HISC/R: An Efficient Hypersparse-Matrix Storage Format for Scalable Graph Processing

Robert Kirchgessner*, Giovanni De La Torre†, Alan D. George‡
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering
University of Florida, Gainesville, FL 32611
*kirchgessner@chrec.org †delatorre@chrec.org ‡george@chrec.org

Vitaliy Gleyzer
Lincoln Laboratory
Massachusetts Institute of Technology
vgleyzer@ll.mit.edu

***Abstract*—The need to analyze increasingly larger graph datasets has driven the exploration of new methods and unique system architectures for graph processing. One such method moves away from the typical edge- and vertex-centric approaches and describes graph algorithms using linear-algebra operations, bringing the added benefits of predictable data-access patterns and ease of implementation. The performance of this approach is limited by the sparse nature of graph adjacency matrices, which leads to inefficient use of memory bandwidth, and reduced scalability in distributed systems. In order to maximize the scalability and performance of these linear-algebra systems, we require new sparse-matrix storage formats capable of maximizing memory throughput and minimizing latency, while maintaining low storage overhead. In this paper, we present an overview of a novel sparse-matrix storage format called Hashed-Index Sparse-Column/Row (HISC/R) which guarantees constant-time row or column access complexity at low storage overhead, while also supporting online non-zero element insertions and deletions. We evaluate the performance of HISC/R using randomly generated Kronecker graphs, demonstrating a 19% reduction in memory footprint, and 40% reduction in memory reads, for sparse matrix/matrix multiplication compared to competing formats.**

I. Introduction

Large-scale graph processing is a key component in modern scientific computing and data analytics, with many commercial and defense applications [1, 2]. Graph-processing applications, however, do not map well to conventional system architectures. Whereas conventional systems focus on maximizing computational throughput and data locality and reuse, graph-processing problems are typically memory-bounded and data-driven, with highly irregular datasets [3]. These problems are further compounded in distributed systems, where the unstructured nature of graph datasets leads to inefficient data partitioning and load imbalances. The need to analyze increasingly larger graph datasets has driven the exploration of new methods, algorithms, and distributed system architectures for graph processing.

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

One such method moves away from the typical edge- and vertex-centric approaches and describes graph algorithms in terms of linear-algebra primitives operating on graph adjacency matrices [4]. This approach brings with it the benefits of the predictable access patterns of linear-algebra operations, and a higher level of abstraction simplifying the implementation and parallelization of many graph algorithms [4]. In order to maximize the scalability and performance of this approach, however, low-overhead storage formats capable of providing scalable, low-latency access to data are critical [5].

Many storage formats have been developed which are optimized for different non-zero distributions and platform architectures. The most commonly used sparse-matrix storage formats for graph processing are Compressed Sparse-Column/Row (CSC/R) and Doubly Compressed Sparse-Column/Row (DCSC/R) [6]. These formats, however, trade-off between storage and lookup complexity, providing either fast lookups at the expense of increased storage overhead, or low storage overhead at the expense of increased access time for unfavorable non-zero distributions.

In order to overcome these limitations, we propose a novel sparse-matrix storage format called Hashed-Index Sparse Column/Row (HISC/R). HISC/R uses a hashed pointer vector and segmented-storage vector which provides constant-time accesses to row or columns of a matrix with low storage overhead, and enables online non-zero insertions and deletions. Additionally, HISC/R optimizes the storage of hypersparse matrices by allowing non-zero elements to be stored directly in the hashed pointer vector. In this paper, we provide an overview of HISC/R, and demonstrate the storage and lookup performance of HISC/R compared to CSC/R and DCSC/R using randomly generated Kronecker graphs. We also show that HISC/R requires up to 40% less memory reads compared to DCSC/R when performing SpGEMM, and uses up to 19% less storage for hypersparse datasets.

The remainder of this paper is organized as follows. Section II overviews competing sparse-matrix storage formats for graph processing. Section III presents and overview of the Hashed-Index Sparse-Column/Row format. Section IV presents our experimental results comparing the storage and lookup performance of HISC/R against competing formats. Finally, Section V presents our conclusions.

