# Parallel Segmented Merge and Its Applications to Two Sparse Matrix Kernels

Weifeng Liu*† Hao Wang‡ Brian Vinter†

**Abstract**

Segmented operations, such as segmented sum, segmented scan and segmented sort, are important building blocks for parallel sparse matrix algorithms. We in this work propose a fast segmented merge kernel that in parallel merges $q$ sub-segments to $p$ segments, both of nonuniform lengths. We implement the segmented merge kernel on GPUs and demonstrate its efficiency on two key operations: sparse transposition and sparse matrix-matrix multiplication.

## 1 Introduction

In this paper, we consider a new primitive operation named *segmented merge* (`segmerge` for short). Assuming we have a key-value array $A$ composed of $n$ key-value pairs and $p$ segments, each segment, say $A_i$, contains $n_i$ key-value pairs and $q_i$ sub-segments, and each sub-segment, say $A_{ij}$, has $n_{ij}$ key-value pairs already sorted according to their keys. The objective of the `segmerge` operation is to let $n_i = \sum_{j=1}^{q_i} n_{ij}$ key-value pairs in each segment $A_i$ ordered. Thus $A$ eventually consists of $p$ sorted segments.

The easiest way to parallelize `segmerge` is to evenly assign a batch of segments to a thread, and each thread can merge sub-segments to segments independently. However, on many-core processors running a large amount of threads, this approach may bring load imbalance problem since the lengths of segments and sub-segments may be imbalanced. As a result, this may make only a few threads much busier than others, and the whole processor is largely underused. Such performance degradation from irregular data distribution is actually not unusual in sparse matrix algorithms such as sparse matrix-vector multiplication [7, 9].

Since the early work by Blelloch et al. [1] reported that segmented operations can be more balanced than row-/column-wise approaches in sparse matrix computations, several new parallel segmented primitives, such as segmented sum [8], segmented scan [3] and segmented sort [5], have been developed for better use of many-core platforms such as GPUs. It is also noticeable that, in practice, nonzeros in each row/column of a sparse matrix are generally sorted in the ascending order according to their column/row indices (though not always [11]). Therefore, if rows/columns (seen as sub-segments) of input sparse matrices are going to be merged into rows/columns (seen as segments) of an output sparse matrix, the `segmerge` primitive can be called to utilize the already ordered sub-segments (i.e., rows/columns of the input). The simplest example of the scenario may be adding multiple sparse matrices into one resulting matrix.

The requirement from sparse matrix kernels and the possibly load imbalance of the naïve parallelization motivate us to design and implement a load balanced and efficient `segmerge` primitive on many-core devices.

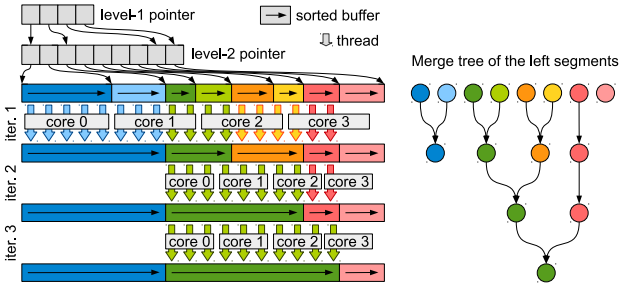



Figure 1: An example showing the proposed segmented merge algorithm. Here the *level-1 pointer* (including three segments) points to the *level-2 pointer* pointing to eight ordered sub-segments continuously stored in the array. After the segmented merge, the eight sub-segments are merged into three ordered segments.

## 2 The Proposed Segmented Merge Algorithm

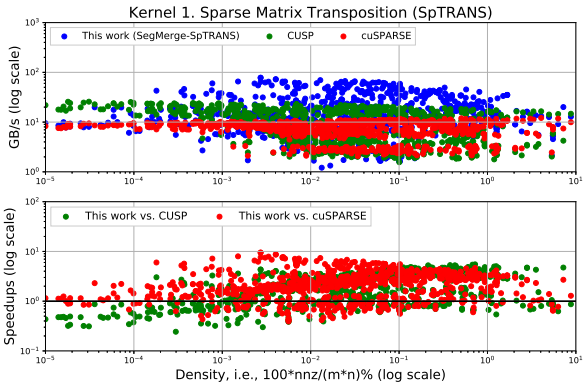
For easier understanding, we in this short paper explain our algorithm by using an example shown in Fig. 1.

First, to record starting and ending positions of segments and sub-segments, we construct two arrays of size $p+1$ and $q+1$ (where $q = \sum_{i=1}^{p} q_i$ is the total number of sub-segments), respectively, as two-level pointers (see the top left of Fig. 1). Overall, our method sees each sub-segment of the same segment as a leaf node of a binary tree (see the right part of Fig. 1), and merges them in the bottom-up manner on this binary tree. The algorithm works in an iterative way and completes when each segment only has one sub-segment.

