Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication using Propagation Blocking

Zhixiang Gu[*‡], Jose Moreira[†], David Edelsohn[†], Ariful Azad[*]

E-mail:gzxultra@fb.com,jmoreira@us.ibm.com,edelsohn@us.ibm.com,azad@iu.edu

[*]Indiana University Bloomington, [†]IBM Corporation, [‡]Facebook Inc.

Abstract

Sparse matrix-matrix multiplication (SpGEMM) is a widely used kernel in various graph, scientific computing and machine learning algorithms. It is well known that SpGEMM is a memory-bound operation, and its peak performance is expected to be bound by the memory bandwidth. Yet, existing algorithms fail to saturate the memory bandwidth, resulting in suboptimal performance under the Roofline model. In this paper, we characterize existing SpGEMM algorithms based on their memory access patterns and develop practical lower and upper bounds for SpGEMM performance. We then develop an SpGEMM algorithm based on the outer product. The newly developed algorithm called PB-SpGEMM saturates memory bandwidth by using the propagation blocking technique and by performing in-cache sorting and merging. For many practical matrices, PB-SpGEMM runs 20%-50% faster than the state-of-the-art heap and hash SpGEMM algorithms on modern multicore processors. Most importantly, PB-SpGEMM attains performance predicted by the Roofline model, and its performance remains stable with respect to matrix size and sparsity.

ACM Reference Format:

1 Introduction

Sparse matrix-matrix multiplication (SpGEMM) is a widely-used kernel in many graph analytics, scientific computing, and machine learning algorithms. In graph analytics, SpGEMM is used in betweenness centrality [9], clustering coefficients, triangle counting [2], multi-source breadth-first search [15], colored intersection search [19], and cycle detection [30] algorithms. In scientific computing, SpGEMM is used in algebraic multigrid [5] and linear solvers. Many machine learning tasks like dimensionality reduction (e.g., NMF, PCA [13]), spectral clustering [18], and Markov clustering [3] rely on an efficient SpGEMM algorithm as well.

In most data analytics applications, SpGEMM has very low arithmetic intensity (AI) measured by the ratio of total floating-point operations to total data movement. For example, when multiplying two Erdős-Rényi random matrices with d nonzeros per column (matrices with d nonzeros uniformly distributed in each column), an algorithm has an AI of just $\frac{1}{16}$ flops/byte (see Sec. 2.3). At this arithmetic intensity, SpGEMM is a memory-bound operation, and SpGEMM's peak performance has an upper bound of $\beta * AI$, where β is the memory bandwidth. Assuming 50GB/s bandwidth available on a multicore processor, the estimated peak performance can be as high as $50/16 = 3.13$ GFLOPS (billions of floating point operations per second). However, state-of-the art parallel algorithms based on heap and hash merging [24] attain no more than 500 MFLOPS for random matrices on a socket of an Intel Skylate processor. No prior work clearly explained these observed performances of SpGEMM algorithms as no standard performance model has been developed to understand SpGEMM's performance.

In this paper, we rely on the Roofline model [29] and develop lower and upper performance bounds for SpGEMM algorithms. In developing practical lower bounds on AI, we considered the fact that an SpGEMM may read data more than once. Even with a tight lower bound on AI, an algorithm can attain peak performance (that is $\beta * AI$) only if it (a) saturates the memory bandwidth, (b) does not have a high latency cost, and (c) makes use of full cache lines of data. We show that these requirements are not satisfied by current column-by-column algorithms, resulting in a performance that is lower than the peak attainable FLOPS.

Here, we develop a new algorithm based on the outer product of matrices. The goal of this algorithm is to eliminate irregular data accesses, increase bandwidth utilization, and attain the performance predicted by the Roofline model. Our algorithm is built upon the expansion-sort-compress paradigm [7, 10]. Given two $n \times n$ matrices, this algorithm performs n outer products to generate intermediate tuples of row index, column index, and the multiplied value. These tuples are then sorted, and duplicate row and column indices are merged to get the final product. To ensure efficient bandwidth utilization, we store intermediate tuples into partially-sorted bins so that each bin can be sorted and merged independently. This technique of binning intermediate data for better bandwidth utilization is called *propagation blocking* (PB) [6]. Prior work has used propagation blocking to improve the bandwidth utilization as well as the overall performance of sparse matrix-vector multiplications [6, 26]. Here we use propagation blocking to regularize data movements in SpGEMM. Hence, our algorithm is named as *PB-SpGEMM*.

All three phases of PB-SpGEMM (expand, sort, and compress or merge) stream data from memory. Hence, every phase has no significant latency overhead, utilizes full cache lines, and attains a

bandwidth close to the STREAM benchmark. Therefore, given the bandwidth of a system and input matrices, PB-SpGEMM's performance matches the prediction of the Roofline model.

Similar to any expansion-sort-compress algorithm, PB-SpGEMM has to store flops intermediate tuples, where flops is the number of multiplications needed by SpGEMM. This can lead to significant data movements when the *compression factor* (the ratio of flops to nonzeros in the output) is greater than four. However, most practical SpGEMM operations have small compression factors. For example, when squaring matrices from the SuiteSparse Matrix Collection [11], more than 80% of SpGEMMs have a compression factor less than 3 and about 99% have a compression factor less than 6 [21]. Hence, for most practical scenarios, PB-SpGEMM performs predictably better than existing heap and hash algorithms. For multiplications with compression factor greater than four, PB-SpGEMM's performance is still predictable, but it can run slower than the alternatives.

We summarize the key contributions of this paper below:

(1) We develop a Roofline performance model for SpGEMM algorithms. This model can reveal the limitations of existing column-by-column SpGEMM algorithms.
(2) We develop an outer-product based SpGEMM algorithm called PB-SpGEMM. We use propagation blocking, in-cache sorting and merging for better bandwidth utilization.
(3) Different phases of PB-SpGEMM utilize bandwidth close to the STREAM benchmark. Given the bandwidth of a system and input matrices, PB-SpGEMM's performance matches the prediction from the Roofline model.
(4) For SpGEMMs with compression factor less than four, PB-SpGEMM is 30% to 50% faster than previous state-of-the-art algorithms for multicore processors.

Our implementation of PB-SpGEMM is publicly available at Bitbucket: https://bitbucket.org/azadcse/pb-spgemm/.

2 A Performance Model for SpGEMM

2.1 Notations

Given two sparse matrices $\mathbf{A} \in \mathbb{R}^{m \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times n}$, SpGEMM computes another potentially sparse matrix $\mathbf{C} \in \mathbb{R}^{m \times n}$. $\mathbf{A}(:, j)$ denotes jth column of $\mathbf{A}$, and $\mathbf{A}(i, j)$ denotes ith entry in the jth column. In our analysis, we consider n-by-n matrices for simplicity. We use Erdős-Rényi (ER) random matrices throughout the paper. An ER matrix with d nonzeros per column has d nonzeros uniformly distributed in each column. Among many representations for sparse matrices, we consider three standard data structures in this paper: Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and Coordinate format (COO). See Langr and Tvrdik [20] for a comprehensive discussion on sparse matrix storage formats.

Given a matrix $\mathbf{A}$, $nnz(\mathbf{A})$ denotes the number of nonzeros in $\mathbf{A}$, and $d(A) = \frac{nnz(\mathbf{A})}{n}$ denotes average nonzeros in a row or column. In computing $\mathbf{C}=\mathbf{AB}$, flops denotes the number of multiplications needed. Throughout the paper, floating point operations only denote multiplications. The *compression factor (cf)* denotes the ratio of flops to nonzeros in the output matrix: $cf=\frac{\text{flops}}{nnz(\mathbf{C})}$. Since at least one multiplication is needed for every output nonzero, $cf \geq 1$.

