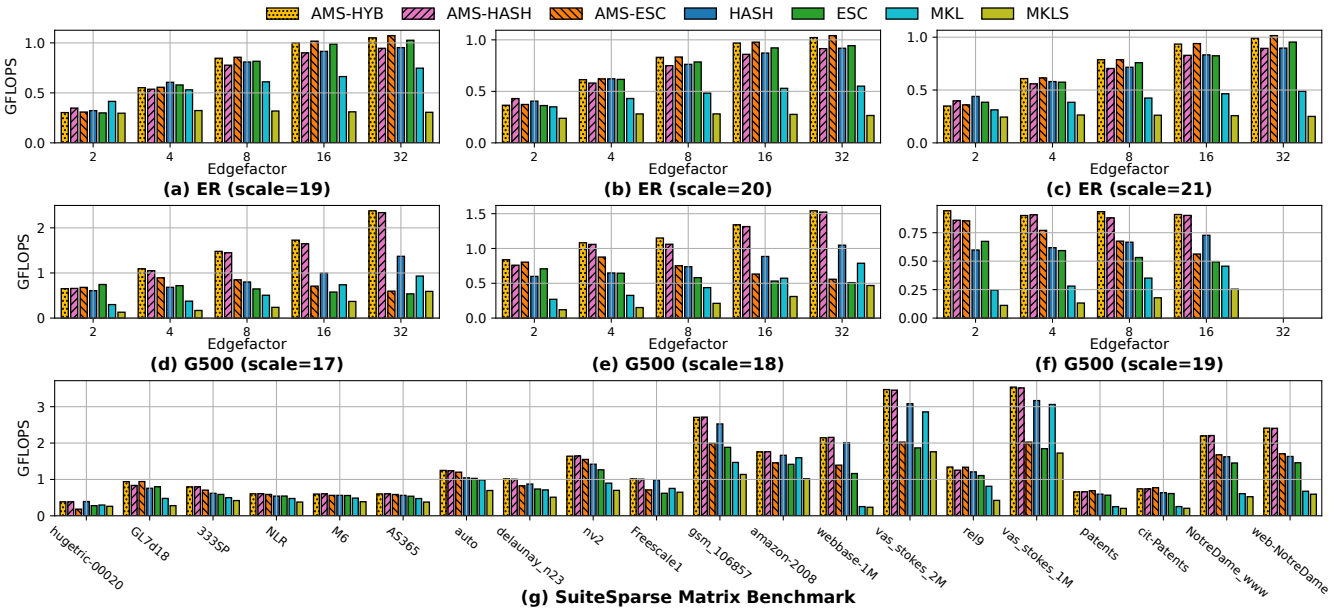


Figure 9: SpGEMM performance on (a)–(f) R-MAT matrices (ER and G500) across varying scales and *edgefactor* values, and (g) 20 representative SuiteSparse matrices. All algorithms run out of memory on G500 at scale 19 and *edgefactor* 32.

"clustering" effects, making flop-count-based models less effective in accurately estimating cost and balancing workloads. In contrast, our AMS approach is agnostic to the underlying SpGEMM algorithm. For both non-sorted and sorted algorithms, AMS achieves performance gains of up to 81.01% and 36.03%, respectively, demonstrating its general applicability across algorithmic variants.

Additionally, Figure 9 shows that AMS-based algorithms may occasionally underperform their flop-count-based counterparts in highly sparse cases (e.g., G500 with scale 17 and *edgefactor* 2), primarily due to overhead from cf-based morsel sorting. Nevertheless, this penalty diminishes as matrix density increases, underscoring the lightweight nature of AMS in addressing load imbalance.

Finally, we compare AMS against the de facto industry solution, Intel MKL. On R-MAT workloads, AMS-HYB surpasses the non-sorted MKL by up to 3.83× and the sorted MKLS by up to 8.45×, while AMS-ESC delivers up to 7.66× over MKLS. The advantage persists on the SuiteSparse matrices of Figure 9(g), where AMS-HYB leads MKL and MKLS on all 20 matrices, while AMS-HASH and AMS-ESC outperform their flop-count-based counterparts on 19 of 20 matrices, by up to 47.46% and 25.92%, respectively, confirming that AMS generalizes to real-world sparsity patterns.

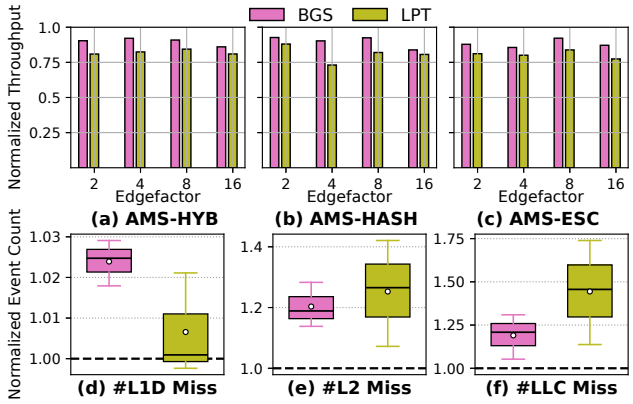


Figure 10: Throughput and cache behavior of CALB compared with the BGS and LPT strategies on G500 workloads (scale=19), all normalized to the corresponding CALB-based execution. (a)–(c): Throughput of AMS-HYB, AMS-HASH, and AMS-ESC. (d)–(f): Distribution of L1D, L2, and LLC misses normalized to CALB across all three AMS variants.

