Title

Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors

Authors

Nagasaka, Yusuke
Matsuoka, Satoshi
Azad, Ariful
et al.

Publication Date

2019-12-01

DOI

10.1016/j.parco.2019.102545

Peer reviewed

Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors

Yusuke Nagasaka[a,*], Satoshi Matsuoka[b,1], Ariful Azad[c], Aydın Buluç[d,2]

[a]*Tokyo Institute of Technology, Tokyo, Japan*
[b]*RIKEN Center for Computational Science, Kobe, Japan*
[c]*Indiana University, Bloomington, Indiana, USA*
[d]*Lawrence Berkeley National Laboratory, Berkeley, California, USA*

Abstract

Sparse matrix-matrix multiplication (SpGEMM) is a computational primitive that is widely used in areas ranging from traditional numerical applications to recent big data analysis and machine learning. Although many SpGEMM algorithms have been proposed, hardware specific optimizations for multi- and many-core processors are lacking and a detailed analysis of their performance under various use cases and matrices is not available. We firstly identify and mitigate multiple bottlenecks with memory management and thread scheduling on Intel Xeon Phi (Knights Landing or KNL). Specifically targeting many-core processors, we develop a hash-table-based algorithm and optimize a heap-based shared-memory SpGEMM algorithm. We examine their performance together with other publicly available codes. Different from the literature, our evaluation also includes use cases that are representative of real graph algorithms, such as multi-source breadth-first search or triangle counting. Our hash-table and heap-based algorithms are showing significant speedups from libraries in the majority of the cases while different algorithms dominate the other scenarios with different matrix size, sparsity, compression factor and operation type. We wrap up in-depth evaluation results and make a recipe to give the best SpGEMM algorithm for target scenario. We build the performance model for hash-table and heap-based algorithms, which supports the recipe. A critical finding is that hash-table-based SpGEMM gets a significant performance boost if the nonzeros are not required to be sorted within each row of the output matrix. Finally, we integrate our implementations into a large-scale protein clustering code named HipMCL, accelerating its SpGEMM kernel by up to 10X and achieving an overall performance boost for the whole HipMCL application by 2.6X.

Keywords: Sparse matrix, SpGEMM, Intel KNL

1. Introduction

Multiplication of two sparse matrices (SpGEMM) is a recurrent kernel in many algorithms in machine learning, data analysis, and graph analysis. For example, bulk of the computation in multi-source breadth-first search [1], betweenness centrality [2], Markov clustering [3], label propagation [4], peer pressure clustering [5], clustering coefficients [6], high-dimensional similarity search [7], and topological similarity search [8] can be expressed as SpGEMM. Similarly, numerical applications such as scientific simulations also use SpGEMM as a subroutine. Typical examples include the Algebraic Multigrid (AMG) method for solving sparse system of linear equations [9], volumetric mesh processing [10], and linear-scaling electronic structure calculations [11].

The extensive use of SpGEMM in data-intensive applications has led to the development of several sequential and parallel algorithms. Most of these algorithms are based on Gustavson's row-wise SpGEMM algorithm [12] where a row of the output matrix is constructed by accumulating a subset of rows of the second input matrix (see Algorithm 1 for details). It is the accumulation (also called merging) technique that often distinguishes major classes of SpGEMM algorithms from one another. Popular data structures for accumulating rows or columns of the output matrix include heap [13], hash [14], and sparse accumulator (SPA) [15].

Recently, researchers have developed parallel heap-, hash-, and SPA-based SpGEMM algorithms for shared-memory platforms [16, 17, 18, 19, 20]. These algorithms are also packaged in publicly-available software that can tremendously benefit many scientific applications. However, when using an SpGEMM algorithm and implementation for a scientific problem, one needs answer to the following questions: (a) what is the best algorithm/implementation for a *problem* at hand? (b) what is the best algorithm/implementation for the *architecture* to be used in solving the problem? These practically important questions remain mostly unanswered for many scientific applications running on highly-threaded

*Corresponding author
Email addresses: nagasaka.y.aa@m.titech.ac.jp (Yusuke Nagasaka), matsu@acm.org (Satoshi Matsuoka), azad@iu.edu (Ariful Azad), abuluc@lbl.gov (Aydın Buluç)
[1]Also with Tokyo Institute of Technology, Department of Mathematical and Computing Science
[2]Also with UC Berkeley, Department of Electrical Engineering and Computer Sciences

architectures. This paper answers both questions in the context of existing SpGEMM algorithms. That means our focus is not to develop new parallel algorithms, but to characterize, optimize and evaluate existing algorithms for real-world applications on modern multi-core and many-core architectures.

Firstly, previous algorithmic work did not pay close attention to architecture-specific optimizations that have big impacts on the performance of SpGEMM. We fill this gap by characterizing the performance of SpGEMM on shared-memory platforms and identifying bottlenecks in memory allocation and deallocation as well as overheads in thread scheduling. We propose solutions to mitigate those bottlenecks. Using microbenchmarks that model SpGEMM access patterns, we also uncover reasons behind the non-uniform performance boost provided by the MCDRAM on KNL. These optimizations result in efficient heap-based and hash-table-based SpGEMM algorithms that outperform state-of-the-art SpGEMM libraries including Intel MKL and Kokkos-Kernels [21] for many practical problems.

Secondly, previous work has narrowly focused on one or two real world application scenarios such as squaring a matrix and studying SpGEMM in the context of AMG solvers [22, 21]. Different from the literature, our evaluation also includes use cases that are representative of real graph algorithms, such as the multiplication of a square matrix with a tall skinny one that represents multi-source breadth-first search and the multiplication of triangular matrices that is used in triangle counting. While in the majority of the cases the hash-table-based SpGEMM algorithm is dominant, we also find that different algorithms dominate depending on matrix size, sparsity, compression factor, and operation type. This in-depth analysis exposes many interesting features of algorithms, applications, and multithreaded platforms.

Thirdly, while many SpGEMM algorithms keep nonzeros sorted within each row (or column) in increasing column (or row) identifiers, this is not universally necessary for subsequent sparse matrix operations. For example, CSparse [23, 24] assumes none of the input matrices are sorted. Clearly, if an algorithm accepts its inputs only in sorted format, then it must also emit sorted output for fairness. This is the case with the heap-based algorithms. However, hash-table-based algorithm do not need their inputs sorted. In this case, we see a significant performance benefit due to skipping the sorting of the output as well.

Fourthly, based on these architecture- and application-centric optimizations and evaluations, we make a recipe for selecting the best-performing algorithm for a specific application scenario. We also build a performance model for hash-table and heap-based algorithms, and show detailed profiling result of them. These performance models and profiling results support the correctness of the recipe both theoretically and empirically. With this recipe, we switch the SpGEMM algorithm inside a large-scale protein clustering application named HipMCL, and show the positive performance impact of selecting the appropriate algorithm in real applications.

Algorithm 1 Gustavson's Row-wise SpGEMM [3]

Input: Sparse matrices A and B
Output: Sparse matrix C

```
1:  set matrix C to ∅
2:  for all a_i* in matrix A in parallel do
3:    for all a_ik in row a_i* do
4:      for all b_kj in row b_k* do
5:        value ← a_ik b_kj
6:        if c_ij ∉ c_i* then
7:          insert (c_ij ← value) to c_i*
8:        else
9:          c_ij ← c_ij + value
10:       end if
11:     end for
12:   end for
13: end for
```

This paper brings various SpGEMM algorithms and libraries together, analyzes them based on algorithm, application, and architecture related features and provides exhaustive guidelines for SpGEMM-dependent applications.

2. Background and Related Work

Let A, B be input matrices, and SpGEMM computes a matrix C such that $C = AB$. When analyzing algorithms in this paper, we assume n-by-n matrices for simplicity. The input and output matrices are sparse and they are stored in a sparse format. The number of nonzeros in matrix A is denoted with $nnz(A)$. Figure 1 shows the skeleton of the most commonly implemented SpGEMM algorithm, which is due to Gustavson [12]. When the matrices are stored using the Compressed Sparse Rows (CSR) format, this SpGEMM algorithm proceeds row-by-row on matrix A (and hence on the output matrix C). Let a_{ij} be the element in i-th row and j-th column of matrix A and a_{i*} be the i-th row of matrix A. The row of matrix B corresponding to each non-zero element of matrix A is read, and each non-zero element of output matrix C is calculated.

SpGEMM computation has two critical issues unlike dense matrix multiplication. Firstly, the pattern and the number of non-zero elements of output matrix are not known beforehand. For this reason, the memory allocation of output matrix becomes hard, and we need to select from two strategies. One is a two-phase method, which counts the number of non-zero elements of output matrix first (symbolic phase), and then allocates memory and computes output matrix (numeric phase). The other is a one-phase method, where we allocate large enough memory space for output matrix and compute. The former requires more computation cost, and the latter uses much more memory space. Second issue is about combining

[3] The **in parallel** keyword does not exist in the original algorithm but is used here to illustrate the common parallelization pattern of this algorithm used by all known implementations.

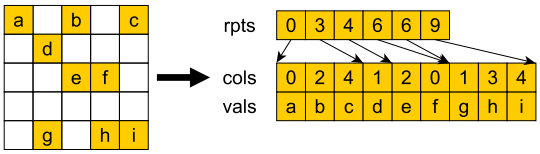


Figure 1: Example of CSR format

the intermediate products (*value* in Fig. 1) to non-zero elements of output matrix. Since the output matrix is also sparse, it is hard to efficiently accumulate intermediate products into non-zero elements. This procedure is a performance bottleneck of SpGEMM computation, and it is important to devise and select better accumulator for SpGEMM.

