High-performance sparse matrix-matrix products on Intel KNL and multicore architectures

Yusuke Nagasaka*
Tokyo Institute of Technology
Tokyo, Japan

Satoshi Matsuoka†
RIKEN Center for Computational Science
Kobe, Japan

Ariful Azad
Lawrence Berkeley National Laboratory
Berkeley, California, USA

Aydın Buluç
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 multi- and 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. 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.

CCS CONCEPTS

• **Computing methodologies** → *Massively parallel algorithms*;

KEYWORDS

Sparse matrix, SpGEMM, Intel KNL

*nagasaka.y.aa@m.titech.ac.jp
†Also with Tokyo Institute of Technology, Department of Mathematical and Computing Sciences.

ICPP '18 Comp, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6523-9/18/08…$15.00
https://doi.org/10.1145/3229710.3229720

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 [17], betweenness centrality [8], Markov clustering [5], label propagation [28], peer pressure clustering [31], clustering coefficients [4], high-dimensional similarity search [1], and topological similarity search [20] 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 [6], volumetric mesh processing [24], and linear-scaling electronic structure calculations [7].

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 [19] where a row of the output matrix is constructed by accumulating a subset of rows of the second input matrix (see Figure 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 [3], hash [26], and sparse accumulator (SPA) [16].

Recently, researchers have developed parallel heap-, hash-, and SPA-based SpGEMM algorithms for shared-memory platforms [2, 10, 18, 22, 32]. 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 answers 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 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 multicore and manycore architectures.

First, 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 resulted in efficient heap-based and hash-table-based SpGEMM algorithms that outperform state-of-the-art SpGEMM libraries including Intel MKL and Kokkos-Kernels [14] for many practical problems.

Second, 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 [14, 27]. 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.

Third, 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 [11, 12] 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.

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. Therefore, 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 [19]. 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.

RowWise_SpGEMM(C, A, B)

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

Figure 1: Pseudo code of Gustavson's Row-wise SpGEMM algorithm. 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.

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 an 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 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) [16]. 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. [27] achieved this by partitioning the data structure of B by columns.

Sulatycke and Ghose [32] 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 [21, 26]. These algorithms first 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 [30] implementation, which was first described for GPUs [18], iteratively merges sorted lists, similar to merge sort. KokkosKernels implementation [14], which we also include in our evaluation, uses a multi-level hash map data structure.

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 [3, 26]. 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

3 ALGORITHMS OPTIMIZATIONS FOR OUR TARGET ARCHITECTURES

Our experiments target Intel Xeon and Xeon Phi architectures. To extract the best performance from these architectures, we perform several optimizations in our SpGEMM algorithms covering light-weight thread scheduling with load-balancing, and inexpensive memory allocation and deallocation schemes. We conducted some preliminary experiments to tune optimization parameters. The effect of these architecture-specific optimizations on SpGEMM algorithm is clearly revealed in Section 4.3.1. Next, we show the optimization schemes for hash-table-based SpGEMM, which is proposed for GPU [26], and heap-based shared-memory SpGEMM algorithms [3]. Additionally, we extend the Hash SpGEMM with utilizing vector registers of Intel Xeon or Xeon Phi. Finally, we uncover the characteristic of MCDRAM based on the memory access pattern of SpGEMM. These microbenchmarks are especially valuable for Intel KNL that offers massive parallelism and has a 16GB high bandwidth memory (MCDRAM) along with traditional DDR memory. Details of evaluation environment are summarized in Table 3.

RowsToThreads(*offset*, *A*, *B*)

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

Figure 2: Load-balanced Thread Assignment

3.1 Light-weight Load-balancing Thread Scheduling Scheme

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 switch to dynamic scheduling in the later part. Here, we experimentally evaluate the cost of these three scheduling options on KNL processors. [1] In load-balanced situation with a large number of iterations, static scheduling takes very little scheduling overhead compared to dynamic scheduling on both Haswell and KNL, as expected. The guided scheduling is also as expensive as dynamic scheduling, especially on the KNL processor. Based on these evaluations, we opt to use static scheduling in our SpGEMM algorithms. To achieve good load-balance with static scheduling, the bundle of rows should be assigned to threads with equal computation complexity. Figure 2 shows how to assign rows to threads. First, 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 14 finds the minimum *id* such that $vec[id]$ is larger than or equal to *value*. Each of these three operations can be executed in parallel.