4.3 Dissecting AMS: CALB and SMTact

4.3.1 Effectiveness of CALB. We now examine the performance benefits of our CALB approach. Designed to alleviate cache contention while maintaining load balance, CALB is evaluated against two alternative scheduling techniques also applicable to our cf-sorted morsels: (1) a bidirectional greedy scheduling scheme (BGS), where threads split into two equal-sized groups processing morsels from opposite ends toward the center, and (2) the classical longest-processing-time-first strategy (LPT) [35], in which all threads start from the high-cf end and progress toward the low-cf end.

Figure 10(a)–(c) reports the throughput of BGS and LPT, normalized to our CALB method, for three AMS-based algorithms on G500 workloads (scale=19). The results show that CALB consistently outperforms both alternatives, achieving 7.90%–19.26% and 8.53%–36.80% higher throughput than BGS and LPT, respectively. While BGS also aims to mitigate contention by avoiding excessive thread concentration on high-cf regions, our three-group division strategy in CALB proves more effective at reducing cache pressure, especially at the high-cf end. In contrast, LPT performs poorly precisely because it schedules all threads to handle the most

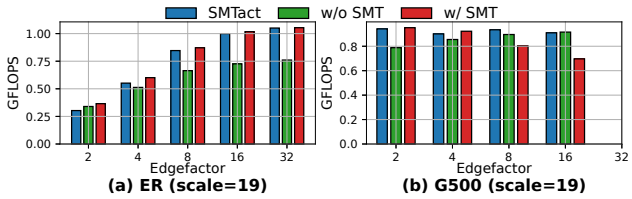


Figure 11: Effectiveness of SMTact in managing SMT to improve AMS-HYB performance on non-skewed (ER) and skewed (G500) workloads.

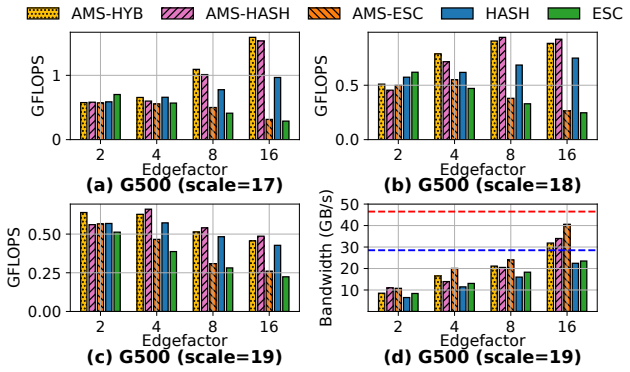


Figure 12: Performance of SpGEMM on G500 workloads. (a)–(c): Measured throughput at G500 scales 17, 18, and 19. (d): Sustained bandwidth measured for different algorithms for the scale-19 workload (the red and blue dashed lines indicate the peak bandwidth of the CXL-augmented memory system (46.50 GB/s) and host DRAM (28.50 GB/s), respectively).

demanding, high-contention tasks simultaneously, thereby exacerbating on-chip cache contention.

To directly attribute this advantage to reduced cache contention, Figure 10(d)–(f) profiles the L1D, L2, and LLC misses incurred by BGS and LPT, again normalized to CALB. While the L1D miss counts remain nearly identical, the divergence emerges deeper in the hierarchy: BGS and LPT incur 1.14–1.28× and 1.07–1.42× more L2 misses than CALB, and LPT additionally exhibits the highest LLC miss counts. This validates the rationale behind CALB: by dispersing threads across the cf spectrum instead of concentrating them at the high-cf end, CALB reduces the aggregate cache pressure of concurrently processed morsels, confirming that effective load balancing must account for cache contention. Notably, LPT has been a seminal scheduling strategy for over 50 years [35], and our findings highlight the need to revisit and refine classical techniques with the idea of cache pressure alleviation.

4.3.2 Effectiveness of SMTact. Our SMTact is designed as an adaptive SMT-enablement strategy by balancing the trade-off between increased computational capability and elevated cache pressure. To evaluate its effectiveness, we compare SMTact against two fixed configurations: SMT always enabled (w/ SMT) and SMT always disabled (w/o SMT). The results, presented in Figure 11, include both non-skewed and skewed workloads. As illustrated, neither fixed configuration is uniformly viable: w/ SMT falls up to 23.9% short of the best configuration on the skewed G500 workloads, where the extra logical threads aggravate cache contention, whereas w/o SMT forfeits up to 28.6% on the non-skewed ER workloads that profit from the additional thread-level parallelism.

Since SMTact ultimately executes as one of these two configurations, its goal is not to surpass both in every case, but to automatically select a configuration at or close to the best. In our experiments, SMTact stays within 8.2% of the best fixed configuration in eight of the nine cases. The exception is the extremely sparse ER matrix at *edgefactor* = 2, where SMTact falls 17.1% short of the best fixed configuration because the sorting overhead introduced by CALB is not yet amortized; this penalty is quickly offset as *edgefactor* or matrix scale increases. Given the wide variability in sparsity patterns in real-world SpGEMM workloads, SMTact thus offers a robust default that avoids the substantial degradation that either fixed policy incurs outside its sweet spot.

