MAGNUS: Generating Data Locality to Accelerate Sparse Matrix-Matrix Multiplication on CPUs

Jordi Wolfson-Pou
Intel Labs
Santa Clara, USA
jordi.wolfson-pou@intel.com

Jan Laukemann
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
jan.laukemann@fau.de

Fabrizio Petrini
Intel Labs
Santa Clara, USA
fabrizio.petrini@intel.com

Abstract

Sparse general matrix-matrix multiplication (SpGEMM) is a critical operation in many applications. Current multithreaded implementations are based on Gustavson's algorithm and often perform poorly on large matrices due to limited cache reuse by the accumulators. We present MAGNUS (Matrix Algebra for Gigantic NUmerical Systems), a novel algorithm to maximize data locality in SpGEMM. To generate locality, MAGNUS reorders the intermediate product into discrete cache-friendly chunks using a two-level hierarchical approach. The accumulator is applied to each chunk, where the chunk size is chosen such that the accumulator is cache-efficient. MAGNUS is input- and system-aware: based on the matrix characteristics and target system specifications, the optimal number of chunks is computed by minimizing the storage cost of the necessary data structures. MAGNUS allows for a hybrid accumulation strategy in which each chunk uses a different accumulator based on an input threshold. We consider two accumulators: an AVX-512 vectorized bitonic sorting algorithm and classical dense accumulation. An OpenMP implementation of MAGNUS is compared with several baselines, including Intel MKL, for a variety of different matrices on three Intel architectures. For matrices from the SuiteSparse collection, MAGNUS is faster than all the baselines in most cases and is often an order of magnitude faster than at least one baseline. For massive random matrices, MAGNUS scales to the largest matrix sizes, while the baselines do not. Furthermore, MAGNUS is close to the optimal bound for these matrices, regardless of the matrix size, structure, and density.

CCS Concepts

• Mathematics of computing → Mathematical software performance; **• Computing methodologies → Shared memory algorithms**; **Linear algebra algorithms**.

Keywords

SpGEMM, Sparse matrices, Multicore CPUs

ACM Reference Format:
Jordi Wolfson-Pou, Jan Laukemann, and Fabrizio Petrini. 2025. MAGNUS: Generating Data Locality to Accelerate Sparse Matrix-Matrix Multiplication on CPUs. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3725773

1 Introduction

The sparse general matrix-matrix multiplication operation (SpGEMM) $C = AB$ is critical to the performance of many applications, including genome assembly [24, 25, 46], machine learning [5, 18, 27, 40, 42], algebraic multigrid [17, 31], and graph analytics [4, 20, 21, 28, 30, 45]. The main challenge of SpGEMM comes from the sparse structures of A and B, leading to unpredictable memory access patterns. Such irregularities pose significant difficulties for modern multicore architectures optimized for regular access patterns and high data reuse. Consequently, many state-of-the-art SpGEMM algorithms struggle to scale effectively for massive, irregular matrices, mainly due to inefficient utilization of the cache hierarchy.

Multithreaded SpGEMM algorithms are typically based on Gustavson's method, where, for each row of A, rows of B are loaded, scaled, *accumulated*, and written to C. The performance of the accumulation step is critical to the overall performance of SpGEMM. Conventional accumulators perform well for certain sparsity patterns, such as banded matrices, where the entire accumulator data structure is accessed infrequently. However, for highly irregular matrices, such as random power-law matrices, frequent accesses to the entire accumulator lead to suboptimal data reuse. As a result, scaling to massive matrices can become prohibitive, especially when the size of the accumulator exceeds the highest level of private cache. Several

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725773

algorithms have been proposed to address caching issues in accumulators, the most recent being the CSeg method [1, 39], where B is partitioned into segments so that only a smaller range of the accumulator is accessed. However, CSeg scales poorly for some datasets; when there are many partitions, the cost of constructing and accessing the partitioning information becomes significant, especially as the number of partitions increases with the matrix dimensions.

We present MAGNUS (**M**atrix **A**lgebra for **G**igantic **NU**merical **S**ystems), a novel algorithm that uses a hierarchical approach to generate the locality needed by the accumulators. The central idea of MAGNUS is to reorder the *intermediate product* of C (arrays of column indices and values generated before accumulation) into cache-friendly chunks that are processed independently by the accumulator. The MAGNUS workflow consists of two main algorithms: the fine- and coarse-level algorithms, the naming of which comes from two-level multigrid methods [17]. The coarse-level algorithm is based on the outer product formulation of SpGEMM and generates the first level of locality. The fine-level algorithm is based on Gustavson's formulation and further reorders the coarse-level chunks. The accumulator is then applied to each fine-level chunk. MAGNUS is input- and system-aware: the number of fine- and coarse-level chunks are chosen based on the matrix parameters and system specifications, where the optimal number of chunks is selected by minimizing the storage requirement of all frequently accessed data structures. Additionally, MAGNUS is accumulator agnostic, where conventional accumulators can be applied to the fine-level chunks. This paper considers two accumulators: AVX-512 vectorized sorting, which is used on chunks with a small number of elements, and dense accumulation, which is used otherwise.

Our experimental results provide two significant contributions: a set of microbenchmarks to motivate the need for MAGNUS, and the comparison of an OpenMP implementation of MAGNUS with six state-of-the-art SpGEMM baselines. For the microbenchmarks, the key building blocks of MAGNUS are tested in isolation and analyzed using Likwid [22]. First, we show that the performance of the accumulators drops significantly if the accumulation data structures do not fit into the L2 cache. Second, we show that with the optimal MAGNUS parameters, the execution time for the building blocks is minimized and performs at near-streaming speed.

For the SpGEMM results, MAGNUS is compared to six state-of-the-art baselines, including CSeg [1] and Intel Math Kernel Library (MKL) [9]. Three matrix test sets are evaluated on three Intel architectures. For the SuiteSparse matrix collection [13], MAGNUS is the fastest method in most cases and is often an order of magnitude faster than at least one baseline. For our second matrix set, which comes from a recursive model to generate power law graphs [7], MAGNUS is always the fastest. With the exception of CSeg, the speedup of MAGNUS over the baselines increases as the matrix size increases. Lastly, we consider massive uniform random matrices [16], which is the most challenging case for our baselines since the uniformity results in frequent accesses to the entire accumulation data structures. This test set demonstrates the need for the two-level approach in MAGNUS, where using the fine-level algorithm alone results in divergence from an ideal performance bound (CSeg also exhibits this poor scaling). However, using the complete MAGNUS algorithm allows scaling to the largest case, where the performance of MAGNUS is close to an ideal bound independent of the matrix size.

2 Background

2.1 Gustavson's Method

For a sparse matrix X, we define n_X, m_X, and nnz_X as the number of rows, columns, and nonzero entries, respectively. The set $\mathcal{S}(X)$ denotes the column indices of all nonzero entries in X. Our notation also extends to individual rows. For example, for row i of X, nnz_{X_i} denotes the number of nonzero entries, and $\mathcal{S}(X_i)$ denotes the set of column indices corresponding to the nonzero entries.

The general sparse matrix-matrix multiplication operation (SpGEMM) is defined as $C = AB$, where A, B, and C are sparse matrices. Multithreaded implementations of SpGEMM are typically based on Gustavson's row-by-row algorithm [26]:

$$C_i = \sum_{j \in \mathcal{S}(A_i)} A_{ij} B_j, \tag{1}$$

i.e., for some row i of C, the rows of B are scaled by the nonzero values in row i of A. These scaled rows are then summed together to give the final row of C. Since each row of C is computed independently, multithreaded implementations typically partition rows among threads, which is the approach we take for MAGNUS.

There are two main ingredients for implementing Gustavson's method: the matrix storage scheme and the algorithm that *accumulates* the scaled rows of B. Compressed sparse row (CSR) format is one of the most popular storage schemes and is especially useful for algorithms such as Gustavson's method that traverse matrices row-wise. The CSR format requires three arrays: $C.col$, $C.val$, and $C.rowPtr$. Arrays $C.col$ and $C.val$ of size nnz_C store the column indices and values, respectively, of the nonzero entries of C, and $C.rowPtr$, of size n_C+1, stores the starting positions of each row in $C.col$ and $C.val$.

Algorithm 1 shows the pseudocode for the *numeric phase* of Gustavson's method. Variables in **bold** are global, meaning they are shared and visible across all threads. The scaled rows of B are summed using a *dense accumulator*, defined as the combination of `denseAccumBuff` and `bitMap`. The column indices in row i of A are loaded as $idx = A.col[A.rowPtr[i]]$, and the rows of B are loaded by reading $B.col$ from $B.rowPtr[idx]$

to $B.rowPtr[idx+1]$. Array $denseAccumBuff$ of size m_C is updated for each column index of B, a companion bitmap stores the nonzero positions in $denseAccumBuff$, and $colBuff$ stores the running list of column indices in C. Besides A, B, and C, all variables are thread-local.

Algorithm 1 can be extended to other types of accumulators, e.g., hash map-based accumulators, where the dense accumulation array and bitmap are replaced by a hash map. In other Gustavson-based algorithms, such as expand-sort-compress (ESC) [11, 23], the *intermediate product* of C is written to an array instead of directly updating the accumulator. This is shown in Algorithm 2, where the intermediate product is generated in the first loop by storing $B.col[k]$ and $A.val[j] \times B.val[k]$ in $colBuff$ and $valBuff$, respectively. The intermediate product is then sorted, duplicates are merged, and the result is written to C (these steps take place in `sortMergeWrite`($colBuff$,$valBuff$)).

Algorithm 1: Gustavson SpGEMM: Dense Accumulation

```
Input: $A$, $B$, $C.rowPtr$
Output: $C.col$, $C.val$
for $i \leftarrow 0$ to $n-1$ do in parallel
    $count \leftarrow 0$, $denseAccumBuff \leftarrow 0$
    /* Read row $i$ of $A$ */
    for $j \leftarrow A.rowPtr[i]$ to $A.rowPtr[i+1]-1$ do
        /* Read row $j$ of $B$ */
        for $k \leftarrow B.rowPtr[A.col[j]]$
          to $B.rowPtr[A.col[j]+1]-1$ do
            /* Multiply and update accumulator */
            $denseAccumBuff[B.col[k]]$
              += $A.val[j] \times B.val[k]$
            if $bitMap[B.col[k]] == 0$ then
                $colBuff[count{+}{+}] \leftarrow B.col[k]$
                $bitMap[B.col[k]] \leftarrow 1$
            end
        end
    end
    /* Write to $C$ */
    $k \leftarrow C.rowPtr[i]$
    for $j \in colBuff$ do
        $C.col[k] \leftarrow j$
        $C.val[k{+}{+}] \leftarrow denseAccumBuff[j]$
        $bitMap[j] \leftarrow 0$
    end
end
```

To compute $C.rowPtr$, which is an input to the numeric phase, an initial *symbolic phase* is required. The symbolic phase typically has the same high-level algorithm as the numeric phase but without performing the multiplication $A.val[j] \times B.val[k]$ and writing to C. For example, in Algorithm 1, the symbolic phase does not include the modifications to $denseAccumBuff$, $C.col$, and $C.val$. Instead, only the bitmap is updated along with a counter that outputs the exact number of nonzero entries for each row of C. Finally, $C.rowPtr$ is computed using a prefix sum on the counters. This type of symbolic phase is known as *precise prediction*, where the number of nonzero entries in C is calculated exactly before the numeric phase.