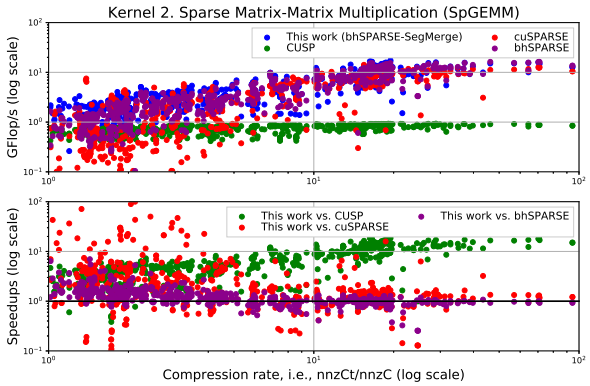
*Department of Computer Science, Norwegian University of Science and Technology, Norway. Email: weifeng.liu@ntnu.no

†Niels Bohr Institute, University of Copenhagen, Denmark. Email: {weifeng.liu, vinter}@nbi.ku.dk

‡Department of Computer Science and Engineering, Ohio State University, USA. Email: wang.2721@osu.edu




(a) Results of three SpTRANS methods, CUSP, cuSPARSE and `segmerge`-based, for transposing $A$ to $A^T$, both in CSR. The speedups are up to 5.5x and 9.49x, respectively.




(b) Results of four SpGEMM methods, CUSP, cuSPARSE, original bhSPARSE [6] and `segmerge`-based bhSPARSE, for computing $A^2$. The speedups are up to 19.06x, 171.27x and 6.98x, respectively.

Figure 2: Performance comparison using 956 square sparse matrices with $100k \leq nnz \leq 200M$ from the SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection [2]).

Furthermore, to achieve a balanced thread assignment, we set each thread's workload to a fixed amount, and count the number of threads required in merging every pair of sub-segments. As a result, each iterative step knows the total number of threads to be issued. Then each sub-segment scatters a group of information, such as global memory offset and segments size, to threads working on it. After that, each thread gathers information by running the partitioning strategy of the standard merge path algorithm [4]. Because now each thread has enough information to run independently, all of them are evenly dispatched to cores (see the left part of Fig. 1) and execute serial merge to complete an iteration.

Because the algorithm does not designate threads to segments already completed (see segments in blue and pink in Fig. 1), a smaller amount of threads may be issued in later iterations. Moreover, because all cores are saturated, our segmented merge method achieves good load balancing in each iteration.

## 3 Performance Evaluation and Conclusion

We use an NVIDIA Titan X Pascal GPU for comparing performance of several parallel implementations for two sparse kernels: sparse transposition (SpTRANS) and sparse matrix-matrix multiplication (SpGEMM) (see [10] and [6] for details). In the last round of SpTRANS, some sorted sub-rows are merged into final rows by calling `segmerge`. As for SpGEMM, the `segmerge` kernel is used for calculating long rows of the resulting matrix assigned to the last bin in [6].

It can be seen in Fig. 2 that the new SpTRANS and SpGEMM algorithms using `segmerge` are in general faster and sometimes much faster (by up to two orders of magnitude) than existing work.


## Acknowledgement

The research has received funding from the EU's H2020 Marie Sklodowska-Curie grant agreement No. 752321.


## References


[1] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors. Technical report, CMU, 1993.

[2] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 2011.

[3] Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli. Fast Scan Algorithms on Graphics Processors. In *ICS '08*, 2008.

[4] O. Green, R. McColl, and D. A. Bader. GPU Merge Path: A GPU Merging Algorithm. In *ICS '12*, 2012.

[5] K. Hou, W. Liu, H. Wang, and W. Feng. Fast Segmented Sort on GPUs. In *ICS '17*, 2017.

[6] W. Liu and B. Vinter. A Framework for General Sparse Matrix-matrix Multiplication on GPUs and Heterogeneous Processors. *Journal of Parallel and Distributed Computing*, 2015.

[7] W. Liu and B. Vinter. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *ICS '15*, 2015.

[8] W. Liu and B. Vinter. Speculative Segmented Sum for Sparse Matrix-Vector Multiplication on Heterogeneous Processors. *Parallel Computing*, 2015.

[9] D. Merrill and M. Garland. Merge-based Parallel Sparse Matrix-vector Multiplication. In *SC '16*, 2016.

[10] H. Wang, W. Liu, K. Hou, and W. Feng. Parallel Transposition of Sparse Data Structures. In *ICS '16*, 2016.

[11] C. Yang, Y. Wang, and J. D. Owens. Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU. In *IPDPS Workshop '15*, 2015.