3.2 Inexpensive Memory Allocation and 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. To contrast this "single" memory management scheme, we considered a "parallel" approach where each thread independently allocates/deallocates equal portion of the total requested memory and accesses only its own allocated memory space. We examined two ways to allocate or deallocate memory; new/delete of C++ and

[1] Detailed benchmark results on scheduling costs can be found in Section 3.1 of our longer technical report [25]

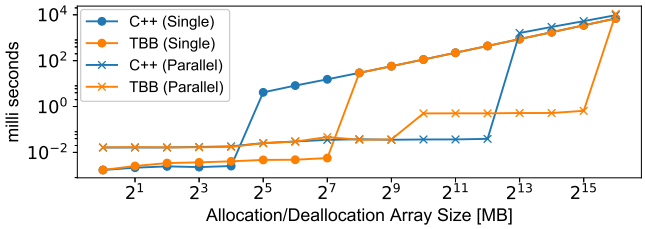

Figure 3: Cost of deallocation on KNL

scalable_malloc/_free provided by Intel TBB (Thread Building Block). Figure 3 shows the results of "single" deallocation and "parallel" deallocation with 256 threads on KNL. All "single" allocators have extremely high cost for large memory: over 100 milliseconds for the deallocation of 1GB memory space. The "parallel" deallocation for large memory chunks is much cheaper than "single" deallocation. The cost of "parallel" deallocation suddenly rises at 8GB (C++) or 64GB (TBB), where each thread allocates 32MB or 256MB, respectively. These thresholds match those of "single" deallocation. On the other hand, the cost of "parallel" deallocation for small memory space becomes larger than "single" deallocation since "parallel" 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 "single" and "parallel" memory allocation/deallocation to simply have performance gain.

3.3 Symbolic and Accumulation

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 AVX-512 or AVX2 instructions.

3.3.1 Hash SpGEMM. We use hash table for accumulator in SpGEMM computation, based on GPU work [26]. Figure 4 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 it. 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 is multiplied by constant number 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 a integer). The hashing algorithm is based on linear probing. Figure 5-(a) shows an example of hash probing on 16 entries hash table.

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

HASH_SPGEMM(C, A, B)

```
1  RowsToThreads(offset, A, B)
2  // Determine hash table size for each thread
3  for tid ← 0 to tnum
4    do size_t ← 0
5       for i ← offset[tid] to offset[tid + 1]
6         do size_t ← max(size_t, flop[i])
7    // Required maximum hash table size is N_col
8    size_t ← min(N_col, size_t)
9    // Return minimum 2^m so that 2^m > size_t
10   size_t ← lowest_p2(size_t)
11 // Allocate rpts_C. After Symbolic, allocate cols_C and vals_C
12 Symbolic(rpts_C, A, B)
13 Numeric(C, A, B)
```

Figure 4: Hash SpGEMM Pseudocode

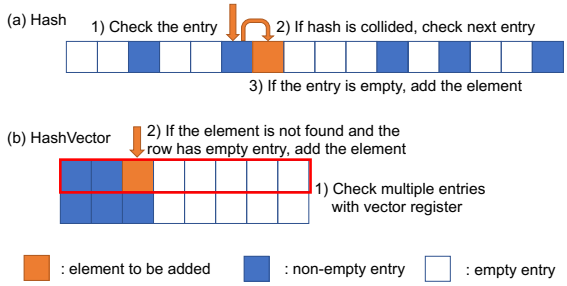

Figure 5: Hash Probing in Hash and HashVector SpGEMM

stored to memory as output. This Hash SpGEMM for multi/many-core processors differs from the GPU version as follows. While a row of output is computed by multiple threads in the GPU version to exploit massive number of threads on GPU, each row is processed by a single thread in the present algorithm. 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.