Since each row of C can be constructed independently of each other, Gustavson's algorithm is conceptually highly parallel. For accumulation, Gustavson's algorithm uses a dense vector and a list of indices that hold the nonzero entries in the current active row. This particular set of data structures used in accumulation are later formalized by Gilbert et al. under the name of sparse accumulator (SPA) [15]. Consequently, a naive parallelization of Gustavson's algorithm requires temporary storage of $O(n\,t)$ where t is the number of threads. For matrices with large dimensions, a SPA-based algorithm can still achieve good performance by "blocking" SPA in order to decrease cache miss rates. Patwary et al. [22] achieved this by partitioning the data structure of B by columns.

Sulatycke and Ghose [16] presented the first shared-memory parallel algorithm for the SpGEMM problem, to the best of our knowledge. Their parallel algorithm, dubbed *IKJ method* due to the order of the loops, has a double-nested loop over the rows and the columns of the matrix A. Therefore, the IKJ method has work complexity $O(n^2 + \text{flop})$ where flop is the number of the non-trivial scalar multiplications (i.e. those multiplications where both operands are nonzero) required to compute the product. Consequently, the IKJ method is only competitive when flop $\geq n^2$, which is rare for SpGEMM. Several GPU algorithms that are also based on the row-by-row formulation are presented [25, 14]. These algorithms bin the rows based on their density due to the peculiarities of the GPU architectures. Then, a poly-algorithm executes a different specialized kernel for each bin, depending on its density. Two recent algorithms that are implemented in both GPUs and CPUs also follow the same row-by-row pattern, only differing on how they perform the merging operation. ViennaCL [26] implementation, which was first described for GPUs [20], iteratively merges sorted lists, similar to merge sort. KokkosKernels implementation [21], which we also include in our evaluation, uses a multi-level hash map data structure.

As Figure 1 shows, the CSR format is composed of three arrays: row pointers array (*rpts*) of length $n + 1$, column indices (*cols*) of length *nnz*, and values (*vals*) of length *nnz*. Array *rpts* indexes the beginning and end locations of nonzeros within each row such that the range $cols[rpts[i] \ldots rpts[i + 1] - 1]$ lists the column indices of row i. The CSR format does not specify whether this range should be sorted with increasing column indices; that decision has been left to the library implementation. As we will show in our results, there are significant performance benefits of operating on unsorted CSR format. Table 1 lists high-level properties of the codes we study in this paper. Heap and Hash are based on our prior work [13, 14]. Since MKL code is proprietary, we do not know the accumulator.

Table 1: Summary of SpGEMM codes studied in this paper

Algorithm	Phases	Accumulator	Sortedness (Input/Output)
MKL	2	-	Any/Select
MKL-inspector	1	-	Any/Unsorted
KokkosKernels	2	HashMap	Any/Unsorted
Heap	1	Heap	Sorted/Sorted
Hash/HashVector	2	Hash Table	Any/Select

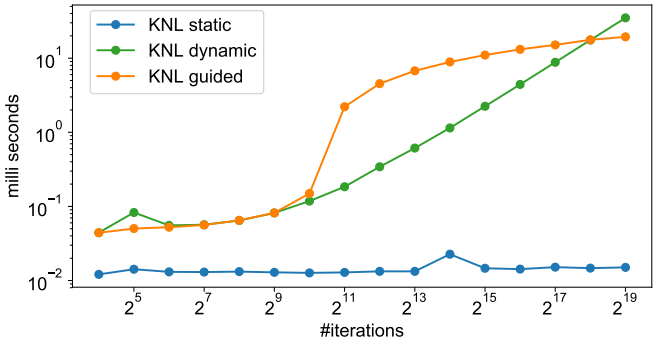


Figure 2: OpenMP scheduling cost on KNL

3. Microbenchmarks on Intel KNL

Our experiments mainly target Intel Xeon Phi architecture. We conducted some preliminary experiments to tune optimization parameters to expose the performance bottlenecks. These microbenchmarks are especially valuable for designing an algorithm of SpGEMM for these architectures. Details of evaluation environment are summarized in Table 3.

3.1. Scheduling Cost of OpenMP

When parallelizing a loop, OpenMP provides three thread scheduling choices: *static*, *dynamic* and *guided*. Static scheduling divides loop iterations equally among threads, dynamic scheduling allocates iterations to threads dynamically, and guided scheduling starts with static scheduling with smaller iterations and switches to dynamic scheduling later. Here, we experimentally evaluate the cost of these three scheduling options on KNL processors by running a simple program, which performs iterations of an empty loop. We measure the time during loop iterations and the result is shown in Figure 2. In the load balanced case with a large number of iterations, static scheduling has little scheduling overhead compared to dynamic scheduling on KNL, as expected. The guided scheduling is also as expensive as dynamic scheduling. Based on these evaluations, we opt to use

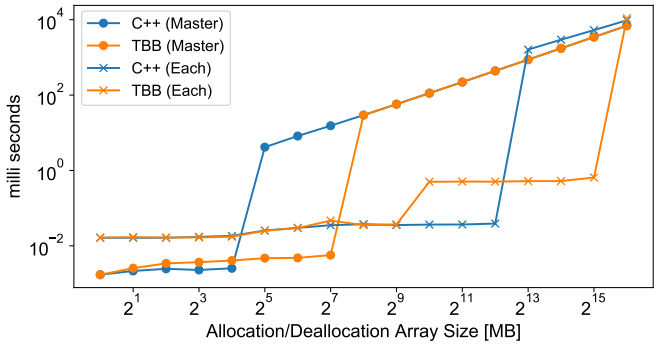


Figure 3: Cost of deallocation on KNL

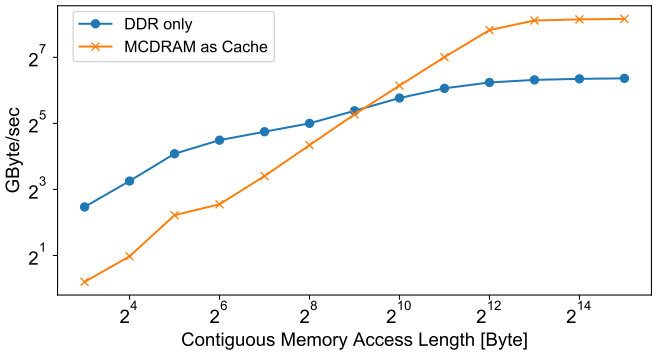


Figure 4: Benchmark result of random memory access with DDR only or MCDRAM as Cache

static scheduling in our SpGEMM algorithms since the scheduling overhead of dynamic or guided scheduling becomes a bottleneck.

3.2. Memory Allocation/Deallocation

To find a suitable memory allocation/deallocation scheme on KNL, we performed a simple experiment: allocate a memory space, access elements on the allocated memory and then deallocate it. In section, we consider two aspects of memory allocation. One is parallelism. Either each thread independently allocates its memory space (called *each* in this paper), or one master thread allocates space for all threads (called *master* in this paper).

The other aspect is the allocator used. We examined three allocators: new/delete of C++, aligned allocation (_mm_malloc/_free), and scalable_malloc/_free provided by Intel TBB (Thread Building Block). Figure 3 shows the results of deallocation cost with 256 threads on KNL. Since aligned allocation showed nearly the same performance as C++, we show only the results of C++ and TBB. All allocators by *master* have extremely high cost for large memory: over 100 milliseconds for the deallocation of 1GB memory space. The *each* deallocation for large memory chunks is much cheaper than *master* deallocation. The cost of *each* deallocation suddenly rises at 8GB (C++) or 64GB (TBB), where each thread allocates 32MB or 256MB, respectively. These thresholds match those of *master* deallocation. On the other hand, the cost of *each* deallocation for small memory space becomes larger than that of *master* deallocation since *each* deallocation causes the overheads of OpenMP scheduling and synchronization. From this result, we compute the amount of memory required by each thread and allocate this amount of thread-private memory in each thread independently in order to reduce deallocation cost in SpGEMM computation, which requires temporally memory allocation and deallocation. In the following experiments in this paper, the TBB is used for both *master* and *each* memory allocation/deallocation to simply have performance gain.

3.3. DDR vs. MCDRAM

Intel KNL implements MCDRAM, which can accelerate bandwidth-bound algorithms and applications. While MCDRAM provides high bandwidth, its memory latency is larger than that of DDR4. In row-wise SpGEMM (Algorithm 1), there are three main types of data accesses for the formation of each row of C. Firstly, there is a unit-stride streaming access pattern arising from access of the row pointers of A as well as the creation of the sparse output vector c_{i*}. Secondly, access to rows of B follows a stanza-like memory access pattern where small blocks (stanzas) of consecutive elements are fetched from effectively random locations in memory. Finally, updates to the accumulator exhibit different access pattern depending on the type of the accumulator (a hash-table, SPA, or heap). The streaming access to the input vector is usually the cheapest of the three and the accumulator access depends on the data structure used. Hence, stanza access pattern is the most canonical of the three and provides a decent proxy to study.

To quantify the stanza bandwidth which we expect to be quite different than STREAM [27], we used a custom microbenchmark that provides stanza-like memory access patterns (read or update) with spatial locality varying from 8 bytes (random access) to the size of the array (i.e. asymptotically the STREAM benchmark). Figure 4 shows a comparison result between DDR only and use of MCDRAM as Cache with scaling the length of contiguous memory access. When the contiguous memory access is wider, both DDR only and MCDRAM as Cache achieve their peak bandwidth, and especially MCDRAM as Cache shows over 3.4× superior bandwidth compared to DDR only. However, the use of MCDRAM as Cache is incompatible with fine-grained memory access. When the stanza length is small, there is little benefit of using MCDRAM. This benchmark hints that it would be hard to get the benefits of MCDRAM on very sparse matrices.

