BANDWIDTH-OPTIMIZED PARALLEL SPGEMM ALGORITHMS USING PROPAGATION BLOCKING

Zhixiang Gu

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Master of Science

in the Department of Intelligent Systems Engineering,

Indiana University

December 2019

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Master of Science.

Committee

Ariful Azad, Ph.D.

Judy Fox, Ph.D.

Minje Kim, Ph.D.

Date: 12/15/2019

ACKNOWLEDGEMENTS

My work and this thesis owe their existence to the help, support and inspiration of many many people.

I would like to thank Prof. Ariful Azad for proposing this project when I approached him asking for a research opportunity. I am grateful to him for the patience, support, motivation, encouragement, generosity and guidance over my master's study.

I want to thank my groupmates - Taufique, Khaledur, and Nicholas, the discussions with them, their technical suggestions, and encouragement are truly valuable.

I'd like to express my deepest gratitude to my family. I remember the first time I said I want to pause my promising engineer career, challenge myself and pursue a higher degree abroad, they supported me without a single doubt. My sincere appreciation also goes to my girlfriend Xuejing, thanks for her understanding, sacrifice and help me through the difficult times. Without all their love and encouragement, this amazing American journey wouldn't even get started.

I am blessed to have many friends at Indiana University: Bo, Changdong, Fanbo, Haici, Hongwei, Jassi, Sahil, Shujun, Qianqian, Weizhe, Yafei, they make my time here quite memorable.

Zhixiang Gu

BANDWIDTH-OPTIMIZED PARALLEL SPGEMM ALGORITHMS USING PROPAGATION BLOCKING

Sparse general matrix-matrix multiplication (SpGEMM) is a key kernel in many graph algorithms, machine learning tasks, and high-performance applications. As it usually is the dominating cost of computation, the performance of SpGEMM is critical. Thus, we've seen many new SpGEMM algorithms and optimization strategies proposed over the years, but few are targeting on bandwidth efficiency, which we believe is the key to deliver higher performance.

In this work, we discuss the advantages and limitations of four different Matrix-Matrix Multiplication formation. Based on the outer-product, we propose a bandwidth-optimized parallel SpGEMM algorithm using propagation blocking. Experiments show that OuterSpGEMM achieves over 80% memory bandwidth in all major phases and significant speedups over state-of-the-art competitors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 Background

Sparse general matrix-matrix multiplication (SpGEMM) is a key kernel in many data analytics applications, and it dominates the overall cost of many iterative methods. For example, SpGEMM is a key kernel in the computation of betweenness centrality [1], clustering coefficients, triangle counting [2], multi-source breadth-first searching [3], colored intersection searching [4], and cycle detection [5]. In scientific computing, SpGEMM is used in algebraic multigrid [6] and linear solvers. Many machine learning tasks like Dimensionality Reduction (e.g., NMF, CX, PCA [7]), Clustering (e.g., Spectral Clustering [8], Markov Clustering (MCL) [9]), etc., rely on an efficient SpGEMM algorithm as well. Additionally, SpGEMM algorithms are applied to evaluate the chained product of sparse Jacobians [10] and optimize join operations on modern relational databases [11].

The fact of the extensive usage and high-performance requirement contributes to the popularity of studying SpGEMM algorithms. The sequential SpGEMM algorithm was first introduced by Gustavson [12] in 1978, after that, considerable effort has been continuously devoted to obtaining higher performance, better scalability and memory/power efficiency. On shared-memory platform, researchers have developed heap-, hash-, and SPA-based SpGEMM algorithms, On distributed system, we have Sparse 1D Algorithm, Sparse Cannon, Sparse SUMMA [13]. Table 1.1 summarizes some of the most widely-used SpGEMM algorithms.

algorithm/software	year	platform	description
Gustavson	1978	CPU - sequential	row-wise
Matlab algorithm [14]	1992	CPU	SPA accumulator
intel MKL	2007	CPU - multicore	
cuSPARSE [15]	2010	GPU	
Sparse SUMMA	2012	distributed	2D matrix partition
Kokkos-Kernels [16]	2014	CPU - manycore	hash-map accumulator
Split-3D-SpGEMM [17]	2015	distributed	3D matrix partition
HashSpGEMM [18]	2016	CPU/GPU	hash-table accumulator
HeapSpGEMM	2018	CPU - multicore	heap accumulator

Table 1.1: A summary of some widely-used sequential and parallel SpGEMM algorithms

1.2 Performance Analysis and Optimization Strategy

Performance of graph algorithms can be extremely non-intuitive, especially considering there has been a large diversity in computer architectures. When designing a graph algorithm, a simple model that offers performance guidelines could be valuable.

The roofline model [19] is just such a model, it combines specific architecture specification and algorithm's operation density into a simple performance chart. From a roofline model, we can easily tell the current limitation of a given kernel.

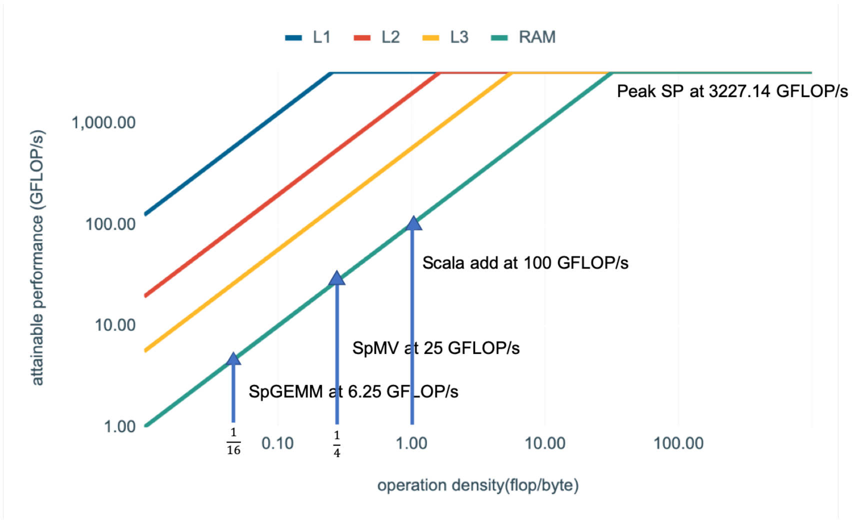


Figure 1.1: The roofline model on a Intel Skylake system

Figure 1.1 is a roofline model plot based on a Intel Skylake system(detailed specifics of the system is described in the Table 4.1). Given a kernel, we draw a vertical line based on operational density(i.e. the ratio of flops executed per byte read), if the line hits the flat green roof, it means the kernel is computation bound, if the line hits the slanted green roof, the kernel is memory bound.

From the roofline performance model we know SpGEMM operations tend to have slowest, in fact, the performance could get even worse due to many practical factors. First, sparse matrix input is stored in special data structures(e.g. CSR, CSC), SpGEMM algorithms need auxiliary variables to access non-zeros and accumulator to store intermediate results, this not only introduces additional data moving but also complicates the cache inference. Second, the number of non-zeros are unknown beforehand, a precise memory allocation before multiplication without overhead is impossible. Third, the non-zero pattern or sparsity structure is unknown in advance, a naive parallelization will most likely be load imbalanced and deliver poor performance. Figure 1.2 shows the performance gap between current state-of-the-art implementation and attainable bound calculated from the roofline model. Clearly, improving the operating density is the top priority.