4.4 Evaluation on Bandwidth-Expanded System

We now evaluate AMS on a bandwidth-expanded memory system, where CXL boosts aggregate bandwidth but introduces higher access latency. As discussed in § 1, load imbalance can inadvertently amplify CXL's latency penalty on straggler threads, negating the bandwidth gain and making effective load balancing particularly critical. For this evaluation, the machine has the same CPU and OS configuration as our previous platform but features host DRAM of 28.50 GB/s bandwidth. Additionally, a CXL expansion memory of 20.95 GB/s bandwidth is installed, yielding a combined system bandwidth of 46.50 GB/s [49, 59].

We evaluate AMS approaches against their vanilla counterparts using skewed (G500) workloads and report the results in Figure 12. As shown, AMS-based methods generally outperform the other methods (Figure 12(a)–(c)). Furthermore, the profiling results confirm that AMS effectively utilizes the additional CXL-augmented bandwidth: AMS-ESC sustains a bandwidth of up to 40.60 GB/s, far exceeding the DRAM-only peak (28.50 GB/s) (Figure 12(d)). In contrast, the bandwidth consumption of the vanilla counterparts does not exceed that of the host DRAM, indicating that AMS indeed improves bandwidth utilization by addressing load imbalance.

We also observe an interesting trend in Figure 12(a)–(c): while AMS-based methods remain competitive on skewed workloads, the performance gap between AMS and the vanilla baselines varies between G500 matrices at scale 17 versus those at scales 18 and 19. Taking the HASH-based variants as an example, AMS-HASH outperforms HASH by 58.77% on the G500 matrix at scale 17, delivering 1.53 GFLOPS versus 0.97 GFLOPS (*edgefactor* 16). However, this advantage diminishes to 23.16% at scale 18, with a similar trend observed at scale 19.

This reduction in the performance gap is mainly attributed to the coincidental load balance occasionally achieved by the static flop-count-based scheme, and it is the extent of this imbalance that largely determines the performance gap of baselines relative to their AMS counterparts. As Figure 3 illustrates, computation and materialization times can exhibit opposing trends across threads.

Table 2: Summary of 6 join-project workloads.

Dataset	$\|user\|$	$\|tuple\|$	Gini	$\|OUT_J\|$	$\|OUT_P\|$	$\|OUT_J\|/\|OUT_P\|$
IMDB [51]	733.47K	7.52M	0.3453	214.77M	176.91M	1.2140
CLH0 [79]	850.00K	2.90M	0.5858	2.43B	2.34B	1.0415
RD23 [82]	6.72K	3.91M	0.5276	4.61B	45.02M	102.3613
BKGN [57]	350.33K	5.15M	0.7242	11.43B	6.77B	1.6892
CLH1 [79]	850.00K	24.05M	0.3862	20.34B	11.41B	1.7823
MVLN [50]	138.49K	20.00M	0.5807	269.61B	16.80B	16.0477

1. $|user|$ and $|tuple|$ are equivalent to the matrix row count and total nnz count, respectively, reflecting the relational-to-sparse-matrix duality; Gini captures the row-wise nnz distribution [58].
2. $|OUT_J|$ and $|OUT_P|$ represent the cardinalities of the join result and the final projected result, respectively.

This is precisely the multi-dimensional skew identified in § 2.2: computation and materialization costs trend in opposite directions, and any single-metric scheme can only coincidentally balance them. In some cases, these opposing patterns may randomly offset each other, resulting in a seemingly balanced overall execution time despite the underlying skew. This balance, however, is merely incidental and heavily dependent on the platform's relative sensitivity to computation versus memory access. Consequently, it fails to provide consistent or portable load balance across architectures.

Previous results on a 179.50 GB/s bandwidth memory corroborate this explanation (see Figure 9(d)–(f)). With this higher bandwidth, AMS-HASH achieves a 45.44% performance increase over HASH (1.52 GFLOPS vs. 1.05 GFLOPS) on the G500 matrix at scale 18, because the increased bandwidth significantly reduces the platform's vulnerability to materialization cost. Collectively, these results validate our claim that AMS provides a robust and portable solution to load balancing across diverse systems.

4.5 Join-Project Query Evaluation

We now validate AMS on relational workloads, where modeling tables as sparse matrices allows SpGEMM to evaluate join-project queries without materializing the intermediate join result [1, 3, 54], a key advantage for many-to-many and self-joins that produce large intermediate cardinalities.