3.3.2 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 [29]. 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. A chunk consists of 8 entries on Haswell since a key (= column index) is represented as 32-bit in our evaluation. In HashVector, the hash indicates the identifier of target chunk in hash table. In order to examine the keys in the chunk, 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. 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 Hash vector SpGEMM can reduce the number of probing caused by hash collision, 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.

3.3.3 Heap SpGEMM. In another variant of SpGEMM [3], 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 we can adapt "parallel" approach for memory management. Thread-private heaps are also managed with "parallel" approach. As with the Hash algorithm, Heap SpGEMM estimates flop via a symbolic step and uses it to balance computational load evenly among threads.

3.4 Efficient Use of 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. First, 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*}. Second, 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.

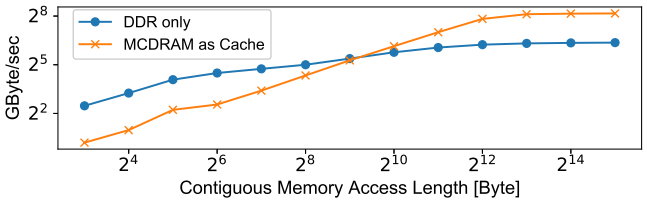

Figure 6: Benchmark result of random memory access with DDR only or MCDRAM as Cache

To quantify the stanza bandwidth which we expect to be quite different than STREAM [23], 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 6 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 EXPERIMENTAL SETUP

4.1 Input Types

We use two types of matrices for the evaluation. We generate synthetic matrix using matrix generator, and take matrices from SuiteSparse Matrix Collection [13]. For the evaluation of unsorted output, the column indices of input matrices are randomly permuted. We use 26 sparse matrices used in [4, 14, 21]. The matrices are listed in Table 2.

We use R-MAT [9], 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.

4.2 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 respectively. We set the number of

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

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	

threads as 68, 136, 204 or 272 for KNL, and 16, 32 or 64 for Haswell. For the evaluation of Kokkos on KNL, we set 256 threads instead of 272 threads since the execution fails on more than 256 threads. We show the result with the best thread count. For the evaluation on KNL, 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".

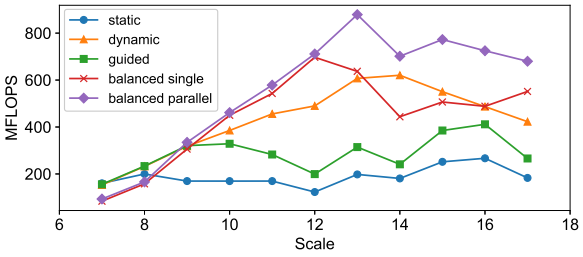

Figure 7: Performance of Heap SpGEMM scaling with size of G500 inputs on KNL with Cache mode

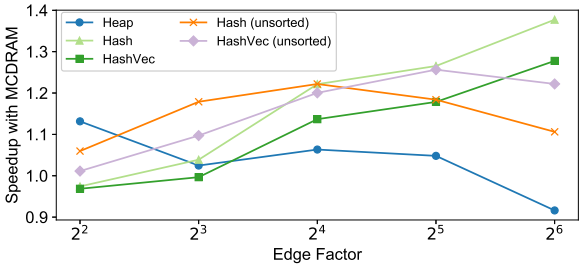

Figure 8: Speedups attained with the use of Cache mode on KNL compared to Flat mode on DDR4. G500 (scale 15) matrices are used with different edge factors.

4.3 Preliminary Evaluation on KNL

4.3.1 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, 'balanced', works well on SpGEMM. For larger inputs, Heap SpGEMM temporally requires large memory use, whose deallocation causes performance degradation. Our "parallel" memory management scheme 'balanced parallel' reduces the overhead of memory deallocation for temporal memory use, and keeps high performance on large size inputs.

