Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms

Md. Mostofa Ali Patwary[1(✉)], Nadathur Rajagopalan Satish[1], Narayanan Sundaram[1], Jongsoo Park[1], Michael J. Anderson[1], Satya Gautam Vadlamudi[1], Dipankar Das[1], Sergey G. Pudov[2], Vadim O. Pirogov[2], and Pradeep Dubey[1]

[1] Parallel Computing Lab, Intel Corporation, Santa Clara, USA
mostofa.ali.patwary@intel.com
[2] Software and Services Group, Intel Corporation, Santa Clara, USA

Abstract. Sparse matrix-matrix multiplication (SpGEMM) is a key kernel in many applications in High Performance Computing such as algebraic multigrid solvers and graph analytics. Optimizing SpGEMM on modern processors is challenging due to random data accesses, poor data locality and load imbalance during computation. In this work, we investigate different partitioning techniques, cache optimizations (using dense arrays instead of hash tables), and dynamic load balancing on SpGEMM using a diverse set of real-world and synthetic datasets. We demonstrate that our implementation outperforms the state-of-the-art using Intel® Xeon® processors. We are up to 3.8X faster than Intel® Math Kernel Library (MKL) and up to 257X faster than CombBLAS. We also outperform the best published GPU implementation of SpGEMM on nVidia GTX Titan and on AMD Radeon HD 7970 by up to 7.3X and 4.5X, respectively on their published datasets. We demonstrate good multi-core scalability (geomean speedup of 18.2X using 28 threads) as compared to MKL which gets 7.5X scaling on 28 threads.

1 Introduction

Sparse Matrix-Matrix Multiplication (SpGEMM) is an important kernel used in many applications in High Performance Computing such as algebraic multigrid solvers [4] and graph analytic kernels [7,10,12,17]. Compared to the efficiency of the corresponding dense GEMM routines, SpGEMM suffers from poor performance on most parallel hardware. The difficulty in optimizing SpGEMM lies in the irregular memory access patterns, unknown pattern of non-zeros in the output matrix, poor data locality and load imbalance during computation. For sparse matrices that have non-zero patterns following power law distributions (e.g. graphs from social network and recommendation system domains), this leads to poor efficiency as some portions of the output are very dense while others are very sparse.

This paper presents an optimized implementation of SpGEMM on two matrices A and B that efficiently utilizes current multicore hardware. We have parallelized SpGEMM through row and column based blocking of A and B respectively.

J.M. Kunkel and T. Ludwig (Eds.): ISC High Performance 2015, LNCS 9137, pp. 48–57, 2015.
DOI: 10.1007/978-3-319-20119-1_4

By using a dense array to accumulate partial (sparse vector) results, we get superior performance compared to previous implementations. We also maintain a CSR input and output format and include data structure transformation and memory allocation costs in our runtime.

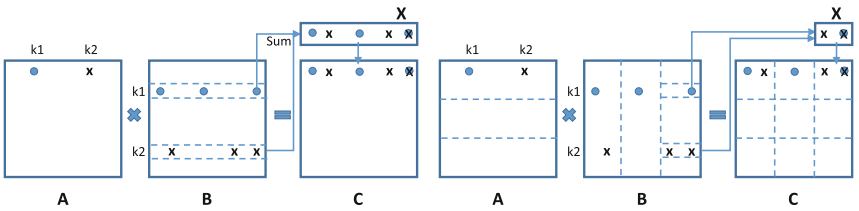
Our main contributions are as follows:

- We present the fastest SpGEMM results on a single node on a variety of different sparse matrices drawn from various domains. Our implementation running on Intel®Xeon® E5-2697 v3 processor based system is faster than Intel® MKL by up to 3.8X and CombBLAS by up to 257X. Our implementation also outperforms previously published GPU implementations [13] on nVidia GTX Titan and on AMD Radeon HD 7970 by up to 7.3X and 4.5X, respectively.
- We have explored different partitioning schemes for SpGEMM. We provide intelligent heuristics that combine row-wise partitioning of one matrix and column-wise partitioning of the second matrix to get the best performance (up to 1.4X improvement) on a single node.
- We divide the matrices into small partitions and perform dynamic load balancing over the partitions, resulting in speedups of up to 1.49X and 1.24X respectively.
- We demonstrate good multi-core scalability (geomean speedup of 18.2X using 28 threads) as compared to MKL, which gets 7.5X scaling on 28 threads.