		Input Access	
		Column wise	Outer Product
Output	Heap/Hash/SPA	[1, 12, 24, 28]	[8]
Formation	ESC	[10, 21]	[27], this

Table 1: Classification of SpGEMM algorithms based on access patterns of input and output matrices. This paper falls into the lower right cell.
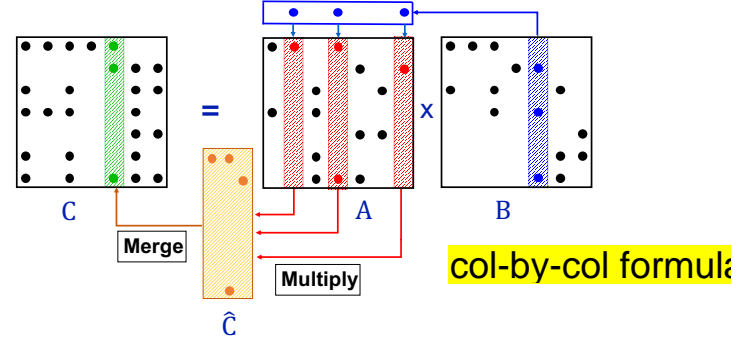

Figure 1: An illustration of column SpGEMM algorithm. To generate $\mathbf{C}(:, 5)$, the algorithm selects 2nd, 4th and 7th columns of $\mathbf{A}$ (shown in red) based on the nonzeros in $\mathbf{B}(:, 5)$ (shown in blue). The selected columns are multiplied by corresponding entries in $\mathbf{B}(:, 5)$, forming entries in $\hat{\mathbf{C}}(:, 5)$ (shown in yellow). Duplicate entries in $\hat{\mathbf{C}}(:, 5)$ are merged to generate $\mathbf{C}(:, 5)$ (shown in green). The merging strategy differs in different column SpGEMM algorithms.

2.2 Classes of SpGEMM algorithms categorized by data access patterns

To compute $\mathbf{C}=\mathbf{AB}$, most algorithms also manipulate an expanded matrix $\hat{\mathbf{C}} \in \mathbb{R}^{m \times n}$ that contains flops unmerged entries. The final output $\mathbf{C}$ can be obtained by merging entries in $\hat{\mathbf{C}}$ with the same row and column indices. Therefore, if we primarily focus on data accesses, SpGEMM algorithms have two distinct phases: (a) input matrices are accessed to form $\hat{\mathbf{C}}$ and (b) duplicate entries in $\hat{\mathbf{C}}$ are merged to form $\mathbf{C}$. Here, merging duplicate entries means adding multiplied values with the same row and column indices. For input accesses, we have two options: (1) access $\mathbf{A}$ and $\mathbf{B}$ column-by-column (or equivalently row-by-row) [1, 10, 12, 24, 28], and (2) access $\mathbf{A}$ column-by-column and access $\mathbf{B}$ row-by-row for outer product [8, 27]. For output formations, we have many options. Prior work used the expand-sort-compress strategy [10, 21, 27] and accumulators based on heap [1], hash table [24], and a dense vector called SPA [14, 28]. Table 1 summarizes prior work based on their data access patterns. Next, we briefly summarize major classes of SpGEMM algorithms and discuss their data access patterns.

Column SpGEMM algorithms based on the heap/hash/SPA accumulator. Some algorithms materialize only one column of $\hat{\mathbf{C}}$, merge duplicate entries in that column and generate the corresponding column of $\mathbf{C}$. Since column-by-column and row-by-row algorithms have similar computational patterns, we only discuss column SpGEMM algorithms in this paper. An illustration of the column SpGEMM algorithm is shown in Fig. 1, where $\mathbf{C}(:, 5)$ is generated by merging a subset of columns in $\mathbf{A}$ determined by nonzeros in $\mathbf{B}(:, 5)$. Most column-by-column algorithms are based
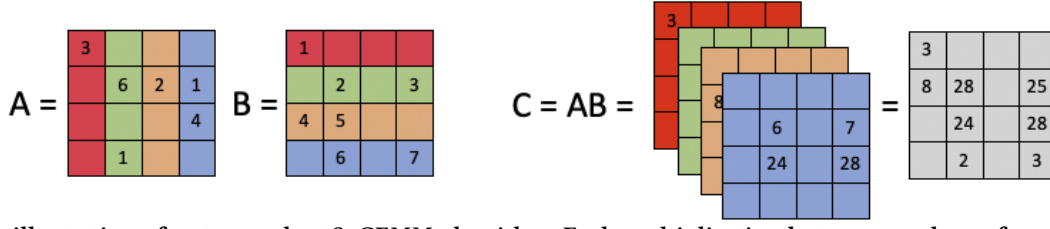
Figure 2: An illustration of outer product SpGEMM algorithm. Each multiplication between a column from A and a corresponding row from B generate a sub 1-rank matrix, those sub 1-rank matrices added up yield the final result C

	No of Accesses				Streamed Access				Cache Line Utilization			
Algorithm	A	B	$\hat{C}$	C	A	B	$\hat{C}$	C	A	B	$\hat{C}$	C
Column SpGEMM (Heap/Hash/SPA)	d	1	0*	1	×	✓	✓	✓	× (when $d < 8$)	✓	✓	✓
ESC (column-wise)	d	1	2	1	×	✓	×	✓	× (when $d < 8$)	✓	✓	✓
ESC (outer product)	1	1	2	1	✓	✓	✓**	✓	✓	✓	✓	✓

* Column SpGEMM generates one column of $\hat{C}$ at a time. ** Using blocking techniques discussed in this paper.

Table 2: Data access patterns in different SpGEMM algorithms when multiplying two ER matrices with d nonzeros per column. If an algorithm does not stream data, it has high latency cost.

on Gustavson's algorithm [17], and they differ from one another based on how they merge entries in $\hat{C}(:, j)$ to obtain $C(:, j)$. Prior work have used heap [1], hash table [25], or a dense vector called SPA [14] for merging columns. A common characteristic of all column-by-column algorithms is that they read from B and write to C one column at a time. However, columns of A may be read irregularly several times based on the nonzero pattern of B.

We explain the access pattern of A when multiplying two ER matrices with d nonzeros per column. In the worst case, each column of A is read d times from memory because columns of A are accessed randomly without any spatial locality. Thus, we read A d times over the execution of the algorithm. Second, if we store matrices in the CSC format, column SpGEMM algorithms have good spatial locality for B, $\hat{C}$ and C, but not for A. Hence, we pay heavy latency costs for irregular accesses of A's columns. Third, if a column of A has very few entries (e.g., when $d \approx 1$), we read a column of A in a cache line, but the whole cache line is not used. Consequently, column SpGEMMs waste memory bandwidth for very sparse matrices. The first row of Table 2 summarizes the data access patterns in column SpGEMM algorithms. Similarly, a row-by-row algorithm (when matrices are stored in the CSR format) has good spatial locality for A, $\hat{C}$ and C, but not for B.

The Expand-Sort-Compress (ESC) algorithms. Algorithms based on the ESC technique generate full $\hat{C}$ before merging duplicate entries. The original ESC algorithm [10] developed for GPUs generates $\hat{C}(:, i)$ using $B(:, i)$. similar to the first step of column SpGEMM algorithm shown in Fig. 1. After the entire $\hat{C}$ is constructed, flops tuples in $\hat{C}$ are sorted and merged to generate the final output. Since sorting can be efficiently performed on GPUs, ESC SpGEMM may perform better than other algorithms on GPUs [10, 21]. The column ESC algorithm has access patterns for A, B, and C similar to the column SpGEMM algorithm. Additionally, $\hat{C}$ is accessed twice (one write after multiplication and one read before merging). The second row of Table 2 summarizes the data access patterns in the column ESC algorithms.

Previous work [10] tried to eliminate reading and writing $\hat{C}$ from memory by partitioning A by rows and multiplying each partition with B. Thus, this approach generates one partition of $\hat{C}$ at a time, which can fit in cache when a large number of partitions is used. However, the partitioned ESC algorithm needs to read B several times and accesses parts of A irregularly in each partition. Hence, the effectiveness of partitioning depends on the number of partitions and nonzero structures of input matrices.