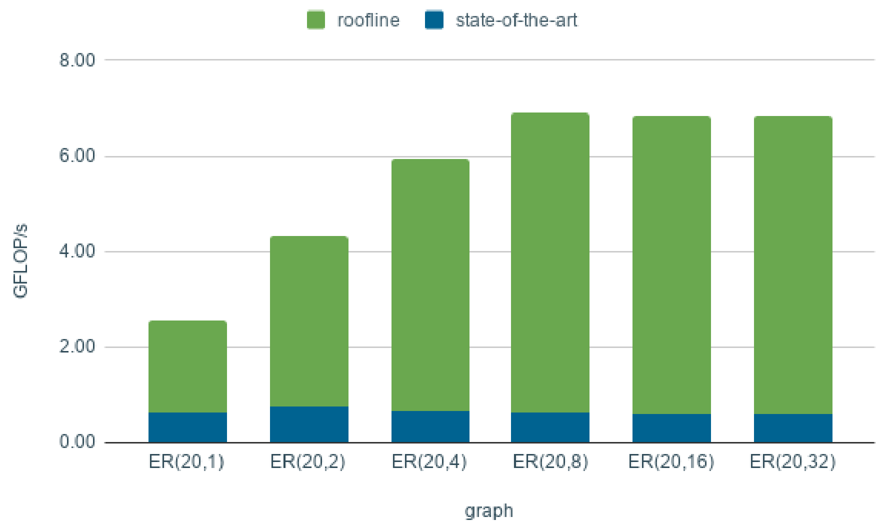


Figure 1.2: A comparison between attainable performance from the roofline model and the practical number from best implementation on Erdős-Rényi matrices

In addition, Figure 1.1 shows that the ceiling of cache hierarchy is much higher than RAM's, which points out a potential optimization strategy, if at some point we have to access duplicated data, SpGEMM could still be efficient if this is done in cache.

1.3 Outline

The rest of thesis is structured as follows,

Chapter 2 introduces four different formations of Matrix Matrix Multiplication. In each formation, we will describe its advantages and limitations. In this work we choose outer-product formation for its promising efficiency in memory bandwidth.

In chapter 3 describes the proposed propagation blocking based outer-product algorithm, we start with a overview, then expound the implementation of symbolic, multiplication, sort and merge, we will discuss the load balance strategy and parameter tuning in the end of this chapter.

Chapter 4 shows the evaluation. We first bring up the environment setup and matrices used in the Evaluation, specifically we will talk about NUMA effect. In the experiment, the proposed OuterSpGEMM algorithm demonstrates high efficiency in memory bandwidth and deliveries good performance, it outperforms the Hash-, Heap- SpGEMM algorithms in most cases.

The end of this thesis is a chapter of discussion, where we point out in some extreme circumstances that our algorithms might not excel in performance. In addition, we will discuss a future work, which partitions the left input matrix and distributes the multiplication to multi sockets, this could completely avoid NUMA effect and deliver even higher performance.

CHAPTER 2
ALGORITHMIC VARIANTS FOR MULTIPLYING SPARSE MATRICES

Given two matrices A $\in \mathbb{R}^{m\times k}$ and B $\in \mathbb{R}^{k\times n}$, where $k, m, n \in \mathbb{N}$, Matrix-Matrix multiplication is the operation that computes

$$C = AB, \text{where } C \in \mathbb{R}^{m\times n}. \tag{2.1}$$

For simplicity, we always consider $n \times n$ matrices unless otherwise specified. In this thesis, we use Matlab like indexing. $A(i, j)$ denotes the matrix entry at ith row and jth column. $A(i, :)$ denotes ith row of the matrix and $A(:, j)$ denotes jth column of the matrix.

Depending on the formation, Matrix-Matrix multiplication can be categorized into several methods, inner-product multiplication, outer-product multiplication, column-wise, and row-wise multiplication. SpGEMM shares the same formation but takes two sparse matrices as input. Although these formations all generate the same result, their performance can vary a lot due to different data access patterns, unused floating point operations and their memory requirements. Hence, different optimization strategies need to be applied to cope with the data access and the underlying sparse matrix representations.

2.1 The inner-product formation

Algorithm 1 describes an inner-product based SpGEMM algorithm. Here, each element can be independently computed. Therefore, inner product multiplication can achieve good performance on large dense matrices. In addition, we can apply cache blocking, matrix partitioning, and tiling to better improve the data locality. However, computing the inner product on sparse matrices is not work efficient because of its $O(n^2)$ complexity (many iterations are wasted just to compute zero entries). Hence, inner-product is almost never

Algorithm 1: Inner product formulation of matrix multiplication

```
for $i \leftarrow 1$ to $M$ do
    for $j \leftarrow 1$ to $N$ do
        $C(i,j) = A(i,:) \times B(:,j)$ ;
```

Algorithm 2: Column-by-column formulation of matrix multiplication

```
for $j \leftarrow 1$ to $N$ do
    for $k \leftarrow$ where $B(k,j) \neq 0$ do
        $C(:,j) \leftarrow C(:,j) + A(:,k) \times B(k,j)$ ;
```

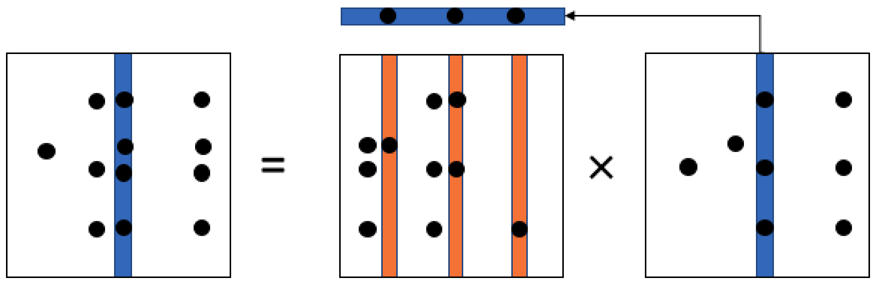


Figure 2.1: Illustration of column-by-column formation.

used in SpGEMM algorithms.

2.2 The column-by-column formation

The column-wise multiplication forms one column of the output matrix C at a time as shown in Algorithm 2. Figure 2.1 illustrates the formation of an output column (shown in blue). Paralleling the column-wise multiplication is relatively easy since threads can construct columns independently without any atomic operations. In addition, the memory requirement of this formation is low since each output column is generated by merging a subset of columns of A. Hence, this formation does not need to store duplicated (that is unmerged) output entries. Likewise, row-wise formation shares the same advantages. In fact, most of parallel SpGEMM algorithms are based on either column-wise or row-wise formulation.

However, column-wise multiplication has some significant limitations. Consider a random matrix where nonzero entris in a row or columns are uniformly distributed (also known as the Erdős-Rényi model [20]). let A be an $n \times n$ matrix with d average nonzeros per column. Hence, A has nd non-zeros in total. If we multiply A with itself to compute A^2, we need nd^2 floating point operations (flops) in expectation. If we perform this computation using the column-wise formation, we may observe two performance bottlenecks in modern multicore processors.

The first issue is the memory latency overehead in accessing columns of A. Notice that we have to read d columns of A to generate a column of C. As a result, a total of nd columns of A will be read. In other words, this algorithm reads matrix A d times. Inefficient cache-line usage is another concern. Since the input matrix is sparse, when we multiply the column i of B with corresponding columns of A, those columns of A are rarely adjacent. On the other hand, modern operating systems transfer data in a cache-line (in many cases, 64 bytes). The goal of doing this is to improve RAM efficiency since RAM works much better if it can transport more data in a row without a new CAS signal. But in our case, the size of a column can be smaller than a cache-line. In that case, every time we read a cache-line size of data, we can only use a portion of the fetched cache line. As a result, memory bandwidth may remain underutilized.

2.3 The outer-product formation

In the outer-product formation, the product is accumulated as the summation of n rank-one matrices, as shown in Figure 2.2. Here, each rank-one sub matrix is the outer product of column k of A and row k of B, which is summarized in Algorithm 3. An outer-product SpGEMM usually follows ESC (expansion-sorting-contraction) [21] formula, which first explicitly expands all temporary products to row-grouped global arrays, then iteratively sorts and merges them to yield the final matrix.

