SAND2016-4513C

Sparse Matrix-Matrix Multiplication for Modern Architectures

Mehmet Deveci * Erik G Boman * Sivansakaran Rajamanickam *

Sparse matrix-matrix multiplication (SPMM) is an important kernel in high performance computing that is heavily used in the graph analytics as well as multigrid linear solvers. Because of its highly sparse structure, it is usually difficult to exploit the parallelism in the modern shared memory architectures. Although there have been various work studying shared memory parallelism of SPMM, some points are usually overlooked, such as the memory usage of the SPMM kernels. Since SPMM is a service-kernel, it is important to respect the memory usage of the calling application in order not to interfere with its execution. In this work, we study memory-efficient scalable shared memory parallel SPMM methods. We study graph compression techniques that reduce the size of the matrices, and allow faster computations. Our preliminary results show that we obtain upto 40% speedups w.r.t SPMM implementation provided in Intel Math Library while using 65% less memory.

1 Introduction

SPMM is a fundamental kernel that is used in various applications in scientific computing such as graph analytics or multigrid solvers. In this work, we study scalable and memory efficient SPMM methods for multicore and many-core architectures. In the literature, most parallel SPMM methods [1] follow Gustavson's algorithm [2]. This algorithm schedules the computations in 1D row-wise fashion, and multiplication results for all entries in a row are found simultaneously. That is, given the multiplication $C = A \times B$, the algorithm performs multiplications in the forms of $C_i = \sum_{k \in A_i} A_{i,k} \times B_k$ in order to find a single row i of C.

There are various memory constraints with the parallelization of Gustavson's algorithm. For example, the size and structure of C is unknown at the beginning of SPMM operation. Although, there exist studies to predict the size of C [3], they still do not provide a robust upper bound for the memory requirements. In the literature, this problem is addressed using various approaches. One approach is to calculate the upper bound for the size of C, and allocate this memory prior to computation [4]. However, the upper bound becomes the number of floating-point operations (FLOPS), which can be significantly larger than the actual size of C. Another approach is to dynamically reallocate the size of C as the computation progresses. However, this approach might also be problematic in the modern architectures with massive number of threads, as memory re-allocations can become bottlenecks in parallel regions. In addition, such re-allocations are not feasible for GPUs. Another approach is to perform a symbolic SPMM operation before the numeric computations to compute the accurate size of C [5]. Although, this approach doubles the number of performed matrix-operations, it allows SPMM to run with minimal memory usage. Symbolic phase needs to be run only once for matrices in which only the numeric values change while the symbolic structures are preserved. In this work, we follow this approach, and we aim to speedup the symbolic phase by performing matrix compression.

Another memory constraint with the parallel Gustavson algorithm is the use of the sparse or dense accumulators. The sequential algorithm uses dense data structures that have the size of the number of columns in B, in order to accumulate the result rows. However, having such thread private arrays is costly on massively threaded architectures. Therefore, sparse accumulators such as heaps or hashmaps are usually preferred in parallel implementations. In this work, we use multi-level hashmaps as sparse accumulators.

2 Algorithms

Algorithm 1 gives the overall structure of our SPMM method. Given two matrices A and B for $C = A \times B$, we compress the symbolic structure of B with an approach similar to [6]. Symbolic structure refers to the underlying graph structure with binary relations, which can be represented using single bits. We compress the rows of B such that 32 columns are represented using a single integer. In this scheme each column is represented with 2 integers (or possibly with long integer). First integer refers to column set (CS) in which the set bit indices denote the indices of the columns. That is, if ith bit in CS is 1, the row has a nonzero entry at ith

*Sandia National Laboratories. Sandia is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

column. The second integer refers to the column set index (CSI) to represent more than 32 columns. With this compression method, the number of non-zeros of B can be reduced up to 32 times. The compression becomes more successful if the column indices in each rows are packed close to each other. This reduces the problem size, and also allows faster union operations using BitwiseOr, which helps to speedup the symbolic phase to find the structure of C. In the following steps, we predict the maximum number of non-zeros in a *row* of C as MAXFLOPS, which is later re-used by each thread in the symbolic SPMM phase as the upper bound for the required memory. Symbolic phase performs unions of rows of compressed B matrix, B_c, to find the memory requirements. In the numeric phase, we perform the actual matrix multiplication.