Outer product algorithms. Fig. 2 shows an illustration of the outer product algorithm. In this formulation, $A(:, i)$ is multiplied with $B(i, :)$ to form a rank-1 outer product matrix. The rank-1 matrices can be merged by using a heap or using ESC. A heap can be used to merge the outer product of $A(:, i)$ and $B(i, :)$ with the current output after every iteration [8]. However, this algorithm is too expensive as it requires n merging operations. Hence, we do not elaborate this algorithm further.

The rank-1 matrices from outer product can also be merged using the ESC strategy. In this case, input matrices are streamed only once. Hence, the outer product can fully utilize cache lines when reading inputs. To sort and merge unmerged tuples in $\hat{C}$, we need to read them from memory. This can significantly increase the memory traffic. Nonetheless, with an efficient blocking technique discussed in this paper, we can stream $\hat{C}$ whenever it is accessed. Hence, we can utilize full cache lines and saturate memory bandwidth to offset increased data accesses. The last row of Table 2 summarizes the data access patterns in the outer-product-based ESC algorithm.

2.3 Arithmetic Intensity (AI) of SpGEMM.

AI is the ratio of total floating-point operations to total data movement (bytes). To compute C=AB, one must read A and B from memory, and write C to memory. Assume that on average, we need b bytes to store a nonzero. Then,

$$\text{AI} \leq \frac{nnz(C) * cf}{[nnz(A) + nnz(B) + nnz(C)] * b} \leq \frac{cf}{b}. \qquad (1)$$
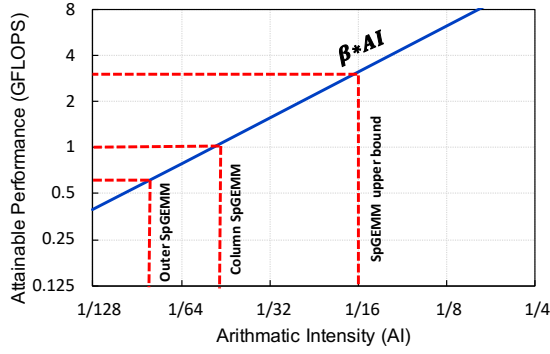
Figure 3: Using the Roofline model to estimate performance when multiplying two Erdős-Rényi matrices on a single-socket Intel Skylake processor. Memory bandwidth β is 50GB/s as measured by the STREAM benchmark.

If we use 4 bytes for indices and 8 bytes for values, then b is 16 bytes (assuming that matrices are stored in the COO format). Here, cf is a property of input matrices and it varies from 1 to 8 for most practical sparse matrices [21]. Even in the best scenario when we read and write matrices just once, the arithmetic intensity of SpGEMM is very low, often less than one. Let's consider multiplying two ER matrices A and B, since cf for ER matrix is close to 1 in expectation [4], according to our model, AI will be around $\frac{1}{16}$ flops/byte. At this AI, SpGEMM's performance is bound by the memory bandwidth.

Peak performance of an SpGEMM algorithm. Suppose, a smart algorithm achieves the best AI of $\frac{cf}{b}$. Let β be the memory bandwidth (BW) of the system measured by the STREAM benchmark. Then, the performance measured by FLOPS is as follows:

$$\text{FLOPS} \leq \beta \frac{cf}{b}. \tag{2}$$

Hence, the *peak* FLOPS for a given problem on a given architecture can be at most $\beta \frac{cf}{b}$ assuming that SpGEMM is bandwidth bound. For example, on an Intel Skylake processor with 50GB/s memory bandwidth, the peak performance for multiplying ER matrices can be at most $50 * 1/16 = 3.13$ GFLOPS as shown in Fig. 3. However, state-of-the-art column SpGEMM algorithms achieve less than 20% of this peak performance as discussed in a recent paper [24].

The primary reasons behind the suboptimal performance of SpGEMM algorithms are: (1) algorithms read/write data multiple times, (2) algorithms access data at random memory locations, (3) algorithms may not fully utilize cache lines. The first problem is inherent to SpGEMM and cannot be completely overcome when input matrices are unstructured. The irregular data access can under utilize bandwidth, impeding the performance of SpGEMM when the compression factor is small. In this paper, we address this irregular access problem and develop an algorithm where all steps of SpGEMM utilize full memory bandwidth. Nevertheless, Equation 1 seems an upper bound that no existing algorithm can attain. Next, we consider a more practical bound on AI.

A more practical bound on AI for SpGEMM. A column SpGEMM algorithm reads the first input matrix several times depending on the nonzero pattern of $\mathbf{B}$. To obtain a lower bound for column

SpGEMM, we assume that the accesses of $\mathbf{A}$ have no temporal or spatial locality, and all accesses of $\mathbf{A}$ incur memory traffic. Hence, in the worst case, the amount of data read from A is $\text{flops}*b = nnz(C)*cf*b$ bytes. Hence AI for column SpGEMM is expressed as follows:

$$\text{AI (Col SpGEMM)} \geq \frac{nnz(C) * cf}{[nnz(C) * cf + nnz(B) + nnz(C)] * b}$$
$$\geq \frac{cf}{(2 + cf)b}. \tag{3}$$

The outer product based algorithm based on the ESC strategy generates all unmerged tuples in $\hat{\mathbf{C}}$, writes all $nnz(\hat{\mathbf{C}}) = \text{flops}$ tuples in memory, reads them again for sorting and merging. Hence, in the worst case, ESC-based algorithms perform additional $2 * \text{flops}$ memory red-write operations, giving us the following bound.

$$\text{AI (Outer ESC)} \geq \frac{nnz(C) * cf}{[nnz(\mathbf{A}) + nnz(\mathbf{B}) + 2 * nnz(\hat{\mathbf{C}}) + nnz(\mathbf{C})] * b}$$
$$= \frac{nnz(C) * cf}{[nnz(A) + nnz(B) + 2 * \text{flops} + nnz(C)] * b}$$
$$\geq \frac{cf}{(3 + 2cf)b}. \tag{4}$$

Fig. 3 shows the lower bounds on AI for various settings of SpGEMM based on Eq. 1, 3, 4. In this example, we consider ER matrices with $cf = 1$ and $b = 16$ bytes. The lowest value of AI from Eq. 3 is 1/48 and from is 1/80. Fig. 3 then shows the corresponding attainable performance ($\beta * \text{AI}$) for a system with $\beta = 50$GB/s. Thus, column SpGEMM should attain at least 1 GFLOPS and the outer ESC algorithm should attain close to 625 MFLOPS.

Our experimental results show that column SpGEMMs attain up to 500 MFLOPS for ER matrices. There are two reasons behind the performance of column SpGEMMs. First, in the worst case, elements of $\mathbf{A}$ are accessed from $nnz(\mathbf{B})$ random locations incurring huge latency overhead. Second, for very sparse matrices, portion of cache lines are wasted, reducing the effective bandwidth. Because of the random accesses in one input matrix, column SpGEMM does not attain the theoretical peak performance shown in Fig. 3. By contrast, the newly-developed PB-SpGEMM algorithm always utilizes the full memory bandwidth. Hence, even with a lower theoretical peak (based on Eq. 4), PB-SpGEMM performs better on matrices with low compression factor.

3 The PB-SpGEMM Algorithm
3.1 Overview of the PB-SpGEMM algorithm

Algorithm 1 provides a high-level description of an SpGEMM algorithm based on the ESC scheme. In the symbolic step, we estimate flops for the current multiplication and allocate memory for unmerged tuples in $\hat{\mathbf{C}}$. Then, multiplied tuples are formed and stored in $\hat{\mathbf{C}}$, which is then sorted and merged to form C.