Outer-product formation has two key limitations. First, the memory requirement is high

Algorithm 3: Outer product formulation of matrix multiplication

```
$C \leftarrow 0$;
for $k \leftarrow 1$ to $K$ do
    $C \leftarrow C + A(:,k) \times B(k,:)$ ;
```

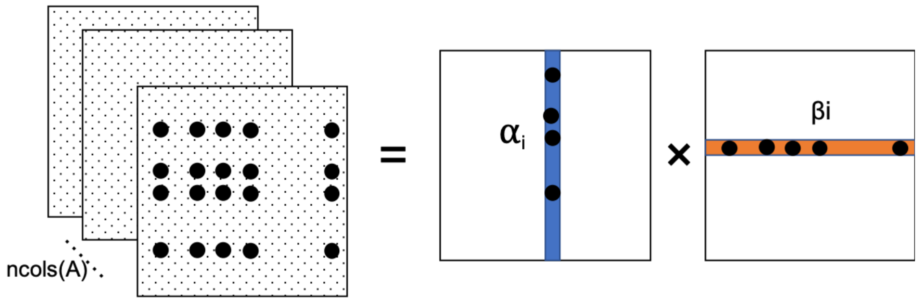


Figure 2.2: Illustration of outer-product formation.

since outer-product needs to store all temporary unmerged product. That means, the total memory requirement can be $O(nd^2)$ which is proportional to the floating point operations in SpGEMM. Second, writing the expanded results back to memory is notoriously hard, since each rank-one matrix is hyper sparse (nnz $<n$) [22]. Hence, directly writing rank-1 matrices may generate random memory traffic. Because of this problem, a naive outer-product SpGEMM can only achieve less than 15% memory bandwidth utilization.

Nonetheless, the benefit of perfect theoretical full cache-line utilization and no redundant memory access to non-zeros indicates the outer-product formation still could be very efficient especially in terms of memory bandwidth utilization. This thesis extensively explores memory bandwidth optimization aspects of outer-product based SpGEMM.

CHAPTER 3
TECHNICAL APPROACH

3.1 Implementation

Outer product iteratively accesses a column from left matrix A and a row from right matrix B, we store A in compressed Sparse Column (CSC) format and B in Compressed Sparse Row (CSR) format, which provides constant-time access, and depends on the requirement, the output can be either CSR or CSC.

Algorithm 5: OuterSpGEMM algorithm

```
  Input: A in CSC, B in CSR
  Output: C in CSR
1 $C \leftarrow 0$;
2 Symbolic (A, B);
3 for $k \leftarrow 1$ to $K$ in parallel do
4     $C \leftarrow$ PB ($C$ + Expand ($A(:, k)$, $B(k, :)$))    ▷ apply propagation blocking
          moving tuples to global bins;
5 for $i \leftarrow 1$ to $nbins$ in parallel do
6     $C_i \leftarrow$ Sort ($C_i$)          ▷ perform radix sort on the key of rowid-colid;
7     $C_i \leftarrow$ Contract ($C_i$)     ▷ merge duplicated tuples by two-pointer method;
8 Convert (C, format=CSR);
```

As shown in the above pseudocode, our OuterSpGEMM extends the naive ESC. Before multiplication, we have a symbolic phase to count the number of non-zeros and sense the non-zero pattern, which will later be used in memory allocation and load balancing. Then, in the numeric phase, we do multiplication but use the propagation blocking to move generated tuples in a batch and group them by bins. We carefully select the number of bins and bin width so that the following sorting and contraction operation can be done in the L2 cache to attain the best possible performance.

3.1.1 Symbolic Phase

To achieve best performance, we need to pre-allocate memory space before doing any computation in order , but unlike dense matrix-matrix multiplication, the pattern and size of the SpGEMM output is unknown beforehand. For outer-product formation, the prior knowledge here is the number of floating-point of operations(also can be considered as the number of non-zero entries generated in the multiplication). An easy way out is using a linked-list [23] or a vector to allocates memory space during run-time. However, the memory space allocated in this way may not be consecutive, thus resulting in poor reading and writing performance.

In this work we use a symbolic method to count flops by simply walking through the row and column pointers of the input matrices. In the CSR/CSC format, the difference of two adjacent pointers is the number of non-zeros for the input row or column, so the product of the two differences is going to be the number of flops of one sub-matrix that contributes to the output of row$_i$.

Algorithm 6: Symbolic Phase

```
Input: A in CSC, B in CSR
Output: flops
$flops \leftarrow$ 0;
for $i \leftarrow firstcolumn$ to $lastcolumn$ in parallel do
    $nnzright \leftarrow B.rowptr[i+1]$ - $B.rowptr[i]$;
    for $j \leftarrow A.colptr[i]$ to $A.colptr[i+1]$ do
        $rowid \leftarrow A.rowids[i]$;
        $flops[rowid]$ += $nnzright$;
```

Doing a symbolic step before multiplication can also help resolve load imbalance issue, we will discuss it in Section 3.2.

3.1.2 Expansion

Multiplication is known as the most tricky part for outer product SpGEMM due to extremely low writing locality, propagation blocking [24] is used in this paper to address this issue. To better describe this algorithm, we separate propagation blocking into binning and accumulating.

We read i_{th} column from left matrix A and a corresponding j_{th} row from right Matrix B to perform outer-product multiplication, but instead of directly writing the triple to a global accumulator according to the row id, each thread will only push triples to its own local bins, which collect triples with a continuous range of rows assigned in the symbolic phase(See Figure 3.1). With binning process applied, writing tuples is always consecutive and we make sure the number of bins and the size of bins is small, usually 1024 bins and 512 bytes, so all the local bins together can easily fit in the cache.

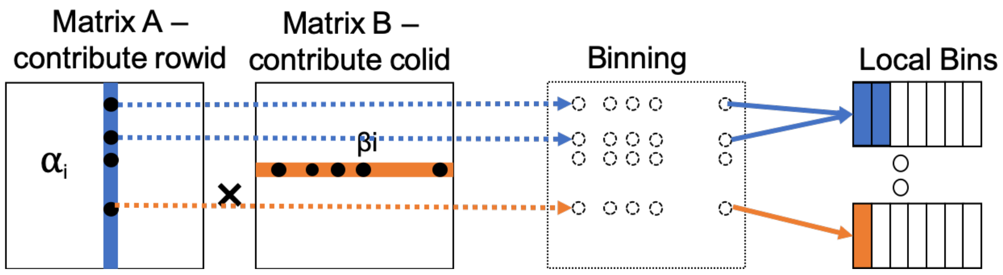


Figure 3.1: Illustration of propagation blocking step in the expansion phase

Since our bin are very small, they can be easily filled after a couple of iterations, accumulator will be called to move the triples to a corresponding global bin(See Figure 3.2). Like local bins, global bins also collect triples within the same row range that assigned in the symbolic phase, except they are shared by all threads. When accumulating, reading can enjoy cache speed, writing is also expected to hit the maximum memory bandwidth speed since data access is all contiguous.

It's worth noting that after the multiplication, there still could be some triples left in the local bins, they were not moved before due to the host bin unfull, in order to get the correct result we need to flush them out.

In addition, our propagation blocking also lowers the latency. Before our algorithm, one of the major concerns of the outer product is the locking (synchronization) cost introduced by the accumulating operation. When different threads accumulate non-zeros of the same

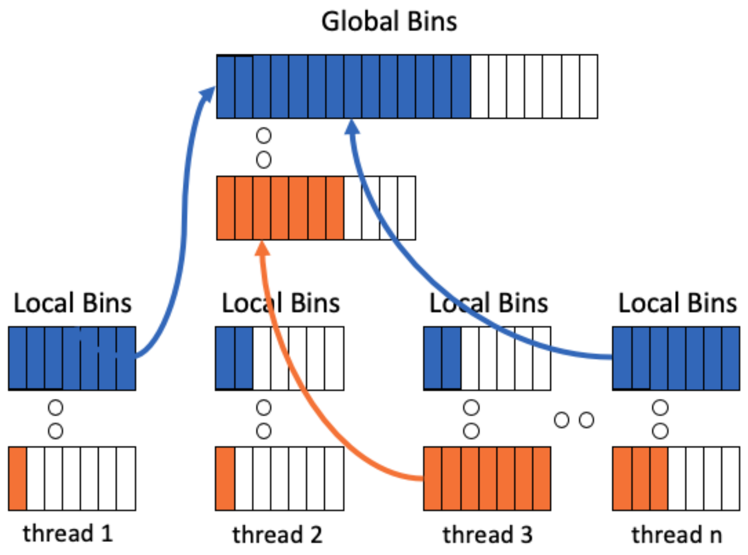


Figure 3.2: Illustration of accumulating step in the expansion phase

vertices, synchronization is unavoidable. But with propagation blocking we use bins to move triples as a batch, this cost can be drastically lowered. The number of bins and the bin width are two important parameters that can affect the expansion and sort performance, in the following sections, we will show how to select them.