2 SpGEMM Algorithm and Optimizations

2.1 Overview

Sparse Matrix-Matrix Multiply (SpGEMM) involves the multiplication of two sparse matrices A of dimension $m \times k$ and B of dimension $k \times n$ to yield a resultant matrix C of dimension $m \times n$. In this paper, for presentation simplicity, we assume we deal with square matrices where $m = n = k$ (represented as n hereafter). However, our techniques are general and applicable to other matrices as well.



(a) Gustavson's algorithm (b) Partitioned SpGEMM

Fig. 1. Data access pattern of Gustavson [11] and Partitioned SpGEMM algorithms.

Consider the following notations. $A_{i,j}$ denotes a single entry of matrix A. $A_{i,:}$ denotes the i^{th} row of matrix A, and $A_{:,i}$ represents the i^{th} column of A. Then the computation of the entire row i of C can be seen to be $C_{i,:} = \sum_k A_{i,k} * B_{k,:}$. This computation is shown in Fig. 1(a). The figure shows the situation for a sparse matrix A, where $A_{i,k}$ is non-zero only for some values of k, and computations only occur on the corresponding rows of B. The product of the scalar $A_{i,k}$ with the non-zeros in $B_{k,:}$ basically scales the elements of the row $B_{k,:}$ and has the same sparsity structure as $B_{k,:}$. This product then needs to be summed into $C_{i,:}$. Note that $C_{i,:}$ is the sum of various sparse vectors obtained from the products above, and is in general sparse, although its non-zero structure and density may be quite different from that of A and B.

Gustavson [11] proposed a single-threaded algorithm for SpGEMM based on the Compressed Sparse Row (CSR) format. This is a straightforward implementation of the computations described in Fig. 1(a). This algorithm can be parallelized over rows of A to run on multi-core processors. However, as we show below, this basic algorithm does not take full advantage of architectural features such as caches present on modern hardware and makes inefficient use of the limited bandwidth available at various levels of the memory hierarchy. We now show the optimizations that we perform to overcome these bottlenecks.

```
1 part = Partition(A, B)
2 #pragma omp parallel for
  schedule(dynamic)
3 forall the p in part do
4     for all rows A_{i,:} in p do
5         Reset X to 0
6         for each nonzero A_{i,j} in A_{i,:} do
7             Load partition B_{j,cols(p)} in p
8             X += A_{i,j} · B_{j,cols(p)}
9         Sparsify X to C_{i,cols(p)}
```

Algorithm 1. Psuedocode for partitioned SpGEMM

2.2 Performance Optimizations

We make a number of improvements to the algorithm described previously.

Adding Sparse Vectors. Whenever a new sparse vector is to be added to the running sum for $C_{i,:}$, the index of each non-zero value in the sparse vector needs to looked up in the running sum. If present, the value of the non-zero element needs to be added to that in $C_{i,:}$, and if not, a new entry is to be created for this non-zero.

We considered various options to efficiently implement this lookup. One approach is to use a hash table to store the non-zero elements of the running sum for $C_{i,:}$, with the key being the column index of the non-zero, and the value being its numerical value. However, we found that this technique had high overheads due to (i) cost of hash key computations and (ii) cost of handling collisions through chaining.

Since the range of elements is known apriori (equal to the matrix dimension n), it is much more efficient to use a dense array to store the running sum. We initialize this array X to zero. When we need to add a new sparse vector, we take each

non-zero entry and simply add its value to $X[c]$, where c is the column index of the non-zero. Finally, once all additions are complete, we need to convert this dense array back to a sparse CSR representation when writing back to C. While the additions themselves have very low overhead with this data structure, it is very inefficient to have to scan the entire dense X array (most of whose elements are zero) and write back only the few non-zero elements into a sparse format. Indeed, our experiments indicate that this scan takes more than 20X the time required for the additions themselves. Hence, in addition to X, we keep a index array that stores the non-zero indices of X. This can be cheaply maintained during the addition process by simply appending a column index c when it is first written to, i.e. when the existing array value $X[c]$ is zero. We then iterate only over this sparse index array when writing to C and reset the corresponding value in X.