Our algorithm follows the exact same principle, but uses outer products and propagation blocking for efficient bandwidth utilization. Fig. 4 explains the propagation blocking idea based on two 4×4 matrices and two bins. After we expand tuples, we partially order them in two bins, where the first bin stores rowid 0 and 1, and the second bin stores rowid 2 and 3. If these bins fit in L2 cache, sorting and merging can be performed efficiently in cache by different threads. In the expansion step, if we directly write a tuple
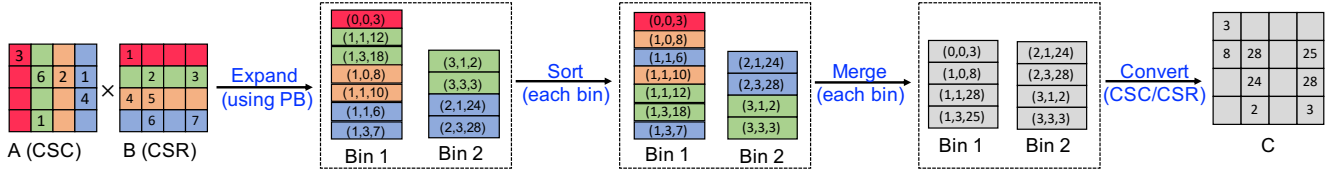
Figure 4: An example of PB-SpGEMM multiplying two 4×4 matrices with two bins, the first bin blocks propagation of tuples with rowid 0 and 1, the second bin blocks rowid 2 and 3. Sorting and merging are performed independently in each bin.
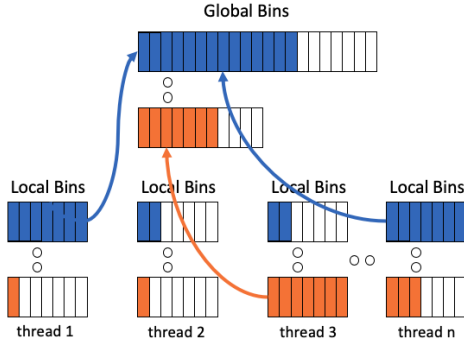

Figure 5: Using thread-private local bins to utilize cache lines. If *nbins* is the total number of global bins, each thread also creates *nbins* number of small local bins to store tuples as they are generated. When a local bin is full, tuples inside will be flushed to the corresponding global bin. This example is using two local bins per thread and two global bins.

to its designated global bin, we may not fully utilize the cache line. Hence, each thread also maintains small local bins that are filled in cache before flushing to global bins in memory. The use of local bins is illustrated in Fig. 5. Hence, our algorithm has several tunable parameters, including (a) nbins: number of global or local bins, and (b) Lbinwidth: the width of local bins. We experimentally select these parameters as will be discussed in the experimental section.

Algorithm 2 described the PB-SpGEMM algorithm. To facilitate outer product operations, input matrices A and B are passed as CSC and CSR formats, respectively. Here, we store C in CSR format, but it can be easily stored in CSC without any additional overhead. Thus, PB-SpGEM can generate output in a desired format needed by iterative applications. Similar to Algorithm 1, PB-SpGEMM has four phases: (a) Symbolic (line 1), (b) Expand (lines 5-18), (c) Sort (line 20), and (d) Compress (line 21). However, Algorithm 2 differs from previous ESC algorithms in two crucial ways: (1) we use outer product to stream data from input matrices, (2) we use propagation blocking to organize the expanded matrix into bins so that all phases of the algorithm saturate the memory bandwidth. We additionally perform a post-processing step (line 22) to convert the output to CSR format. Next, we discuss these phases in detail.

3.2 Symbolic Phase

In the symbolic phase, we estimate the memory requirement for $\hat{C}$, estimate the number of bins and allocate space for global bins (Gbin). Algorithm 3 describes the symbolic step. We compute flops for the current multiplication using an outer-product style computation. The loop in Line 2 accesses A column by column and B row by row. If there is a nonzero entry in $A(rowid, i)$, it must be

---

Algorithm 1: ESC-SpGEMM algorithm

Input: A , B
Output: C
1 $\hat{C} \leftarrow$ Symbolic (A, B)   ▷ Create space for $\hat{C}$;
2 $\hat{C} \leftarrow$ Expand (**A**, **B**)   ▷ Create unmerged tuples ;
3 Sort ($\hat{C}$) ▷ perform radix sort using (rowid, colid) as keys;
4 C ← Compress ($\hat{C}$)   ▷ merge duplicated tuples ;

Algorithm 2: PB-SpGEMM algorithm

Input: $A \in \mathbb{R}^{m \times k}$ in CSC, $B \in \mathbb{R}^{k \times n}$ in CSR, nbins
Output: $C \in \mathbb{R}^{m \times n}$ in CSR
1 GBin ← Symbolic(**A**, **B**) ▷ expanded tuples ($\hat{C}$) partitioned into *nbins* global bins;
2 Gbinwidth ← $\lfloor m/\text{nbins} \rfloor$ ▷ Global bin width ;
3 Lbinwidth ← 512 ▷ Local bin width, default set to 512 bytes ;
4 LBin ← Create a 3D array of size nthreads × nbins × Lbinwidth   ▷ LBin[T_i] is local bin, private for thread T_i, nthreads is the number of threads ;
5 **for** $i \leftarrow 1$ **to** k *in parallel* **do**
6     $T_i \leftarrow$ The thread executing the ith iteration ;
7   **for** *(rowid, i, aval)* $\in$ $A(:, i)$ **do**
8   **for** *(i, colid, bval)* $\in$ $B(i :,)$ **do**
9   binid ← $\lfloor$rowid / Gbinwidth$\rfloor$ ▷ the last bin holds excess tuples;
10   **if** *size(LBin[T_i][binid]) = Lbinwidth* **then**
11   MemCopy (GBin[binid], LBin[T_i][binid]) ;
12   LBin[T_i][binid] ← ϕ ;
13   mval ← aval × bval ;
14   Append (LBin[T_i][binid], (rowid, colid, mval));

15 **for** *all thread T_i in parallel* **do**
16   **for** *binid* ← 1 **to** *nbins* **do**
17   **if** *size(LBin[T_i][binid]) ≠ 0* **then**
18   MemCopy (GBin[binid], LBin[T_i][binid]) ;

19 **for** *binid* ← 1 **to** *nbins in parallel* **do**
20   GBin[binid] ← Sort (GBin[binid]) ▷ perform radix sort in the ith global bin using (rowid, colid) as keys;
21   GBin[binid] ← Compress (GBin[binid])   ▷ merge duplicated tuples by two-pointer method;
22 C ← ConvertCSR($GBin$);

---

multiplied by all nonzeros in the ith row of **B**. Hence, line 5 adds $nnz(B(i, :)) \times nnz(A(:, i))$ to the flops count. After we compute flops,

---

Algorithm 3: Symbolic Phase (Static Schedule)

Input: A in CSC, B in CSR
Output: Gbin, nbins

1 flops ← 0;
2 **for** $i \leftarrow 1$ **to** k *in parallel* **do**
3      $nnz(\mathbf{B}(i,:)) \leftarrow \mathbf{B}.rowptr[i+1]$ - $\mathbf{B}.rowptr[i]$;
4      $nnz(\mathbf{A}(:,i)) \leftarrow \mathbf{A}.colptr[i+1]$ - $\mathbf{A}.colptr[i]$;
5   flops $+ = nnz(\mathbf{B}(i,:)) \times nnz(\mathbf{A}(:,i))$

6 nbins ← flops / L2_CACHE_SIZE ;
7 Gbin ← MemAlloc(flops)   ▷ allocate shared array to store tuples, no initialization needed

---

we compute the number bins (line 6) so that each global bin fits in the L2 cache in the sorting and merging phase. We then allocate memory for global bins (line 7). Algorithm 3 needs only $O(n)$ time and attains higher memory bandwidth by streaming just row and column pointer arrays of **B** and **A**, respectively.

Note that Algorithm 3 is much simpler than symbolic steps used in column SpGEMM algorithms that need to estimate $nnz(\mathbf{C})$. An outer product algorithm can also be developed without a symbolic phase. For example, a linked-list can be used to dynamically append expanded tuples [27]. However, the dynamic memory allocations in parallel sections could result in poor performance.