3.1.3 Sorting

OuterSpGEMM uses radix sort to sort unmerged tuples.

Radix sort is an integer sorting algorithm, it groups the keys of data by radix that share the same significant position(key) and value (numeric value). Figure 3.3 demonstrates an example of using LSD radix sort to sort a simple integer array. In base 10 (the decimal system), this array needs a total of 3 passes to get sorted.

The main motivation to use radix sort is better performance. Radix sort is not comparison-based, which means it will not be restricted by the $\Omega(n \log n)$ lower bound, in fact, given proper optimization, it can perform at linear time level. Here are two optimizations we've done in this work:

First, the performance of radix sort is strongly related to the number of bits in the key.

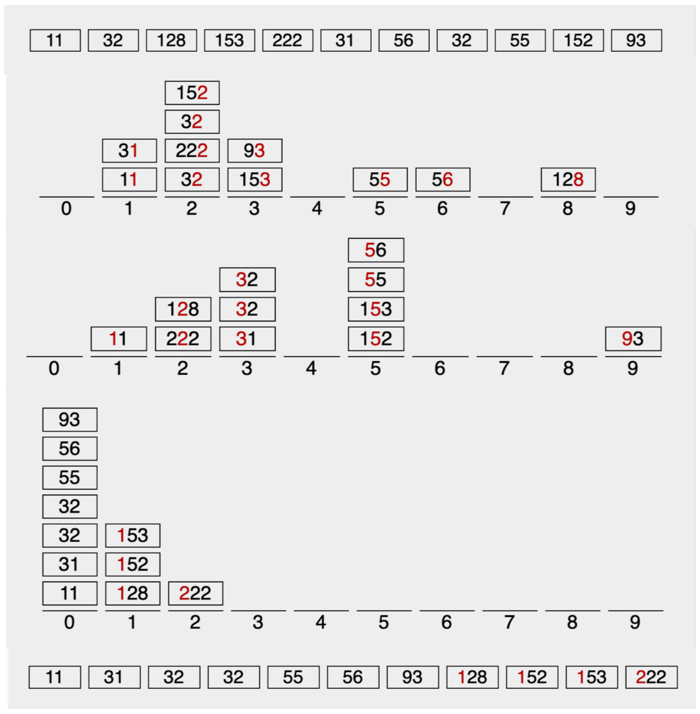


Figure 3.3: Using radix sort to sort a small integer array

In our case, we are using tuples to represent the generated flops, but radix sort can only sort on integer keys, so we concat the rowid and column id, and then the sorting key can be as simple as a 64-bit integer. However, a 64-bit integer key needs 8 passes to get sorted(8-bit per pass). Since in we are storing tuples in bins, we can use the modular index in the bin and concat it with column id, potentially squeeze them into a 32-bit integer, doing so we are able to save significant amount of data transfer.

Second, radix sort moves data between the source array and buffer array for multiple times, the performance will improve significantly if the dataset can be fit into cache. Modern computer cache hierarchy can be very complicated, in Intel Skylake system, L3 cache is usually larger, but it's non-inclusive and shared by all the cores in the socket, which makes the performance non-predictable. In order to get a sustainable performance, we choose to use L2 cache. Let's $flops$ be the number of flops generated in the multiplication, and we use n bins to block them, we need to make sure $flops/n$ is smaller than the size of one L2 cache. More detailed discussion will be shown in next sections.

In our preliminary evaluation, we compare the performance against some most popular algorithms like the std::sort, vergesort, timsort, qsort, pdqsort, the radix sort outperforms all of them.

3.1.4 Contraction

During contraction phase we merge duplicated tuples, duplicated tuples are tuples that share the same key(i.e. (rowid, colid) pair), in which case we merge them to one tuple but summing the numeric value.

In this work, we use a two-pointer method, which only walks the array once. The first pointer (p_1) walks through the array, the second pointer (p_2) maintain 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 triples are the same, simply add the numeric value of the first triple to the second, if not, we move p_2 to the next location and copy the triple2 there, keep doing this until the p_1 reach the end of the array.

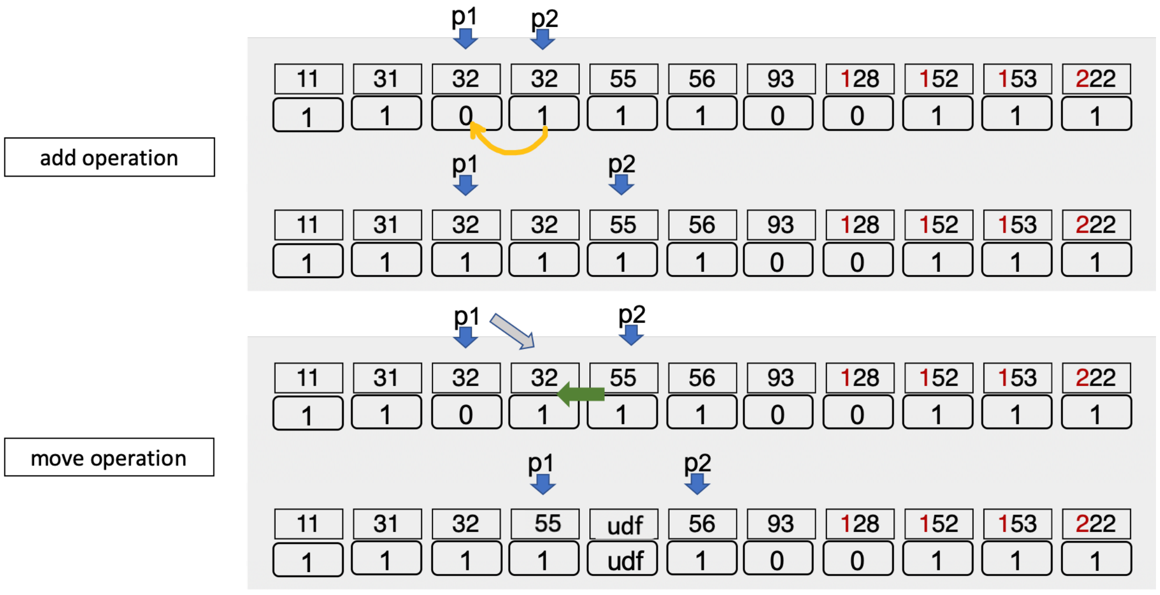


Figure 3.4: Illustration of the in-place two pointers merge method, when the keys of the elements are equal, perform add operation, otherwise, perform move operation

3.2 Scheduling

Static, dynamic, guided are the three main scheduling policies provided in OpenMP.

Static scheduling divides the loop into fixed-sized chunks by the number of threads multiplied by the chunk size. Dynamic scheduling employs a internal work queue to assign a chunk-sized block of work(loop iterations) to threads, and when a thread finishes its work, it gets a new assigned work from the top of the queue until the queue is empty. Guided scheduling looks similar to dynamic scheduling, but the chunk size starts off large and decreases according to the work amount left.

Related work [18, 25] has shown that in a well-balanced situation, we should choose static scheduling due to lowest overhead, it's best to avoid dynamic/guided scheduling because the scheduling cost can be huge on large loops.

Nagasaka also proposed a light-weight load balancing scheme [18], it assigns threads with equal amount of continuous work before symbolic and numeric phase. Our work is a bit more complex, When dealing with scale-free graph, we identify the load imbalance coming from two major parts. First, since the input graph is scale-free, the number of non-zeros in some rows/columns can be significantly larger than the others. Second, when multiplying two scale-free graphs, load imbalance is mirrored to our global bins, the number of flops in some bins could be significantly larger than others, which makes sort out-of-cache.

We extend this balancing technique to a 2-way load balancing scheme, which is shown in the algorithm 7. It not only feeds equal amount of work to threads, which keeps the expansion phase balanced, but also keeps a similar amount of tuples in global bins to be sorted by using variable bins that block tuples from different range of column-row pairs.