4.3.2 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 8 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 6. The limited benefit stems from the fact that SpGEMM frequently requires indirect 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

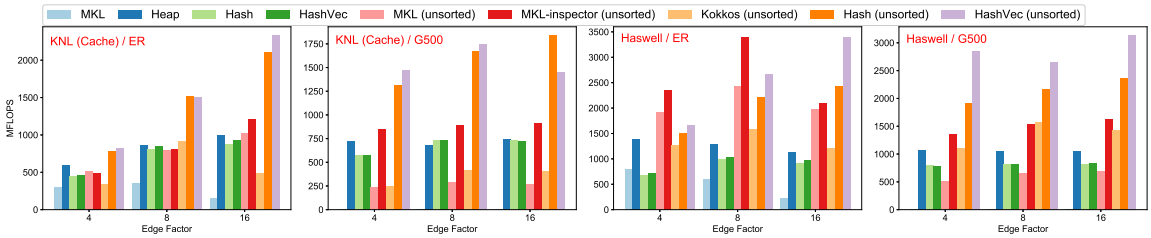

Figure 9: Scaling with increasing density (scale 16) on KNL (left) and Haswell (right)

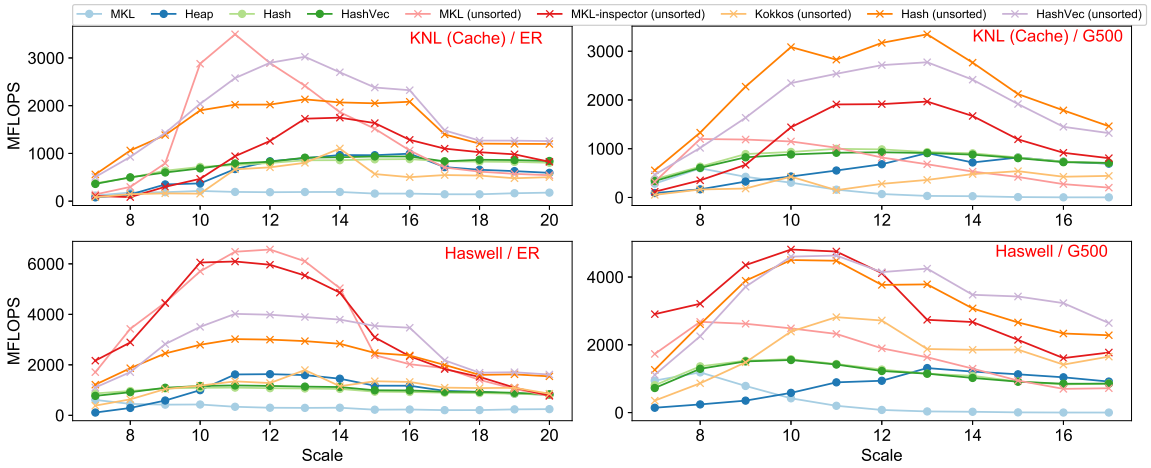

Figure 10: Scaling with size on KNL (top) and Haswell (bottom), both with edge factor 16

point the memory requirement of Heap SpGEMM surpasses the capacity of MCDRAM.

5 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.

5.1 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).

5.1.1 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, but such performance gains are not so pronounced for G500 matrices. For G500 matrices, we see significant performance difference between KNL and Haswell results. While Hash shows superior performance on KNL, HashVector achieves much higher performance on Haswell. Also for G500 matrices, we see that MKL (both sorted and unsorted) and MKL-inspector achieves a peak in performance at edgecount 8, with performance degrading as the matrices get denser or sparser than that sweet spot. Hash and HashVector might not have peaked at these density ranges we experimented since they get faster as the matrices get denser.

5.1.2 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 (top) and Haswell (bottom). 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 overcome. Especially, Hash and HashVector keep high performance even for large scale matrices. This performance trend becomes more clear on Haswell. 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 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.