3.3 Expand

Lines 5-18 in Algorithm 2 describes the expand phase of PB-SpGEMM. In the expand phase, a thread reads $\mathbf{A}(:,i)$ and $\mathbf{B}(i,:)$ and performs their outer product. The binid is computed by rowid of the multiplied tuple (rowid comes from the rowid in $\mathbf{A}(:,i)$). The newly-formed tuple is appended to its designated local bin in line 14. Once a local bin is full, it is flushed to the corresponding global bin (line 10-12). After the multiplication, there still could be some tuples left in the local bins because bins were not full at termination. Lines 15-18 send the partially full local bins to global bins.

With the local binning, we always write tuples in multiple of cache lines. We make sure the number of local bins and the size of local bins are small. Typically, we create 1024 bins and 512 bytes per local bin so that all local bins for a thread easily fit in the cache.

3.4 Sorting

After the multiplication and propagation blocking phase, the expanded matrix $\hat{\mathbf{C}}$ is stored in the COO format, partitioned into several bins. Then, we sort $\hat{\mathbf{C}}$ to bring tuples with the same (rowid, colid) pair close to one another for merging in the next phase. Here, each bin sorts tuples independently in parallel because bins do not share tuples with the same rowid.

The sorting algorithm uses the (rowid, colid) pairs as keys and the multiplied values as payloads. For this purpose, we use an in-place radix sort (similar to American flag sort [23]) that groups the keys by individual bytes sharing the same significant byte position. In the worst case, this in-place radix sort needs $\Theta(b)$ passes over the data, where b is the number of bytes needed to store a key. Hence, Radix sort can be faster than comparison-based sorting when keys are stored in fewer bytes.

Preparing integer keys for Radix sort. In our algorithm, we use 4-byte integers for row and column indices. We concatenate

the rowid and colid to form a combined 8-byte integer key for radix sort. With 8-byte keys, radix sort may need 8 passes over the data, which can incur significant data transfers. We can reduce the key space by using the fact that bins are already grouped by consecutive row indices. For example, if the input is a 1M × 1M matrix and we create 1K bins to block the propagation, the rowids of tuples within the same bin will be in adjacent 1k, then we only need 10 bits to represent the remainder rather than a full 32-bit integer, and we still have 32-10=22 bits to store colid. Furthermore, if we assume that matrices have at most 1M rows and columns, we can use 20 bits for colid and 20-10=10 bits for rowid (assuming 1K bins). Hence, in most practical cases, we can potentially squeeze keys into 4-byte integers, needing four passes over the data for sorting.

In-cache sorting. Since we sort $\hat{\mathbf{C}}$ containing flops tuples, four passes over the data directly from memory can be the performance bottleneck. Fortunately, bins can help in this case if tuples in a bin fit in L3 or L2 cache. In many practical problems, we can indeed fit a bin into L2 cache. For example, consider an ER matrix **A** with 1M rows, 1M columns and 4M nonzeros. When computing $\mathbf{A}^2$, we generate 16M tuples in expectation. If 1K bins are used, each bin will contain 16K tuples. If we use 4 byte keys (as described in the previous paragraph) and 8 byte payloads, we need 192KB memory to store all tuples in a bin. This can easily fit in L2 cache of most modern processors. Multiplications with high compression ratios and matrices with denser rows can create problems for some bins as tuples may not fit in L2 cache. In these cases, we either use more bins or create bins with variable ranges of rows. Hence, our algorithm reads a bin from memory and perform radix sort on data stored in cache. The sorted data can then be compressed while it is still in the cache. Overall, the sorting phase reads $(b * \text{flops})$ bytes.

3.5 Compression

After we sort each bin, tuples with the same (rowid, colid) pair are stored in adjacent locations. In the compression step, we sum numeric values from tuples with the same (rowid, colid). Similar to sorting, compression can be performed independently in each bin.

As tuples are already sorted within a bin, compression is done by scanning the tuples in the sorted order. We implement this using two in-place pointers, which only walk the array once. The first pointer (p_1) walks through the array, the second pointer (p_2) maintains the location to be merged. Every time when p_1 points to a new location it checks with p_2, if the keys of the two tuples are the same, simply add the numeric value of the first tuple to the second, if not, we move p_2 to the next location and copy the tuple2 there, keep doing this until the p_1 reach the end of the array.

Table 2 summarizes the computational complexity, bandwidth and latency costs, in-cache operations and parallelism schemes for all phases of PB-SpGEMM.

4 Experiment Setup

4.1 Software

We evaluate the performance of PB-SpGEMM against some state-of-the-art column SpGEMM algorithms, namely HeapSpGEMM, HashSpGEMM and HashVecSpGEMM. All of these algorithms have memory access patterns of a typical column SpGEMM algorithm discussed earlier.

Phase	Comp. Complexity	Bandwidth cost	Latency	In-cache operations	Parallelism
Expand	$O(\text{flops})$	reading $(b * (nnz(\mathbf{A}) + nnz(\mathbf{B})))$ bytes writing $(b * \text{flops})$ bytes	Negligible	manipulating local bins	cols of $\mathbf{A}$ and rows of $\mathbf{B}$ per thread
Sort	$O(\text{flops})$	reading $(b * \text{flops})$ bytes	Negligible	shuffling $(4 * b * \text{flops})$ bytes	bins per thread
Compress	$O(\text{flops})$	writing $(b * nnz(\mathbf{C}))$ bytes	Negligible	read/write $(b * \text{flops})$ bytes	bins per thread

Table 3: Computational complexity and data access patterns in different phases of PB-SpGEMM.

	Skylake-SP	POWER9
CPU Model	Intel Xeon Platinum 8160	IBM POWER9
Architecture	x86_64	ppc64le
#Sockets	2	2
#Cores/socket	24	20
Clock	2.1GHz	3.8GHz
L2 cache	1024KB/core	512KB/two cores
L3 cache	33792KB/socket	10240KB/two cores
Memory Size	250GB	1TB
Bandwidth	100GB/s	250GB/s

Table 4: The configuration of hardware used for evaluation

	Copy	Scale	Add	Triad
single socket	47.40	46.85	54.00	57.04
dual socket	97.73	87.43	107.00	108.42

**Table 5: Stream benchmark result of the evaluation platform**

- **HeapSpGEMM** uses heaps to merge columns [1]. To multiply two $n \times n$ ER matrices with d average nonzeros per column, HeapSpGEMM complexity is $O(nd^2 \log d)$, where the $\log d$ term comes from manipulating heaps. Each column of $\mathbf{C}$ can be formed in parallel using thread-private heaps.
- **HashSpGEMM** uses a hash table to merge columns[24, 25]. Its complexity is $O(nd^2)$ for ER matrices assuming that hash tables have limited collisions. Each column of $\mathbf{C}$ can be formed in parallel using thread-private hash tables.
- **HashVecSpGEMM** is a variant of hash algorithm, which utilizes vector registers for hash probing [24]. HashVecSpGEMM may preform better when the collision in the hash table is high.

Previous work [24] has shown that the optimized heap and hash algorithms largely outperform Intel MKL and Kokkos-kernel [12]. Considering this, we will not include those algorithms and software in our evaluation. All implementations are compiled by GCC-8.2.0, with flags "-fopenmp" "-O3" "-m64" "-march=native" enabled.