Algorithm 2: Gustavson SpGEMM: Expand-Sort-Compress (ESC)

```
Input: $A$, $B$, $C.rowPtr$
Output: $C.col$, $C.val$
for $i \leftarrow 0$ to $n-1$ do in parallel
    $count \leftarrow 0$
    /* Read row $i$ of $A$ */
    for $j \leftarrow A.rowPtr[i]$ to $A.rowPtr[i+1]-1$ do
        /* Read row $j$ of $B$ */
        for $k \leftarrow B.rowPtr[A.col[j]]$
          to $B.rowPtr[A.col[j]+1]-1$ do
            /* Multiply and update accumulator */
            $colBuff[count] \leftarrow B.col[k]$
            $valBuff[count{+}{+}] \leftarrow A.val[j] \times B.val[k]$
        end
    end
    /* Sort, merge, and write to $C$ */
    sortMergeWrite($colBuff$,$valBuff$)
end
```

On modern CPUs, maximizing cache reuse is crucial to the performance of any application. In SpGEMM, the accumulator is the most frequently accessed, where the amount of reuse is determined by the sparsity pattern of A and B. For optimal performance, the dense accumulator should be confined to the L2 cache, which is the highest level of private cache. This efficient cache utilization occurs naturally in specific matrix structures, such as banded matrices or matrices that yield a highly sparse C. However, for matrices that produce "large" intermediate products (where "large" refers to both the number of nonzero elements and a wide distribution of column index values), SpGEMM faces significant challenges. A prominent example is random power-law matrices that model social networks [7]. For such matrices, the size of $denseAccumBuff$ often exceeds the capacity of the L2 cache and the large intermediate product results in frequent accesses to the entire $denseAccumBuff$ array. Consequently, $denseAccumBuff$ must frequently be evicted from and reloaded to the L2 cache, resulting in suboptimal performance. This breakdown in locality presents a substantial obstacle for current SpGEMM algorithms, as we will demonstrate through microbenchmarks and SpGEMM results.

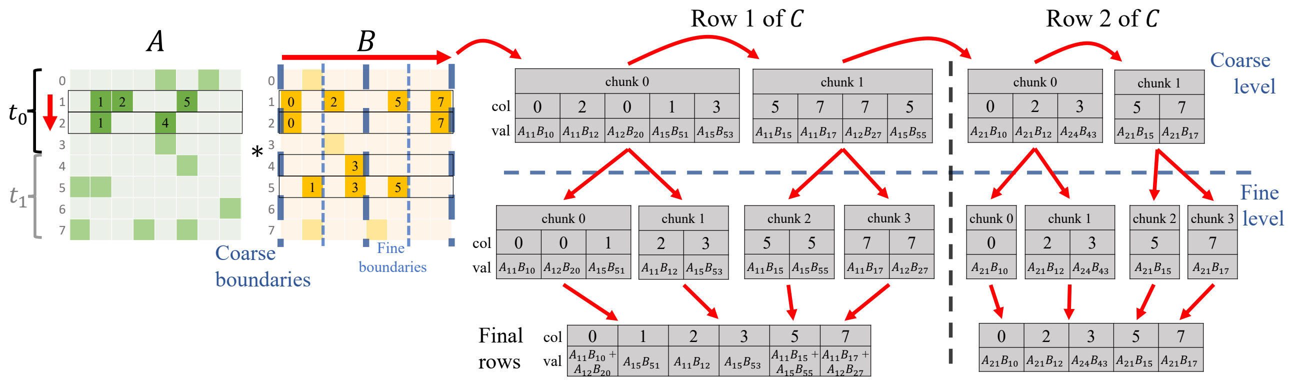

Figure 1: Example workflow of MAGNUS, where two threads multiply two 8×8 matrices. The computation of rows 1 and 2 assigned to thread t_0 is shown. Two coarse- and fine-level chunks are used for each row.

2.2 Related Work

Optimizations related to the accumulation step have been the primary focus of research on SpGEMM. For load balancing, a common approach is based on the observation that rows with different intermediate sizes require different accumulation strategies [8, 11, 15, 23, 33, 34, 37, 38]. In [11], A is reordered by increasing intermediate size. Other approaches group rows based on the size of the intermediate product, where a different accumulator is used for different groups [8, 15, 33, 34, 37, 38, 44]. In [33], five different accumulation strategies are used, including priority queues and sorting algorithms. Developing new accumulators is another topic that has been widely studied, especially for modern multicore machines with vector processors [6, 29, 35, 36, 43, 47, 48]. A common approach is to optimize sorting algorithms [6, 11, 23, 30, 32, 34] or data structures such as heaps [3, 35, 36] and hash maps [2, 10, 14, 32, 35, 36, 42]. In Section 4.3, MAGNUS is compared with hash map and heap-based approaches, which are considered state-of-the-art [1, 9, 35, 36].

Perhaps most relevant to MAGNUS are recent works on improving the cache behavior of accumulators, proposed in [39] and improved in the CSeg method [1]. The core concept is to partition the columns of B into segments, where the number of segments is chosen so that the dense accumulator fits in the L2 cache. An additional *high-level summary matrix* is used to store the segmentation information. CSeg was shown to be overall faster than many state-of-the-art libraries mentioned above. For a more extensive overview of SpGEMM research, including distributed memory algorithms, see [19].

3 MAGNUS

3.1 Overview

MAGNUS uses a novel hierarchical algorithm to generate two levels of locality. The *coarse-level algorithm* reorders the intermediate product into discrete chunks and the *fine-level algorithm* further subdivides and accumulates the coarse-level chunks. The number of chunks used in both algorithms is based on optimal parameters that are computed using the input matrix properties and the target system specifications. These parameters are optimal in the sense that they minimize the storage cost of all frequently accessed arrays. The levels are generated using a set of basic operations, including histogramming and prefix summing. Combining the building blocks and the optimal parameters creates the locality required by the accumulators.

For sufficiently "small" matrices (as discussed later in the derivation of the MAGNUS parameters), the fine-level algorithm alone provides an adequate level of data locality. This algorithm is based on Gustavson's method, similar to most SpGEMM algorithms. However, achieving scalability for "massive" matrices requires both fine- and coarse-level locality. Here, massive matrices are those in which the data structures required by the fine-level algorithm, including the accumulator, exceed the capacity of the L2 cache. The coarse-level algorithm employs an outer product-based approach, necessitating an additional pass over the intermediate product, which increases the total data volume. This additional cost means that using the standalone fine-level algorithm wherever possible is advantageous, which is why we reserve the coarse-level algorithm only for massive matrices. Since the intermediate product is generated in both approaches, MAGNUS can be classified as an ESC-type algorithm [11].

Figure 1 shows a simple workflow for MAGNUS, where two threads are used to multiply two 8×8 matrices. The example shows how rows 1 and 2 assigned to thread t_0 are computed, where two chunks are used for both the coarse and fine levels. The outer product-based approach traverses the submatrix of A (corresponding to rows 1 and 2) column by column, and the highlighted rows of B are traversed row by row. This traversal generates the four coarse-level chunks. Each chunk is reordered again to get the eight fine-level chunks, where the accumulator is applied to get the final result. It is important to note that all coarse-level chunks are generated before executing the fine-level algorithm. However, the fine level is processed depth-first: for each coarse-level chunk, the fine level is generated and then immediately accumulated (similar to a Gustavson-based method) before proceeding to the next coarse-level chunk.

The key property of MAGNUS is that the range of column indices in each fine-level chunk is significantly smaller than m_C (the number of columns of C). This allows the dense accumulator to fit in the L2 cache when m_C exceeds the L2 cache capacity. In Figure 1, the per-chunk range of column indices is two, compared to $m_C = 8$. For example, the column indices in the fourth fine-level chunk of either row fall within the range $[6,7]$. Although not shown in the figure, the column indices in each chunk are shifted into the chunk-local range $[0,1]$, which means that we only need a dense accumulator of length two. If we consider a theoretical system with an L2 cache capable of storing a dense accumulator with a maximum of two elements, each fine-level chunk can be accumulated with minimal L2 cache misses. In contrast, if we used Gustavson's method with a conventional dense accumulator, L2 cache misses would occur frequently after loading the first two elements in the second row of B.

The high-level steps of MAGNUS are: (1) the setup phase, which includes calculating the optimal number of chunks (this will be discussed later in Section 3.5), computing the intermediate product size for each row, and categorizing each row; (2) the symbolic phase; and (3) the numeric phase. The setup phase is inexpensive compared to the symbolic and numeric phases: calculating the optimal number of chunks has constant time complexity, while the remaining setup steps involve a highly parallel single pass over the rows of C, with time complexity $O(n_C/t)$, where t is the number of threads. Row categorization is necessary because not all rows require locality generation.

MAGNUS categorizes each row based on its structure and the system's specifications:

1. **Sort**: If the number of intermediate elements is less than the dense accumulation threshold, we can directly apply a sort-based accumulator, as in Algorithm 2. The dense accumulation threshold will be described later in this section.
2. **Dense accumulation**: If the *intermediate row length* fits into the L2 cache, we can directly apply dense accumulation to the row, as in Algorithm 2. This is because the range of column indices does not exceed the size of the L2 cache. The intermediate row length is the difference between the minimum and maximum column index of the intermediate product.
3. **Fine level:** If $s_{finelevel} < s_{L2}$, the fine-level algorithm can be applied, where $s_{finelevel}$ is the number of bytes required to store all necessary fine-level data structures (this is discussed later in this section).
4. **Coarse level:** The coarse-level algorithm is applied to all remaining rows, where the fine-level algorithm is applied to each coarse-level chunk.