Regarding workloads, we adopt widely used datasets from prior work [27, 47, 50, 84], and additionally evaluate on an industrial OLAP ClickHouse benchmark [79] with two sampled workloads: CLH0, a uniform 5% sample of users, and CLH1, the top 5% of users by pageviews (see Table 2 for key statistics). All these workloads exhibit skew to varying degrees, as quantified by the Gini coefficient in Table 2. Notably, the uniform sample exhibits a higher Gini coefficient than the top-user sample (0.59 versus 0.39), because selecting only the heaviest users excludes the low-activity tail, whereas uniform sampling largely preserves the original population's skew. As for the query form, we perform a self-join (i.e., $R = S$) with projection in the form:

```
SELECT DISTINCT R.user, S.user
FROM R, S
WHERE R.item = S.item AND R.user <> S.user
```

This query measures user-wise similarity, where "item" abstracts the shared attribute across scenarios, e.g., URL in ClickHouse (CLH0 and CLH1) and movie in MovieLens (MVLN). We compare AMS-HYB against DIM3 [50], the state-of-the-art SpGEMM-based join-project method; NPHJ-PRJ, which projects on-the-fly with a fast

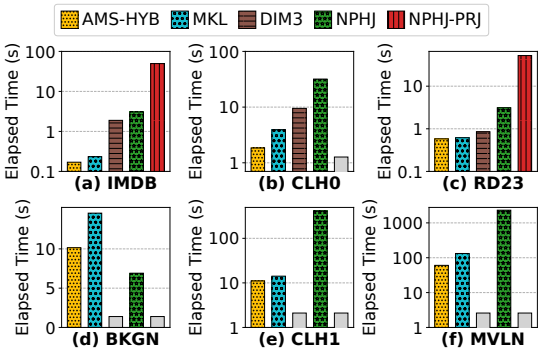


Figure 13: Execution time across six join-project workloads, including the tuple-to-CSR conversion for matrix-based methods; gray bars denote algorithms that run out of memory, and every vertical axis but (d) is logarithmic.

hash table [80] during the probing phase of non-partitioned hash join (NPHJ) [9, 46, 78], also bypassing intermediate materialization; the industry-standard MKL with non-sorted output; and NPHJ itself as a join-only reference. Traditional "join-then-project" [27, 50] and partitioned hash join [5–7, 15] are omitted as they exhaust memory and cannot finish any run. Following prior work [3, 50], we model join-project as a Boolean matrix product, letting AMS and DIM3 terminate the accumulation of a cell once it is known to be nonzero, while the proprietary MKL offers only a numerical product and cannot exploit this property.

It should be noted that in this evaluation, the table inputs are in the original row-wise tuple format ⟨row-index, column-index⟩, e.g., ⟨user, URL⟩ for the ClickHouse workloads, where a recorded pair is the nonzero entry itself, which is exactly the coordinate format (COO) of a sparse Boolean matrix [30, 56, 76]. The NPHJ-based methods consume these tuples directly, whereas the matrix-based methods require CSR and therefore incur a preliminary conversion pass. We perform this tuple-to-CSR conversion following the common practice of mainstream sparse-matrix libraries [37, 93], and charge its time to all matrix-based methods, though the cost is low enough that the input format is not a performance concern (as detailed in our full technical report [45]).

Figure 13 presents the join-project results. As shown, AMS-HYB consistently achieves the best performance among all join-project variants, demonstrating its effectiveness in addressing relational join-project queries. Surprisingly, AMS-HYB even outperforms the join-only baseline NPHJ in five of the six cases, highlighting the superiority of SpGEMM methods in handling joins with large output cardinalities. Comparing the two runners-up, MKL and DIM3, DIM3 is 3.06× slower and fails to finish on some workloads due to out-of-memory errors. This is because DIM3 follows an "estimate-partition-multiplication" paradigm, where a few dense rows under skew inflate its output-size estimate far beyond the actual result, so the over-allocated partitions exhaust memory on BKGN, CLH1, and MVLN, the three workloads with the largest join outputs. Notably, AMS-HYB outperforms MKL by up to 2.19× on MVLN and 2.13× on CLH0, confirming that its ability to address multi-dimensional skew and alleviate on-chip cache contention translates directly to these relational workloads. Regarding NPHJ-PRJ, it completes only

Table 3: M_l construction on DBLP V10 (seconds).

Meta-path	$\|row\|$	$\|nnz\|$	Gini	AMS-HYB	MKL	Speedup
APCPA	1.77M	4.98M	0.62	0.052	0.117	2.24×
CPAPC	5.08K	4.98M	0.78	0.067	0.120	1.80×
APTPA	1.77M	48.48M	0.60	0.158	0.529	3.34×
TPAPT	120.69K	48.48M	0.94	0.235	0.382	1.62×
Geomean	–	–	–	0.107	0.231	2.16×

on IMDB and RD23, the two workloads with the smallest projected output. This is because the on-the-fly projection hash table must grow during probing, typically doubling in size, and thus exhausts all available memory. NPHJ-PRJ is moreover an order of magnitude slower than NPHJ, indicating the significant cost of the on-the-fly projection and underscoring the importance of SpGEMM-based methods.

4.6 PathSim Meta-Path Construction

We finally evaluate AMS on PathSim [86], a meta-path-based similarity measure for heterogeneous information networks, focusing on the intermediate matrices it must construct. Given a symmetric meta-path, PathSim scores node pairs through a commuting matrix $M = M_l M_l^T$, whose left half M_l is a chain of sparse matrix products. Constructing M_l is therefore an SpGEMM workload, and it dominates the cost of answering a query.