Figure 3.5 shows a preliminary evaluation of OuterSpGEMM using static scheduling, dynamic scheduling and the proposed 2-way balanced scheduling on scale 16 R-MAT matrices. R-MAT matrix follows power law distribution, some rows contain significant more

Algorithm 7: OuterSpGEMM 2-way balanced-scheduling algorithms

```
Input: A in CSC, B in CSR
Output: (1) an array m, works as a lookup table, that points tuples to
        corresponding global bin; (2) an array f sets the loop work offset for
        threads
$flops \leftarrow 0$                              ▷ count flop by output row;
$work \leftarrow 0$                           ▷ count per loop generated flop;
for $i \leftarrow firstcolumn$ to $lastcolumn$ in parallel do
    $nnzleft \leftarrow A.colptr[i+1]$ - $A.colptr[i]$;
    $nnzright \leftarrow B.rowptr[i+1]$ - $B.rowptr[i]$;
    $work[i] \leftarrow nnzleft \times nnzright$;
    for $j \leftarrow A.colptr[i]$ to $A.colptr[i+1]$ do
        $flops[[A.rowids[i]]$ += $nnzright$;
$flops_{ps} \leftarrow$ PrefixSum ($flops$);
$avg_{work} \leftarrow$ Sum ($work$) / $nthreads$   ▷ assign average amount of work to threads;
$avg_{flops} \leftarrow$ Sum ($flops$) / $nbins$        ▷ assign average amount tuples to bins;
for $tid \leftarrow 1$ to $nthreads$ do
    $m[tid] \leftarrow$ LowBound ($flops_{ps}$, $avg_{flops} \cdot tid$);
$cur\_bin\_id \leftarrow 0$;
$cur\_volumn \leftarrow 0$;
for $rowid \leftarrow firstcolumn$ to $lastcolumn$ do
    $m[rowid]$ = $cur\_bin\_id$;
    $cur\_volumn$ += $work[rowid]$;
                ▷ if a thread complains about too much work, assign next;
    if if $cur\_volumn > avg_{work}$ then
        $cur\_bin\_id$ ++;
        $cur\_volumn$ = 0;
```

non-zeros than others. As discussed before, the naive static scheduling assigns even number of rows(work) to threads, resulting in heavy load imbalance. Dynamic scheduling doesn't do much better due to schedule overhead and sorting out of cache. Our balanced scheduling demonstrated significant improvement.

3.3 Selecting proper parameters for bins

Two parameters are important to the performance of this algorithm, the local bin width and the number of bins.

The local bins are used to improve data locality when writing tuples. Since modern

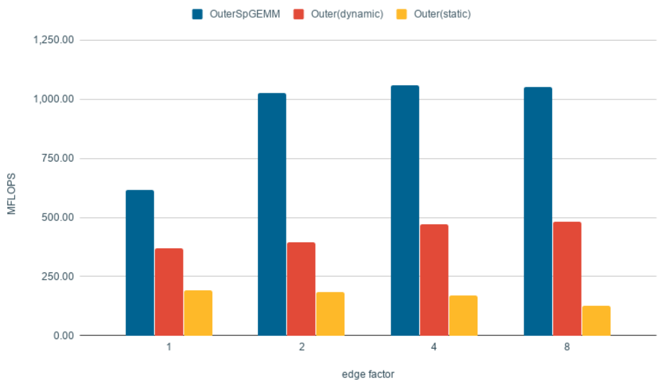
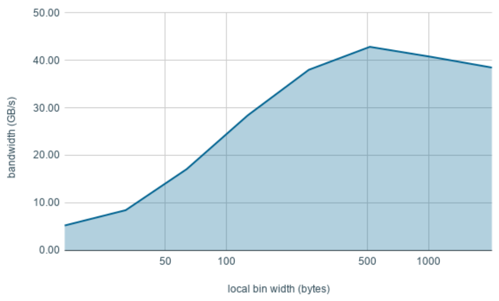
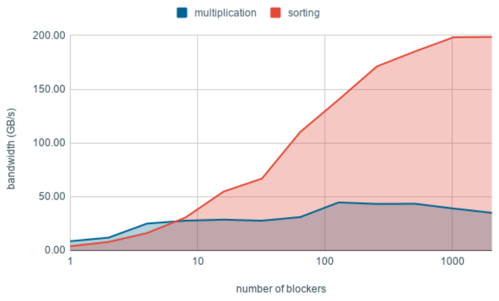


Figure 3.5: Evaluate three different scheduling policies on RMAT(scale=16)



(a) Select local bin width



(b) Select number of bins, 12M flops, 1MB L2 cache

Figure 3.6: Select two important parameters for OuterSpGEMM, local bin width and number of bins

operating system transfers data one cache line at a time, if our bin width is bigger than a cache line, we should hit the peak memory bandwidth. Also, increase the width of bin reduce the atomic communication cost between local bins and global bins. Once the size of local bins is bigger than the size of last line cache, the performance drops due to cache miss(See left chart in Figure 3.6).

Ideally, the size of bin to be as large as possible but all local bins could still fit in L2 cache. let $n\_threads$ be the number of threads used, $size\_L2$ as the size of total last level cache, n_bins as the number of bins for each thread, the maximum local bin width is:

$$S = \frac{size\_L2}{n\_threads \times n\_bins} \tag{3.1}$$

In OuterSpGEMM, we select 512 bytes for the local bin width due to consistent observation of better performance.

On the other hand, the parameter $nbins$ is a trade-off between locality in the propagation blocking and the sorting phase. Increasing the number of bins will decrease the average number of flops in each global bin so the dataset to be sorted can be fit into lower level cache, enjoying faster cache speed and lower latency, but binning phase will need to pay more memory latency and have a higher cache miss.

We want to find a balanced point where these two phases can both get good performance, the maximum number of bins can be calculated. Given a SpGEMM (C = $A \times B$), let $flops$ be the number of total tuples generated, the size of L2 cache divide by the size of tuple is the maximum size of array to be sorted in cache, together we have a inequality:

$$\frac{flops}{nbins} < \frac{size\_L2/size\_tuple}{2 \cdot k} \tag{3.2}$$

We added a $(2 \cdot k)$ to the denominator of the right side of the inequality 3.2 because in the radix sort we need to keep both source array, buffer array and some counters in cache. Based on our experience, the value of k is around 1.8.

CHAPTER 4
EVALUATION

For the evaluation, we compare the performance of OuterSpGEMM against HashSpGEMM, HashVecSpGEMM, HeapSpGEMM, which are implemented based on state-of-the-art algorithms. All the softwares share the same libraries, so we can exclude the concern addressed in some related research that it could affect the result.

4.1 Evaluation Platform Setup

Evaluation are primarily conducted on a single node from IU FutureSystem supercomputer, this particular server is equipped with Intel Skylake dual-socket processor, it has 24 cores per socket and a total of 33MB last line cache(L3 cache). Detailed platform information is described in Table 4.1.

	FutureSystems server
CPU	**Intel(R) Xeon(R) Platinum 8160**
micro-architecture	Skylake
#Sockets	2
#Cores/socket	24
Clock	2.1GHz
L1(i/d) cache	32KB/32KB
L2 cache	1024K
L3 cache	33792K
Memory	
Size	250GB
Bandwidth	100GB/s
Software	
OS	Red Hat Enterprise Linux Server 7.6 (Maipo)
Compiler	gcc version 8.2.0
Option	-g -dynamic -fopenmp -O3

Table 4.1: Overview of Evaluation Environment setup

As we mentioned in the previous introduction chapter, SpGEMM is purely memory bound, so a large factor of performance is how fast the data can be transferred between processors and memory. Measuring these is important to establish a baseline for the system under test, and for performance analysis.

So we exam this carefully with the STREAM benchmark [26], which uses a synthetic benchmark to measure sustainable memory bandwidth of four simple vector kernels (Copy, Scale, Add and Triad). Our experiment result (see Table 4.2) shows this system has a memory bandwidth roughly around 100GB/s in practice.

	Copy	Scale	Add	Triad
single socket	47404.1	46847.0	54002.5	57042.7
dual socket	97731.9	87432.6	107004.7	108419.3

Table 4.2: Stream benchmark result of the evaluation platform

However, this could be more complicated if we consider a multi-socket system where Non-Uniform Memory Access (NUMA) [27] is enabled, local memory latencies and cross-socket memory bandwidth and memory latencies will vary significantly. Table 4.3 describes the local access and cross-socket access memory bandwidth from STREAM copy kernel, as well as the memory latency stats that reported by Intel Memory Latency Checker.