Partitioning Schemes. The computation of individual rows of C can be done independently, and hence rows of C (and the computation on the corresponding rows of A) can be trivially partitioned among the threads.

However, we need to pay careful attention to cache behavior during the sparse addition. Depending on the number of distinct cache lines of the dense array X that are touched during the update of a row of C, the X data structure may not completely reside in a close enough level of the cache hierarchy. Since this update is in the inner loop of the code, this can significantly affect performance. Specifically, for the datasets we describe in the evaluation section, we see a number of potential misses to the private second level (L2) cache. These misses are usually captured in the shared last level (LLC) cache, however we do find a significant performance impact due to L2 misses. In general for larger data sets, one could see misses to LLC as well. Hence we need a general scheme to localize accesses to the X array through blocking.

Blocking accesses to X in an efficient manner is non-trivial. Without any modifications, blocking along a row of B is difficult to achieve (as there are only few non-zeros per row on average). We hence propose to change the data structure of matrix B, and store it in a partitioned manner. We store individual CSRs for each partition of B. The number of partitions required for B depends on the L2 cache size. Figure 1(b) shows this scheme, where accesses to X get blocked. There are, of course, overheads in creating this blocked CSR data structure from the original CSR, and it may not be worthwhile to perform this transformation. We discuss this shortly.

Algorithm 1 shows the overall pseudocode for our algorithm. We use a hybrid scheme where we partition both the rows of A and the columns of B. Each partition updates a 2D block of matrix C. Each block of C is computed independently as an SpGEMM of the corresponding row partition of A and column partition of B.

As mentioned before, there is overhead in creating the blocked CSR representations of B. Further, each block of B is much sparser than the full row of B, and accesses to row pointers are not well amortized. This can lead to some bandwidth

overhead when reading B as well. Consequently it only makes sense to perform the transformation when we know that the benefits when accessing X are large. If we know that, for a significant fraction of rows, the number of non-zeros of the final X after all updates is greater than the L2 cache size (usually 256 KB, or 64 K single-precision numbers), then blocking accesses to X makes sense. Since we do not apriori know the density of the output rows, we need to estimate it. We use a simple upper-bound estimate described in [13], where we merely count the number of multiplications involved in computing each row of C. This can be done cheaply without any actual floating point operations by merely looking at the non-zero structure of A and the row pointer array of B. This usually takes just 1-2 % of overall runtime to compute. We use these estimates $e\_nnz(i)$ for each row i to define an overall metric : $e\_nnz = \frac{\sum_{i:e\_nnz(i)>64K} e\_nnz(i)}{\sum_i e\_nnz(i)}$. If this is sufficiently large (greater than 30 %), our results show that we should partition B.

Dynamic Scheduling. Different partitions in the computations described in Algorithm 1 have differing amounts of computation and store different numbers of non-zeros. This leads to severe load imbalance. We have found that reducing the size of each partition so that the total number of partitions is 6–10 times the number of threads leads to significantly better load balance.

3 Experimental Results

3.1 Experimental Setup

We used an Intel® Xeon®[1] E5-2697 v3 processor based system for the experiments. The system consists of two processors, each with 14-cores running at 2.6 GHz (a total of 28 cores) with 36 MB L3 cache and 64 GB memory. The system is based on the Haswell microarchitecture and runs Redhat Linux (version 6.5). All our code is developed using C/C++ and is compiled using the Intel® C++ compiler[2] (version: 15.0.1) using the -O3 flag. We pins threads to cores for efficient NUMA behavior by setting KMP_AFFINITY to *granularity=fine,compact,1* in our experiments [2].