Algorithm 1 SPMM for $C = A \times B$

Require: Matrices A, B

1: $B_c \leftarrow$ Compress(B)
2: $maxRowNNZ \leftarrow$ getMaxNNZ(A, B_c)
3: $cmem \leftarrow$ Symbolic_Spmm(A, B_c, $maxRowNNZ$)
4: $C \leftarrow$ Numeric_Spmm(A, B, $cmem$)
5: **return** C

3 Preliminary Results

We evaluate the performance of the proposed method on the single nodes of the Shepard cluster at Sandia. A single node in Shepard has 64 cores at 2.3 GHz with 128 GB memory. The proposed method is implemented using the Kokkos Library in Trilinos, and compiled using the version within the Trilinos 12.2 release, with icc 16.0.190. In the experiments, we study matrix multiplications in the forms of $P^T \times A \times P$ and $A \times A^T$. The matrices are selected from a Laplace3D problem where A is a square matrix with $15,625,000$ rows and $109,000,000$ non-zeros, and P is a rectangular matrix with $15,625,000$ rows, $1,969,824$ columns, and $57,354,176$ non-zeros. We compare our proposed method (KK) with the SPMM implementation provided in Math Kernel Library (MKL) library for multiplications that often occur in algebraic multigrid setup. . Table 1 gives the strong scaling results with the average speedups of KK and MKL w.r.t sequential MKL.

MKL performs the SPMM within a single phase. Its peak memory usage 68%, 65% and 7.5% more than KK on the multiplications for $R \times A$, $A \times A^T$, and $A \times P$ and $RA \times P$ than KK. Although, KK doubles the matrix operations, it usually obtains better speedups than MKL on $A \times A^T$, $A \times P$, and $P^T \times A$. The compression speeds up KK's symbolic phase by the amount of the reduction, where it was able to reduce the number of non-zeros in A by 27.7%. However, it only reduced it by 6.7% and 2.9% on P and AP matrices.

Table 1: Strong scaling speedups for Laplace3D SPMM

	$A \times A^T$		$A \times P$		$P^T \times AP$		$P^T \times A$		$P^T A \times P$	
	KK	MKL	KK	MKL	KK	MKL	KK	MKL	KK	MKL
1	0.63	1.00	0.72	1.00	0.65	1.00	0.68	1.00	0.65	1.00
2	1.27	1.93	1.40	1.91	1.31	1.93	1.36	1.97	1.28	1.91
4	2.17	3.50	2.82	2.88	2.15	3.10	2.39	2.63	2.21	2.83
8	4.93	6.90	5.22	5.47	3.90	4.51	3.97	4.09	4.23	6.45
16	9.28	9.35	10.07	6.71	5.81	6.52	5.95	5.93	6.66	7.54
32	17.08	12.22	19.17	8.67	10.74	9.67	9.28	9.09	10.56	10.50
64	12.61	10.75	14.41	14.14	9.12	11.33	9.03	7.42	9.19	12.00

4 Ongoing Work

There are various ongoing efforts to extend our shared memory SPMM work. Firstly, the current work is being extended to GPUs. We would like to study 2D partitioning of C using the massive number of threads provided by GPUs. Secondly, we are studying ordering methods to allow better quality compressions as well as better cache locality. Compression mechanism will depends on the column order of B. In the experiments so far the compression mechanism is able to achieve 27% reduction at the most. This can be improved by using different column orderings that reduce the bandwidth or minimum logarithmic gap arrangement. We are studying the ordering heuristics that minimizes the column sets of B_c that is, it orders the columns as consecutive as possible. We believe such orderings are important in different applications such as for improving the cache locality in Sparse Matrix-Vector multiplications. Moreover, we would like to study methods that re-orders the rows of A in such a way that consecutive rows have similar columns so that the cache-locality can be exploited for the accesses to rows of B.

References

[1] M. M. A. Patwary et al., "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *High Performance Computing*. Springer, 2015

[2] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[3] E. Cohen, "Structure prediction and computation of sparse matrix products," *Journal of Combinatorial Optimization*, vol. 2, no. 4, pp. 307–332, 1998.

[4] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrixmatrix multiplication for the GPU," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 4, p. 25, 2015.

[5] J. Demouth, "Sparse matrix-matrix multiplication on the gpu," in *GPU Technology Conference*, 2012.

[6] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring for manycore architectures." *IPDPS*, 2016, to appear.