5.1.3 Scaling with Thread Count. Figure 11 shows the scalability analysis of KNL on ER and G500 matrices with scale=16

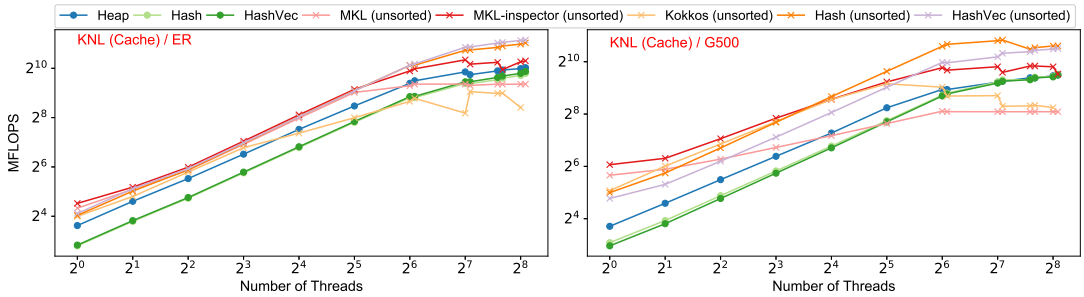

Figure 11: Strong scaling with thread count on KNL with ER (left) and G500 inputs (right). Data used is of scale 16 with edge factor 16

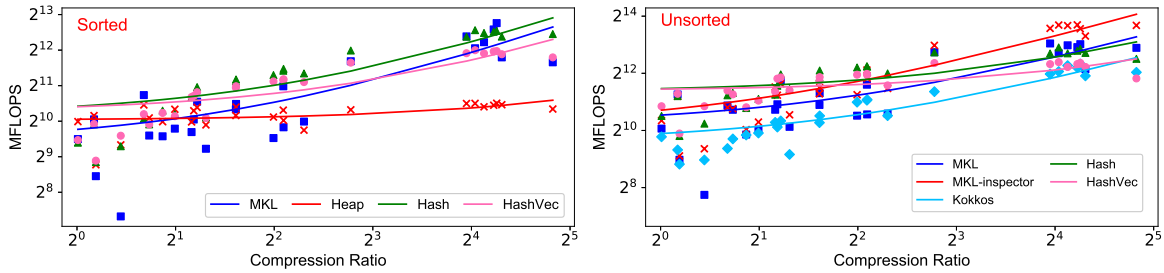

Figure 12: Scaling with compression ratio of SuiteSparse matrices on KNL. The algorithms that operate on sorted matrices (both input & output) are on the left and those that operate on unsorted matrices are on the right.

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 improvement over 68 threads. On the other hand, Heap and Hash/HashVector get further improvement over 64 threads.

5.1.4 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). Lines in the graph are linear fitting for each kernel. First 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 becomes worse on high compression ratio matrices as well as the evaluation with sorted output. MKL-inspector shows significant improvement especially for the matrices with high compression ratio.

Comparing sorted and unsorted versions of algorithm that provide the flexibility, we see consistent performance boost of keeping the output sorted. In particular, the harmonic mean of the speedups achieved operating on unsorted data over all real matrices we have studied from the SuiteSparse collection on KNL is 1.58× for MKL, 1.63× for Hash, and 1.68× for HashVector.

5.1.5 Profile of Relative Performance of SpGEMM Algorithms. We compare the relative performance of different SpGEMM algorithms with performance profile plots [15]. 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 13 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.

5.2 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 [5], a matrix is multiplied with a subset of its column, representing another use case of multiplying a square matrix with