4.2 Hardware

In our experiments, we use a dual-socket Intel Skylake system and an IBM POWER9 system as described in Table 4. Since most of our experiments are conducted on the Skylake processor, we examine its memory system carefully using the STREAM benchmark [22]. Table 5 shows the sustainable memory bandwidth for Copy, Scale, Add and Triad benchmarks on single and dual sockets of the Skylake system. Hence, we expect that PB-SpGEMM should attain ~ 55 GB/s on a single socket and ~ 105 GB/s on two sockets of Skylake. While PB-SpGEMM indeed attains ~ 55 GB/s in every phase, its dual socket performance falls short of STREAM benchmark. Hence, we primarily focus the single-socket performance because
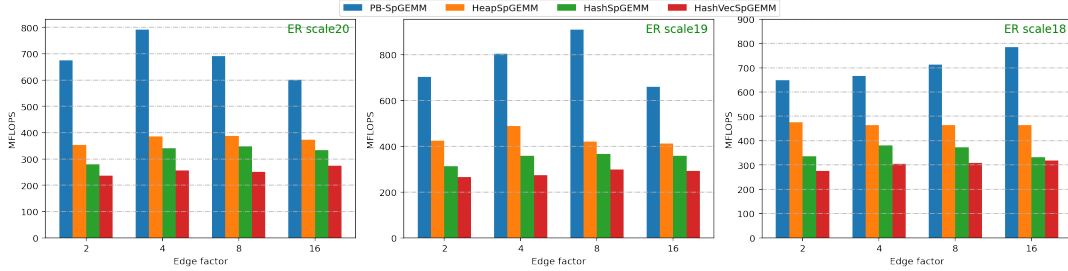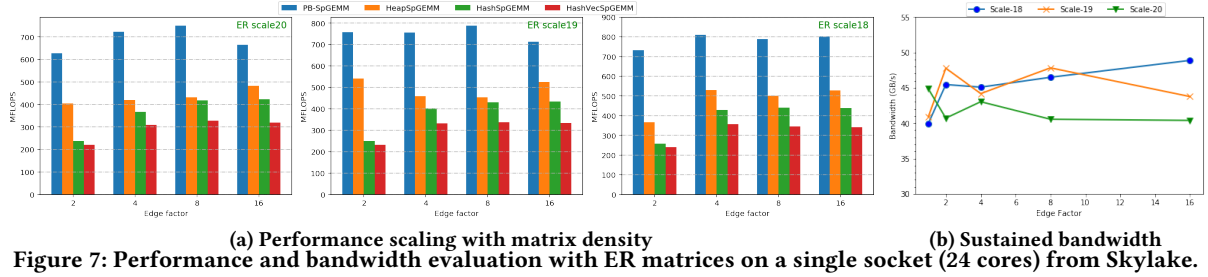
Graph	n(A)	nnz(A)	d(A)	flops	nnz(C)	cf
2cubes_sphere	101.5K	1.6M	16.23	27.5M	9.0M	3.06
amazon0505	410.2K	3.4M	8.18	31.9M	16.1M	1.98
cage12	130.2K	2.0M	15.61	34.6M	15.2M	2.14
cant	62.5K	4.0M	64.17	269.5M	17.4M	15.45
hood	220.5K	9.9M	44.87	562.0M	34.2M	16.41
m133_b3	200.2K	800.8K	4.00	3.2M	3.2M	1.01
majorbasis	160.0K	1.8M	10.94	19.2M	8.2M	2.33
mc2depi	525.8K	2.1M	3.99	8.4M	5.2M	1.6
offshore	259.8K	4.2M	16.33	71.3M	69.8M	3.05
patents_main	240.5K	560.9K	2.33	2.6M	2.3M	1.14
scircuit	171.0K	958.9K	5.61	8.7M	5.2M	1.66
web-Google	916.4K	5.1M	5.57	60.7M	29.7M	2.04

Table 6: A list of real-world graphs used in our evaluation

(a) Local bin width   (b) The number of bins
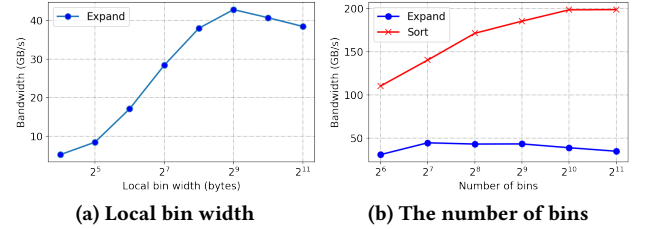
Figure 6: Impact of bin width and the number of bins, data based on ER matrices with scale 20 and edge factor 4.

memory bandwidth is harder to predict in Non-Uniform Memory Access (NUMA) domains. We also show some results with both sockets and explain dual socket performance in Sec. 5.5. When experimenting with a single socket, we set "OMP_PLACES" to "cores", "OMP_PROC_BIND" to "close", and restrict the memory allocation to NUMA node 0 by using "numactl –membind=0".

4.3 Dataset

We followed a recent paper [24] to select test matrices. We use the R-MAT recursive matrix generator to generate synthetic matrices. RMAT has four parameters a, b, c, and d. ER matrices are generated with a=b=c=d=0.25, and Graph-500 matrices are generated with a=0.57, b=c=0.19, d=0.05. Here, we refer to the latter graphs as RMAT. For random graphs, edge factor denotes the average number of nonzeros in a row or column. A matrix of scale k has 2^k rows and columns. We also use 12 real-world matrices from SuiteSparse Matrix Collection [11] as shown in Table 6. In our experiments, we randomly generate two random matrices and multiply them or multiply a real matrix with itself. These computations cover some representative applications such as Markov clustering [3] and triangle counting [2]. If an input matrix fit in cache, then column SpGEMM is expected to perform better than PB-SpGEMM. Hence, we did not explore multiplying rectangular matrices.

(a) Performance scaling with matrix density

(b) Sustained bandwidth

Figure 7: Performance and bandwidth evaluation with ER matrices on a single socket (24 cores) from Skylake.

Figure 8: Performance evaluation with ER matrices on a single socket (20 cores) from Power9.

5 Results

5.1 Select parameters of PB-SpGEMM

In the expand phase, local bins are used to improve data locality, the propagation of tuples will be blocked into small bins, and they are moved to global shared memory. A local bin should be large enough to have good utilization of a cache line so that it can be sent to global bins without wasting memory bandwidth. Furthermore, the total size of local bins per thread should be smaller than the L2 cache. Hence, number of bins and local bin width are two parameters upon which the performance of PB-SpGEMM depends. Fig. 6a shows our experiment where we vary the width of local bins to observe its impact on the performance of the expand phase. Smaller local bins do not utilize full cache lines, resulting in reduced sustained bandwidth. Based on this experiment, we used 512 bytes for every thread-private bin in all experiments in the paper.

The number of bins (*nbins*), on the other hand, is a tradeoff between the expand and sort phases. Recall that sorting is performed independently in each bin. Hence, increasing the number of bins ensures that tuples in a global bin fit in the cache. However, increasing the number of bins also reduces the average bin size, which may reduce the bandwidth utilization of the expand phase. Fig. 6b shows the impact of *nbins* over the expand and sorting phases. With more bins, radix sort can be entirely performed in L2 cache. Hence, in-cache sorting bandwidth can be as high as 200 GB/s. Hence, the number of bins is determined by L2/L3 cache size and total number of flop, and for most practical matrices, we use 1K or 2K bins.

5.2 Overall performance of PB-SpGEMM

At first we discuss the overall performance of PB-SpGEMM with respect to state-of-the-art column SpGEMM algorithms. Here, we consider both ER, RMAT, and real matrices as discussed in Sec 4.3.

Performance with ER random matrices. Fig. 7a compares the performance of PB-SpGEMM, HeapSpGEMM, HashSpGEMM, and HashVec-SpGEMM with ER matrices of various scales and edge factors on a single socket of the Skylake server. Here, we multiply

two ER matrices with the same scale and edge factor. For a given scale, the performance of PB-SpGEMM is stable (between 700 and 800 MFLOPS) and is better than column SpGEMM algorithms for all edge factors considered. This performance of PB-SpGEMM can be explained by Fig. 7b that reports the sustained bandwidth of PB-SpGEMM. We observed that the sustained bandwidth on a single socket is between 40 and 50 GB/s, which are close to the STREAM benchmark. Since ER matrices have $cf = 1$, the lower bound on AI is $\frac{1}{80}$ flops/byte according to Eq. 4. Hence, the performance of PB-SpGEMM should be at least $40 * \frac{1}{80} = 500$ MFLOPS when the sustained bandwidth is 40 GB/s and at least $50 * \frac{1}{80} = 625$ MFLOPS when the sustained bandwidth is 50 GB/s. Fig. 7a confirms that PB-SpGEMM's performance remains close to this lower bound estimate. By comparison, hash and heap algorithms have lower performance primarily because of their irregular memory accesses. However, as we increase the edge factor, the performance of column SpGEMM may increase because of increased utilization of cache lines.