	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 4.3: NUMA bandwidth and latency

NUMA systems are quite popular these days because they are capable of scaling to many more cores than traditional architectures, but surprisingly, most existing SpGEMM algorithms have not yet optimized for it. This work doesn't intend to cover NUMA awareness or any platform specifics, in order to fairly compare the performance of algorithm itself, we set ”OMP_PLACES” to ”cores”, ”OMP_PROC_BIND” to ”close”, the memory allocation is restricted in numa node 0 by ”numactl –membind=0”, in this way, we ensure the memory access is local node only.

We will describe a potential improvement in the discussion section, which splits the matrix to multiple partition and distributes them to multi sockets, in this way we can avoid NUMA effect.

4.2 Dataset

Related work [18] has shown that different SpGEMM algorithms can dominate others depending on the aspect ratio (i.e. ratio of its dimensions), density, sparsity structure, and size (i.e. dimensions) of its inputs. In order to thoroughly and fairly compare the algorithms' performance, the dataset has to respect these metrics.

In our experiments, we use the R-MAT recursive matrix generator [28] to synthetic matrices for algorithm feature study and preliminary performance evaluation. Erdős-Rényi is binomial random graph, we can set the R-MAT seed parameters to a=b=c=d=0.25 to ensure uniformly random. The parameters for G-500 are set to a=0.57, b=c=0.19, d=0.05 to demonstrate strong power-law degree distribution. In addition, 12 real-world matrices from SuiteSparse Matrix Collection [29][30] are added to the dataset, the following table 4.4 summarizes the key characteristics of the real-world graphs that used in our experiments.

In the following sections, we denote the d as the edge factor, nnz as the number of the non-zeros, $flops$ as the floating point operations or tuples generated during multiplication, CR as the compression ratio, n is the number of rows/columns of the matrix. For example, Given a matrix with scale m and edge factor d, the n and nnz should be 2^n, $2^n d$ respectively.

4.3 Performance

We first study the performance on synthetic matrices.

Erdős-Rényi is uniformly random model and it is the very basic case in SpGEMM.

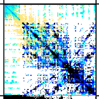
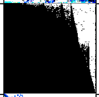
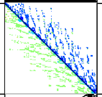
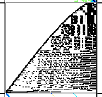
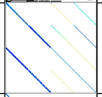
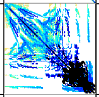
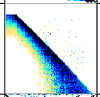
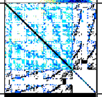
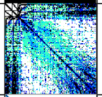
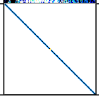
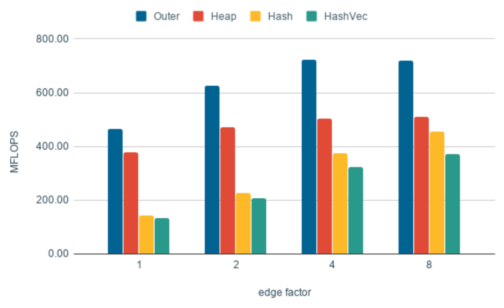
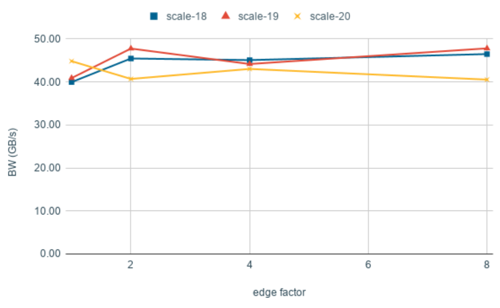
Graph	thumb	n(A)	nnz(A)	d(A)	flops(C)	nnz(C)	CR(C)
2cubes_sphere		101.5K	1.6M	16.23	27.5M	9.0M	3.06
amazon0505		410.2K	3.4M	8.18	31.9M	16.1M	1.98
cage12		130.2K	2.0M	15.61	34.6M	15.2M	2.14
m133_b3		200.2K	800.8K	4.00	3.2M	3.2M	1.01
majorbasis		160.0K	1.8M	10.94	19.2M	8.2M	2.33
mc2depi		525.8K	2.1M	3.99	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
hood		220.5K	9.9M	44.87	562.0M	34.2M	16.41
cant		62.5K	4.0M	64.17	269.5M	17.4M	15.45

Table 4.4: A list of the real-world graphs used in our evaluation

Figure 4.1 describes the performance and bandwidth on scale-20 graphs with increasing density. We can tell from the figure that OuterSpGEMM shows good performance and it almost hit the maximum bandwidth. R-MAT, on the other hand, follows power law distribution, and it is a good case to evaluate the performance with a load imbalanced situation, Figure 4.2 shows the result on scale-16 R-MAT graphs with increasing density, similarly, OuterSpGEMM wins the most of the time. Although on low edge factor case, as many rows don't have even one non-zero, our local bins are never full and the bin-to-bin copy is less efficient, thus a slightly lower bandwidth.

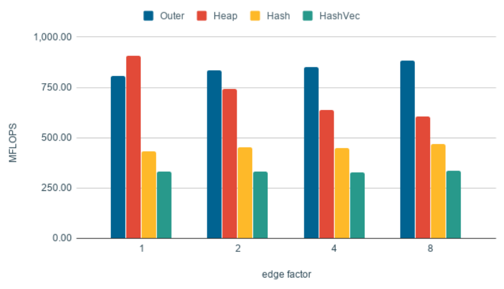


(a) Performance evaluation on Erdős-Rényi matrices with fix scale 20 and increasing edge factor(density)

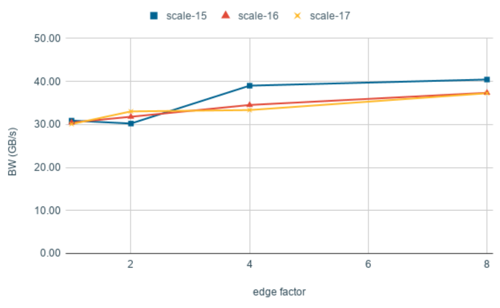


(b) Evaluation of Memory bandwidth of Outer-SpGEMM during the expansion phase

Figure 4.1: Performance evaluation on Erdős-Rényi matrices using 24 threads, with a fix scale 20 and increasing density



(a) Performance evaluation on R-MAT matrices with fix scale 16 and increasing edge factor(density)



(b) Evaluation of Memory bandwidth of Outer-SpGEMM during the expansion phase

Figure 4.2: Performance evaluation on R-MAT matrices using 24 threads, with a fix scale 16 and increasing density

We also study the performance on real-world graphs described in table 4.4. The result 4.3 meets our expectation, HashSpGEMM gets significant boost when the output has a high compression ratio, HeapSpGEMM maintains consistent numbers no matter of compression ratio changes, it gets better when the output is sparse. In most cases(CR < 3), our OuterSpGEMM outperforms them all.

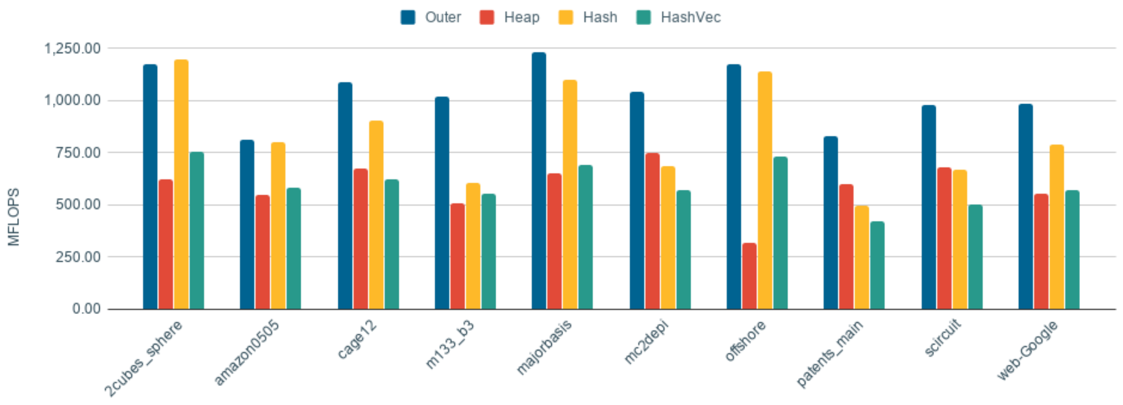


Figure 4.3: Performance evaluation on real-world graphs

4.4 Scalability Analysis

Scalability is important to graph algorithms since many of them are ran on many-core, multi-core and distributed architecture where tens or hundreds of cores are available, we'd like to see a performance boost if put more resources. However, SpGEMM operation is memory bound, in other word, the performance will keep increasing until at some point it stops, and the peak performance is determined by maximum memory bandwidth available.