We evaluate on the AMiner DBLP V10 citation network [89], comprising 1.77M authors (A), 3.08M papers (P), 5,078 venues (C), and 120,685 terms (T), with 9.48M author-paper, 2.57M venue-paper, and 22.64M paper-term edges. We select four symmetric meta-paths that place each node type in the row dimension: APCPA and APTPA relate authors through shared venues and terms, whereas TPAPT and CPAPC relate terms and venues through shared authors. As Table 3 summarizes, they differ widely in both shape and skew, and form two transpose pairs (APCPA/CPAPC and APTPA/TPAPT) that perform the same work over row counts differing by 350× and 15×. All four exhibit pronounced row-wise skew (Gini 0.60–0.94), reflecting prevalent skew in real-world graphs. We compare AMS-HYB against the industry-standard MKL and report the SpGEMM time of the M_l construction.

As reported in Table 3, AMS-HYB outperforms MKL on all four meta-paths, by 1.62× to 3.34×, giving a geometric-mean speedup of 2.16×. These gains span a 350× range of row counts, from the 5,078-row CPAPC to the 1.77M-row APTPA, indicating that the benefit of AMS does not depend on a particular matrix shape. Within each transpose pair, however, the gain is consistently larger for the variant with more rows (2.24× against 1.80×, and 3.34× against 1.62×), as the greater computational work associated with the larger row count helps amortize the cost of cf-based morsel sorting. Overall, AMS extends its advantage beyond matrix squaring and relational queries to meta-path construction.

5 RELATED WORK

SpGEMM has been extensively studied in scientific computing and graph analytics [25, 71], and has more recently emerged in relational query processing [27, 42, 43, 50, 56]. These relational works, however, target algorithmic mapping (e.g., DIM3 [50]) and kernel choice from local density (e.g., SPALM [56]), rather than the load imbalance and cache contention that AMS addresses. Beneath such mappings and choices lies the kernel itself, where Gustavson's row-row formulation [39] dominates for its natural parallelizability, especially when paired with the dense array accumulator SPA [33]. However, SPA's high memory overhead prompted more memory-efficient alternatives such as hash tables [69, 75], heaps [68], and radix sort [25].

Load balancing in parallel SpGEMM has also drawn much attention. Flop-count-based schemes [68, 69] have emerged as the dominant solution, alongside static strategies using blocks [16, 17] or graph partitioning [2]. These approaches, however, often fail to account for the multi-dimensional skew of real-world workloads, resulting in imbalanced execution. As a more adaptive alternative, dynamic scheduling offers greater flexibility. For instance, Patwary et al. [72] proposed a dynamic task scheduling scheme that partitions work across both row and column dimensions. However, task partitioning across two dimensions introduces substantial synchronization overhead, which limits its practical applicability. Similarly, techniques leveraging work-stealing [13, 14, 101] and cache-access modeling [38] have been explored, but either incur significant cache-coherence overhead or fail to account for complicated LLC sharing, making them less favorable than flop-count-based approaches. Critically, none of them account for the on-chip cache contention caused by concurrently executing dense submatrices, which AMS explicitly addresses alongside multi-dimensional skew.

Prior work has likewise studied task granularity in SpGEMM through one- and two-dimensional blocking [16, 17], sparse tiles [40, 70], and adaptive panels [100]. These approaches balance spatial locality against the flexibility to distribute work across threads. A similar trade-off arises in data systems, where morsel-driven parallelism [29, 61] distributes fixed-size groups of *input* tuples across workers. AMS adopts the morsel abstraction but varies the input size to maintain a roughly fixed *output* volume per morsel. It also orders morsel execution to reduce shared-cache contention, which conventional morsel-driven parallelism does not explicitly address. The approaches are therefore complementary, and AMS's techniques could benefit morsel-driven engines by using partition-size estimates to guide dispatch order and SMT activation for hash-based operators under skew.

6 CONCLUSION

This paper presents adaptive morsel scheduling (AMS), a morsel-wise dynamic scheduling framework for two co-existing challenges in SpGEMM, namely *multi-dimensional skew* and *on-chip cache contention*. AMS balances skewed workloads without aggravating cache contention through contention-aware load balancing (CALB), judiciously activates SMT via data-conscious simultaneous multi-threading activator (SMTact), and adaptively selects the per-morsel accumulator with hybrid accumulation mechanism (HYB). Across matrix squaring, relational queries, and graph analytics, AMS significantly outperforms the state of the art on skewed workloads, remains comparable on non-skewed ones, and improves bandwidth utilization, all of which hold on both conventional memory and CXL-expanded memory systems.

REFERENCES

[1] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. 2018. LevelHeaded: A Unified Engine for Business Intelligence and Linear Algebra Querying. In *ICDE*. IEEE Computer Society, 449–460.

[2] Kadir Akbudak and Cevdet Aykanat. 2014. Simultaneous Input and Output Matrix Partitioning for Outer-Product-Parallel Sparse Matrix-Matrix Multiplication. *SIAM J. Sci. Comput.* 36, 5 (2014).

[3] Rasmus Resen Amossen and Rasmus Pagh. 2009. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*. 121–126.

[4] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *IPDPS*. IEEE Computer Society, 22–31.

[5] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (2013), 85–96.

[6] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*. IEEE Computer Society, 362–373.

[7] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2015. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1754–1766.

[8] Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. 2016. Hypergraph Partitioning for Sparse Matrix-Matrix Multiplication. *ACM Trans. Parallel Comput.* 3, 3 (2016), 18:1–18:34.

[9] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *SIGMOD Conference*. ACM, 168–180.