4. Architecture Specific Optimization of SpGEMM

Based on our findings of performance critical bottlenecks on Intel KNL, we design SpGEMM algorithms taking account in architecture specific issues. Firstly, we show light-weight thread scheduling scheme with load-balancing for SpGEMM. Next, we show the optimization schemes for hash-table-based SpGEMM, which is proposed for GPU [14], and heap-based shared-memory SpGEMM algorithms [13]. Additionally, we extend the Hash SpGEMM with utilizing vector registers of Intel Xeon Phi. Finally, we show which accumulator works well for target scenario from the theoretical point of view by estimating each accumulation cost.

Algorithm 2 RowsToThreads

Input: Sparse matrices A and B
Output: Array $offset$

```
1: {1. Set FLOP vector}
2: for i ← 0 to m in parallel do
3:    flop[i] ← 0
4:    for j ← rpts_A[i] to rpts_A[i + 1] do
5:       rnz ← rpts_B[cols_A[j] + 1] − rpt_B[cols_A[j]]
6:       flop[i] ← flop[i] + rnz
7:    end for
8: end for
9: {2. Assign rows to thread}
10: flop_ps ← ParallelPrefixSum(flop)
11: sum_flop ← flop_ps[m]
12: tnum ← omp_get_max_threads()
13: ave_flop ← sum_flop / tnum
14: offset[0] ← 0
15: for tid ← 1 to tnum in parallel do
16:    offset[tid] ← lowbnd(flop_ps, ave_flop * tid)
17: end for
18: offset[tnum] ← m
```

Algorithm 3 Hash SpGEMM

Input: Sparse matrices A and B
Output: Sparse matrix C

```
1: offset ← RowsToThreads(A, B)
2: {Determine hash-table size for each thread}
3: tnum ← omp_get_max_threads()
4: for tid ← 0 to tnum in parallel do
5:    size_t ← 0
6:    for i ← offset[tid] to offset[tid + 1] do
7:       size_t ← max(size_t, flop[i])
8:    end for
9:    {Required maximum hash-table size is N_col}
10:   size_t ← min(N_col, size_t)
11:   {Return minimum 2^n so that 2^n > size_t}
12:   size_t ← lowest_p2(size_t)
13: end for
14: Symbolic(rpts_C, A, B)
15: Numeric(C, A, B)
```

4.1. Light-weight Load-balancing Thread Scheduling Scheme

To achieve good load-balance with static scheduling, the bundle of rows should be assigned to threads with equal computation complexity before symbolic or numeric phase. Algorithm 2 shows how to assign rows to threads. We count flop of each row, then do prefix sum. Each thread can find the start point of rows by binary search. lowbnd(vec, $value$) in line 16 finds the minimum id such that $vec[id]$ is larger than or equal to $value$. Each of these operations can be executed in parallel.

4.2. Symbolic and Numeric Phases

We optimized two approaches of accumulation for KNL. One is hash-table-based algorithm and the other is heap-based algorithm. Furthermore, we add another version of Hash SpGEMM, where hash probing is vectorized with AVX2 or AVX-512 instructions.

4.2.1. Hash SpGEMM

We use hash-table for accumulator in SpGEMM computation, based on GPU work [14]. Algorithm 3 shows the algorithm of Hash SpGEMM for multi- and many-core processors. We count a flop per row of output matrix. The upper limit of any thread's local hash-table size is the maximum number of flop per row within the rows assigned to the thread. Each thread once allocates the hash-table based on its own upper limit and reuses that hash-table throughout the computation by reinitializing for each row. Next is about hashing algorithm we adopted. A column index is inserted into hash-table as key. Since the column index is no less than 0, the hash-table is initialized by storing -1. The column index (= key is multiplied by constant number, c, and divided by hash-table size to compute the remainder. In order to compute modulus operation efficiently, the hash-table size is set as 2^n (n is an integer). The hashing algorithm is based on linear probing. Figure 5-(a) shows an example of hash probing on 16 entries hash-table. The index of hash table in first attempt is calculated as $h = (key * c)\%16$, and if that produces a collision, the next is $(h + 1)\%16$.

In symbolic phase, it is enough to insert keys to the hash-table. In numeric phase, however, we need to store the resulting value data. Once the computation on the hash-table finishes, the results are sorted by column indices in ascending order (if necessary), and stored to memory as output. The Hash SpGEMM for multi/many-core processors computes each row of output matrix by single thread. On the other hand, multiple threads are assigned to a row of output matrix in the GPU version of Hash SpGEMM in order to exploit massive number of threads on GPU. Due to this design for GPU version, the Hash SpGEMM on GPU requires some form of mutual exclusion since multiple threads access the same entry of the hash-table concurrently. We were able to remove this overhead in our present Hash SpGEMM for multi/many-core processors.

4.2.2. HashVector SpGEMM

Intel Xeon or Xeon Phi implements 256 and 512-bit wide vector register, respectively. This vector register reduces instruction counts and brings large benefit to algorithms and applications, which require contiguous memory access. However, sparse matrix computation has indirect memory access, and hence it is hard to utilize vector registers. In this paper, we utilize vector register for hash probing in our Hash SpGEMM algorithm. The vectorization of hash probing is based on Ross [28]. Figure 5-(b) shows how HashVector algorithm works hash probing. The size of hash-table is 16, same as (a), and it is divided into chunks based on vector width (=256-bit in this case). A chunk consists of 8 entries when a key (= column index) is represented as 32-bit. In HashVector,

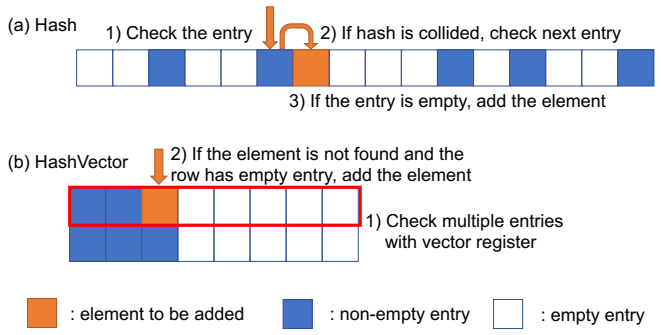


Figure 5: Hash probing in Hash and HashVector SpGEMM

the hash indicates the identifier of target chunk in hash-table. In order to examine the keys in the chunk (Figure-5-(b)-(1)), we use comparison instruction with vector register. If the entry with target key is found, the algorithm finishes the probing for the element in symbolic phase. In numeric phase, the target entry in chunk is identified by __builtin_ctz function, which counts trailing zeros, and the multiplied value is added to the value of the entry. If the algorithm finds no entry with the key, the element is pushed to the hash-table as Figure 5-(b)-(2) shows. In HashVector, new element is pushed into the table in order from the beginning. The entries in chunk are compared with the initial value of hash-table, -1, by using vector register. The beginning of empty entries can be found by counting the number of bit flags of comparison result. When the chunk is occupied with other keys, the next chunk is to be checked in accordance with linear probing. Since HashVector SpGEMM can reduce the number of probing caused by hash collisions, it can achieve better performance compared to Hash SpGEMM. However, HashVector requires a few more instructions for each check. Thus, HashVector may degrade the performance when the collisions in Hash SpGEMM are rare.

4.2.3. Heap SpGEMM

In another variant of SpGEMM [13], we use a priority queue (heap) – indexed by column indices – to accumulate each row of C. To construct c_{i*}, a heap of size $nnz(a_{i*})$ is allocated. For every nonzero a_{ik}, the first nonzero entry in b_{k*} along with its column index is inserted into the heap. The algorithm iteratively extracts an entry with the minimum column index from the heap, accumulates it to c_{i*}, and inserts the next nonzero entry from the last extracted row of B into the heap. When the heap becomes empty, the algorithm moves to construct the next row of C.

Heap SpGEMM can be more expensive than hash- and SPA- based algorithms because it requires logarithmic time to extract elements from the heap. However, from the accumulator point of view, Heap SpGEMM is space efficient as it uses $O(nnz(a_{i*}))$ memory to accumulate c_{i*} instead of $O(\text{flop}(c_{i*}))$ and $O(n)$ memory used by hash- and SPA-based algorithms, respectively.

Our implementation of Heap SpGEMM adopts the one-phase method, which requires larger memory usage for temporally keeping the output. In parallel Heap SpGEMM, because rows of C are independently constructed by different threads, this temporary memory use for keeping output is thread-independent and the memory allocation/deallocation is done by each thread. Thread-private heaps are also managed by each thread. As with the Hash algorithm, Heap SpGEMM estimates flop via a symbolic step and uses it to balance computational load evenly among threads. Each thread is assigned multiple rows with the equal number of floating operations.

4.2.4. Performance Estimation for a Recipe

To estimate how our algorithms would perform in practice, we analytically analyze their computation costs. As described, Heap SpGEMM requires logarithmic time to extract elements to the heap. The complexity of Heap SpGEMM is:

$$T_{heap} = \sum_{i=1}^{n} (\text{flop}(c_{i*}) \cdot \log nnz(a_{i*})) \quad (1)$$

On the other hand, Hash SpGEMM has $O(1)$ cost to explore its hash-table if there are no hash collisions. We introduce the collision factor c, which is the average number of probes needed to find or insert a key. When $c = 1$, no hash collision occurs during SpGEMM computation. In addition to this hash probing cost, Hash SpGEMM requires sorting, which takes $O(n \log n)$ time, if the application needs sorted output. The computational complexity of Hash SpGEMM is:

$$T_{hash} = \text{flop} \cdot c + \sum_{i=1}^{n} (nnz(c_{i*}) \cdot \log nnz(c_{i*})) \quad (2)$$

From (1) and (2), Hash SpGEMM tends to achieve superior performance to Heap SpGEMM when $nnz(c_{i*})$ or $\text{flop}(c_{i*})/nnz(c_{i*})$ is large. Denser input matrices make output matrix denser too. Also, the multiplication of input matrices with regular non-zero patterns outputs a regular matrix, and in that case, $\text{flop}(c_{i*})/nnz(c_{i*})$ is large. Thus, we can guess that Hash becomes a better choice when the input matrices are dense or have regular structures.