The accumulation parameters mentioned above will be discussed in Section 3.4, including a description of the sorting algorithm. The first two categories imply that some rows possess intrinsic locality and do not benefit from the locality-generation algorithms in MAGNUS. After the rows are categorized, the rows of C are computed category-first to ensure that data specific to a particular category are cached for as long as possible. In our OpenMP implementation of MAGNUS, we use a parallel for loop with dynamic scheduling to traverse the rows in each category with a `no wait` clause. The `no wait` clause ensures that threads proceed to the next category without unnecessary synchronization.

For simplicity, we assume m_C is a power of two in our descriptions of the algorithms in MAGNUS. We use precise prediction for the symbolic phase, but for brevity, we will only describe the numeric phase. See Section 2 for clarity on the differences between the symbolic and numeric phases.

3.2 The Fine-level Algorithm

The fine-level algorithm has the following steps for each row: histogram, prefix sum, reorder, and accumulate, where $n_{chunksFine}$ is the number of fine-level chunks. As in Gustavson's method, each row (or coarse-level chunk in cases where the coarse-level algorithm is applied) is computed before moving on to the next row. This means that the intermediate product is generated only for a single row (or coarse-level chunk) at any given time, unlike in outer-product-based approaches.

Pseudocode for the fine-level algorithm is shown in Algorithm 3, where the input is the column indices and values of a single coarse-level chunk for row i of C. For rows that only require fine-level locality, A and B are read directly as in Algorithm 1, i.e., the loop headers on lines 3 and 12 of Algorithm 3 are replaced with the nested loop headers on lines 4 and 6 of Algorithm 1, respectively. The notation $arr \leftarrow \alpha$ means that the entire array arr is initialized to the value α. The C-style notation $arr[i]$ means that element i of arr is accessed, and $\&arr[i]$ means that the array is accessed starting at element i.

The first step towards reordering the intermediate product is to compute $offsetsFine$ (the chunk offsets) using a histogram and a prefix sum operation. The array $offsetsFine$ stores the start and end locations of each fine-level chunk in $colFine$ and $valFine$. The buffers $colFine$ and $valFine$ are typical of any ESC-type algorithm where the intermediate product must be explicitly stored. In the case of MAGNUS, they store the reordered intermediate product. In the histogram step, the column indices are mapped to chunks as $col/chunkLenFine$, where $chunkLenFine = m_{C_{maxL2}}/n_{chunksFine}$ is the *chunk*

Algorithm 3: MAGNUS fine-level algorithm applied to a single coarse-level chunk

```
   Input: $colCoarse$, $valCoarse$
   Output: $C_{i,rangeCoarse}$
1  /* Histogram */
2  $countsFine \leftarrow 0$
3  for $col \in colCoarse$ do
4      $chunk \leftarrow col \gg shiftFine$
5      $countsFine[chunk]$++
6  end
7  /* Prefix sum */
8  $offsetsFine[0] \leftarrow 0$
9  $\&offsetsFine[1] \leftarrow$ inclusiveScan($countsFine$)
10 /* Reorder */
11 $countsFine \leftarrow 0$
12 for $\{col,val\} \in \{colCoarse,valCoarse\}$ do
13     $chunk \leftarrow col \gg shiftFine$
14     $\ell \leftarrow offsetsFine[chunk] + countsFine[chunk]$++
15     $colFine[\ell] \leftarrow col - chunk \times chunkLenFine$
16     $valFine[\ell] \leftarrow val$
17 end
18 /* Accumulation */
19 for $j \leftarrow 0$ to $n_{chunksFine} - 1$ do
20     $k \leftarrow offsetsFine[j]$
21     $C_{i,rangeFine_j} \leftarrow$ accum($\&colFine[k]$,$\&valFine[k]$)
22 end
```

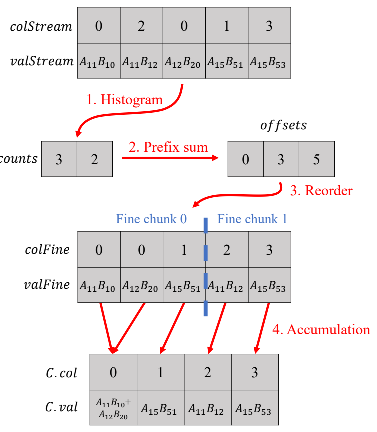
colStream
0 2 0 1 3
valStream
$A_{11}B_{10}$ $A_{11}B_{12}$ $A_{12}B_{20}$ $A_{15}B_{51}$ $A_{15}B_{53}$
1. Histogram
counts
3 2
2. Prefix sum
offsets
0 3 5
3. Reorder
Fine chunk 0
Fine chunk 1
colFine
0 0 1 2 3
valFine
$A_{11}B_{10}$ $A_{12}B_{20}$ $A_{15}B_{51}$ $A_{11}B_{12}$ $A_{15}B_{53}$
4. Accumulation
C.col
0 1 2 3
C.val
$A_{11}B_{10}+A_{12}B_{20}$ $A_{15}B_{51}$ $A_{11}B_{12}$ $A_{15}B_{53}$

Figure 2: Data structure-view of applying the fine-level algorithm to the first chunk from Figure 1.

length of the fine-level chunks. The chunk length is the local range of the column indices within a chunk, where column indices are shifted into the range $[0, chunkLenFine)$ as shown on line 15. The value of $m_{C_{maxL2}}$ is equal to m_C if the fine-level algorithm is used alone, or equal to the coarse chunk length if the coarse-level algorithm is used ($m_{C_{maxL2}}$ is discussed in more detail in Section 3.5). To optimize the mapping, the division operation is replaced with a bitwise operation by restricting $n_{chunksFine}$ to a power of two, ensuring that $m_C/n_{chunksFine}$ is also a power of two. Therefore, the mapping becomes $col \gg shiftFine$, where $shiftFine = \log_2(chunkLenFine)$ and $\gg$ is the bitwise right-shift operator.

After the histogram is computed, the chunk offsets are computed using a prefix sum (inclusive scan) of the histogram. To reorder the input (shown in the second loop), the histogramming phase is repeated, where the histogram is used to track the current number of elements in each chunk. Elements are reordered by writing the input coarse-level chunk at the position $offsetsFine[chunk] + countsFine[chunk]$ in $colFine$ and $valFine$. As mentioned previously, the column indices are shifted into the local range of each chunk as $col - chunk \times chunkLenFine$ to allow for cache-efficient accumulation.

Finally, a call to `accum()` for each chunk invokes either sort-based accumulation or dense accumulation, where the size of $denseAccumBuff$ (see Algorithm 1) is now reduced from $m_{C_{maxL2}}$ to $chunkLenFine$. After the accumulation step, the column indices are shifted back into the correct range before writing to C. The variable $rangeFine_j$ denotes the range of column indices of the fine-level chunk j (i.e., $[j \times chunkLenFine, (j+1) \times chunkLenFine)$), and $rangeCoarse$ is the range of column indices of the input coarse-level chunk. Figure 2 shows the workflow of the fine-level algorithm in terms of the data structures from Algorithm 3, where the input is the first chunk from the example in Figure 1.

The fine-level algorithm requires two additional arrays: $countsFine$ and $offsetsFine$, both of size $n_{chunksFine}$. Alongside $denseAccumBuff$ and $bitMap$, the goal is to keep $countsFine$, $offsetsFine$, and the active cache lines of $colFine$ and $valFine$ in the L2 cache. The active cache lines must be considered since we are writing to $colFine$ and $valFine$ at noncontiguous positions. In our implementations, we use non-temporal streaming stores when writing to $colFine$ and $valFine$, which avoids polluting the L2 cache. Non-temporal stores are intrinsic functions used on Intel processors (e.g., `_mm512_stream_si512()`) that write to memory without evicting cached data, allowing us to retain the accumulator and fine-level data structures in the L2 cache while streaming the intermediate product. Section 3.5 shows how we choose the optimal number of chunks that minimizes the total storage cost of these cached arrays.

3.3 The Coarse-level Algorithm

As the columns of C increase, the storage of the fine-level data structures eventually exceeds the size of the L2 cache. An initial coarse level must be generated for such matrices, providing the first level of locality. To generate the coarse level, we use a modified outer product-based algorithm, where the intermediate product is generated and reordered for all rows that require coarse-level locality before any accumulation occurs. The reordered intermediate product is organized into discrete coarse-level chunks that can be handled independently by the fine-level algorithm.

The outer product-based approach is used because the coarse-level algorithm computes only the intermediate product without performing any accumulation. Therefore, maximizing the reuse of the input matrices rather than the accumulator is beneficial, which is a well-known property of outer product-based SpGEMM algorithms [23, 48]. The combination of our reordering algorithm with the outer product formulation makes the coarse-level algorithm similar to propagation blocking-based algorithms [23].

Conventional outer product algorithms typically generate the intermediate product by multiplying the columns of A, stored in CSC format, with the rows of B, stored in CSR format. The coarse-level algorithm in MAGNUS follows the same scheme, but on the subset of the rows categorized as coarse-level rows. Therefore, a CSC version of the submatrix $\hat{A}$ of A is constructed, where $\hat{A}$ only includes these coarse-level rows. Each thread performs the following steps on its list of coarse-level rows, which are stored in the array *coarseRowsC*:

(1) For all $i \in coarseRowsC$, generate *coarseRowsB*, i.e., the unique set of rows in B required to perform the outer product. This is done by iterating through the nonzero entries in $\hat{A}$ and setting a bitmap, where *coarseRowsB* is the list of set bits.
(2) Construct the thread-local CSC submatrix $\hat{A}^{CSC}$ using the well-known approach for the conversion of CSR to CSC: a histogramming stage computes the number of nonzero entries per row, a prefix sum computes $\hat{A}^{CSC}.colPtr$, and $\hat{A}^{CSC}.colPtr$ is then used to construct the arrays $\hat{A}^{CSC}.row$ and $\hat{A}^{CSC}.val$. This step is inexpensive since the time complexity of outer product-based SpGEMM algorithms is dominated by the generation of the intermediate product in the following step. Specifically, the time complexity of constructing $\hat{A}$ is $O(nnz_{\hat{A}})$, while generating the intermediate product requires $O(\sum_{(i,j)\in\mathcal{S}(\hat{A})} nnz_{B_j})$ time, which is best-case $O(nnz_{\hat{A}})$ (i.e., each row in B contains only one nonzero value) and worst-case $O(nnz_{\hat{A}} m_B)$ (i.e., each row in B is dense).
(3) Perform a histogram step by reading $\hat{A}^{CSC}$ and the rows of B corresponding to *coarseRowsB*. A prefix sum of the resulting histogram generates the coarse-level offsets.
(4) Generate the coarse level by again reading $\hat{A}^{CSC}$, reading the rows of B corresponding to *coarseRowsB*, and writing the result in *colCoarse* and *valCoarse* at the positions determined by the offsets. As in the fine-level algorithm, we use non-temporal streaming stores for the reordering.
(5) For each coarse-level chunk, apply the fine-level algorithm.