978-1-5090-3867-1/16 $31.00 © 2016 IEEE
DOI 10.1109/IA3.2016.18

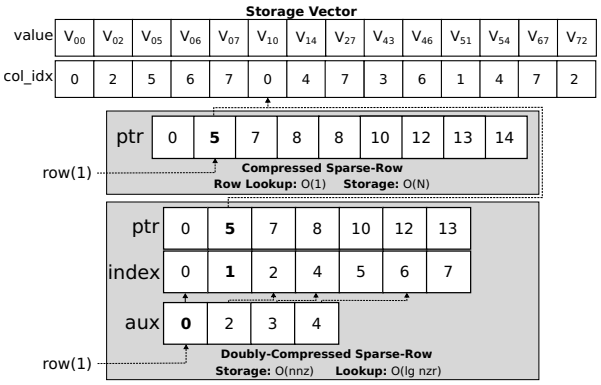

Fig. 1. Comparison of indexing techniques used by CSC/R and DCSC/R.

II. Sparse-Matrix Format Overview

Various sparse-matrix storage formats [7, 8, 9, 10] have been developed to maximize algorithmic performance for different non-zero distributions and hardware architectures. We define an optimal format as one that enables constant-time lookup complexity for row or column elements while maintaining $O(nnz)$ storage, where nnz is the number of non-zero elements in the matrix. In practice, however, sparse-matrix storage formats must compromise between maximizing lookup performance or minimizing storage overhead.

Compressed Sparse-Column/Row (CSC/R) is the most commonly used sparse-matrix storage format for graph processing due to its simplicity and good performance [11]. CSC/R encodes matrices using three vectors: the pointer, index, and value vectors, as shown in Figure 1. The value and index vectors are sparse vectors which store the corresponding values and non-major indices of the non-zero elements of CSC/R in column/row-major order. The pointer vector is a dense vector which contains an offset into the index and value vectors for the start of each column/row for CSC/R. The dense pointer vector enables constant-time indexing into the start of rows and columns at the expense of significant storage overhead when dealing with distributed sparse or hypersparse matrices.

To overcome this storage limitation, the Doubly Compressed Sparse-Column/Row format [6] replaces the dense pointer vector with a sparse pointer vector, only storing entries for non-zero rows or columns as shown in Figure 1. Since the pointer vector is sparse, another index vector is used to store the row/column index associated with each pointer. By using a sparse pointer vector, we must now search for each row/column, increasing the lookup complexity to $O(nzc/r)$, where nzc/r is the number of non-zero columns/rows. In order to minimize the search overhead, DCSC/R introduces an AUX array which breaks the non-zero rows/columns into blocks and stores a pointer to the first non-zero of each block. Although DCSC/R solves the scalability issues of CSC/R by eliminating the dense pointer vector, the introduction of a sparse vector requires a search on lookup and may significantly increase the lookup overhead and limit performance.

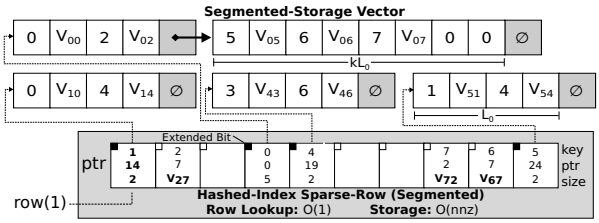

Fig. 2. Overview of HISC/R using segmented storage.

III. Hashed-Index Sparse-Column/Row

Hashed-Index Sparse-Column/Row (HISC/R) uses a hashed pointer vector rather than the dense pointer vector used in CSC/R, or the sparse pointer vector used in DCSC/R, as illustrated in Figure 2. When looking up the non-zero values of a column or row, we use a hash function to generate an offset into the hashed pointer vector, verify the key, and use the pointer and size entries to iterate over the non-zero elements. The hash-table type and load factor, α, determines the achievable storage and lookup performance of HISC/R.