5. Experimental Setup

5.1. Input Types

We use two types of matrices for the evaluation. We generate synthetic matrices using matrix generator, and take matrices from SuiteSparse Matrix Collection [29]. For the evaluation of unsorted output, the column indices of input matrices are randomly permuted. We use 26 sparse matrices used in [25, 6, 21]. The matrices are listed in Table 2.

We use R-MAT [30], the recursive matrix generator, to generate two different non-zero patterns of synthetic matrices represented as ER and G500. ER matrix represents Erdős-Rényi random graphs, and G500 represents graphs with power-law degree distributions used for Graph500 benchmark. These matrices are generated with R-MAT seed parameters; a=b=c=d=0.25 for ER matrix and a=0.57, b=c=0.19, d=0.05 for G500 matrix. A scale m matrix represents 2^m-by-2^m. The *edge factor* parameter for the generator is the average number of non-zero elements per row (or column) of the matrix. In other words, it is the ratio of nnz to n.

Table 2: Matrix data used in our experiments (all numbers are in millions)

Matrix	n	$nnz(A)$	flop(A^2)	$nnz(A^2)$
2cubes_sphere	0.101	1.65	27.45	8.97
cage12	0.130	2.03	34.61	15.23
cage15	5.155	99.20	2,078.63	929.02
cant	0.062	4.01	269.49	17.44
conf5_4-8x8-05	0.049	1.92	74.76	10.91
consph	0.083	6.01	463.85	26.54
cop20k_A	0.121	2.62	79.88	18.71
delaunay_n24	16.777	100.66	633.91	347.32
filter3D	0.106	2.71	85.96	20.16
hood	0.221	10.77	562.03	34.24
m133-b3	0.200	0.80	3.20	3.18
mac_econ_fwd500	0.207	1.27	7.56	6.70
majorbasis	0.160	1.75	19.18	8.24
mario002	0.390	2.10	12.83	6.45
mc2depi	0.526	2.10	8.39	5.25
mono_500Hz	0.169	5.04	204.03	41.38
offshore	0.260	4.24	71.34	23.36
patents_main	0.241	0.56	2.60	2.28
pdb1HYS	0.036	4.34	555.32	19.59
poisson3Da	0.014	0.35	11.77	2.96
pwtk	0.218	11.63	626.05	32.77
rma10	0.047	2.37	156.48	7.90
scircuit	0.171	0.96	8.68	5.22
shipsec1	0.141	7.81	450.64	24.09
wb-edu	9.846	57.16	1,559.58	630.08
webbase-1M	1.000	3.11	69.52	51.11

5.2. Experimental Environment

We evaluate the performance of SpGEMM on a single node of the Cori supercomputer at NERSC. Cori system consists of two partitions; one is Intel Xeon Haswell cluster (Haswell), and another is Intel KNL cluster. We use nodes from both partitions of Cori. Details are summarized in Table 3. Each performance number in the following part is the average of ten executions.

The Haswell and KNL processors provide hyperthreading with 2 or 4 threads for each core. We set the number of threads as 68, 136, 204 or 272 for KNL, and 32 or 64 for Haswell. For the evaluation of Kokkos, we set 128 or 256 threads instead of 136 or 272 threads whenever the execution with Kokkos fails on non-powers of two threads. We show the result with the best thread count. We set "quadrant" cluster mode, and mainly "Cache" memory mode. To select DDR4 or MCDRAM with "Flat" memory mode, we use "numactl -p". The thread affinity is set as "KMP_AFFINITY='granularity=fine',scatter".

5.3. Preliminary Evaluation

5.3.1. DDR vs MCDRAM

We examine the benefit of using MCDRAM over DDR memory by squaring G500 matrices with or without using MCDRAM on KNL. Figure 6 shows the speedup attained with the Cache mode against the Flat mode on DDR for various matrix densities. We observe that Hash SpGEMM algorithms can be benefitted, albeit moderately, from MCDRAM when denser matrices are multiplied. This observation is consistent with the benchmark shown in Figure 4. The limited benefit stems from the fact that SpGEMM frequently requires indirect

Table 3: Overview of Evaluation Environment (Cori system)

	Haswell cluster	KNL cluster
CPU	**Intel Xeon Processor E5-2698 v3**	**Intel Xeon Phi Processor 7250**
#Sockets	2	1
#Cores/socket	16	68
Clock	2.3GHz	1.4GHz
L1 cache	32KB/core	32KB/core
L2 cache	256KB/core	1MB/tile
L3 cache	40MB per socket	-
Memory		
DDR4	128GB	96GB
MCDRAM	-	16GB
Software		
OS	SuSE Linux Enterprise Server 12 SP3	
Compiler	Intel C++ Compiler (icpc) ver18.0.0	
Option	-g -O3 -qopenmp	

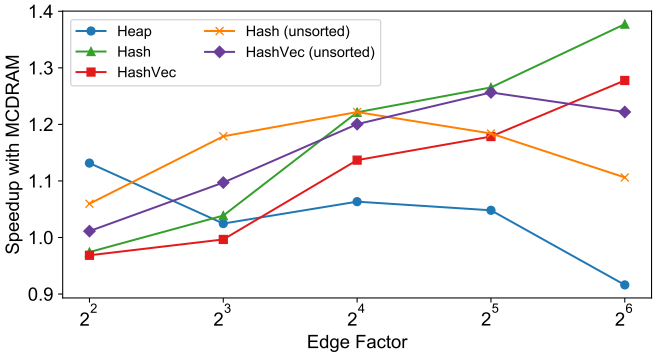


Figure 6: Speedups attained with the use of Cache mode on KNL compared to Flat mode on DDR4. G500 (scale 15) matrices are used with different edge factors.

fine-grained memory accesses often as small as 8 bytes. On denser matrices, MCDRAM can still bring benefit from contiguous memory accesses of input matrices. By contrast, Heap SpGEMM is not benefitted from high-bandwidth MCDRAM because of its fine-grained memory accesses. The performance of Heap SpGEMM even degrades when edge factor is 64 at which point the memory requirement of Heap SpGEMM surpasses the capacity of MCDRAM.

5.3.2. Advantage of Performance Optimization on KNL for SpGEMM

We examined performance difference between OpenMP scheduling and ways to allocate memory. Figure 7 shows the performance of Heap SpGEMM for squaring G500 matrices with edge factor 16. When simply parallelizing SpGEMM by row, we cannot achieve higher performance because of load imbalance with static scheduling or expensive scheduling overhead with dynamic/guided scheduling. On the other hand, our light-weight load-balancing thread scheduling scheme based on the number of floating operations (flop) with static scheduling works well on SpGEMM. For larger inputs, Heap SpGEMM temporally requires large memory usage, whose deallocation causes performance degradation. Memory management scheme by each thread reduces the overhead of

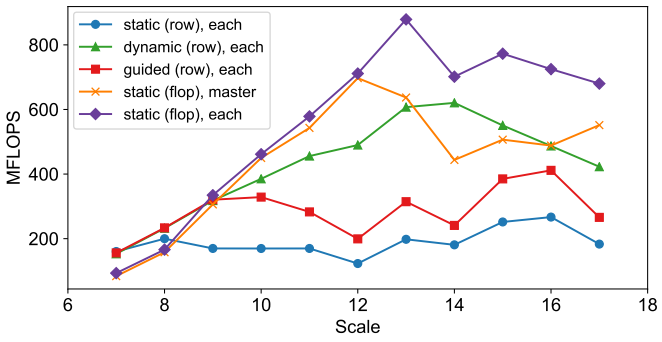


Figure 7: Performance of Heap SpGEMM scaling with size of G500 inputs on KNL with Cache mode. First argument in legend is about parallelism, and second is about memory allocation/deallocation scheme.

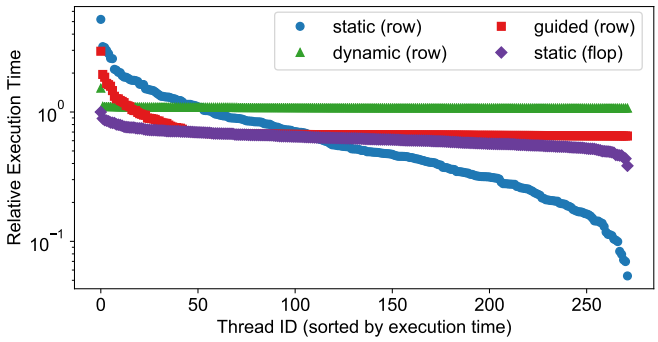


Figure 8: Execution time of each thread for SpGEMM with G500 matrix (scale = 13, edge factor=16)

memory deallocation for temporal memory use compared to that by one master thread, and keeps high performance on larger size input matrices. We are also showing more detailed thread behavior in Figure 8. We evaluate the execution time of each thread with static, dynamic or guided option parallelized by row and with static option parallelized by flop on 272 threads. The input matrix is G500 with scale=13 and edge factor=16. Since we focus on the load-balance in Figure 8, the memory management scheme by one master thread (static(flops), master) in Figure 7 is excluded. The execution time of the busiest thread in "static (flop)" is set as 1, and the relative execution time of each thread to the baseline is plotted in descending order. The scheduling algorithms, which are parallelizing SpGEMM by row, assign the task to each thread without considering the actual computational cost. As a result, the "static (row)" shows terrible load imbalance, and the performance of "static (row)" is determined by the longest run among all threads even though some threads finish their tasks in 1% of the execution time of the slowest thread. The execution time of almost all threads with "dynamic (row)" algorithm is about the same, while the execution time is longer on the whole due to the overhead of thread scheduling. The "guided (row)" algorithm can reduce the overhead of thread scheduling compared to "dynamic (row)" algorithm, and achieves better load balance than "static (row)". However, some of threads still take much more execution time compared to other threads, that we can see the same issue in the "static (row)". Our approach parallelized by flop with static scheduling shows not only good load balance but also less overhead of thread scheduling.