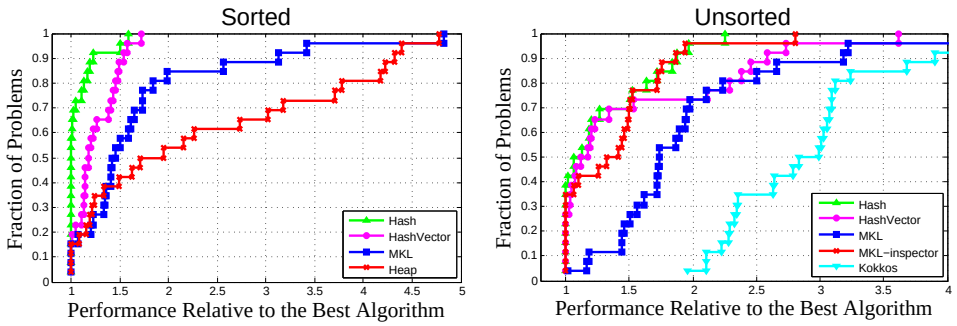

Figure 13: Performance profiles of SuiteSparse matrices on KNL using sorted (left) and unsorted (right) algorithms.

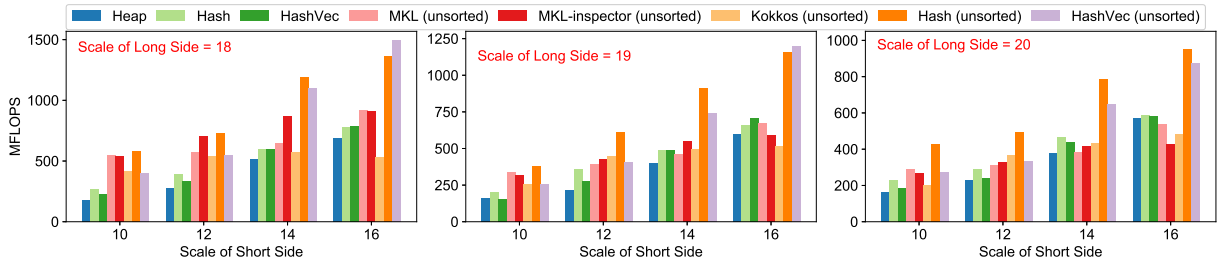

Figure 14: SpGEMM between square and tall-skinny matrices on KNL (scales 18, 19, and 20)

a tall-skinny matrix. In our evaluations, we generate the tall-skinny matrix by randomly selecting columns from the graph itself. Figure 14 shows the result of SpGEMM between square and tall-skinny matrices. We set scale as 18, 19 or 20 for square matrix, and as 10, 12, 14 or 16 for short size of tall-skinny 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.

5.3 Triangle counting

We also evaluate the performance of SpGEMM used in triangle counting [4]. 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 15 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. One big difference from A^2 is that Heap performs the best for inputs with low compression ratios.

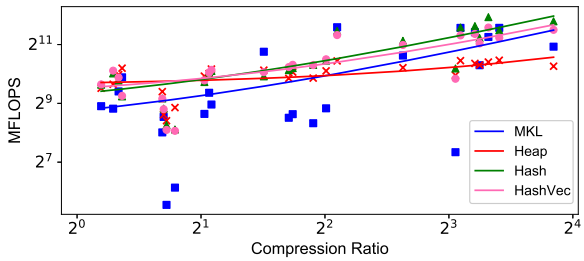

Figure 15: The performance of SpGEMM between L and U triangular matrices when used to count triangles on KNL

6 CONCLUSIONS

We studied the performance of computing the multiplication of two sparse matrices on multicore and Intel KNL architectures. This primitive, known as SpGEMM, has recently gained attention in the GPU community, but there has been relatively less work on CPUs and 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 multicore and Intel KNL architectures. Performance evaluation shows that our optimized SpGEMM algorithms largely overcome 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 ($\approx$ 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 than, MCDRAM shines as matrices get denser because then SpGEMM becomes primarily bandwidth bound and can take advantage of

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

the higher bandwidth available on MCDRAM. The second recipe is for the users. As summarized in Table 4, our results show that different codes dominate on different inputs, and we clarify which SpGEMM algorithm works well. 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, compression ratio also effects the dominant code. 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.