A. Hashed pointer vector

Each bucket in the hashed pointer vector consists of three entries: the key (column/row index for HISC/R respectively), a pointer into the non-zero value/index vectors, and the size of the current column or row. In order for HISC/R to achieve high performance, we need a function $h : M \rightarrow \{0, ..., B - 1\}$ for a hash table with B buckets, that provides sufficient uniformity regardless of the non-zero distribution. We select the initial number of buckets, B, targeting a load factor of 0.75 based on the expected number of non-zero rows or columns in the matrix. HISC/R uses tabulation hashing [12], a strongly universal$_3$ family of hash functions [13] which provides good uniformity guarantees regardless of the non-zero row/column distribution with low computational complexity.

Although simple collision-resolution techniques such as linear or quadratic probing, and double hashing, provided good lookup performance when $\alpha < 0.6$, we must turn to more complex hash-table designs to achieve higher load factors. Hopscotch hashing [14] combines various techniques from multiple-choice and relocation hashing, linear probing, and chaining to provide a compromise between lookup performance and load factor. Our experiments indicates that hopscotch hashing outperforms the other explored hash-table types for HISC/R by achieving a load factor of up to 83%, with an average of 1.4 probes per lookup.

B. Segmented-storage vector

Although HISC/R can use a single index/value array similar to CSC/R and DCSC/R, we instead use a segmented-storage vector which enables online insertions and deletions. The segmented-storage vector breaks rows or columns with more than one non-zero element into variable-sized sublists with initial size L_0. Each sublist contains either null or a pointer to the next sublist of size $k^d L_0$, where k is a customizable multiplier, and d is the current sublist depth. Unused elements

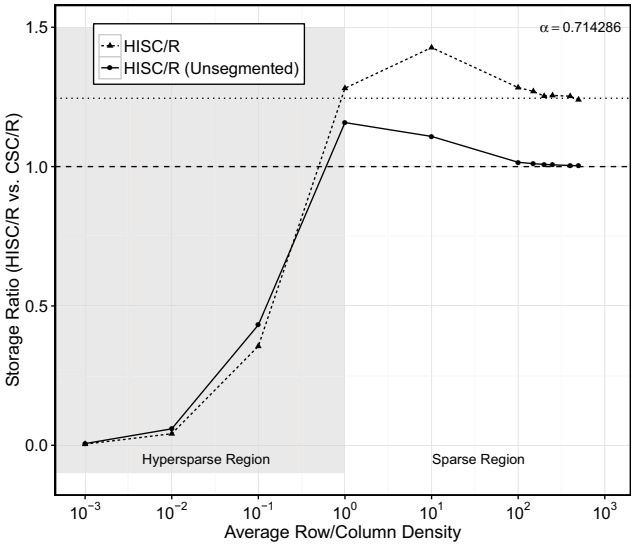

Fig. 3. Comparison of HISC/R and CSC/R storage performance for randomly generated scale-30 Kronecker matrices.

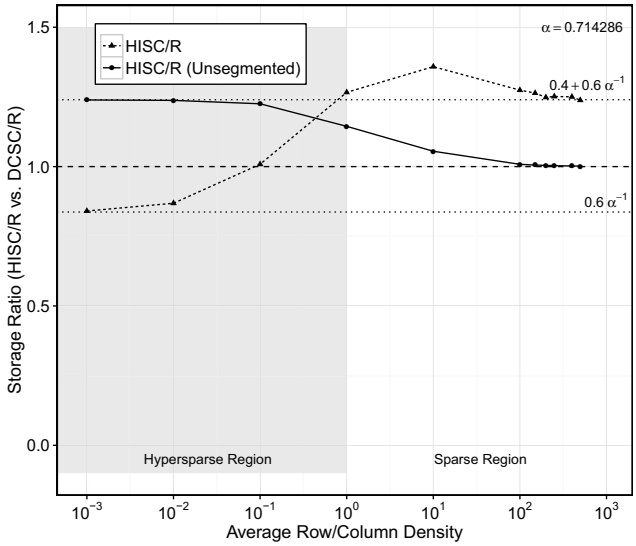

Fig. 4. Comparison of HISC/R and DCSC/R storage performance for randomly generated scale-30 Kronecker matrices.

of each sublist are initialized to zero to indicate that they can be inserted into, as shown in Figure 2. The parameters L_0 and k provide a method to tune the segmented-storage vector, allowing users to minimize the storage overhead and number of segments for a particular dataset. The optimal values for these parameters depend on the properties of the matrix being stored, and can be determined experimentally.