6. Experimental Results

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. To provide a comprehensive and fair assessment, we evaluate SpGEMM codes under several different scenarios. For the case where input and output matrices are sorted, we evaluate MKL, Heap and Hash/HashVector, and for the case where they are unsorted, we evaluate MKL, MKL-inspector, KokkosKernels (with 'kkmem' option) and Hash/HashVector.

6.1. Squaring a matrix

Multiplying a sparse matrix by itself is a well-studied SpGEMM scenario. Markov clustering is an example of this case, which requires A^2 for a given doubly-stochastic similarity matrix. We evaluate this case extensively, under using real and synthetically generated data. For synthetic data, we provide experiments with varying density (for a fixed sized matrix) and with varying size (for a fixed density).

6.1.1. Scaling with Density

Figure 9 shows the result of scaling with density. When output is sorted, MKL's performance degrades with increasing density. When the output is not sorted, increased density generally translates into increased performance. The performance of all codes except MKL increases significantly as the ER matrices get denser until edge factor is 16. On the other hand, the evaluation result on G500 matrices with edge factor 32 shows further performance boost. In SpGEMM operation between ER matrices with edge factor 32, the average non-zero elements per row of output matrix or hash table size for each row is about 32^2. As a result, the memory accesses to the accumulators appear to cause L1 cache misses, degrading the performance of SpGEMM compared to the case between matrices with edge factor 16. For G500 matrices, Hash shows superior performance on KNL while HashVector is the best performer on Haswell.

6.1.2. Scaling with Input Size

Evaluation is running on ER and G500 matrices with scaling the size from 7 to 20 or 17 respectively. The edge factor is fixed as 16. Figure 10 shows the results on KNL. MKL family with unsorted output shows good performance for ER matrices with small scale. However, for large scale matrices, MKL goes down, and Heap and Hash outperform. Especially, Hash and HashVector keep high performance even for large scale matrices. When the scale is about 13, the performance gap between sorted and unsorted is large, and it becomes smaller when the scale is getting large. This is because the cost of computation with hash-table or heap becomes larger, and the advantage of removing sorting phase becomes relatively smaller. For G500 matrices, whose non-zero elements of each row are skewed, the performance of

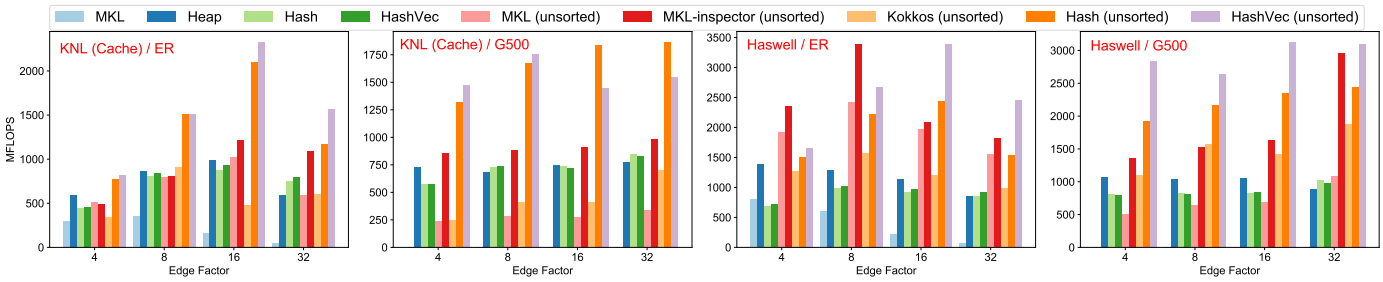


Figure 9: Scaling with increasing density (scale 16) on KNL and Haswell

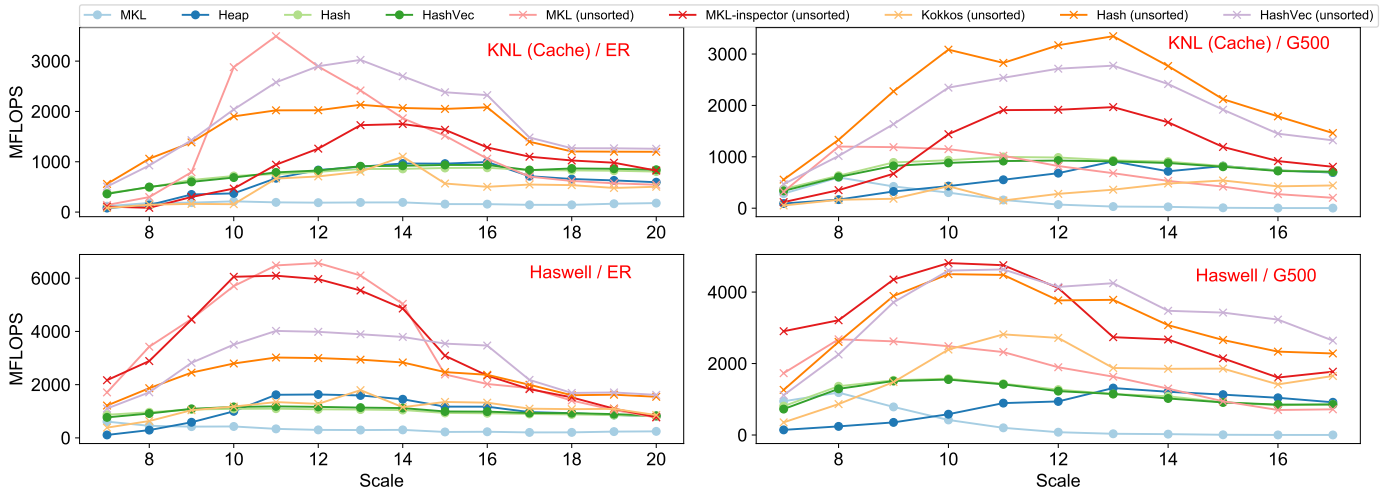


Figure 10: Scaling with size on KNL and Haswell with edge factor 16

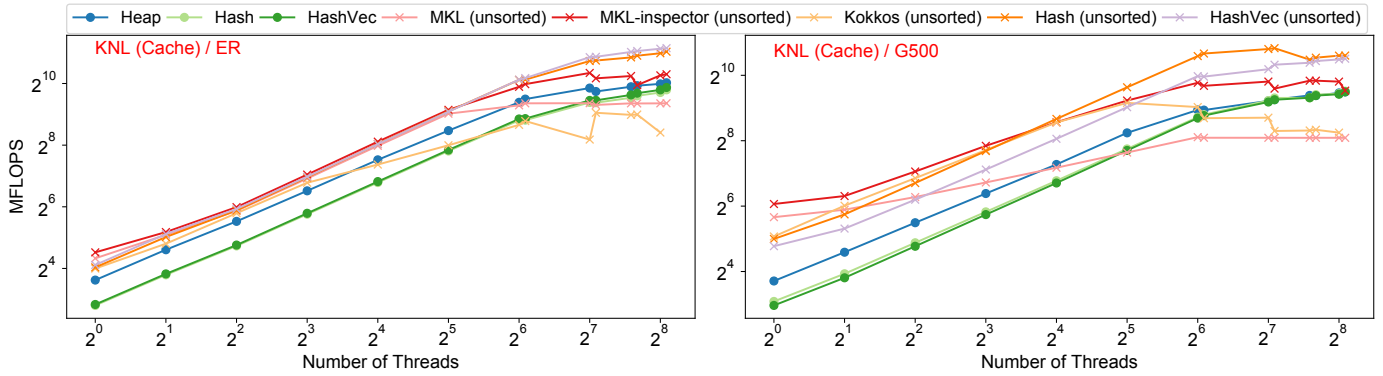


Figure 11: Strong scaling with thread count on KNL with ER (left) and G500 inputs (right). Data used is of scale 16 with edge factor 16

MKL is terrible even if its output is unsorted. Since there is no issue about load-balance in Heap and Hash kernels, they show stable performance as ER matrices.

6.1.3. *Comparison with CSparse*

CSparse is the fastest single threaded code to the best of our knowledge and it also is the code that powers Matlab's sparse implementation. Figure 14 shows the result of SpGEMM (A^2) on single thread. The matrixs scale is 16, and its edge factor is also 16. Since the non-zero elements in each column of output matrix with CSparse are unsorted, the figure includes only the result with the kernels for unsorted output. The performance of CSparse is about same as that of MKL. Indeed, CSparse shows better performance than Heap and Hash on single thread. However, CSparse does not support multi-thread. On the other hand, our kernels can take advantage of multi-thread.