[10] Richard Barrett, Michael Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. 1994. *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM.

[11] Nathan Bell, Steven Dalton, and Luke N Olson. 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152.

[12] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD Conference*. ACM, 37–48.

[13] Robert D. Blumofe. 1994. Scheduling Multithreaded Computations by Work Stealing. In *FOCS*. IEEE Computer Society, 356–368.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP*. ACM, 207–216.

[15] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*. Morgan Kaufmann, 54–65.

[16] Aydin Buluç and John R. Gilbert. 2008. Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication. In *ICPP*. IEEE Computer Society, 503–510.

[17] Aydin Buluç and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *IPDPS*. IEEE, 1–11.

[18] Aydin Buluç and John R. Gilbert. 2011. The Combinatorial BLAS: design, implementation, and applications. *Int. J. High Perform. Comput. Appl.* 25, 4 (2011), 496–509.

[19] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *SDM*. SIAM, 442–446.

[20] YuAng Chen and Yeh-Ching Chung. 2022. Workload Balancing via Graph Reordering on Multicore Systems. *IEEE Trans. Parallel Distributed Syst.* 33, 5 (2022), 1231–1245.

[21] Yu Chen and Ke Yi. 2017. Two-Level Sampling for Join Size Estimation. In *SIGMOD Conference*. ACM, 759–774.

[22] Helin Cheng, Wenxuan Li, Yuechen Lu, and Weifeng Liu. 2023. HASpGEMM: Heterogeneity-Aware Sparse General Matrix-Matrix Multiplication on Modern Asymmetric Multicore Processors. In *ICPP*. ACM, 807–817.

[23] Transaction Processing Performance Council. 2021. TPC BENCHMARKTM DS Standard Specification Revision 3.2.0. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v3.2.0.pdf Accessed: October 2026.

[24] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[25] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)* 41, 4 (2015), 1–20.

[26] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1:1–1:25.

[27] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2020. Fast join project query evaluation using matrix multiplication. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1213–1223.

[28] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. 2020. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *Proc. VLDB Endow.* 13, 9 (2020), 1598–1613.

[29] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. 2019. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. *Proc. VLDB Endow.* 12, 12 (2019), 2218–2229.

[30] Patrick J. Flynn, Xinyao Yi, Erik Saule, Gokcen Kestor, and Yonghong Yan. 2025. SpMM-Bench: Performance Characterization of Sparse Formats for Sparse-Dense Matrix Multiplication. In *IPDPS (Workshops)*. IEEE, 149–158.

[31] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. 2023. A Systematic Survey of General Sparse Matrix-matrix Multiplication. *ACM Comput. Surv.* 55, 12 (2023), 244:1–244:36.

[32] Sen Gao, Shengliang Lu, Shixuan Sun, Yuchen Li, and Bingsheng He. 2025. An Efficient Memoization Engine for Concurrent Graph Query Processing. In *ICDE*. IEEE, 1015–1028.

[33] John R Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM journal on matrix analysis and applications* 13, 1 (1992), 333–356.

[34] John R. Gilbert, Steven P. Reinhardt, and Viral B. Shah. 2006. High-Performance Graph Algorithms from Parallel Sparse Matrices. In *PARA (Lecture Notes in Computer Science)*, Vol. 4699. Springer, 260–269.

[35] Ronald L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics* 17, 2 (1969), 416–429.

[36] Zhixiang Gu, Jose Moreira, David Edelsohn, and Ariful Azad. 2020. Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication using Propagation Blocking. In *SPAA*. ACM, 293–303.

[37] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen. https://libeigen.gitlab.io/ Accessed: October 2026.

[38] Jihu Guo, Rui Xia, Jie Liu, Xiaoxiong Zhu, and Xiang Zhang. 2024. CAMLB-SpMV: An Efficient Cache-Aware Memory Load-Balancing SpMV on CPU. In *ICPP*. ACM, 640–649.

[39] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (1978), 250–269.

[40] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *PPoPP*. ACM, 300–314.

[41] Chuhe Hong, Qinglin Wang, Runzhang Mao, Yuechao Liang, Rui Xia, and Jie Liu. 2024. SaSpGEMM: Sorting-Avoiding Sparse General Matrix-Matrix Multiplication on Multi-Core Processors. In *ICPP*. ACM, 1166–1175.

[42] Xiao Hu. 2025. Output-optimal algorithms for join-aggregate queries. *Proceedings of the ACM on Management of Data* 3, 2 (2025), 1–27.

[43] Xiao Hu and Ke Yi. 2020. Parallel algorithms for sparse matrix multiplication and join-aggregate queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 411–425.

[44] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. TCUDB: Accelerating Database with Tensor Processors. In *SIGMOD Conference*. ACM, 1360–1374.

[45] Wentao Huang, Qiming Huang, Yunhong Ji, Mian Lu, Yuqiang Chen, Bingsheng He, and Kian-Lee Tan. 2026. Taming Multi-Dimensional Skew in Sparse Matrix Multiplication with Contention-Aware Scheduling [Technical Report]. https://github.com/fukien/fukien.github.io/blob/master/assets/works/VLDB-2027/contention-aware-spgemm/main-tr.pdf

[46] Wentao Huang, Yunhong Ji, Xuan Zhou, Bingsheng He, and Kian-Lee Tan. 2023. A Design Space Exploration and Evaluation for Main-Memory Hash Joins in Storage Class Memory. *Proc. VLDB Endow.* 16, 6 (2023), 1249–1263.