A SpGEMM algorithm with good scalability actually means (1), the algorithm's performance can increase given more threads if memory is not bound (2), the performance gain per thread is high.

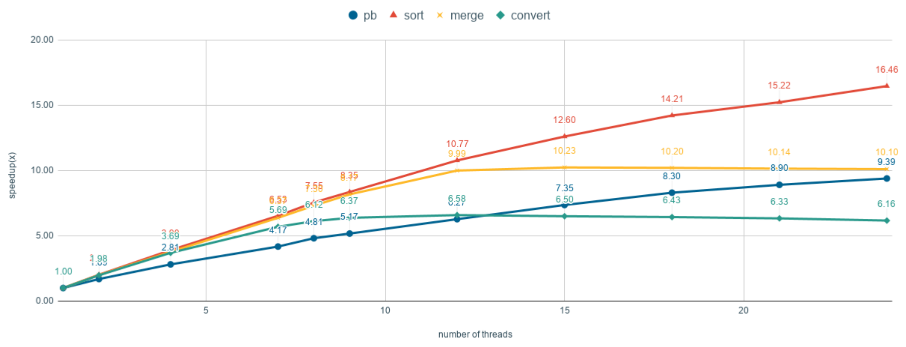


Figure 4.4: OuterSpGEMM scalability analysis on ER(20, 8), with increasing number of threads

Figure 4.4 shows the performance speedup of four key phases of OuterSpGEMM, com-

	pb	sort	merge	convert
bandwidth(GB/s)	48.13	171.83	72.03	44.98

Table 4.5: Sustainable bandwidth speed achieved in four major steps

pared to their single-thread performance. Propagation blocking(expansion), merge and convert stop increasing performance at about 12 threads, where they achieved maximum memory bandwidth available on a single NUMA socket. Sort, as expected, goes a bit further benefiting from using cache. Table 4.5 describes the sustainable bandwidth achieved these four major steps.

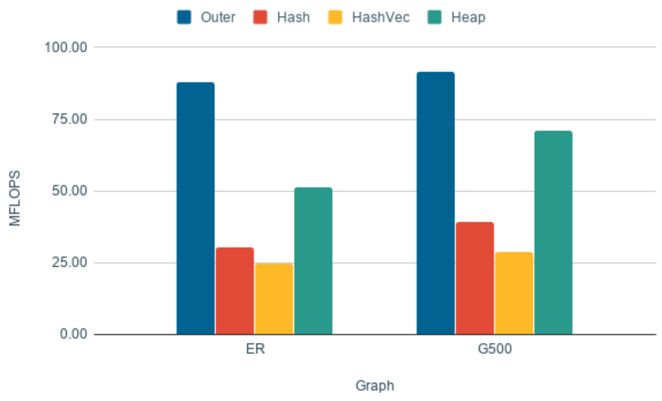


Figure 4.5: Evaluation of single thread performance on ER(16,16) and R-MAT(16,16)

It is also worth noting that due to a higher memory bandwidth utilization and cache awareness, OuterSpGEMM achieves its best performance faster than many other algorithms(using less number of threads). As shown in the Figure 4.5, the single-thread performance is higher as well, which makes OuterSpGEMM a good fit in experimental or non-supercomputing environments like MATLAB.

CHAPTER 5
DISCUSSION

In this work we studied four different matrix-matrix multiplication and implemented a bandwidth-optimized parallel SpGEMM algorithm using propagation blocking. According to our experiments, the OuterSpGEMM achieves satisfactory memory bandwidth utilization, scalability, and it outperforms state-of-the-art HashSpGEMM and HeapSpGEMM in many general cases. However, we have to admit OuterSpGEMM has some limitations(there is yet no perfect SpGEMM algorithm), that's why so many new SpGEMM algorithms are still coming up, they are tailored for different applications and architectures and will shine in different scenarios. At the end of this thesis we want to discuss a few things and future work.

One of the concern is the sensitivity to compression ratio. Our OuterSpGEMM expands and sorts on unmerged entries(i.e. $O(flops)$), but HashSpGEMM operates on merged entries(i.e. $O(nnz)$). In other word, HashSpGEMM can enjoy can significant performance boost on high compression ratio graphs, but OuterSpGEMM and HeapSpGEMM can't.

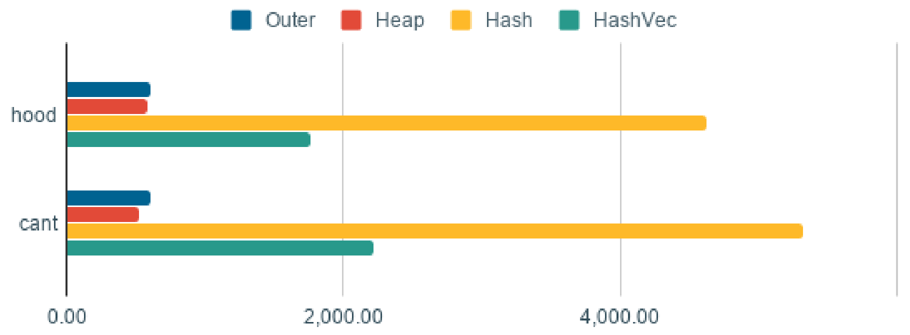
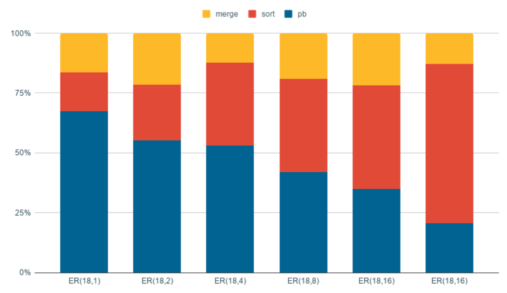


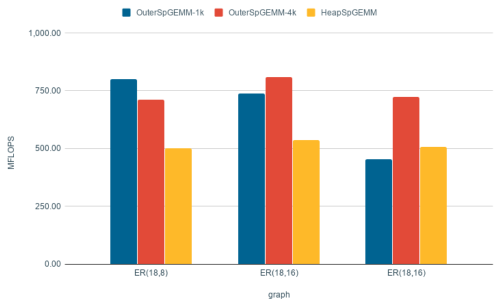
Figure 5.1: Evaluation on high compression graphs

Figure 5.1 shows the experiment ran on *hood* and *cant*, the compression ratio of these two graphs is higher than 15, hash based algorithms clearly dominated the game and is to 10x speedup faster over the others. In practice, such cases are not common but indeed exists, we can implement a hybrid SpGEMM that falls back to HashSpGEMM when a high compression ratio detected(CR > 3).

One other issue is the sensitivity to number of flops. OuterSpGEMM sorts tuples before merging, it moves tuples back-forward multiple times so it is very important to keep the sort partition in the cache. In Figure 5.2 the left chart shows that if the matrix is big enough the sort phase could out of cache thus dominating the execution time, and performance suffers.



(a) Evaluation of Memory bandwidth of OuterSpGEMM during the expansion phase



(b) Performance evaluation on Erdős-Rényi matrices with fix scale 18 and increasing edge factor(density)

Figure 5.2: Execution time percentage analysis on ER matrices, scale=18

As shown in section 3.3, the size of sort partition is related to number of bins since it equals to $flops/n\_bins$. At some point we have to increase the number of bins, the right chart in figure 5.2 shows the performance improvement using 4k bins(the red bar) over 1k bins(the blue bar), which trades the expansion phase bandwidth efficiency for better sorting and overall performance.