Similar performance is observed on the Power9 system as shown in Fig. 8. As observed on Skylake, PB-SpGEMM performs better than column SpGEMM algorithms and its performance remains relatively stable for various matrix size and sparsity.

Performance with RMAT random matrices. Fig. 9a compares the performance of SpGEMM algorithms with RMAT matrices of various scales and edge factors on a single socket of the Skylake server. Here, we multiply two RMAT matrices with the same scale and edge factor. As with the ER matrices, the performance of PB-SpGEMM remains between 700 and 900 MFLOPS and is generally better than column SpGEMM algorithms. However, Fig. 9b shows that PB-SpGEMM attains lower sustained bandwidth between 30 and 40 GB/s. The reason behind this lower than STREAM bandwidth is the skewed degree distributions in RMAT matrices, resulted in variable-size bins. Load imbalance in different bins makes the expansion phase less bandwidth efficient.

Similar performance is observed on the Power9 system as shown in Fig. 10. As observed on Skylake, PB-SpGEMM performs better

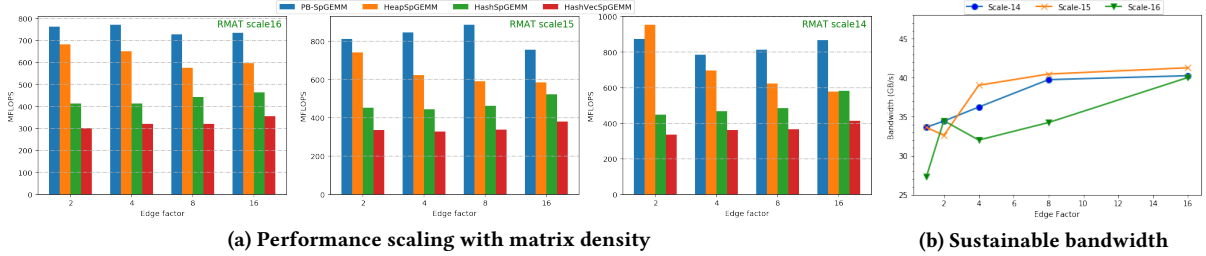(a) Performance scaling with matrix density

(b) Sustainable bandwidth

Figure 9: Performance and bandwidth evaluation with RMAT matrices on a single socket (24 cores) from Skylake.
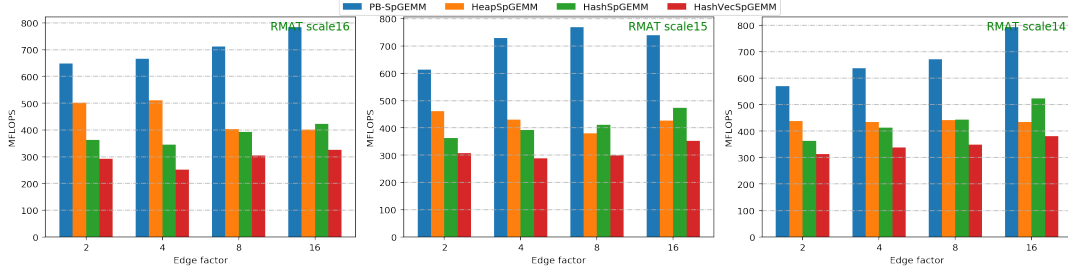

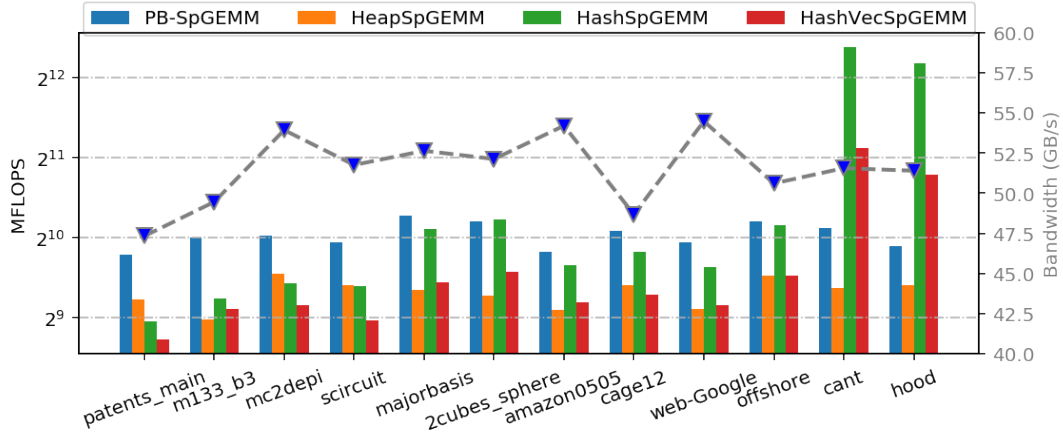Figure 10: Performance evaluation with RMAT matrices on a single socket (20 cores) from Power9.

Figure 11: Performance of SpGEMM when multiplying real matrices on a single socket of Skylake. From left to right, we sort matrices in the ascending order of the compression factor. Blue triangles denote the sustained bandwidth of PB-SpGEMM.

than column SpGEMM algorithms and its performance remains relatively stable for various matrix size and sparsity.

Performance with real matrices. Fig. 11 shows the performance of SpGEMM algorithms when squaring real matrices on a single socket of the Skylake server. As before, the sustained bandwidth of PB-SpGEMM is between 47 and 55 GB/s and its performance is relatively stable. In Fig. 11, we sort matrices in the ascending order of the compression factor (from left to right). PB-SpGEMM is generally faster than its peers except for the last two matrices (cant and hood) that have very high compression factors.

5.3 Scalability

Fig. 12 shows the strong scaling of SpGEMM algorithms from 1 to 24 threads within a socket of the Skylake processor. We observe that PB-SpGEMM runs faster than column SpGEMM on all concurrencies. All SpGEMM algorithms scale well within a socket.

On 24 cores, PB-SpGEMM attains about 16× speedup for ER matrices and 10× speedup for RMAT matrices. For RMAT matrices, PB-SpGEMM does not scale well on high thread counts because of the load imbalance caused by highly skewed nonzero distributions.

To improve the load balance, we used variable length bins tailored for irregular matrices. Consider a scale-free matrix which follows a strong inverse power law, if we assign the same number of rows to bins, the first one or two bins will be holding most flops. To solve this, we first perform a prefix sum counting the number of flops by row, which enables us to assign variable rows to bins while keeping the number of flops evenly distributed across bins. Our implementation used a simple lookup table for row-bin inference, it came with a cost, so we observed relatively lower effective bandwidth as per Fig. 9(b), but it did give us reasonably good performance advantage over the others.
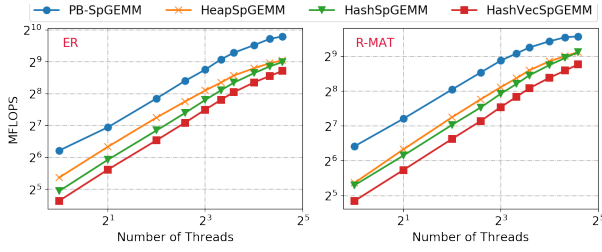
Figure 12: Scalability on ER (left) and R-MAT matrices (right). Both are scale 16 matrices with edge factor of 16.
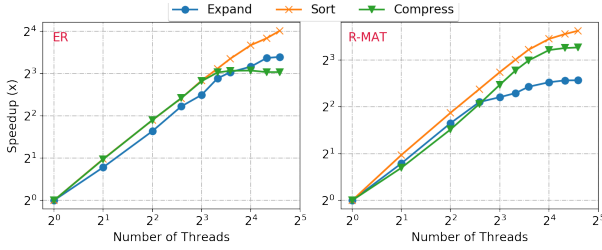

Figure 13: Scalability breakdown on ER (left) and R-MAT matrices (right). Both are scale 16 matrices with edge factor 16.