[47] Wentao Huang, Yuchen Li, Yuan Fang, Ju Fan, and Hongxia Yang. 2020. BiANE: Bipartite Attributed Network Embedding. In *SIGIR*. ACM, 149–158.

[48] Wentao Huang, Mian Lu, and Kian-Lee Tan. 2026. Hash Joins Meet CXL: A Fresh Look. In *CIDR*. www.cidrdb.org.

[49] Wentao Huang, Mo Sha, Mian Lu, Yuqiang Chen, Bingsheng He, and Kian-Lee Tan. 2024. Bandwidth Expansion via CXL: A Pathway to Accelerating In-Memory Analytical Processing. In *VLDB Workshops*. VLDB.org.

[50] Zichun Huang and Shimin Chen. 2022. Density-optimized intersection-free mapping and matrix multiplication for join-project operations. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2244–2256.

[51] IMDb.com, Inc. 2024. IMDb Non-Commercial Datasets. https://developer.imdb.com/non-commercial-datasets/ Accessed: October 2026.

[52] Intel. 2024. Intel® oneAPI Math Kernel Library (oneMKL). https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html Accessed: October 2026.

[53] Myung-Hwan Jang, Yunyong Ko, Hyuck-Moo Gwon, Ikhyeon Jo, Yongjun Park, and Sang-Wook Kim. 2023. SAGE: a storage-based approach for scalable and efficient sparse generalized matrix-matrix multiplication. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. 923–933.

[54] Mahmoud Abo Khamis, Xiao Hu, and Dan Suciu. 2025. Fast Matrix Multiplication meets the Submodular Width. *Proc. ACM Manag. Data* 3, 2 (2025), 98:1–98:26.

[55] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proc. VLDB Endow.* 2, 2 (2009), 1378–1389.

[56] Junyoung Kim and Kenneth A. Ross. 2026. SPALM: A Sparsity-Pattern-Adaptive Library for Matrices. *Proc. ACM Manag. Data* 4, 3 (2026), 236:1–236:27.

[57] Denis Kotkov, Alan Medlar, Alexandr Maslov, Umesh Raj Satyal, Mats Neovius, and Dorota Glowacka. 2022. The Tag Genome Dataset for Books. In *Proceedings of the 2022 ACM SIGIR Conference on Human Information Interaction and Retrieval (CHIIR '22)*. 5. https://doi.org/10.1145/3498366.3505833

[58] Jérôme Kunegis and Julia Preusse. 2012. Fairness on the web: alternatives to the power law. In *WebSci*. ACM, 175–184.

[59] Georgiy Lebedev, Hamish Nicholson, Musa Ünal, Sanidhya Kashyap, and Anastasia Ailamaki. 2025. Demystifying CXL Memory Bandwidth Expansion for Analytical Workloads. In *VLDB Workshops*. VLDB.org.

[60] Suhyun Lee, Chaemin Lim, Jinwoo Choi, Heelim Choi, Chan Lee, Yongjun Park, Kwanghyun Park, Hanjun Kim, and Youngsok Kim. 2024. SPID-Join: A Skew-resistant Processing-in-DIMM Join Algorithm Exploiting the Bank- and Rank-level Parallelisms of DIMMs. *Proc. ACM Manag. Data* 2, 6 (2024), 251:1–251:27.

[61] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.

[62] Zhonggen Li, Xiangyu Ke, Yifan Zhu, Yunjun Gao, and Yaofeng Tu. 2025. HC-SpMM: Accelerating Sparse Matrix-Matrix Multiplication for Graphs with Hybrid GPU Cores. In *ICDE*. IEEE, 501–514.

[63] Chunxu Lin, Wensheng Luo, Yixiang Fang, Chenhao Ma, Xilin Liu, and Yuchi Ma. 2024. On Efficient Large Sparse Matrix Chain Multiplication. *Proc. ACM Manag. Data* 2, 3 (2024), 156.

[64] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. 2025. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *ASPLOS (2)*. ACM, 1203–1217.

[65] Jinshu Liu, Hanchen Xu, Daniel S. Berger, Marcos K. Aguilera, and Huaicheng Li. 2026. Performance Predictability in Heterogeneous Memory. In *ASPLOS (2)*. ACM, 1413–1429.

[66] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, and Christopher M. Jermaine. 2017. Scalable Linear Algebra on a Relational Database System. In *ICDE*. IEEE Computer Society, 523–534.

[67] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-Efficient Aggregation: Hashing Is Sorting. In *SIGMOD Conference*. ACM, 1123–1136.

[68] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydin Buluç. 2018. High-Performance Sparse Matrix-Matrix Products on Intel KNL and Multicore Architectures. In *ICPP Workshops*. ACM, 34:1–34:10.

[69] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydin Buluç. 2019. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Comput.* 90 (2019).

[70] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: a tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *PPoPP*. ACM, 90–106.

[71] Taehyeong Park, Seokwon Kang, Myung-Hwan Jang, Sang-Wook Kim, and Yongjun Park. 2023. Orchestrating Large-Scale SpGEMMs using Dynamic Block Distribution and Data Transfer Minimization on Heterogeneous Systems. In *ICDE*. IEEE, 2456–2459.