[1] Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

[2] Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel micro-architecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

Table 1. Structural properties of the datasets. C denotes the resultant matrix.

Name	Rows	nnz	Avg Degree	Max Degree	nnz (C)	Time (sec)	Type
harbor	46,835	4,701,167	100	289	7,900,918	0.0681	Symmetric
hood	220,542	10,768,436	49	77	34,242,181	0.0802	Symmetric
qcd	49,152	1,916,928	39	39	10,911,745	0.0373	Asymmetric
consph	83,334	6,010,480	72	81	26,539,737	0.0675	Symmetric
pwtk	217,918	11,634,424	53	180	32,772,237	0.0821	Symmetric
PR02R	161,070	8,185,136	51	92	30,969,454	0.0756	Asymmetric
mono	169,410	5,036,288	30	719	41,377,965	0.0965	Asymmetric
webbase	1,000,005	3,105,536	3	4,700	51,111,997	0.1597	Asymmetric
audikw	943,695	39,297,771	42	346	164,772,225	0.2053	Asymmetric
mou.gene	45,101	14,506,196	322	8033	190,430,984	0.9016	Asymmetric
cage14	1,505,785	27,130,349	18	82	236,999,813	0.2469	Asymmetric
dielFilt	1,102,824	45,204,422	41	271	270,082,366	0.2687	Asymmetric
rmat_er	262,144	16,777,150	64	181	704,303,135	0.7538	Asymmetric
rmat_g	262,144	16,749,883	64	3283	1,283,506,475	1.1721	Asymmetric
rmat_b	262,144	16,749,883	64	54250	1,648,990,052	2.4519	Asymmetric

We used both real-world and synthetic matrices for performance analysis. Table 1 shows the structural properties of the datasets. Our real-world matrices consist of 12 datasets collected from the Florida Matrix collection [8] covering many applications including structural engineering, web connectivity, electromagnetics, and medical science. Six of these datasets (*harbor*, *hood*, *qcd*, *pwtk*, *mono*, and *webbase*) are the same as those used in [13]. We specifically picked datasets where the authors' implementation shows the best performance and augmented this set with the largest datasets considered in that work. To increase the diversity of use cases considered, we additionally included 6 more datasets, which are up to an order of magnitude larger (in terms of nonzeros) than those previously experimented.

We also generated 3 types of synthetic datasets using the Graph500 RMAT data generator [14] by varying the RMAT parameters (similar to previous work [6]). These are (i) *rmat_b* (parameters 0.55, 0.15, 0.15, 0.15), (ii) *rmat_g* (parameters 0.45, 0.15, 0.15, 0.25), and (iii) *rmat_er* (0.25, 0.25, 0.25, 0.25). These matrices vary mainly in terms of degree distributions (e.g. *rmat_b* is highly skewed) to cover a wider range of applications.

The seven largest datasets (*audikw*, *mou.gene*, *cage14*, *dielFilt*, and the three synthetic datasets) are treated as unsymmetric to avoid large SpGEMM output matrices C that overflow memory limits. The symmetry of each dataset is tabulated in Table 1.

3.2 Experimental Results

We first compare the performance of our SpGEMM implementation with state-of-the-art results. To do so, we consider four available implementations,