6.1.4. *Scaling with Thread Count*

Figure 11 shows the scalability analysis of KNL on ER and G500 matrices with scale=16 and edge factor=16. Each kernel is executed with 1, 2, 4, 8, 16, 32, 64, 68, 128, 136, 192, 204, 256 or 272 threads. We do not show the result of MKL with sorted output since it takes much longer execution time compared to other kernels. All kernels show good scalability until around 64 threads, but MKL with unsorted output has no

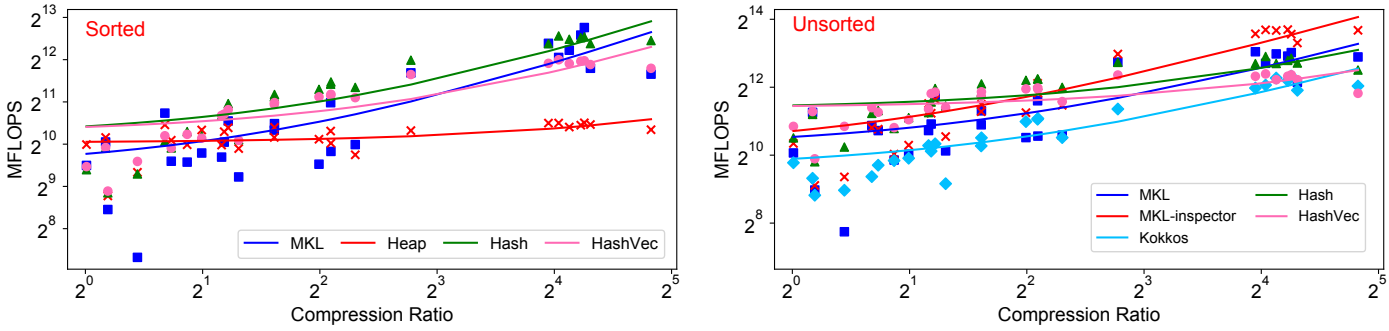


Figure 12: Scaling with compression ratio of SuiteSparse matrices on KNL. The algorithms that operate on sorted matrices (both input & output) are on the left and those that operate on unsorted matrices are on the right.

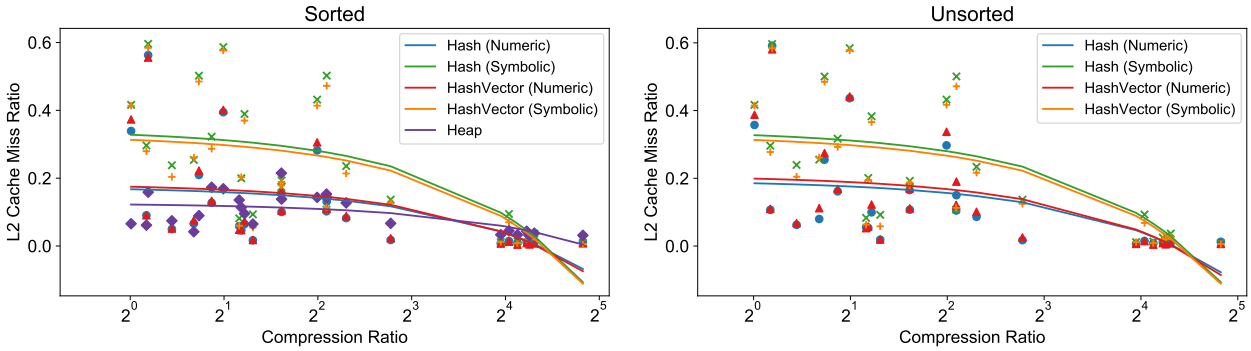


Figure 13: L2 cache miss ratio scaling with compression ratio of SuiteSparse matrices on KNL

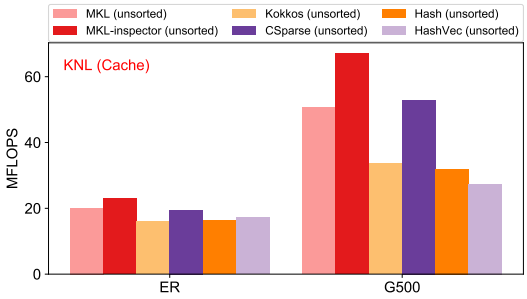


Figure 14: Comparison result of CSparse on single thread

improvement over 68 threads. On the other hand, Heap and Hash/HashVector get further improvement over 64 threads. This is because our thread scheduling scheme based on the number of floating operations in each row works well and brings better load-balancing.

6.1.5. Sensitivity to Compression Ratio on Real Matrices

We evaluate SpGEMM performance on 26 real matrices listed in Table 2 on KNL. Figure 12 shows the result with sorted output and unsorted output respectively in ascending order of compression ratio (= flop / number of non-zero elements of output). The compression ratio is getting larger when the density of the matrix is large. The compression ratio can be the proxy of memory access locality and we can easily predict the performance boost of keeping the output sorted by using the compression ratio of target SpGEMM operation and the performance model proposed in Section 4.2.4. Lines in the graph are linear fitting for each kernel. Firstly, we discuss the result with sorted matrices. The performance of Heap is stable regardless of compression ratio while MKL gets better performance with higher compression ratio. The matrices about graph processing with low compression ratio cause load imbalance and performance degradation on MKL. In contrast, Hash outperforms MKL on most of matrices, and shows high performance independent from compression ratio. For unsorted matrices, we add KokkosKernels to the evaluation, but it underperforms other kernels in this test. The performance of Hash SpGEMM is best for the matrices with low compression ratio and underperforms on high compression ratio matrices. MKL-inspector shows significant improvement especially for the matrices with high compression ratio.

We are also showing L2 cache miss ratio of Heap, Hash and HashVector algorithms on same matrix data in Figure 13. We use likwid [31] to profile the L2 cache behavior, and evaluate only SpGEMM part of each algorithm. The cache miss ratio of Hash or HashVector decreases with increased compression ratio both in sorted and unsorted cases. This trend is same as the performance trend showed in Figure 12. When the compression ratio is high, the entries in same hash-table is frequently accessed. Since our Hash and HashVector algorithms limit the size of hash-table by the number of flop, the cache hit ratio is likely to increase when the hash-table is repeatedly accessed. As a result, Hash and HashVector can

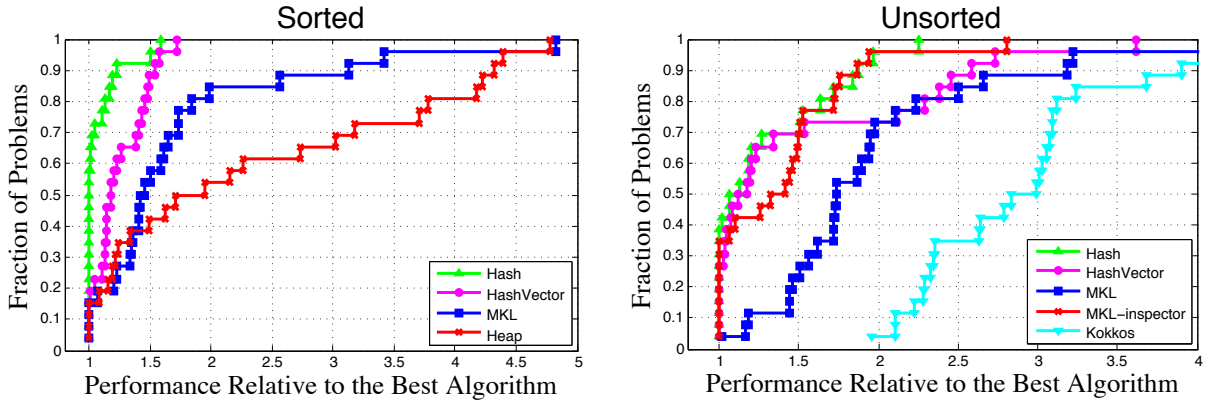
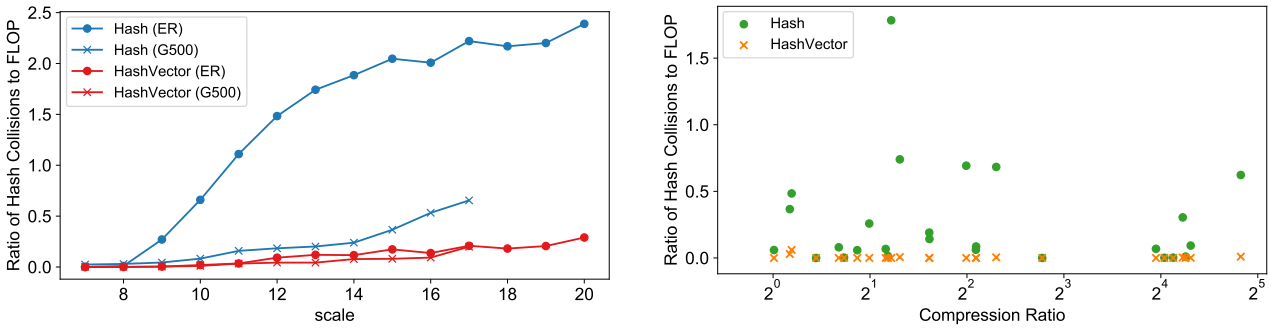


Figure 15: Performance profiles of SuiteSparse matrices on KNL using sorted (left) and unsorted (right) algorithms.



(a) Synthetic matrices with edge factor = 16

(b) SuiteSparse matrices scaling with compression ratio

Figure 16: Ratio of hash collisions in SpGEMM on KNL

reduce the ratio of L2 cache miss and achieve better performance on high compression ratio matrices. On the other hand, Heap algorithm shows low L2 cache miss ratios even on matrices with low compression ratio. Therefore, the ratio of L2 cache misses does not change significantly with compression ratio. This is why the performance of Heap algorithm is stable regardless of compression ratio as seen in Figure 12.