Steps 3-5 are shown in Algorithm 4. The same set of basic building blocks is used as in the fine-level algorithm (histogramming, prefix summing, and reordering). The key difference is that the chunks for all rows are generated at the same time. The prefix sum is modified to compute the chunk offsets within a row based on the offsets from the previous rows. Just as in the fine-level algorithm, we generate the reordered intermediate product using non-temporal streaming stores. In the final loop, the fine-level algorithm is applied to each coarse-level chunk for each row. Figure 3 shows the data structures for the example in Figure 1.

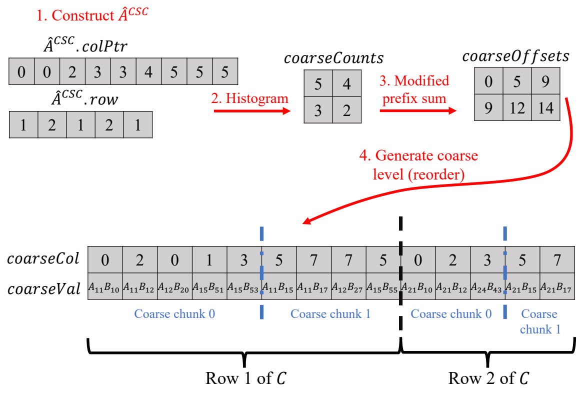

Figure 3: Data structure-view of the coarse-level algorithm for the example in Figure 1.

The coarse-level algorithm utilizes a set of data structures similar to that of the fine-level algorithm, but without an accumulator. The absence of a cached accumulator enables the coarse-level algorithm to generate more chunks than the fine-level algorithm. However, this comes at the cost of increased data volume due to the additional pass over the intermediate product. Despite this tradeoff, for a sufficiently large matrix, the increased data volume proves more efficient than the frequent cache misses incurred by the fine-level algorithm, as we will demonstrate in our microbenchmark results.

Similarly to other outer product-based approaches, the memory requirement for the coarse-level algorithm is higher than the fine-level algorithm since the intermediate product for multiple rows must be stored. In the worst case, the memory requirement is proportional to the sum of the outer products of all rows, which may exceed the memory capacity of certain memory-constrained systems. Therefore, we use a batching approach, where we collect rows of C into *coarseRowsC* until one of two conditions is met: (1) the memory limit of our system is reached, or (2) the storage requirement for generating the coarse-level chunks (*countsCoarse*, *offsetsCoarse*, and the small buffers for the non-temporal streaming stores) exceeds the L2 cache size. If either condition is met, the batch of rows in *coarseRowsC* is

Algorithm 4: MAGNUS coarse-level algorithm

```
Input: $\hat{A}^{CSC}$, $B$, $coarseRowsB$, $coarseRowsC$
Output: $C_{coarseRowsC}$
$countsCoarse \leftarrow 0$
/* Histogram */
for $i \in coarseRowsB$ do
    for $j \leftarrow \hat{A}^{CSC}.colPtr[i]$ to $\hat{A}^{CSC}.colPtr[i+1]-1$ do
        for $k \leftarrow \mathbf{B.rowPtr}[i]$ to $\mathbf{B.rowPtr}[i+1]-1$ do
            $chunk \leftarrow \mathbf{B.col}[k] \gg shiftCoarse$
            $countsCoarse[\hat{A}^{CSC}.row[j]][chunk]$++
        end
    end
end
/* Prefix sum */
$offsetsCoarse[0][0] \leftarrow 0$
for $i \in coarseRowsC$ do
    $\&offsetsCoarse[i][1] \leftarrow$ inclusiveScan($countsCoarse[i]$)
    $offsetsCoarse[i+1][0] \leftarrow offsetsCoarse[i][n_{chunksCoarse}]$
end
/* Reorder */
$countsCoarse \leftarrow 0$
for $i \in coarseRowsB$ do
    for $j \leftarrow \hat{A}^{CSC}.colPtr[i]$ to $\hat{A}^{CSC}.colPtr[i+1]-1$ do
        for $k \leftarrow \mathbf{B.rowPtr}[i]$ to $\mathbf{B.rowPtr}[i+1]-1$ do
            $chunk \leftarrow \mathbf{B.col}[k] \gg shiftCoarse$
            $\ell \leftarrow offsetsCoarse[\hat{A}^{CSC}.row[j]][chunk] + countsCoarse[\hat{A}^{CSC}.row[j]][chunk]$++
            $colChunks[\ell] \leftarrow \mathbf{B.col}[k] - chunk \times chunkLenCoarse$
            $valChunks[\ell] \leftarrow \hat{A}^{CSC}.val[j] \times \mathbf{B.val}[k]$
        end
    end
end
/* Apply fine-level algorithm to each coarse-level chunk */
for $i \in coarseRowsC$ do
    for $j \leftarrow 0$ to $n_{chunksCoarse}-1$ do
        $k \leftarrow offsetsCoarse[i][j]$
        $C_{i,rangeCoarse_j} \leftarrow$ fineLevel($\&colCoarse[k]$,$\&valCoarse[k]$)
    end
end
```

computed via steps 2-5. This batching process is repeated until all rows requiring coarse-level locality have been computed.

3.4 Accumulation

MAGNUS is accumulator-agnostic, allowing for portability and flexibility: only the storage requirements of the desired accumulators are needed to compute the optimal MAGNUS parameters (the optimal parameters are derived in Section 3.5. For portability, accumulators optimized for specific architectures can be selected without changing the locality-generation algorithms. This is evident in Algorithm 3, where the `accum()` function requires only an input chunk. For flexibility, MAGNUS allows for a hybrid approach in which each chunk chooses an accumulator based on the chunk characteristics. In this paper, we consider two accumulators: AVX-512 vectorized bitonic sorting [6] and classical dense accumulation. For chunks with a small number of elements, sorting is performed. Otherwise, dense accumulation is used. When visiting each chunk, a threshold is used to choose the accumulator. This threshold is based on how the sorting algorithm is implemented: quicksort partitions the array, and then hard-coded, vectorized bitonic sorting networks sort the partitions. We found experimentally that dense accumulation is faster than sorting unless the sort size is small enough to bypass the quicksort algorithm and directly use bitonic sorting. For more details, see Section 4.1.1 and [6].

3.5 Choosing the Number of Chunks

The optimal number of chunks is chosen based on the following input parameters, which are readily available for end users: $s_{cacheLine}$, s_{L2}, and m_C. Parameters $s_{cacheLine}$ and s_{L2} are the cache line and L2 cache sizes, respectively, which are retrieved by querying the underlying system, e.g., by using standard Linux commands. Parameter m_C is the number of columns of C, which is already included in the CSR data structure of B since $m_C = m_B$.

As explained in Section 3.2, the goal is to retain in the L2 cache certain data structures needed by the fine-level algorithm. For simplicity, assume m_C is a power of two (m_C is ceiled to the nearest power of two otherwise). Choosing the optimal number of fine-level chunks corresponds to selecting the value of $n_{chunksFine}$ that minimizes the convex function

$$s_{fineLevel} = \frac{m_C s_{denseAccum}}{\boldsymbol{n_{chunksFine}}} + \boldsymbol{n_{chunksFine}} s_{chunkFine}, \qquad (2)$$

where $s_{fineLevel}$ is the storage requirement for the L2-cached fine-level data structures. The first term is the storage requirement of the dense accumulator, where the number of elements in the underlying dense accumulator array is $m_C / n_{chunksFine}$. For the numeric phase, $s_{denseAccum} = s_{val} + 1$, since we need $denseAccumBuff$ and $bitMap$ (see Algorithm 1). For the symbolic phase, $s_{denseAccum} = 1$ since we only need $bitMap$. The second term is the storage requirement for reordering, where the storage cost per fine-level chunk is $s_{chunkFine} = s_{histoType} + s_{prefixSumType} + 2s_{cacheLine}$. The parameters $s_{histoType}$ and $s_{prefixSumType}$ denote the size of the histogram and prefix sum array data types, respectively, which are both four bytes. The term $2s_{cacheLine}$ accounts for the storage of active cache lines when noncontiguously writing to $colFine$ and $valFine$ during the reordering phase.

The value of $n_{chunksFine}$ that minimizes Equation 2 is

$$\boldsymbol{n_{chunksFine}} = \sqrt{\frac{m_C s_{denseAccum}}{s_{chunkFine}}}, \tag{3}$$

rounded to the nearest power of two, which is the number of fine-level chunks used when the fine-level algorithm is used alone.

By plugging in $n_{chunksFine}$ from Equation 3 into Equation 2 we get

$$s_{fineLevel} = 2\sqrt{m_C s_{denseAccum} s_{chunkFine}}, \tag{4}$$

which is the total storage requirement of the fine-level algorithm when the optimal number of fine-level chunks is used. When m_C becomes sufficiently large, $s_{fineLevel}$ exceeds the size of the L2 cache, at which point the coarse-level algorithm is applied. We can now derive the number of fine- and coarse-level chunks when both levels of locality are used. In this case, we first determine $m_{C_{maxL2}}$, which is the maximum number of columns in which $s_{fineLevel} \leq s_{L2}$. To calculate $m_{C_{maxL2}}$, we replace m_C with $m_{C_{maxL2}}$ in Equation 4 and solve $s_{fineLevel} = s_{L2}$ for $m_{C_{maxL2}}$. This gives us

$$m_{C_{maxL2}} = \frac{s_{L2}^2}{4 s_{denseAccum} s_{chunkFine}}, \tag{5}$$

which is constant and is floored to the nearest power of two. Therefore,

$$\boldsymbol{n_{chunksFine}} = \sqrt{\frac{m_{C_{maxL2}} s_{denseAccum}}{s_{chunkFine}}}, \tag{6}$$

is the optimal number of fine-level chunks when both levels of locality are used, and the optimal number of coarse-level chunks is

$$\boldsymbol{n_{chunksCoarse}} = m_C / m_{C_{maxL2}}. \tag{7}$$

Equations 5 and 6 show that for a sufficiently large value of m_C, the number of fine-level chunks stops growing once the coarse-level algorithm is applied. At this point, the number of coarse-level chunks begins to grow while maintaining a fixed number of fine-level chunks, ensuring that the fine-level data structures fit into the L2 cache.