In cases where there is only one non-zero in a row or column, as is the common case for hypersparse matrices, we store the non-zero value directly in the hash table. The extended bits shown in Figure 2 are used to indicate whether we are storing a non-zero value or the start of a segmented vector in the hash table. When storing a non-zero directly in the hash table, we place the major index in the key, the minor index in the pointer position, and the value in the size position.

IV. Results

In this section we compare the storage and lookup performance of HISC/R with CSC/R and DCSC/R. Our experiments use randomly generated scale-30 Kronecker matrices [15] (adjacency matrix of size 2^{30} by 2^{30}) with edge factors varying from 10^{-3} to 10^{3}. We use HISC/R with hopscotch hashing and segmented storage with the parameter values $L_0 = k = 2$. Unsegmented HISC/R uses a single value and index array similar to CSC/R and DCSC/R, and does not store non-zero elements in the hashed pointer vector. The figures are divided into hypersparse and sparse regions in order to illustrate the storage format behavior for different levels of sparsity.

A. Storage comparison

We compare storage performance by calculating the ratio of the number of bytes to store matrices using HISC/R to CSC/R and DCSC/R. Figure 3 compares the storage performance of HISC/R with CSC/R. HISC/R and unsegmented HISC/R greatly outperforms CSC/R in the hypersparse region, as indicated by the storage ratio approaching zero asymptotically. As the matrix density approaches an average of one non-zero per row/column, the overhead from the dense pointer vector of CSC/R decreases, causing the storage ratio to increase. The storage ratio of HISC/R peaks around 1.4 at an average of 10 non-zero elements per row/column and then asymptotically approaches 1.25 as the matrix becomes denser.

Figure 4 compares the storage performance of HISC/R with DCSC/R. HISC/R achieves a storage ratio of 0.85 in the hypersparse region by storing non-zero elements directly in the hashed pointer vector. Unsegmented HISC/R approaches a storage ratio of 1.25 in the hypersparse region due to the unused buckets in the hashed pointer vector. As the matrix becomes denser, the number of rows and columns with more than one non-zero element increases, increasing the number of storage segments. Due to the unused elements in the segmented-storage vectors, the storage ratio peaks around 1.35 in the sparse region at 10 non-zero elements per row/column, and then approaches 1.25 asymptotically as the matrix becomes denser. Optimizing the parameters L_0 and k for the dataset being stored would minimize this overhead, and will be explored in our future work.

B. Performance comparison

We evaluate the lookup performance of HISC/R using sparse generalized matrix-matrix multiplication (SpGEMM). SpGEMM is a key kernel used for many graph-processing applications including all-nodes shortest paths, and betweenness centrality. We measure the total memory read operations required for SpGEMM when using CSC/R, DCSC/R, and HISC/R, for varying degrees of sparsity. We measure only the total memory reads needed to perform SpGEMM, and do not assume a particular hardware architecture. We compare the percent improvement of HISC/R over CSC/R and DCSC/R in

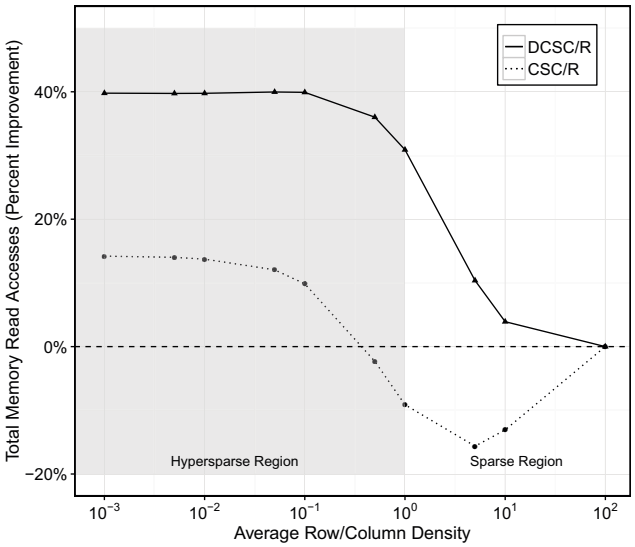

Fig. 5. Comparison of total reads required to perform sparse matrix/matrix multiplication using HISC/R compared with CSC/R and DCSC/R.

terms of the reduction in total memory accesses required to perform the computation.