CODE

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

REFERENCES

[1] Sandeep R Agrawal, Christopher M Dee, and Alvin R Lebeck. 2016. Exploiting accelerators for efficient high dimensional similarity search. In *PPoPP*. ACM.
[2] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. 2016. Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication. In *ICS*. ACM, New York, NY, USA, Article 36.
[3] Ariful Azad, Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. 2016. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing* 38, 6 (2016), C624–C651.
[4] Ariful Azad, Aydın Buluç, and John Gilbert. 2015. Parallel triangle counting and enumeration using matrix algebra. In *IPDPSW*.
[5] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic acids research* (2018).
[6] Grey Ballard, Christopher Siefert, and Jonathan Hu. 2016. Reducing communication costs for sparse matrix multiplication within algebraic multigrid. *SIAM Journal on Scientific Computing* 38, 3 (2016), C203–C231.
[7] Nicolas Bock, Matt Challacombe, and Laxmikant V Kalé. 2016. Solvers for O(N) Electronic Structure in the Strong Scaling Limit. *SIAM Journal on Scientific Computing* 38, 1 (2016), C1–C21.
[8] Aydın Buluç and John R. Gilbert. 2011. The Combinatorial BLAS: Design, Implementation, and Applications. *IJHPCA* 25, 4 (2011), 496–509.
[9] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.
[10] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)* 41, 4 (2015), 25.
[11] Timothy A Davis. [n. d.]. private communication. ([n. d.]).
[12] Timothy A Davis. 2006. *Direct methods for sparse linear systems*. SIAM.
[13] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.
[14] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2017. Performance-Portable Sparse Matrix-Matrix Multiplication for Many-Core Architectures. In *IPDPSW*. IEEE, 693–702.
[15] Elizabeth D Dolan and Jorge J Moré. 2002. Benchmarking optimization software with performance profiles. *Mathematical programming* 91, 2 (2002), 201–213.
[16] John R Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 333–356.
[17] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. 2007. High performance graph algorithms from parallel sparse matrices. In *PARA*. 260–269.
[18] Felix Gremse, Andreas Hofter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. 2015. GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging. *SIAM Journal on Scientific Computing* 37, 1 (2015), C54–C71.
[19] Fred G Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM TOMS* 4, 3 (1978), 250–269.
[20] Guoming He, Haijun Feng, Cuiping Li, and Hong Chen. 2010. Parallel SimRank computation on large graphs with iterative aggregation. In *SIGKDD*. ACM.
[21] Weifeng Liu and Brian Vinter. 2014. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *IPDPS*. IEEE, 370–381.
[22] Kiran Matam, Siva Rama Krishna Bharadwaj Indarapu, and Kishore Kothapalli. 2012. Sparse Matrix-Matrix Multiplication on Modern Architectures. In *HiPC*. IEEE.
[23] John D. McCalpin. 1991-2007. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Technical Report. University of Virginia.
[24] Johannes Sebastian Mueller-Roemer, Christian Altenhofen, and André Stork. 2017. Ternary Sparse Matrix Representation for Volumetric Mesh Subdivision and Processing on GPUs. In *Computer Graphics Forum*, Vol. 36.
[25] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2018. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. *arXiv preprint arXiv:1804.01698* (2018).
[26] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. In *ICPP*. IEEE, 101–110.
[27] Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G Pudov, Vadim O Pirogov, and Pradeep Dubey. 2015. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In *ISC*. Springer, 48–57.
[28] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E* 76, 3 (2007), 036106.
[29] Kenneth A Ross. 2007. Efficient Hash Probes on Modern Processors. In *ICDE*. IEEE, 1297–1301.
[30] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Andreas Morhammer, Tibor Grasser, Ansgar Jüngel, and Siegfried Selberherr. 2016. ViennaCL—Linear Algebra Library for Multi-and Many-Core Architectures. *SIAM Journal on Scientific Computing* 38, 5 (2016), S412–S439.
[31] Viral B. Shah. 2007. *An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity*. Ph.D. Dissertation. University of California, Santa Barbara.
[32] Peter D Sulatycke and Kanad Ghose. 1998. Caching-efficient multithreaded fast multiplication of sparse matrices. In *IPPS/SPDP*. IEEE.