Note that we only derive optimal parameters for dense accumulation. This is because the sort-based accumulator does not require any additional storage since the arrays storing the intermediate product are directly sorted. The same analysis can also be applied to other accumulators (e.g., hash maps) by modifying Equation 2.

4 Experimental Results

4.1 Microbenchmarks

This section aims to establish our motivation by evaluating the core building blocks of MAGNUS in isolation through microbenchmarking. The input and output in our microbenchmarks are streams consisting of two arrays of the same size: one of unsigned integers (the *index array*) and the other of floating-point numbers (the *value array*), which emulate the column indices and values, respectively, in the intermediate product. There are two critical parameters to which SpGEMM accumulators are sensitive: the number of elements in the stream and the maximum value of the elements in the index array (the *stream length*). The index array is uniformly random in the range [0,stream length), which emulates the range of column indices in the matrix or a MAGNUS chunk.

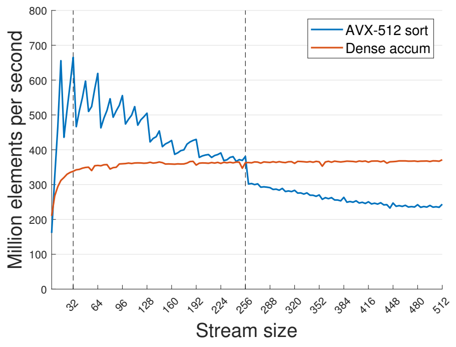

Figure 4: Comparison of accumulators used in MAGNUS: AVX-512 vectorized sorting and dense accumulation. A single core of the SPR system is used (see Table 1). The rate (millions of elements per second) versus the number of accumulated elements is shown. The dashed lines indicate where the sorting achieves peak performance (32 elements) and where the performance of dense accumulation overtakes that of sorting (256 elements).

First, we show that increasing the stream size and length degrades the performance of conventional accumulators. We then benchmark the building blocks of MAGNUS to demonstrate how the introduction of locality generation can improve the performance of the dense accumulator. For these experiments, four-byte types were used (`uint32_t` and `float`) on one core of the SPR system (see Table 1). We used Likwid [22] to collect performance metrics, providing insight into the cache behavior of the building blocks of MAGNUS. Likwid is a performance monitoring tool that reports detailed CPU hardware metrics, including the volume of L2-to-L3 cache evictions.

4.1.1 Accumulators. We first consider the two accumulators used by MAGNUS: AVX-512 vectorized bitonic sorting [6] and dense accumulation. Figure 4 shows the rate (in millions of elements per second) versus the stream size for a fixed stream length of 2^{18}, which is the stream length for which the dense accumulation arrays fit into the L2 cache. There are two important sizes to consider: 32 and 256. At a size of 32, the sorting achieves peak performance, processing nearly 700 million elements per second. Therefore, targeting sort sizes as close as possible to 32 is ideal. In MAGNUS, we

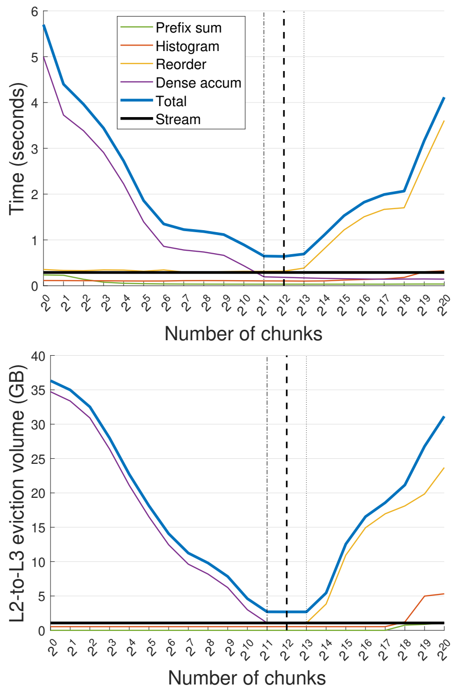

Figure 5: Wall-clock time (top) and L2-to-L3 cache evictions (bottom) versus the number of chunks for a set of microbenchmarks that test the performance of the building blocks of MAGNUS. A single core of the SPR system is used (see Table 1). *Total* denotes the sum of the building block times, and *Stream* is a standard streaming benchmark that serves as the peak-performance baseline. The middle vertical dashed line denotes the optimal number of fine-level chunks. The left and right dashed lines denote the point at which the storage requirement of the reorder and dense accumulation arrays exceed the L2 cache size, respectively.

do precisely this: we combine consecutive chunks until the difference between the sort size and 32 is minimized. Dense accumulation overtakes sorting at a stream size of 256 and is $\approx 1.5\times$ faster for a sort size of 512. The 256 threshold originates from the sorting algorithm, which partitions the array into 256 parts (for more information, see [6]). MAGNUS uses this threshold when selecting an accumulator within a chunk.

4.1.2 Building Blocks of MAGNUS. In this experiment, MAGNUS is deconstructed into its building blocks: histogramming, prefix summing, reordering, and accumulation. For the algorithmic details of each building block, see the associated comments in Algorithm 3 and Algorithm 4. For example, the algorithm for histogramming is on lines 3-6 in Algorithm 3 and lines 3-10 in Algorithm 4. Figure 5 shows time and the volume of L2-to-L3 cache evictions (measured using Likwid [22]) versus the number of chunks for a stream size and length of 2^{29} elements. This stream length results in the size of the dense accumulation array varying from 2^{29} to $2^{29}/2^{20} = 512$ as the number of chunks increases. This emulates the per-chunk dense accumulation in MAGNUS. The time for a standard streaming benchmark is shown, which consists of performing contiguous reads from the input arrays and contiguous writes to the output arrays. This serves as our peak performance baseline, where the total time (i.e., the sum of all the building block times) cannot exceed the streaming time. The left and right dashed lines are the points at which the reorder and dense accumulation data structures exceed the L2 cache size. The middle vertical dashed line shows the optimal number of fine-level chunks (derived in Section 3.5), which is calculated using the stream length in place of m_C in Equation 3.

The total time, which closely approximates the performance of the fine-level algorithm, is dominated by dense accumulation and reordering. The time for reordering increases significantly past 2^{13}, where the storage requirement (active cache lines, histogram array, and prefix sum array) exceeds the L2 cache size, as seen by the increase in L2-to-L3 cache evictions. For dense accumulation, performance improves as the number of chunks increases due to the reduced size of the dense accumulation array; this is a key result of our locality generation approach. The total execution time reaches a minimum at the optimal number of fine-level chunks, where both dense accumulation and reordering achieve optimal cache behavior. At this point, the total time is ≈ 2.2 times the streaming time. Although lower-level optimizations, such as vectorized histogramming, could further decrease this slowdown, maintaining this reasonably small multiple of the peak performance is crucial for scaling to massive matrices. We will show in Section 4.3 that MAGNUS can maintain a similar multiple of the peak performance for massive random matrices, while other SpGEMM baselines cannot.

Figure 6 illustrates the impact of varying the stream length, where the optimal number of fine-level chunks is chosen for each value of the stream length. Past a stream length of 2^{31}, the total time rises sharply due to the high volume of L2-to-L3 cache evictions. This behavior highlights the necessity of the coarse-level algorithm in MAGNUS: even when the optimal number of fine-level chunks is used, the total storage cost of all data structures can exceed the L2 cache capacity when the number of columns of C is sufficiently large. In MAGNUS, the coarse-level algorithm automatically activates after the breaking point at 2^{31}. Consequently, each coarse-level chunk contains column indices in the range $[0, 2^{31})$, allowing the fine-level data structures to fit into the L2 cache. The fine-level algorithm is then applied to each coarse-level chunk and is cache-efficient.

In summary, our microbenchmarks give us two key conclusions. First, when both the stream length and size are large, neither sort-based nor dense accumulation performs optimally. However, the relatively inexpensive reordering

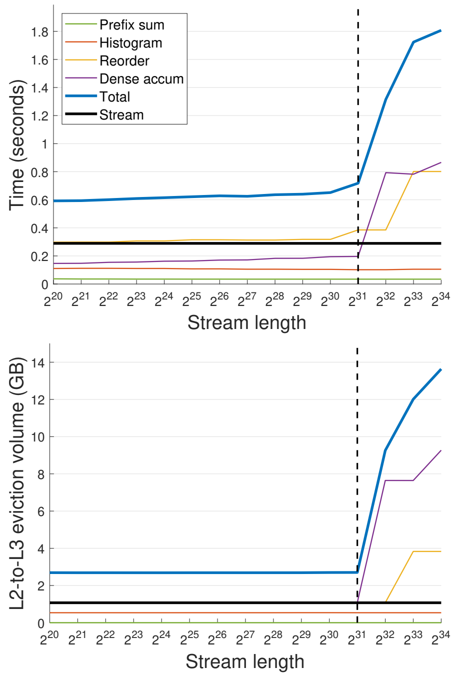

Figure 6: Wall-clock time (top) and L2-to-L3 cache evictions (bottom) versus the stream length of the input stream for a set of microbenchmarks that test the performance of the building blocks of MAGNUS. For each stream length value, the optimal number of fine-level chunks is chosen (see Section 3.5). *Total* denotes the sum of the building block times, and *Stream* is a standard streaming benchmark that serves as the peak-performance baseline. The vertical dashed line denotes the threshold at which the fine-level data structures exceed the L2 cache capacity.

mechanism in MAGNUS effectively mitigates these issues by reducing both of these quantities. For chunks with a small number of elements, sorting can be applied. Otherwise, dense accumulation is more efficient due to the short stream length per chunk, which allows the accumulation data structures to fit in the L2 cache.

Second, our microbenchmarks provide insight into the overall performance of the coarse- and fine-level algorithms. The reordering microbenchmark closely approximates the coarse-level algorithm since the dominant cost of the coarse-level algorithm is its reordering phase. Therefore, the coarse-level algorithm is approximated to perform at near-streaming speed up to the breaking point of 2^{13} chunks. For realistic matrices, this breaking point is likely not reached since that would require the multiplication of matrices with more than $2^{31}2^{13} = 2^{44}$ columns. The performance of the fine-level algorithm is closely approximated by *Total*, which is dominated by the time for reordering and dense accumulation. The significant decline in the performance of *Total* beyond a stream length of 2^{31} underscores the importance of the initial, near-streaming-speed pass over the intermediate product provided by the coarse-level algorithm. Rather than using the fine-level algorithm beyond this 2^{31} threshold, a comparatively inexpensive initial reordering step yields better performance. Note that these thresholds (2^{13} and 2^{31}) are system-dependent; MAGNUS automatically calculates these thresholds using the input system parameters (see Section 3.5).