Comparing sorted and unsorted versions of algorithms that provide the flexibility of choosing either option, we see consistent performance boost of keeping the output sorted. In Hash SpGEMM, for example, sorting requires computation complexity equal to the second term of the equation (2) in section 4.2.4. This implies that the unsorted version of SpGEMM operation brings larger benefit on the matrix with lower compression ratio. Such performance boost with unsorted SpGEMM can be seen in other methods, and in particular, the geometric mean of the speedups achieved operating on unsorted data over all real matrices we have studied from the SuiteSparse collection on KNL is 1.62× for MKL, 1.68× for Hash, and 1.74× for HashVector. These performance numbers simply show the impact of keeping the output matrix unsorted.

6.1.6. Profile of Relative Performance of SpGEMM Algorithms

We compare the relative performance of different SpGEMM algorithms with performance profile plots [32]. To profile the relative performance of algorithms, the best performing algorithm for each problem is identified and assigned a relative score of 1. Other algorithms are scored relative to the best performing algorithm, with a higher value denoting inferior performance for that particular problem. For example, if algorithm A and B solve the same problem in 1 and 3 seconds, their relative performance scores will be 1 and 3, respectively. Figure 15 shows the profiles of relative performance of different SpGEMM algorithms for all 26 matrices from Table 2. Hash is clearly the best performer for sorted matrices as it outperforms all other algorithms for 70% matrices and its runtime is always within 1.6× of the best algorithm. Hash is followed by HashVector, MKL and Heap algorithms in decreasing order of overall performance. For unsorted matrices, Hash, HashVector and MKL-inspector all perform equally well for most matrices (each of them performs the best for about 40% matrices). They are followed by MKL and KokkosKernels, with the latter being the worst performer for unsorted matrices.

6.1.7. Reduction Ratio of Hash Collisions with HashVector

Our HashVector algorithm is designed to reduce the hash collisions by utilizing wide vector registers implemented in Intel KNL. Figure 16 shows the ratio of hash collisions to the flop in SpGEMM operation on synthetic matrices generated by R-MAT and SuiteSparse matrices, respectively. HashVector can largely reduce the ratio of hash collisions from Hash on ER matrices compared to G500 matrices. This gap of reduced

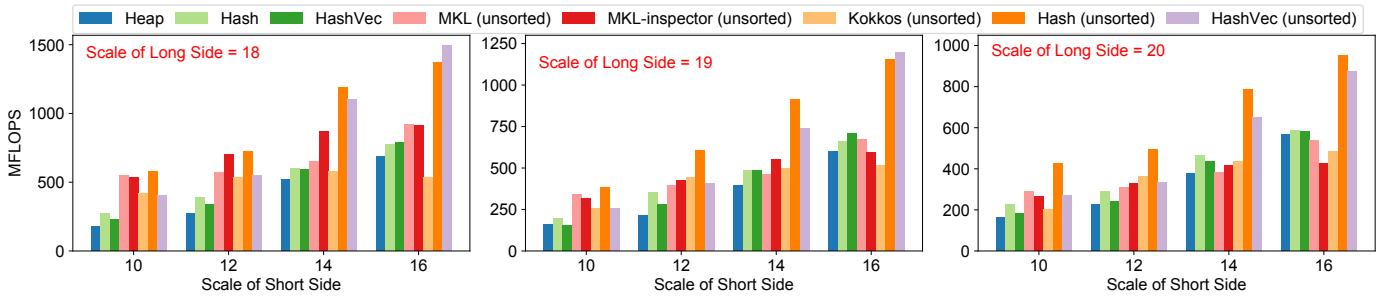


Figure 17: SpGEMM between square and tall-skinny matrices on KNL (scales 18, 19, and 20)

hash collisions between ER and G500 matrices exposes the difference of best algorithm between ER and G500 matrices. HashVector achieves better performance on ER matrices, while Hash is superior to HashVector on G500 matrices. Since the flop of SpGEMM with G500 matrix is larger than that of ER matrix, the performance degradation due to hash collisions is relatively small. On SuiteSparse matrices, HashVector algorithm causes very few hash collisions. However, Hash outperforms HashVector on most of matrices as Figure 12 shows. This is because the hash collisions in Hash algorithm are not so many and reducing hash collisions does not contributed the performance improvement. The increase of the instructions with HashVector dominates the performance of SpGEMM. As a result, HashVector with large reduction of hash collisions is best for ER matrices, and Hash outperforms HashVector on G500 matrices and SuiteSparse matrices.

6.2. *Square x Tall-skinny matrix*

Many graph processing algorithms perform multiple breadth-first searches (BFSs) in parallel, an example being Betweenness Centrality on unweighted graphs. In linear algebraic terms, this corresponds to multiplying a square sparse matrix with a tall-skinny one. The left-hand-side matrix represents the graph and the right-hand-side matrix represent the stack of frontiers, each column representing one BFS frontier. In the memory-efficient implementation of the Markov clustering algorithm [3], a matrix is multiplied with a subset of its column, representing another use case of multiplying a square matrix with a tall-skinny matrix. In our evaluations, we generate the tall-skinny matrix by randomly selecting columns from the graph itself. Figure 17 shows the result of SpGEMM between square and tall-skinny matrices. A square matrix is generated with scale as 18, 19 or 20. The scale of short side of tall-skinny matrix is set as 10, 12, 14 or 16, and a long side is same as the square matrix. The non-zero pattern of generated matrix is G500 with edge factor=16. The result of square x tall-skinny follows that of A^2 (upper right in Figure 10). Both for sorted and unsorted cases, Hash or HashVec is the best performer.

6.3. *Triangle counting*

We also evaluate the performance of SpGEMM used in triangle counting [6]. The original input is the adjacency matrix of an undirected graph. For optimal performance in triangle counting, we reorder rows with increasing number of nonzeros. The algorithm then splits the reordered matrix A as $A = L + U$, where L is a lower triangular matrix and U is an upper triangular matrix. We evaluate the SpGEMM performance of the next step, where $L \cdot U$ is computed to generate all wedges. After preprocessing the input matrix, we compute SpGEMM between the lower triangular matrix L and the upper triangular matrix U. Figure 18 shows the result with sorted output respectively in ascending order of compression ratio on KNL. Lines in the graph are linear fitting for each kernel. Basically, the result shows similar performance trend to that of A^2. Hash and HashVector generally overwhelm MKL for any compression ratio. Since the triangle counting preprocesses the input matrix before SpGEMM operation, the characteristic of input matrix such as block structure is lost. As the result, the compression ratio tends to decrease compared to the case of A^2, and sorting output takes larger percentage of execution time. While the performance of MKL and Hash for triangle counting degrade compared to the case of computing A^2, the performance of Heap is consistent regardless of compression ratio as our performance estimation in Section 4.2.4 shows. Heap performs the best for inputs with low compression ratios.

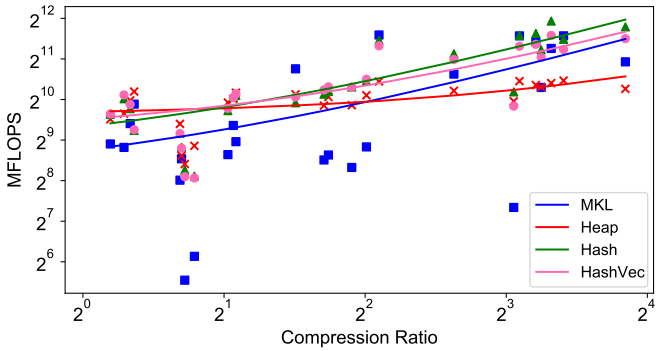


Figure 18: The performance of SpGEMM between L and U triangular matrices when used to count triangles on KNL

6.4. *Empirical Recipe for SpGEMM on KNL*

From our evaluation results, best algorithm is different by target use case and inputs. We summarize the recipe to choose best algorithm for SpGEMM on KNL in Table 4. The recipe for real data is based on compression ratio, which effects the dominant code. For low compression ratios, especially $L \cdot U$

Table 4: Summary of best SpGEMM algorithms on KNL

(a) Real data specified by compression ratio (CR)

		High CR (> 2)	Low CR (≤ 2)
A x A	Sorted	Hash	Hash
	Unsorted	MKL-inspector	Hash
L x U	Sorted	Hash	Heap

(b) Synthetic data specified by sparsity (edge factor, EF) and non-zero pattern

		Sparse (EF ≤ 8)		Dense (EF > 8)	
		Uniform	Skewed	Uniform	Skewed
A x A	Sorted	Heap	Heap	Heap	Hash
	Unsorted	HashVec	HashVec	HashVec	Hash
TallSkinny	Sorted	-	Hash	-	HashVec
	Unsorted	-	Hash	-	Hash

where output is sparser, Heap shines. In the other cases, Hash and MKL-inspector dominate the high-compression ratio scenarios because Hash has better locality of memory access to hash-table on high compression ratio matrices and can keep low cache miss ratio. On synthetic data, our hash-table-based implementations dominate others for almost cases, and Heap works well for sparser matrices or uniform data. We note that A^2 for uniform input matrices shows low compression ratio. Also, the performance gap between Hash and HashVector comes from how many hash collisions occur in each algorithm. This empirical recipe was already predicted by our analysis in Section 4.2.4.

7. Impact on the HipMCL clustering software

HipMCL [3] is a high-performance distributed-memory implementation of the Markov cluster algorithm, which itself is commonly used for clustering protein similarity networks. The algorithm performs concurrent random walks from all vertices, which is implemented by squaring the sparse similarity matrix. In the context of protein similarity networks, compression ratios are often high in the first few iterations but they drop rapidly as the algorithm starts to converge. An overwhelming majority of the time is spent during those first few iterations. The current implementation of HipMCL uses the Heap SpGEMM implementation within Combinatorial BLAS [33]. We expect that hash algorithm can provide a significant performance boost because most of the time is spent in iterations with high compression ratios.