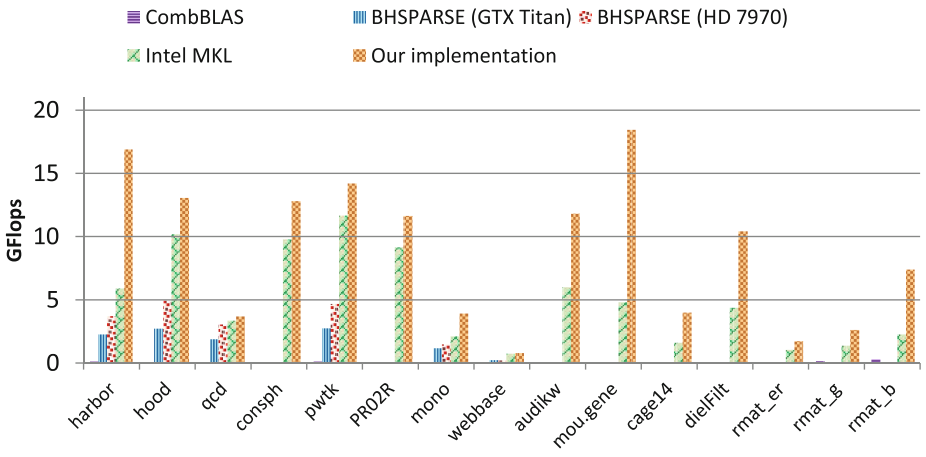
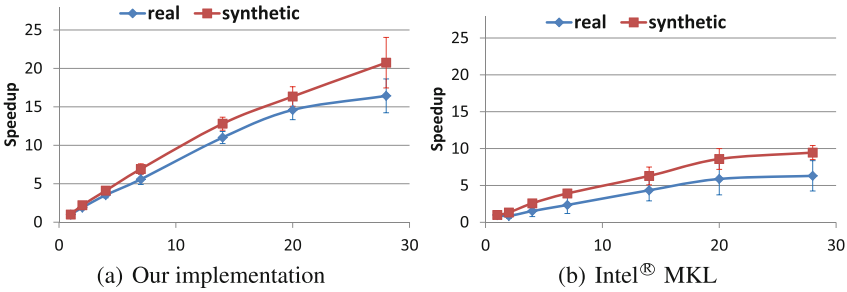


Fig. 2. Performance comparison of CombBLAS, Intel® MKL, BHSPARSE on nVidia GTX Titan GPU and on AMD Radeon HD 7970 GPU, and our implementation.

(i) the Combinatorial BLAS Library (CombBLAS v1.3, [1]), (ii) Intel® Math Kernel Library (MKL, version 11.2.1 [3]), (iii) BHSPARSE implementation on the nVidia GeForce GTX Titan GPU reported in [13], and (iv) BHSPARSE implementation on the AMD Radeon HD 7970 GPU reported in [13]. These represent some of the most recent SpGEMM publications that perform well on modern hardware [13]. We consider all 15 datasets in the comparison. However, since the GPU SpGEMM code is not available online, we used the performance results of the 6 overlapping datasets from [13]. Figure 2 shows the performance of these implementations in GFlops. As can be seen, our code is able to achieve up to 18.4 GFlops (geomean 6.6 GFlops) whereas CombBLAS, MKL, BHSPARSE on nVidia GTX Titan, and BHSPARSE on AMD Radeon HD 7970 GPU achieve up to 0.3, 11.7, 2.8, and 5.0 (geomean of 0.09, 3.6, 1.5, and 2.1) GFlops respectively. CombBLAS uses the DCSC matrix format that involves an additional layer of indirection (more cache line accesses) that leads to poor performance. MKL does not partition the columns of B and uses a static partitioning scheme that can cause performance degradation. We note that our performance results are up to 7.3X (geomean 3.9X) compared to the best BHSPARSE GPU implementation. This is despite the nearly 2× higher peak flops for the GPU cards compared to the Intel® Xeon® processor used. We attribute this to the impact of the algorithmic optimizations such as partitioning techniques, cache optimizations, and dynamic load balancing (more details are in Sect. 2).

We next demonstrate scalability using the 7 largest datasets in Table 1. Figure 3 shows the scalability of our algorithm and Intel® MKL. We ignore CombBLAS for this comparison as it is significantly slower (76X slower on average) and also the GPU implementations due to unavailability of code. Figure 3(a) and Fig. 3(b) show scaling results for our algorithm and MKL respectively. As can be seen, we achieve up to 28X speedup using 28 threads with respect to single