4.2 Test Configuration for SpGEMM

In the next section, we compare an OpenMP implementation of MAGNUS to a diverse set of state-of-the-art baselines: CSeg [1], Intel MKL [9], vectorized hash/heap-based algorithms [35, 36], SuiteSparse:GraphBLAS [12], and Kokkos [14, 41]. For MKL, we use the sparse BLAS inspector-executor API, i.e., the function `mkl_sparse_spmm()`. CSeg is the only baseline that implements a locality-generation algorithm, and to the best of our knowledge, it is the only algorithm other than MAGNUS that does so.

We report the total SpGEMM time for all algorithms, where the total time is the sum of the pre-processing, symbolic, numeric, and post-processing phases. For example, the total time for CSeg is the sum of the time taken to construct the high-level summary matrix and perform the symbolic and numeric phases. In contrast, for MKL, we measure only the time of the call to `mkl_sparse_spmm()`, as that is the only operation exposed to us. For MAGNUS, the total time is the sum of the setup, symbolic, and numeric phases. We perform one warm-up run and then extract the time by taking the average of the next 10 runs. We found that the time did not vary significantly between many runs. Our test systems are shown in Table 1, all of which use Intel processors. We utilized all available threads, including hyperthreads, for all SpGEMM runs, resulting in the fastest execution time for both the baseline implementations and MAGNUS.

Table 1: Hardware specifications of the test systems. All systems are a single multisocket node with Intel CPUs.

Architecture	Skylake (SKX)	Sapphire Rapids (SPR)	Emerald Rapids (EMR)
Xeon Model	Gold 6140	Gold 6438M	Platinum 8592+
Sockets	4	2	2
Total cores and threads	72 and 144	64 and 128	128 and 256
L1 size per core	32 KB	48 KB	48 KB
L2 size per core	1 MB	2 MB	2 MB
L3 size per socket	24.75 MB	60 MB	320 MB
Memory	2 TB	4 TB	1 TB

We consider three important matrix test sets: matrices from the SuiteSparse collection [13], recursive model power-law matrices (R-mats) [7], and uniform random matrices (i.e., those from the Erdős–Rényi (ER) model) [16]. For the SuiteSparse and R-mat test sets, we consider the operation A^2 for square A, which is the de facto standard for evaluating SpGEMM algorithms. The configuration for the uniform random matrix set will be discussed later in this section. Table 2

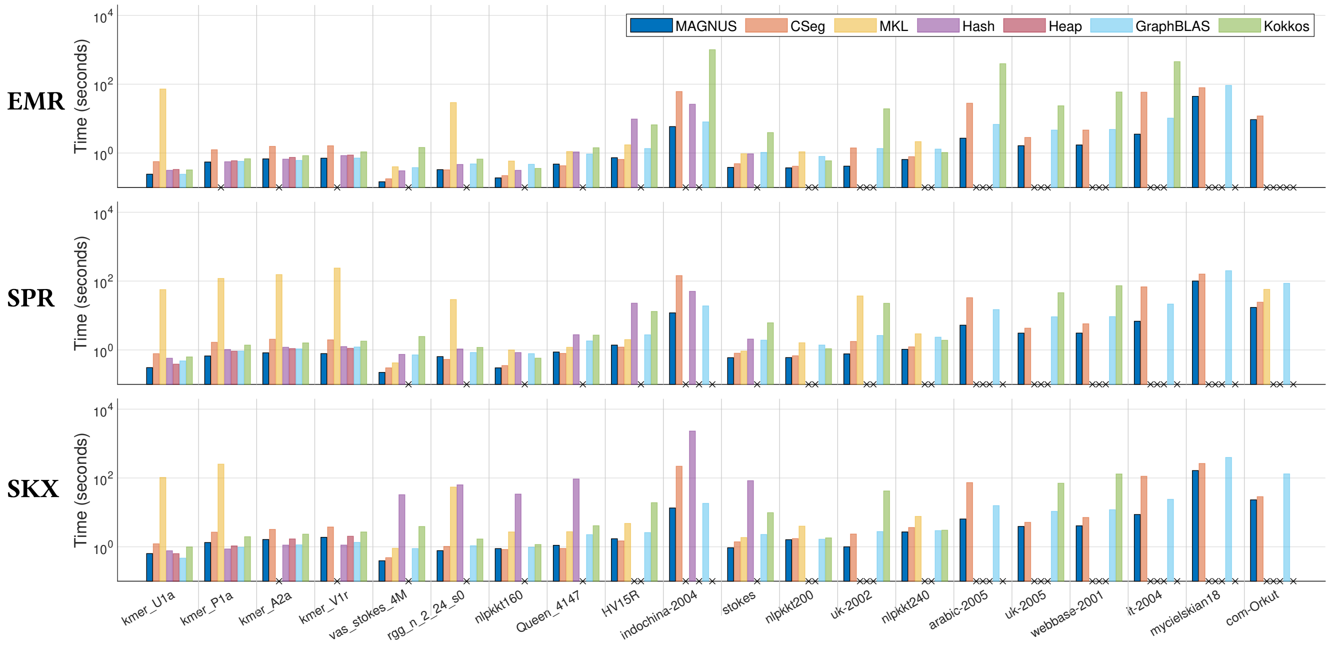

Figure 7: Wall-clock time in log scale for the SuiteSparse matrix collection. The ×-shaped markers denote failed runs (out-of-memory or segmentation faults) of the baselines. All available threads were used for each system.

Table 2: Properties of the SuiteSparse matrices.

Matrix	n_A	nnz_A	nnz_{A^2}/n_{A^2}	nnz_{A^2}
kmer_U1a	67,716,231	138,778,562	3.3	222,262,359
kmer_P1a	139,353,211	297,829,984	3.8	531,367,449
kmer_A2a	170,728,175	360,585,172	3.6	622,660,207
kmer_V1r	214,005,017	465,410,904	3.9	824,450,881
vas_stokes_4M	4,382,246	131,577,616	188.6	826,486,449
rgg_n_2_24_s0	16,777,216	265,114,400	49.4	828,639,073
nlpkkt160	8,345,600	229,518,112	148.7	1,241,294,184
Queen_4147	4,147,110	329,499,284	362.2	1,501,950,816
HV15R	2,017,169	283,073,458	876.5	1,768,066,720
indochina-2004	7,414,866	194,109,311	263.3	1,952,630,542
stokes	11,449,533	349,321,980	184.7	2,115,146,825
nlpkkt200	16,240,000	448,225,632	149.4	2,425,937,704
uk-2002	18,520,486	298,113,762	172.5	3,194,986,138
nlpkkt240	27,993,600	774,472,352	149.8	4,193,781,224
arabic-2005	22,744,080	639,999,458	366.0	8,323,612,632
uk-2005	39,459,925	936,364,282	227.4	8,972,400,198
webbase-2001	118,142,155	1,019,903,190	114.0	13,466,717,166
it-2004	41,291,594	1,150,725,436	340.2	14,045,664,641
mycielskian18	196,607	300,933,832	195,076.4	38,353,378,617
com-Orkut	3,072,441	234,370,166	16,220.6	49,836,711,933

shows the set of SuiteSparse matrices used in our experiments, where nnz_{A^2}/n_{A^2} is the average number of nonzero entries per row of A^2, which is a measure of the sparsity of A^2. These matrices are the largest 20 (in terms of the total number of nonzero entries in A) in which both the A and A^2 (the result of SpGEMM in our experiments) fit into memory for MAGNUS and at least one baseline. Table 3 shows the R-mats with an average of 16 nonzero entries per row, where the table is organized by increasing *scale*, e.g., the scale-18 R-mat has 2^{18} rows. The standard Graph500 parameters were used to generate these R-mats ($a = .57$ and $b = c = .19$). The scale-23 matrix is the largest in which both the input and output fit into memory for MAGNUS and at least one baseline on the SPR system.

Table 3: Properties of the RMat16 matrices (R-mats with an average of 16 nonzero entries per row). The standard Graph500 parameters are used ($a = .57$ and $b = c = .19$).

Scale	n_A	nnz_A	nnz_{A^2}/n_{A^2}	nnz_{A^2}
18	262,144	4,194,304	5,141.7	1,347,858,618
19	524,288	8,388,608	7,072.6	3,708,083,907
20	1,048,576	16,777,216	9,479.9	9,940,402,266
21	2,097,152	33,554,432	12,377.9	25,958,392,028
22	4,194,304	67,108,864	16,387.1	68,732,382,095
23	8,388,608	134,217,728	22,253.2	186,673,674,064

4.3 SpGEMM Results

We first consider the SuiteSparse matrix collection. Figure 7 shows the execution time in logarithmic scale for each matrix and each system. In some cases, out-of-memory errors and segmentation faults caused some of the baselines to fail, denoted by the missing bars. MAGNUS is often faster than all baselines and is often orders of magnitude faster than at least one baseline. CSeg is sometimes slightly faster than MAGNUS (up to 1.23×) for Queen_4147, HV15R, rgg_n_2_24_s0, and nlpkkt160. For these matrices, MAGNUS places all rows into the dense accumulation category due to their banded structure, i.e., these matrices do not require locality generation to be efficiently multiplied. This suggests that in the absence of locality generation, the accumulators in CSeg may be slightly

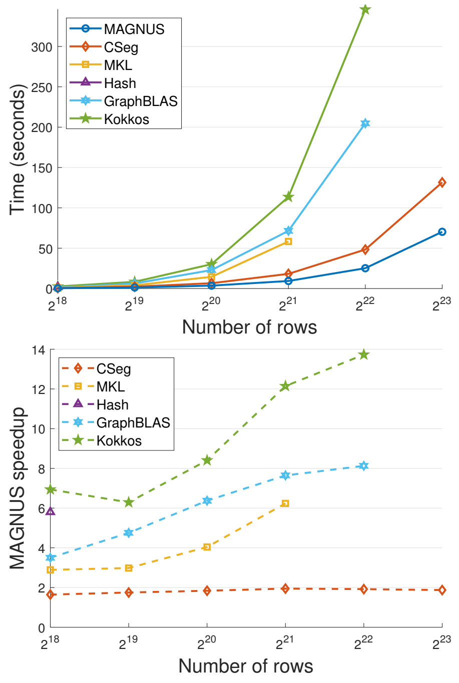

Figure 8: Wall-clock time and speedup versus number of rows for the RMat16 matrix set on the SPR system (see Table 1). The speedup is the ratio of the time of the baselines to that of MAGNUS.

more optimized for banded matrices. Similarly, the kmer matrices also do not require locality generation, where Hash, Heap, and GraphBLAS are slightly faster than MAGNUS in several cases. These matrices are not banded but are highly sparse, leading to an intermediate product per row that is less than our dense accumulation threshold. Therefore, sort- or hash map-based accumulators are most effective, as shown by MAGNUS, Hash, and Heap having the fastest times (most rows in MAGNUS are placed into the sort category).