5.4 Breakdown of performance

Generally, expand and sort consume a major fraction of PB-SpGEMM's execution time. For ER matrices, expand takes 50%, sort takes 30% and compress takes 15% of the total runtime. For RMAT, the ratio is as follows: expand: 40%, sort: 40%, compress: 15%. RMAT matrices typically need more bins and variable load per bin, which require more time in sorting. Fig. 13 shows the scalability of different steps of PB-SpGEMM. In-cache sorting scales linearly as expected. However, the expand step stops scaling linearly when the memory bandwidth is nearly saturated.

5.5 Dual-socket Performance

Thus far, we have only considered the performance of SpGEMM algorithms on a single socket of Skylake and Power9 processors. Fig. 14 shows the performance of SpGEMM algorithms using both sockets of the Skylake processor. PB-SpGEMM still runs faster for ER matrices, but runs slightly slower than heap algorithm for RMAT matrices. This lower-than-expected performance of PB-SpGEMM on NUMA systems is due to inter-socket communication contentions. If a bin is allocated on socket-1 in the expand phase and sorted by a thread from socket-2, the performance of PB-SpGEMM depends on cross-socket memory bandwidth. We checked cross-socket memory bandwidth empirically by placing data on a socket and accessing it from the other socket in a STREAM copy kernel. Table 7 shows the local and remote access bandwidth and latency. Memory latency was measured by Intel Memory Latency Checker. We observe that cross-socket access is much slower than local access. Since PB-SpGEMM's performance relies on saturating the memory bandwidth, it is affected by the cross-socket bandwidth. Note that column SpGEMM algorithms are not significantly affected by the cross-socket bandwidth because they generate one column at a time, where the active column usually stays in cache.

In the Master's thesis [16] of the first author, we improved the dual socket performance by partitioning $\mathbf{A}$ into two $\frac{n}{2} \times n$ matrices and multiply each part with $\mathbf{B}$ independently in two sockets.
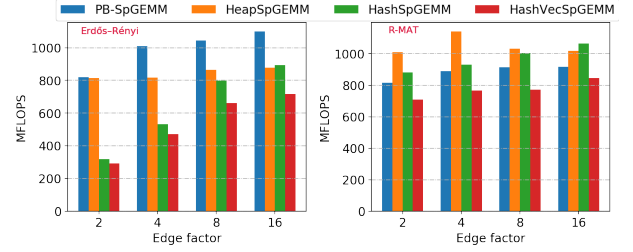

Figure 14: Multi-socket performance of SpGEMM algorithms on the Skylake system.

	NUMA socket 0	NUMA socket 1
NUMA socket 0	50.26GB/s and 88.1ns	33.36GB/s and 147.4ns
NUMA socket 1	34.06GB/s and 146.7ns	50.12GB/s and 88.3ns

Table 7: NUMA local and cross-socket memory bandwidth and latency on Skylake

This partitioned PB-SpGEMM partially mitigates the cross-socket bandwidth problem, but it does not perform uniformly well for all matrices due to the additional cost of reading $\mathbf{B}$ more than once.

6 Conclusions

With the rise of sparse and irregular data, SpGEMM has emerged as an important operation in many scientific domains. Over the past decade, the state-of-the-art of parallel SpGEMM algorithms has progressed significantly. However, understanding the performance of SpGEMM algorithms remains a challenge without an established performance model. Relying on the fact that SpGEMM is a bandwidth-bound operation, we used the Roofline model to develop bounds for SpGEMM algorithms based on column-by-column merging and the expand-sort-compress strategy. We conclude the paper with the following key findings:

(1) We can estimate the arithmetic intensity (AI) of an SpGEMM algorithm based on the compression factor of the multiplication and number of bytes needed to store each nonzero.
(2) The attainable performance of an algorithm is $(AI * \beta)$, where β is the memory bandwidth. This peak performance can only be attained if the algorithm saturates the bandwidth. We showed that existing column SpGEMM algorithms do not attain peak performance according to the Roofline model because of irregular data accesses and underutilization of cache lines.
(3) We develop a new algorithm based on outer product of matrices. This algorithm called PB-SpGEMM uses propagation blocking to group multiplied tuples into bins and then sort and merge tuples independently in each bin.
(4) PB-SpGEMM approximately saturates the memory bandwidth in all of its three phases and attains performance as predicted by the Roofline model.
(5) On a single socket, PB-SpGEMM performs better than the best column SpGEMM algorithms for multiplications with compression factors less than four.
(6) For multiplications with compression factors greater than four, HashSpGEMM is the best performer.

References

[1] Ariful Azad, Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 38(6):C624–C651, 2016.

[2] Ariful Azad, Aydın Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *IPDPSW*, 2015.

[3] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. HipMCL: A high-performance parallel implementation of the markov clustering algorithm for large-scale networks. *Nucleic acids research*, 2006.

[4] Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. Communication optimal parallel multiplication of sparse random matrices. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 222–231, 2013.

[5] Grey Ballard, Christopher Siefert, and Jonathan Hu. Reducing communication costs for sparse matrix multiplication within algebraic multigrid. *SIAM Journal on Scientific Computing*, 38(3):C203–C231, 2016.

[6] Scott Beamer, Krste Asanović, and David Patterson. Reducing pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 820–831. IEEE, 2017.

[7] Nathan Bell, Steven Dalton, and Luke N Olson. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. *SIAM Journal on Scientific Computing*, 34(4):C123–C152, 2012.

[8] Aydin Buluc and John R Gilbert. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11. IEEE, 2008.

[9] Aydin Buluç and John Gilbert. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, 04 2008.

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

[11] Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[12] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *IPDPSW*, pages 693–702. IEEE, 2017.

[13] Xu Feng, Yuyang Xie, Mingye Song, Wenjian Yu, and Jie Tang. Fast randomized pca for sparse data. In *ACML*, 2018.

[14] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[15] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. High performance graph algorithms from parallel sparse matrices. In *PARA*, pages 260–269, 2007.

[16] Zhixiang Gu. Bandwidth-optimized parallel SpGEMM algorithms using propagation blocking. Master's thesis, Indiana University, 2019.

[17] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM TOMS*, 4(3):250–269, 1978.

[18] Yu Jin and Joseph JáJá. A high performance implementation of spectral clustering on cpu-gpu platforms. *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 825–834, 2016.

[19] Haim Kaplan, Micha Sharir, and Elad Verbin. Colored intersection searching via sparse rectangular matrix multiplication. In *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*, SCG '06, pages 52–60. ACM, 2006.

[20] Daniel Langr and Pavel Tvrdik. Evaluation criteria for sparse matrix storage formats. *IEEE Transactions on parallel and distributed systems*, 27(2):428–440, 2015.

[21] Junhong Liu, Xin He, Weifeng Liu, and Guangming Tan. Register-aware optimizations for parallel sparse matrix–matrix multiplication. *International Journal of Parallel Programming*, 47(3):403–417, 2019.

[22] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, 1991-2007.

[23] Peter M McIlroy, Keith Bostic, and M Douglas McIlroy. Engineering radix sort. *Computing systems*, 6(1):5–27, 1993.

[24] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Computing*, 90:102545, 2019.

[25] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA Pascal GPU. In *ICPP*, pages 101–110. IEEE, 2017.

[26] Muhammet Mustafa Ozdal. Improving efficiency of parallel vertex-centric algorithms for irregular graphs. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2265–2282, 2019.

[27] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[28] 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. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In *ISC*, pages 48–57. Springer, 2015.

[29] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[30] Raphael Yuster and Uri Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2004.