[72] Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G. Pudov, Vadim O. Pirogov, and Pradeep Dubey. 2015. Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms. In *ISC (Lecture Notes in Computer Science)*, Vol. 9137. Springer, 48–57.

[73] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[74] Moinuddin K. Qureshi. 2014. Memory Scaling is Dead, Long Live Memory Scaling. https://hps.ece.utexas.edu/yale75/qureshi_slides.pdf Accessed: October 2026.

[75] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proceedings of the VLDB Endowment* 9, 3 (2015), 96–107.

[76] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[77] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQL-Storm: Taking Database Benchmarking into the LLM Era. *Proc. VLDB Endow.* 18, 11 (2025), 4144–4157.

[78] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD Conference*. ACM, 1961–1976.

[79] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.* 17, 12 (2024), 3731–3744.

[80] Malte Skarupke. 2017. I Wrote The Fastest Hashtable. Probably Dance. https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/ Accessed: October 2026.

[81] SuiteSparse Matrix Collection. 2018. VLSI/vas_stokes_1M. https://sparse.tamu.edu/VLSI/vas_stokes_1M Accessed: October 2026.

[82] Ruixuan Sun, Ruoyan Kong, Qiao Jin, and Joseph A. Konstan. 2023. Less Can Be More: Exploring Population Rating Dispositions with Partitioned Models in Recommender Systems. In *UMAP (Adjunct Publication)*. ACM, 291–295.

[83] Wenbo Sun, Asterios Katsifodimos, and Rihan Hai. 2023. Accelerating Machine Learning Queries with Linear Algebra Query Processing. In *SSDBM*. ACM, 13:1–13:12.

[84] Wenbo Sun, Asterios Katsifodimos, and Rihan Hai. 2023. An Empirical Performance Comparison between Matrix Multiplication Join and Hash Join on GPUs. In *ICDEW*. IEEE, 184–190.

[85] Wenbo Sun, Asterios Katsifodimos, and Rihan Hai. 2025. Accelerating machine learning queries with linear algebra query processing. *Distributed Parallel Databases* 43, 1 (2025), 8.

[86] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S. Yu, and Tianyi Wu. 2011. PathSim: Meta Path-Based Top-K Similarity Search in Heterogeneous Information Networks. *Proc. VLDB Endow.* 4, 11 (2011), 992–1003.

[87] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *MICRO*. ACM, 105–121.

[88] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High performance graph analytics made productive. *Proc. VLDB Endow.* 8, 11 (2015), 1214–1225.

[89] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2017. DBLP-Citation-Network V10. AMiner Citation Network Dataset, release 10. https://lfs.aminer.cn/lab-datasets/citation/dblp.v10.zip Landing page: https://www.aminer.org/citation. 1.85 GB archive, 4 JSON shards, 3,079,007 papers. Accessed: October 2026.

[90] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *EuroSys*. ACM, 818–833.

[91] Micron Technology. 2023. Memory Scaling is Dead, Long Live Memory Scaling. https://sg.micron.com/content/dam/micron/global/public/products/white-paper/cxl-impact-dram-bit-growth-white-paper.pdf Accessed: October 2026.

[92] James D. Trotter, Sinan Ekmekçibasi, Johannes Langguth, Tugba Torun, Emre Düzakin, Aleksandar Ilic, and Didem Unat. 2023. Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs. In *SC*. ACM, 31:1–31:13.

[93] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, et al. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17, 3 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[94] Aleksandar Vitorovic, Mohammed Elseidy, and Christoph Koch. 2016. Load balancing and skew resilience for parallel joins. In *ICDE*. IEEE Computer Society, 313–324.

[95] Jeffrey Scott Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (1985), 37–57.

[96] TaiNing Wang and Chee-Yong Chan. 2020. Improved Correlated Sampling for Join Size Estimation. In *ICDE*. IEEE, 325–336.

[97] Marcel Weisgut, Daniel Ritter, Florian Schmeller, Pınar Tözün, and Tilmann Rabl. 2025. CXL-Bench: Benchmarking Shared CXL Memory Access. In *VLDB Workshops*. VLDB.org.

[98] Samuel Williams. 2008. Williams/webbase-1M. https://sparse.tamu.edu/Williams/webbase-1M Accessed: October 2026.

[99] Guoqing Xiao, Chuanghui Yin, Tao Zhou, Xueqi Li, Yuedan Chen, and Kenli Li. 2024. A Survey of Accelerating Parallel Sparse Linear Algebra. *ACM Comput. Surv.* 56, 1 (2024), 21:1–21:38.

[100] Dezhong Yao, Sifan Zhao, Tongtong Liu, Gang Wu, and Hai Jin. 2024. Ap-SpGEMM: Accelerating Large-scale SpGEMM with Heterogeneous Collaboration and Adaptive Panel. *ACM Transactions on Architecture and Code Optimization* (2024).

[101] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, Jonathan Berry, and Ümit V Çatalyürek. 2018. Fast triangle counting using cilk. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[102] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2020. Speeding up SpMV for power-law graph analytics by enhancing locality & vectorization. In *SC*. IEEE/ACM, 86.