In all other cases, MAGNUS categorizes rows as a mixture of sorting, dense accumulation, and fine-level locality, where a significant number of rows require the fine-level algorithm (for all SuiteSparse matrices, the coarse-level algorithm is not needed). MAGNUS is the fastest method for 76% of the 60 test cases (20 matrices across three systems) and is always the fastest for the 11 largest matrices. MAGNUS is up to 16.6, 306.7, 172.5, 1.4, 5.7, and 171.8 times faster than CSeg, MKL, Hash, Heap, GraphBLAS, and Kokkos, respectively, and is only 1.4 times slower than any of the baselines in the worst case. The low peak speedup over Heap is due to the high failure rate of Heap, which only ran to completion for the kmer matrices.

Figure 8 shows the time and speedup versus the number of rows for the RMat16 matrix set on the SPR system, where the speedup is the ratio of the time of the baselines to that of MAGNUS. The SPR system, which has the largest memory, allows us to scale to the largest RMat16 matrix. Showing results for the other two systems does not provide additional insight. Since we are using the standard Graph500 parameters, a small average number of nonzero entries per row (16 in this case) produces a high amount of fill-in in C (as shown in Table 3) because many of the nonzero entries in A are clustered in the top left corner. Unlike banded and other highly sparse structures that produce less fill-in and more regular access patterns, the mixture of a random distribution with clustering in the RMat16 matrices is problematic for conventional accumulators since a high data volume across a wide range of column indices results in data structures that do not fit into the L2 cache. This is demonstrated in Figure 8, which illustrates the poor scaling of all baselines, except for CSeg, as the scale increases. Heap failed in all cases and Hash failed in all but the smallest case. MAGNUS is 6.2, 5.8, 8.1, and 13.7 times faster than MKL, Hash, GraphBLAS, and Kokkos, respectively, for the largest matrices. The scaling of CSeg and MAGNUS demonstrates the importance of locality generation. Although MAGNUS is ≈ 1.8 times faster than CSeg, they both scale at a similar rate.

Lastly, we consider uniform random matrices. Unlike the R-mats, a small number of nonzero entries per row does not produce a high amount of fill-in in C. However, as we scale up the number of columns and nonzero entries per row of B, the uniform distribution of the column indices results in frequent accesses to the entire accumulator. For conventional accumulators, this becomes cache-inefficient if no locality generation strategy is used. Since the performance of conventional accumulators is sensitive to the number of columns of C and not to the number of rows (see Section 4.1), we consider the nonsquare case where C has 4096 rows and a variable number of columns. This allows us to scale to massive matrices without exceeding the memory limit of our systems. Furthermore, only the rows of B that depend on the nonzero entries in A are generated, saving additional memory.

Figure 9 shows results for increasing the number of columns with a fixed average number of nonzero entries per row of 2048. Hash and Heap failed in all cases. The black line is the ideal performance bound, which is calculated by dividing the minimum required data volume for Gustavson-based SpGEMM by the system bandwidth, i.e., $T_{ideal} = \frac{n_{readVol}+n_{writeVol}}{r_{bw}}$, where the system bandwidth r_{bw} was measured using a streaming microbenchmark. The ideal read volume is

$$n_{readVol} = 2(n_A+1)s_{rowPtr} + nnz_A(4s_{rowPtr} + 2s_{colIdx} + s_{val}) + n_{interProd}(2s_{colIdx} + s_{val}), \tag{8}$$

where s_{rowPtr} is the size of the CSR row pointer type (`size_t` in our implementation), s_{colIdx} is the size of the column index type (`uint32_t` or `uint64_t` depending on the size of the matrix), and s_{val} is the matrix coefficient value type (`float` in our experiments). The first term denotes the read volume of the row pointers of A; the second term denotes the read

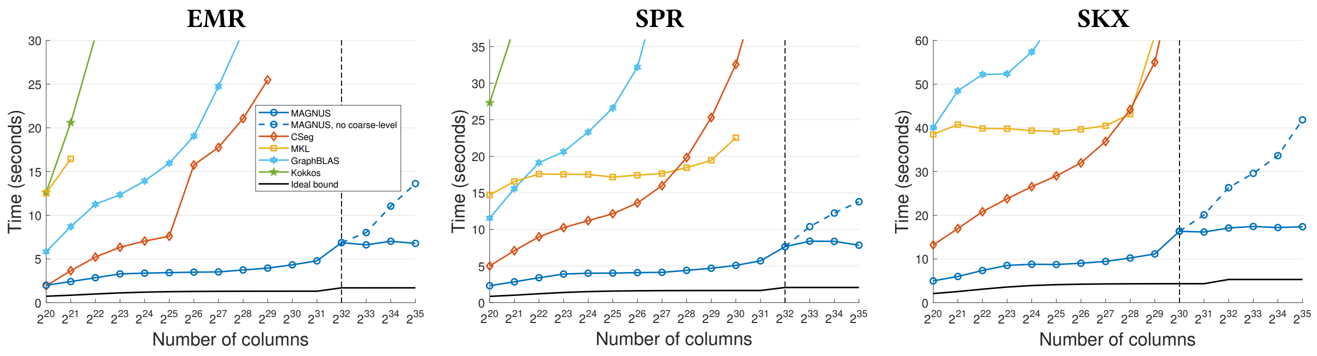

Figure 9: Time versus number of columns of C for the uniform random matrix test set. A fixed average number of nonzero entries per row of 2048 is used. The vertical dashed line denotes the point at which MAGNUS begins to use the coarse-level algorithm.

volume of the nonzero entries in A and the row pointers in B; and the third term, which is the asymptotically dominant term, denotes the read volume of the nonzero entries in the rows of B. The number of elements in the intermediate product is defined as $n_{interProd} = \sum_{(i,j)\in\mathcal{S}(A)} nnz_{B_j}$. The factors of two account for the symbolic and numeric phases. The factor of four accounts for the symbolic phase, the numeric phase, and the need to read both the start and end row pointers for each row of B (in contrast to the reading of rows of A, the rows of B are read nonconsecutively).

The ideal write volume is

$$n_{writeVol} = (n_C+1)s_{rowPtr} + nnz_C(s_{colIdx}+s_{val}), \qquad (9)$$

with the first term corresponding to writing the row pointers of C and the second term to writing the nonzero entries. The overall data volume is the sum of the read and write data volumes. This ideal bound does not account for various costs, such as NUMA effects, synchronization overheads, or the performance of accessing intermediate data structures (e.g., the accumulator), which can have a significant impact on the performance of SpGEMM algorithms. Additionally, this bound does not consider cached rows of B, as reflected in the expression for $n_{interProd}$, which assumes that previously read rows of B are not reused. For this reason, we only consider this bound for the uniform random matrices, where there is minimal opportunity for row reuse in B.

Figure 9 shows that MAGNUS maintains an average multiple of $\approx$ 2.7 of the ideal bound before applying the coarse-level algorithm and $\approx$ 3.5 afterward. The increase in the multiple is due to the increase in data volume incurred by the outer product (additionally, the slight increase in the ideal bound at 2^{32} is due to a change from `uint32_t` to `uint64_t` for s_{colIdx}). The 2.7 multiple is consistent with the multiple from our microbenchmarks. In contrast, the baselines, including CSeg, diverge from the ideal bound. This suggests that the locality generation method in CSeg, which explicitly constructs an auxiliary segmented matrix, does not scale if a large number of segments is required. The vertical dashed line shows the point at which MAGNUS starts to place rows in the coarse-level category. For SKX, the crossover point occurs at 2^{30}, compared to 2^{32} for SPR and EMR, due to the smaller L2 cache size in SKX. The dashed blue line shows MAGNUS with the coarse-level algorithm turned off, where MAGNUS diverges from the ideal bound. This shows the necessity of multiple levels of locality, especially for massive matrices where the fine-level data structures do not fit into the L2 cache.

5 Conclusion

On modern CPUs, current SpGEMM algorithms often scale poorly to massive matrices due to inefficient use of the cache hierarchy. We present MAGNUS, a novel algorithm for locality generation, where the intermediate product is reordered into cache-friendly chunks using a hierarchical two-level approach. MAGNUS consists of two algorithms that create multiple levels of locality: the fine- and coarse-level algorithms. The coarse-level algorithm generates a set of coarse-level chunks, and the fine-level algorithm further subdivides the coarse-level chunks into cache-friendly fine-level chunks. An accumulator is then applied to each fine-level chunk, where a dense or sort-based accumulator is selected based on a threshold on the number of elements in the chunk. MAGNUS is input- and system-aware: the chunk properties are determined using the matrix dimensions and the system cache sizes.

Our experimental results compare MAGNUS with several state-of-the-art baselines for three matrix sets on three Intel architectures. MAGNUS is faster than all the baselines in most cases and is often an order of magnitude faster than at least one baseline. More importantly, MAGNUS scales to massive, uniform random matrices, the most challenging test sets that we consider. This challenging case highlights the importance of the locality-generation techniques in MAGNUS, which allows MAGNUS to scale with an ideal performance bound independent of the matrix properties. In contrast, the baselines diverge from this bound as the matrix size increases.