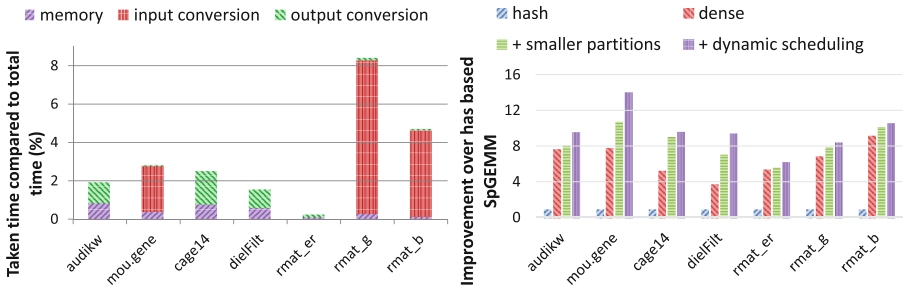
(a) Our implementation (b) Intel® MKL

Fig. 3. Scalability of our implementation and Intel® MKL on real (geomean of largest 4) and synthetic (geomean of largest 3) datasets with {1, 2, 4, 7, 14, 20, 28} threads.

thread performance (geomean 18.2X) on our datasets (for real-world datasets: max 19.6X, and geomean of 16.4X; for synthetic datasets: max 28X, and geomean of 20.7X). MKL shows up to 9.9X speedup (geomean of 7.5X) on the same datasets (for real-world datasets:max 7.3X, and geomean of 6.3X; for synthetic datasets: max 9.9X, and geomean of 9.5X). The bar at each point shows the standard deviation on the scalability of our code as observed in our experiments. In general, scalability is better for dense datasets (*mou.gene* scales by about 20X on 28 cores); as also for more skewed datasets (*rmat_b* scales near linearly - 28X on 28 cores) where a significant portion of runtime is spent in dense areas of the matrix. For such matrices, there is more reuse of data structures and SpGEMM is more compute bound.

Figure 4(a) shows the timing breakdown in our algorithm. Among the 4 steps (memory allocation, computation, and input and output data structure conversion), computation is the dominant part and the other 3 take only 0.4 %, 2.6 %, and 0.3 %, respectively on average. For some datasets such as *mou.gene*, *rmat_g* and *rmat_b*, where there are dense regions of computation, as determined by our metric defined in Sect. 2, we partition columns of B and input data structure conversion times increase (e.g. *rmat_g* 7.9 %). This is still however small and our computation times still dominate; overall runtimes for such datasets are up to 1.4X (average 1.22X) faster with column partitioning. We verified our reasoning by counting L2 cache misses using hardware counters. Our analysis shows that for such datasets, L2 cache misses went down by 1.3-2.3X when we performed column partitioning. This ties in well with our performance gains.

We now discuss the impact of various optimizations described in Sect. 2 (Fig. 4(b)) on our SpGEMM code. We take the hash-based SpGEMM implementation as the baseline for the comparison. The main performance gain comes from using a dense array instead of hash tables during the addition phase. This yields up to 9.2X speedup (with a geomean of 6.3X). This is due to the overheads of using hash tables as explained in Sect. 2. Increasing the number of partitions improves performance by up to 49 % (with a geomean of 14.5 %). Using dynamic scheduling gives an additional 24.9 % (with a geomean of 10.2 %) performance boost. This is due to better load balancing among the parallel threads.



(a) Timing breakdown (computation time is not shown for readability purposes)

(b) Optimization breakdown

Fig. 4. Performance analysis of our implementation

4 Related Work

Gustavson introduced an SpGEMM algorithm with a work complexity proportional to the number of nonzeros in A, number of total flops, number of rows in C, and the number of columns in C [11]. This algorithm, and how it relates to our work, is described in detail in Sect. 2. A similar algorithm is used by Matlab, which processes a column of C at a time, and uses a dense vector with values, indices, and valid flags, for accumulating sparse partial results [9]. Buluc and Gilbert address the case of hypersparse matrices, where the number of non-zeros is less than the number of columns or rows [5]. The authors introduce the doubly compressed sparse column (DCSC) format, and two new SpGEMM algorithms to handle this case.