Figure 5 presents the percent improvement in the total number of memory reads required by HISC/R compared to CSC/R and DCSC/R. HISC/R provides an improvement of up to 40% and 14%, compared to DCSC/R and CSC/R for hypersparse datasets, respectively. The improvement of HISC/R in the hypersparse region is a result of storing non-zero elements directly in the hashed pointer vector, reducing the number of indirect memory accesses compared to CSC/R and DCSC/R. In the sparse region, HISC/R requires up to 16% more memory accesses than CSC/R due to having to decode additional pointers for the segmented-storage vectors. As the matrices become denser, this overhead decreases as the additional segment lookups are amortized by the increasing segment size.

V. Conclusions

In this paper, we present an overview of Hashed-Index Sparse-Column/Row (HISC/R), a novel sparse-matrix storage format optimized for graph-processing applications. HISC/R provides $O(1)$ lookup complexity and $O(nnz)$ storage complexity while also enabling runtime insert and delete operations, enabling matrices to be constructed directly without using expensive intermediate storage formats. We show that HISC/R requires significantly less storage than CSC/R and up to 19% less than DCSC/R for hypersparse datasets when maintaining an average hash-table load-factor of 71%. Additionally, we show HISC/R provides up to a 14% and 40% improvement in terms of memory reads compared to CSC/R and DCSC/R, respectively, when performing matrix multiplication with hypersparse datasets. The reduction in the total number of memory accesses and favorable storage performance for hypersparse datasets make HISC/R uniquely suited for scalable graph processing.

Acknowledgement

This work was supported by CHREC members and the I/UCRC Program of the National Science Foundation under Grant No. IIP-1161022. We gratefully acknowledge tools and devices provided by Altera.

References

[1] F. Riaz and K. M. Ali. “Applications of Graph Theory in Computer Science”. In: *Computational Intelligence, Communication Systems and Networks (CICSyN), 2011 Third International Conference on*. 2011, pp. 142–145.

[2] L. Ball. “Automating social network analysis: A power tool for counter-terrorism”. In: *Security Journal* 29.2 (2016), pp. 147–168.

[3] A. Lumsdaine et al. “Challenges in Parallel Graph Processing”. In: *Parallel Processing Letters* 17.01 (2007).

[4] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2011.

[5] A. Eisenman et al. “Parallel Graph Processing: Prejudice and State of the Art”. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE ’16. New York, NY, USA: ACM, 2016, pp. 85–90.

[6] A. Buluc and J. R. Gilbert. “On the representation and multiplication of hypersparse matrices”. In: *2008 IEEE International Symposium on Parallel and Distributed Processing* (2008), pp. 1–11.

[7] W. Armstrong and A. P. Rendell. “Runtime sparse matrix format selection”. In: *Procedia Computer Science* (2010), pp. 135 –144.

[8] N. Bell and M. Garland. “Efficient Sparse Matrix-Vector Multiplication on CUDA”. In: *Nvidia Technical Report* (2008), pp. 1–32.

[9] E. Montagne and A. Ekambaram. “An optimal storage format for sparse matrices”. In: *Information Processing Letters* 90.2 (2004), pp. 87–92.

[10] I. Imecek, D. Langr, and P. Tvrdik. “Minimal Quadtree Format for Compression of Sparse Matrices Storage”. In: *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (2012).

[11] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[12] M. Patrascu and M. Thorup. “The Power of Simple Tabulation Hashing”. In: (2011), pp. 1–48.

[13] M. N. Wegman and J. L. Carter. “New hash functions and their use in authentication and set equality”. In: *Journal of Computer and System Sciences* 22.3 (1981).

[14] M. Herlihy, N. Shavit, and M. Tzafrir. “Hopscotch Hashing”. In: *Distributed Computing: 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24*. 2008, pp. 350–364.

[15] J. Leskovec. “Kronecker Graphs : An Approach to Modeling Networks”. In: 11 (2010), pp. 985–1042.