References

[1] Xiaojing An and Ümit V. Çatalyürek. 2021. Column-Segmented Sparse Matrix-Matrix Multiplication on Multicore CPUs. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 202–211. https://doi.org/10.1109/HiPC53243.2021.00034

[2] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. 2016. Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication. In *Proceedings of the 2016 International Conference on Supercomputing (ICS)* (Istanbul, Turkey). Association for Computing Machinery, New York, NY, USA, Article 36, 12 pages. https://doi.org/10.1145/2925426.2926273

[3] Ariful Azad, Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. 2016. Exploiting Multiple Levels of Parallelism in Sparse Matrix-Matrix Multiplication. *SIAM J. Sci. Statist. Comput.* 38, 6 (2016), C624–C651. https://doi.org/10.1137/15M104253X

[4] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS) Workshop*. 804–811. https://doi.org/10.1109/IPDPSW.2015.75

[5] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic Acids Res.* 46, 6 (January 2018), e33–e33. https://doi.org/10.1093/nar/gkx1313

[6] Berenger Bramas. 2017. A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake. *Int. J. Adv. Comput. Sci. Appl.* 8, 10 (2017). https://doi.org/10.14569/IJACSA.2017.081044

[7] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining (SDM)*. SIAM, 442–446. https://doi.org/10.1137/1.9781611972740.43

[8] Helin Cheng, Wenxuan Li, Yuechen Lu, and Weifeng Liu. 2023. HASpGEMM: Heterogeneity-Aware Sparse General Matrix-Matrix Multiplication on Modern Asymmetric Multicore Processors. In *Proceedings of the 52nd International Conference on Parallel Processing (ICPP)*. Association for Computing Machinery, New York, NY, USA, 807–817. https://doi.org/10.1145/3605573.3605611

[9] Intel Corporation. 2023. Developer Reference for Intel oneAPI Math Kernel Library (MKL) for C.

[10] NVIDIA Corporation. 2023. cuSPARSE Library.

[11] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing Sparse Matrix-Matrix Multiplication for the GPU. *ACM Trans. Math. Software* 41, 4, Article 25 (October 2015), 20 pages. https://doi.org/10.1145/2699470

[12] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Software* 45, 4, Article 44 (December 2019), 25 pages. https://doi.org/10.1145/3322125

[13] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1, Article 1 (December 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[14] Mehmet Deveci, Christian Trott, and Siva Rajamanickam. 2018. Multi-threaded Sparse Matrix-Matrix Multiplication for Many-Core and GPU Architectures. *Parallel Comput.* 78 (January 2018). https://doi.org/10.1016/j.parco.2018.06.009

[15] Zhaoyang Du, Yijin Guan, Tianchan Guan, Dimin Niu, Linyong Huang, Hongzhong Zheng, and Yuan Xie. 2022. OpSparse: A Highly Optimized Framework for Sparse General Matrix Multiplication on GPUs. *IEEE Access* 10 (2022), 85960–85974. https://doi.org/10.1109/ACCESS.2022.3196940

[16] P. Erdös and A. Rényi. 1959. On Random Graphs I. *Publicationes Mathematicae Debrecen* 6 (1959), 290.

[17] R.D. Falgout. 2006. An introduction to algebraic multigrid. *Comput. Sci. Eng.* 8, 6 (2006), 24–33. https://doi.org/10.1109/MCSE.2006.105

[18] Xu Feng, Yuyang Xie, Mingye Song, Wenjian Yu, and Jie Tang. 2018. Fast randomized PCA for sparse data. In *Asian conference on machine learning*. PMLR, 710–725. https://proceedings.mlr.press/v95/feng18a.html

[19] 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. *Comput. Surveys* 55, 12, Article 244 (March 2023), 36 pages. https://doi.org/10.1145/3571157

[20] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. 2007. High-Performance Graph Algorithms from Parallel Sparse Matrices. In *Applied Parallel Computing: State of the Art in Scientific Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 260–269. https://doi.org/10.1007/978-3-540-75755-9_32

[21] Vitaliy Gleyzer, Andrew J. Soszynski, and Edward K. Kao. 2020. Leveraging Linear Algebra to Count and Enumerate Simple Subgraphs. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–8. https://doi.org/10.1109/HPEC43674.2020.9286191

[22] Thomas Gruber, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2014. Overhead Analysis of Performance Counter Measurements. In *43rd International Conference on Parallel Processing Workshops (ICCPW)*. 176–185. https://doi.org/10.1109/ICPPW.2014.34

[23] Zhixiang Gu, Jose Moreira, David Edelsohn, and Ariful Azad. 2020. Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication using Propagation Blocking. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (Virtual Event, USA). Association for Computing Machinery, New York, NY, USA, 293–303. https://doi.org/10.1145/3350755.3400216

[24] Giulia Guidi, Marquita Ellis, Daniel Rokhsar, Katherine Yelick, and Aydın Buluç. 2021. BELLA: Berkeley Efficient Long-Read to Long-Read Aligner and Overlapper. In *Proceedings of the 2021 SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. 123–134. https://doi.org/10.1137/1.9781611976830.12

[25] Giulia Guidi, Oguz Selvitopi, Marquita Ellis, Leonid Oliker, Katherine A. Yelick, and Aydin Buluç. 2021. Parallel String Graph Construction and Transitive Reduction for De Novo Genome Assembly. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 517–526. https://doi.org/10.1109/IPDPS49936.2021.00060

[26] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Software* 4, 3 (September 1978), 250–269. https://doi.org/10.1145/355791.355796

[27] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in deep learning: pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.* 22, 1, Article 241 (Jan. 2021), 124 pages. https://dl.acm.org/doi/abs/10.5555/3546258.3546499

[28] Haim Kaplan, Micha Sharir, and Elad Verbin. 2006. Colored intersection searching via sparse rectangular matrix multiplication. In *Proceedings of the Twenty-Second Annual Symposium on Computational Geometry (SCG)* (Sedona, Arizona, USA). Association for Computing Machinery, New York, NY, USA, 52–60. https://doi.org/10.1145/1137856.1137866

[29] Valentin Le Fèvre and Marc Casas. 2023. Efficient Execution of SpGEMM on Long Vector Architectures. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (Orlando, FL, USA). Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/3588195.3593000

[30] Jiayu Li, Fugang Wang, Takuya Araki, and Judy Qiu. 2019. Generalized Sparse Matrix-Matrix Multiplication for Vector Engines and Graph Applications. In *2019 IEEE/ACM Workshop on*

Memory Centric High Performance Computing (MCHPC). 33–42. https://doi.org/10.1109/MCHPC49590.2019.00012

[31] Ruipeng Li, Björn Sjögreen, and Ulrike Meier Yang. 2021. A New Class of AMG Interpolation Methods Based on Matrix-Matrix Multiplications. *SIAM J. Sci. Comput.* 43, 5 (2021), S540–S564. https://doi.org/10.1137/20M134931X

[32] Junhong Liu, Xin He, Weifeng Liu, and Guangming Tan. 2019. Register-Aware Optimizations for Parallel Sparse Matrix-Matrix Multiplication. *Int. J. Parallel Program.* 47, 3 (June 2019), 403–417. https://doi.org/10.1007/s10766-018-0604-8

[33] Weifeng Liu and Brian Vinter. 2014. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 370–381. https://doi.org/10.1109/IPDPS.2014.47

[34] Weifeng Liu and Brian Vinter. 2015. A framework for general sparse matrix–matrix multiplication on GPUs and heterogeneous processors. *J. Parallel and Distrib. Comput.* 85 (2015), 47–61. https://doi.org/10.1016/j.jpdc.2015.06.010

[35] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2018. High-Performance Sparse Matrix-Matrix Products on Intel KNL and Multicore Architectures. In *Workshop Proceedings of the 47th International Conference on Parallel Processing (ICPP)* (Eugene, OR, USA). Association for Computing Machinery, New York, NY, USA, Article 34, 10 pages. https://doi.org/10.1145/3229710.3229720

[36] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2019. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Comput.* 90 (2019), 102545. https://doi.org/10.1016/j.parco.2019.102545

[37] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing (ICPP)*. 101–110. https://doi.org/10.1109/ICPP.2017.19

[38] Mathias Parger, Martin Winter, Daniel Mlakar, and Markus Steinberger. 2020. spECK: accelerating GPU sparse matrix-matrix multiplication through lightweight analysis. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (San Diego, California). Association for Computing Machinery, New York, NY, USA, 362–375. https://doi.org/10.1145/3332466.3374521

[39] 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 *High Perform. Comput.* Springer International Publishing, Cham, 48–57. https://doi.org/10.1007/978-3-319-20119-1_4

[40] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Dipankar Srinivasan, Sudarshan an 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

[41] Sivasankaran Rajamanickam, Seher Acer, Luc Berger-Vergiat, Vinh Dang, Nathan Ellingwood, Evan Harvey, Brian Kelley, Christian R. Trott, Jeremiah Wilke, and Ichitaro Yamazaki. 2021. Kokkos Kernels: Performance Portable Sparse/Dense Linear Algebra and Graph Kernels. arXiv:2103.11991

[42] Oguz Selvitopi, Md Taufique Hussain, Ariful Azad, and Aydın Buluç. 2020. Optimizing High Performance Markov Clustering for Pre-Exascale Architectures. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 116–126. https://doi.org/10.1109/IPDPS47924.2020.00022

[43] Hikaru Takayashiki, Hotaka Yagi, Hiroki Nishimoto, and Naoki Yoshifuji. 2023. A New Sparse General Matrix-matrix Multiplication Method for Long Vector Architecture by Hierarchical Row Merging. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. Association for Computing Machinery, New York, NY, USA, 756–759. https://doi.org/10.1145/3624062.3625131

[44] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2019. Adaptive sparse matrix-matrix multiplication on the GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)* (Washington, District of Columbia). Association for Computing Machinery, New York, NY, USA, 68–81. https://doi.org/10.1145/3293883.3295701

[45] Michael M. Wolf, Jonathan W. Berry, and Dylan T. Stark. 2015. A task-based linear algebra Building Blocks approach for scalable graph analytics. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2015.7322450

[46] Kathy Yelick, Aydın Buluç, Muaaz Awan, Ariful Azad, Bowei Brock, Rob Egan, Saliya Ekanayake, Marquita Ellis, Evangelos Georganas, Giulia Guidi, Steven Hofmeyr, Oguz Selvitopi, Cristina Teodoropol, and Leonid Oliker. 2020. The parallelism motifs of genomic data analysis. *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci.* 378, 2166 (2020), 20190394. https://doi.org/10.1098/rsta.2019.0394

[47] Chao Zhang, Maximilian Bremer, Cy Chan, John Shalf, and Xiaochen Guo. 2022. ASA: Accelerating Sparse Accumulation in Column-wise SpGEMM. *ACM Trans. Archit. Code Optim.* 19, 4, Article 49 (September 2022), 24 pages. https://doi.org/10.1145/3543068

[48] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. GAMMA: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Virtual, USA). Association for Computing Machinery, New York, NY, USA, 687–701. https://doi.org/10.1145/3445814.3446702