In addition, as discussed in section 4.1, NUMA effect could play a significant role in memory bandwidth and latency, it's hard to avoid and most SpGEMM algorithms have to comply and pay the price. Figure 5.3 describes a potential way out, a NUMA-aware outer-product SpGEMM algorithm.

Given a $C = A \times B$, partition the left matrix A into an upper half A_{p1} and lower half

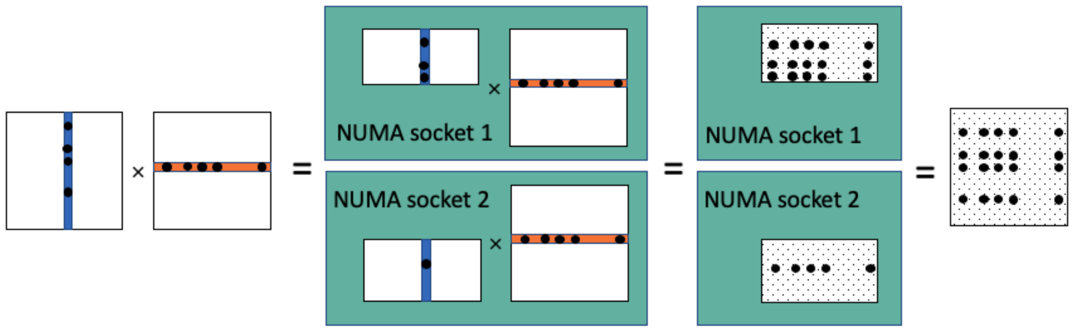


Figure 5.3: A potential NUMA-aware outer-product SpGEMM algorithm

A_{p2}, then we assign the upper multiplication $A_{P1} \times B$ to NUMA socket 1, and the lower multiplication goes to NUMA socket 2. In iterative methods where SpGEMM serves as the key primitive and is called multiple times, threads pinned in each NUMA socket first initialize their own A partition so the memory will be allocated in corresponding socket, in this way, cross-socket memory access should be eliminated.

Furthermore, this method could extend to multi-socket systems and/or split the A into multiple partitions. An extra benefit is lower memory/cache requirement, since each time we will be dealing with a much smaller matrix and generating smaller amount of flops.

Appendices

APPENDIX A
SOURCE CODE

Our OuterSpGEMM implementation on share memory architectures is opensource software and licensed under the Apache License, Version 2.0, it is available at bitbucket(`https://bitbucket.org/azadcse/outerspgemm/`).

We also include the source code for HashSpGEMM and HeapSpGEMM, implemented by Yusuke Nagasaka.

A script to generate synthetic matrices and download real-world matrices can be found in the repository as well.

REFERENCES

[1] Aydin Buluç and John R Gilbert. “The Combinatorial BLAS: Design, Implementation, and Applications”. In: *Int. J. High Perform. Comput. Appl.* 25.4 (Nov. 2011), pp. 496–509.

[2] Ariful Azad, Aydin Buluç, and John Gilbert. “Parallel Triangle Counting and Enumeration Using Matrix Algebra”. In: *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IPDPSW ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 804–811. ISBN: 978-1-4673-7684-6.

[3] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. “High-performance Graph Algorithms from Parallel Sparse Matrices”. In: *Proceedings of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing*. PARA’06. Ume&#229;, Sweden: Springer-Verlag, 2007, pp. 260–269. ISBN: 3-540-75754-6, 978-3-540-75754-2.

[4] 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. Sedona, Arizona, USA: ACM, 2006, pp. 52–60. ISBN: 1-59593-340-9.

[5] 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*. SODA ’04. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2004, pp. 254–260. ISBN: 0-89871-558-X.

[6] Grey Ballard, Christopher M. Siefert, and Jonathan J. Hu. “Reducing Communication Costs for Sparse Matrix Multiplication within Algebraic Multigrid”. In: *SIAM J. Scientific Computing* 38 (2015).

[7] Xu Feng, Yuyang Xie, Mingye Song, Wenjian Yu, and Jie Tang. “Fast Randomized PCA for Sparse Data”. In: *ACML*. 2018.

[8] Yu Jin and Joseph JáJá. “A High Performance Implementation of Spectral Clustering on CPU-GPU Platforms”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2016), pp. 825–834.

[9] Ariful Azad, Georgios Pavlopoulos, Christos Ouzounis, Nikos Kyrpides, and Aydin Buluç. “HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks”. In: *Nucleic Acids Research* 46 (Jan. 2018), pp. 1–11.

[10] Andreas Griewank and Uwe Naumann. “Accumulating Jacobians as chained sparse matrix products”. In: *Math. Program.* 95 (Mar. 2003), pp. 555–571.

[11] Rasmus Resen Amossen and Rasmus Pagh. “Faster Join-projects and Sparse Matrix Multiplications”. In: *Proceedings of the 12th International Conference on Database Theory*. ICDT ’09. St. Petersburg, Russia: ACM, 2009, pp. 121–126. ISBN: 978-1-60558-423-2.

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

[13] Aydin Buluç and John Gilbert. “Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication”. In: Oct. 2008, pp. 503–510. ISBN: 978-0-7695-3374-2.

[14] John R. Gilbert, Cleve. Moler, and Robert. Schreiber. “Sparse Matrices in MATLAB: Design and Implementation”. In: *SIAM Journal on Matrix Analysis and Applications* 13.1 (1992), pp. 333–356. eprint: `https://doi.org/10.1137/0613024`.

[15] NVIDIA Corporation. *The NVIDIA CUDA Sparse Matrix library*.

[16] H. C. Edwards and C. R. Trott. “Kokkos: Enabling Performance Portability Across Manycore Architectures”. In: *2013 Extreme Scaling Workshop (xsw 2013)*. 2013, pp. 18–24.

[17] 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”. In: *SIAM J. Scientific Computing* 38 (2016).

[18] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydin Buluç. “Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors”. In: *Parallel Computing* 90 (Aug. 2019), p. 102545.

[19] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures”. In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76.

[20] P. Erdös and A. Rényi. “On Random Graphs I”. In: *Publicationes Mathematicae Debrecen* 6 (1959), p. 290.

[21] Steven Dalton, Luke Olson, and Nathan Bell. “Optimizing Sparse Matrix&Mdash;Matrix Multiplication for the GPU”. In: *ACM Trans. Math. Softw.* 41.4 (Oct. 2015), 25:1–25:20.

[22] Aydin Buluç and John Gilbert. “On the representation and multiplication of hypersparse matrices”. In: Apr. 2008, pp. 1–11.

[23] 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, pp. 724–736.

[24] S. Beamer, K. Asanović, and D. Patterson. “Reducing Pagerank Communication via Propagation Blocking”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 820–831.

[25] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. “High-performance sparse matrix-matrix products on Intel KNL and multicore architectures”. In: *Proceedings of the 47th International Conference on Parallel Processing Companion*. 2018, pp. 1–10.

[26] John D. McCalpin. “Memory Bandwidth and Machine Balance in Current High Performance Computers”. In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25.

[27] Christoph Lameter. “NUMA (Non-Uniform Memory Access): An Overview”. In: *Queue* 11.7 (July 2013), 40:40–40:51.

[28] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. “R-MAT: A Recursive Model for Graph Mining”. In: *Proceedings of the 2004 SIAM International Conference on Data Mining*, pp. 442–446. eprint: `https://epubs.siam.org/doi/pdf/10.1137/1.9781611972740.43`.

[29] Paolo Boldi and Sebastiano Vigna. “The WebGraph Framework I: Compression Techniques”. In: *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[30] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. “Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks”. In: *Proceedings of the 20th international conference on World Wide Web*. Ed. by Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar. ACM Press, 2011, pp. 587–596.

VITA

2018 - present	Master student, Indiana University Bloomington
2019, summer	Software Engineer Intern, Facebook Inc.
2016 - 2018	Software Engineer, Xiachufang Technology Inc., Beijing
2012 - 2016	B.S., Beijing University of Technology

FIELD OF STUDY

Major Field: Computer Engineering, Intelligent Systems Engineering