There have also been several efforts to optimize the performance of SpGEMM for parallel and heterogeneous hardware. Sulatycke and Ghose analyzed the cache behavior of different loop-orderings of SpGEMM with sparse A and B and dense C, and proposed a cache-efficient parallel algorithm that divided work across rows of A and C [16]. Siegel et al. created a framework for running SpGEMM on a cluster of heterogeneous (2x CPU + 2x GPU) nodes, and demonstrated a significant improvement in load-balance through dynamic scheduling as compared to a static approach [15]. Zhu et al. propose a custom hardware implementation to accelerate SpGEMM [18]. They use custom logic integrated with 3D stacked memory to retrieve columns from A, and merge intermediate results into C using content-addressable memories (CAMs). Liu and Vintner describe an SpGEMM algorithm and implementation for GPUs [13]. In this implementation, rows of C are divided into bins based on an upper-bound of the size of intermediate results and processed using different routines.

5 Conclusion and Future Work

In this paper we investigated Sparse Matrix-Matrix Multiplication (SpGEMM), an important kernel used extensively in many applications including linear

solvers and graph analytics. To improve SpGEMM efficiency on multicore platforms, we performed different optimization techniques such as using dense arrays, implementing column-wise partitioning, and dynamic scheduling. We showed state-of-the-art results on both real-world and synthetic datasets. We are up to 3.8X faster than Intel® MKL and up to 257X faster than CombBLAS. We are also up to 7.3X better than the best published GPU implementation. Our code shows good scalability of 18.2X using 28 threads, as compared to MKL that achieves 7.5X speedup. In the future, we intend to extend these optimizations in a distributed setup.

References

1. Combinatorial Blas v 1.3. http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/
2. Thread affinity interface. https://software.intel.com/en-us/node/522691
3. Intel math kernel library (2015). https://software.intel.com/en-us/intel-mkl
4. Bell, N., Dalton, S., Olson, L.N.: Exposing fine-grained parallelism in algebraic multigrid methods. SIAM J. Sci. Comput. **34**(4), C123–C152 (2012)
5. Buluc, A., Gilbert, J.: On the representation and multiplication of hypersparse matrices. In: Proceedings of IPDPS, pp. 1–11, April 2008
6. Buluç, A., Gilbert, J.R.: Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. CoRR abs/1109.3739 (2011)
7. Chan, T.M.: More algorithms for all-pairs shortest paths in weighted graphs. SIAM J. Comput. **39**(5), 2075–2089 (2010)
8. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1:1–1:25 (2011)
9. Gilbert, J., Moler, C., Schreiber, R.: Sparse matrices in matlab: design and implementation. SIAM J. Matrix Anal. Appl. **13**(1), 333–356 (1992)
10. Gilbert, J.R., Reinhardt, S., Shah, V.B.: High-performance graph algorithms from parallel sparse matrices. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 260–269. Springer, Heidelberg (2007)
11. Gustavson, F.G.: Two fast algorithms for sparse matrices: multiplication and permuted transposition. ACM Trans. Math. Softw. **4**(3), 250–269 (1978)
12. Kaplan, H., Sharir, M., Verbin, E.: Colored intersection searching via sparse rectangular matrix multiplication. In: Symposium on Computational Geometry, pp. 52–60. ACM (2006)
13. Liu, W., Vinter, B.: An efficient GPU general sparse matrix-matrix multiplication for irregular data. In: Proceedings of IPDPS, pp. 370–381. IEEE (2014)
14. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the graph 500. Cray User's Group (2010)
15. Siegel, J., et al.: Efficient sparse matrix-matrix multiplication on heterogeneous high performance systems. In: IEEE Cluster Computing, pp. 1–8 (2010)
16. Sulatycke, P., Ghose, K.: Caching-efficient multithreaded fast multiplication of sparse matrices. In: Proceedings of IPPS/SPDP 1998, pp. 117–123, March 1998
17. Vassilevska, V., Williams, R., Yuster, R.: Finding heaviest h-subgraphs in real weighted graphs, with applications. CoRR abs/cs/0609009 (2006)
18. Zhu, Q., Graf, T., Sumbul, H., Pileggi, L., Franchetti, F.: Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware. In: IEEE HPEC, pp. 1–6 (2013)