The results, shown in Table 5, support our hypothesis. The Hash algorithm is up to 10X faster than the Heap algorithm for the SpGEMM time alone. The Hash SpGEMM algorithm accelerates the whole HipMCL pipeline by up to a factor of 2.6X. These impressive performance improvements have significant positive implications because HipMCL is often used in really large datasets that take several hours to cluster on O(1000) of nodes.

In addition to testing the Hash algorithm, we also implemented a hybrid algorithm that chooses Hash or Heap SpGEMM implementation depending on the compression ratio. Instead of making a single coarse decision per iteration, our Hybrid algorithm estimates the compression ratios for each column and chooses a different implementation per column.

Table 5: Execution time of HipMCL application on single KNL node using Heap, Hash and Hybrid algorithms [sec]

Dataset	Target	Heap	Hash	Hybrid
Viruses	SpGEMM	20.38	5.14	5.13
	Total	62.59	47.87	47.77
Archaea	SpGEMM	7700.50	717.07	716.42
	Total	11535.00	4550.22	4539.07
Eukarya	SpGEMM	18448.10	1941.93	1964.34
	Total	26717.00	10241.60	10284.80

Specifically, if the compression ratio of the column is larger than 2, then it uses the Hash algorithm. Otherwise, it uses the Heap algorithm. This threshold of 2 is motivated by the experimental data shown earlier in Figure 12. The hybrid algorithm performs only marginally better than the Hash algorithm. This has two reasons. First, an overwhelming majority of the time is spent in high compression ratio computations, thus there is limited benefit that can be gained by choosing a different algorithm in the other regime. Second, Hash algorithm is only slightly worse than the Heap algorithm for low compression ratio scenarios.

However, we believe that this Hybrid approach might be more valuable in other iterative applications that thread a finer line between low and high compression ratio computations. One potential candidate is the multi-source betweenness centrality implementation [2, 34] that uses SpGEMM as its computational workhorse.

8. Conclusions

We studied the performance of computing the multiplication of two sparse matrices on Intel architectures. This primitive, known as SpGEMM, has recently gained attention in the GPU community, but there has been relatively less work on other accelerators. We have tried to fill that gap by evaluating publicly accessible implementations, including those in proprietary libraries. From architecture point of view, we develop the optimized Heap and Hash SpGEMM algorithms for Intel architectures. Performance evaluation shows that our optimized SpGEMM algorithms largely outperform Intel MKL and Kokkos-kernel.

Our work provides multiple recipes. One is for the implementers of new algorithms on highly-threaded x86 architectures. We have found that the impact of memory allocation and deallocation to be significant enough to warrant optimization as without them SpGEMM performance does not scale well with increasing number of threads. We have also uncovered the impact of MCDRAM for the SpGEMM primitives. When the matrices are sparser than a threshold (≈ 4 nonzeros on average per row), the impact of MCDRAM is minimal because in that regime the computation becomes close to latency bound. On the other hand, MCDRAM shines as matrices get denser because then SpGEMM becomes primarily bandwidth bound and can take advantage of the higher bandwidth available on MCDRAM. The second recipe is for the users. Our results show that different codes dominate on different inputs. We clarify which SpGEMM algorithm

works well by building performance model and showing detailed performance results and profiling data. For example, MKL is a perfectly reasonable option for small matrices with uniform nonzero distributions. However, our heap and hash-table-based implementations dominate others for larger matrices. Similarly, the compression ratio also affects the dominant code. Based on the recipe, the hash-table-based algorithm is applied to HipMCL application and provides significant performance boost. Our results also highlight the benefits of leaving matrices (both inputs and output) unsorted whenever possible as the performance savings are significant for all codes that allow both options. Finally, this optimization strategy for acquiring these two recipes is beneficial for optimization of SpGEMM on future architectures.

Acknowledgement

This work was partially supported by JST CREST Grant Number JPMJCR1303 and JPMJCR1687, and performed under the collaboration with DENSO IT Laboratory, inc., and performed under the auspices of Real-World Big-Data Computation Open Innovation Laboratory, Japan. The Lawrence Berkeley National Laboratory portion of this research is supported by the DoE Office of Advanced Scientific Computing Research under contract DE-AC02-05CH11231. This research was partially supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

Code

Our implementations of the SpGEMM algorithms on Intel KNL and multi-core architectures are available at https://bitbucket.org/YusukeNagasaka/mtspgemmlib.

References

[1] J. R. Gilbert, S. Reinhardt, V. B. Shah, High performance graph algorithms from parallel sparse matrices, in: PARA, 2007, pp. 260–269.

[2] A. Buluç, J. R. Gilbert, The Combinatorial BLAS: Design, implementation, and applications, IJHPCA 25 (4) (2011) 496–509.

[3] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, A. Buluç, HipMCL: A high-performance parallel implementation of the markov clustering algorithm for large-scale networks, Nucleic acids research.

[4] U. N. Raghavan, R. Albert, S. Kumara, Near linear time algorithm to detect community structures in large-scale networks, Physical review E 76 (3) (2007) 036106.

[5] V. B. Shah, An interactive system for combinatorial scientific computing with an emphasis on programmer productivity, Ph.D. thesis, University of California, Santa Barbara (June 2007).

[6] A. Azad, A. Buluç, J. Gilbert, Parallel triangle counting and enumeration using matrix algebra, in: IPDPSW, 2015.

[7] S. R. Agrawal, C. M. Dee, A. R. Lebeck, Exploiting accelerators for efficient high dimensional similarity search, in: PPoPP, ACM, 2016.

[8] G. He, H. Feng, C. Li, H. Chen, Parallel SimRank computation on large graphs with iterative aggregation, in: SIGKDD, ACM, 2010.

[9] G. Ballard, C. Siefert, J. Hu, Reducing communication costs for sparse matrix multiplication within algebraic multigrid, SIAM Journal on Scientific Computing 38 (3) (2016) C203–C231.

[10] J. S. Mueller-Roemer, C. Altenhofen, A. Stork, Ternary sparse matrix representation for volumetric mesh subdivision and processing on GPUs, in: Computer Graphics Forum, Vol. 36, 2017.

[11] N. Bock, M. Challacombe, L. V. Kalé, Solvers for O(N) electronic structure in the strong scaling limit, SIAM Journal on Scientific Computing 38 (1) (2016) C1–C21.

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

[13] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, S. Williams, Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication, SIAM Journal on Scientific Computing 38 (6) (2016) C624–C651.

[14] Y. Nagasaka, A. Nukada, S. Matsuoka, High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA Pascal GPU, in: ICPP, IEEE, 2017, pp. 101–110.

[15] J. R. Gilbert, C. Moler, R. Schreiber, Sparse matrices in MATLAB: Design and implementation, SIAM Journal on Matrix Analysis and Applications 13 (1) (1992) 333–356.

[16] P. D. Sulatycke, K. Ghose, Caching-efficient multithreaded fast multiplication of sparse matrices, in: IPPS/SPDP, IEEE, 1998.

[17] K. Matam, S. R. K. B. Indarapu, K. Kothapalli, Sparse Matrix-Matrix Multiplication on Modern Architectures, in: HiPC, IEEE, 2012.

[18] S. Dalton, L. Olson, N. Bell, Optimizing sparse matrix—matrix multiplication for the GPU, ACM Transactions on Mathematical Software (TOMS) 41 (4) (2015) 25.

[19] P. N. Q. Anh, R. Fan, Y. Wen, Balanced hashing and efficient GPU sparse general matrix-matrix multiplication, in: ICS, ACM, New York, NY, USA, 2016.

[20] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, U. Naumann, GPU-accelerated sparse matrix-matrix multiplication by iterative row merging, SIAM Journal on Scientific Computing 37 (1) (2015) C54–C71.

[21] M. Deveci, C. Trott, S. Rajamanickam, Performance-portable sparse matrix-matrix multiplication for many-core architectures, in: IPDPSW, IEEE, 2017, pp. 693–702.

[22] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, P. Dubey, Parallel efficient sparse matrix-matrix multiplication on multicore platforms, in: ISC, Springer, 2015, pp. 48–57.

[23] T. A. Davis, Direct methods for sparse linear systems, SIAM, 2006.

[24] T. A. Davis, private communication.

[25] W. Liu, B. Vinter, An efficient GPU general sparse matrix-matrix multiplication for irregular data, in: IPDPS, IEEE, 2014, pp. 370–381.

[26] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jüngel, S. Selberherr, ViennaCL—linear algebra library for multi-and many-core architectures, SIAM Journal on Scientific Computing 38 (5) (2016) S412–S439.

[27] J. D. McCalpin, Stream: Sustainable memory bandwidth in high performance computers, Tech. rep., University of Virginia (1991-2007).

[28] K. A. Ross, Efficient hash probes on modern processors, in: ICDE, IEEE, 2007, pp. 1297–1301.

[29] T. A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Transactions on Mathematical Software (TOMS) 38 (1) (2011) 1.

[30] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-MAT: A recursive model for graph mining, in: Proceedings of the SIAM International Conference on Data Mining, SIAM, 2004, pp. 442–446.

[31] J. Treibig, G. Hager, G. Wellein, Likwid: A lightweight performance-oriented tool suite for x86 multicore environments, in: Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, 2010.

[32] E. D. Dolan, J. J. Moré, Benchmarking optimization software with performance profiles, Mathematical programming 91 (2) (2002) 201–213.

[33] A. Buluç, J. R. Gilbert, The Combinatorial BLAS: Design, implementation, and applications, The International Journal of High Performance Computing Applications 25 (4) (2011) 496–509.

[34] A. Buluç, T. Mattson, S. McMillan, J. Moreira, C. Yang, Design of the GraphBLAS API for C, in: IEEE Workshop on Graph Algorithm Building Blocks, IPDPSW, 2017.