A Scale-Free Structure for Power-Law Graphs

Richard Veras, Tze Meng Low and Franz Franchetti
Department of Electrical and
Computer Engineering
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
Email: {rveras, lowt, franzf}@cmu.edu

***Abstract*—Many real-world graphs, such as those that arise from the web, biology and transportation, appear random and without a structure that can be exploited for performance on modern computer architectures. However, these graphs have a scale-free graph topology that can be leveraged for locality. Existing sparse data formats are not designed to take advantage of this structure. They focus primarily on reducing storage requirements and improving the cost of certain matrix operations for these large data sets. Therefore, we propose a data structure for storing real-world scale-free graphs in a sparse and hierarchical fashion. By maintaining the structure of the graph, we preserve locality in the graph and in the cache. For synthetic scale-free graph data we outperform the state of the art for graphs with up to 10^7 non-zero edges.**

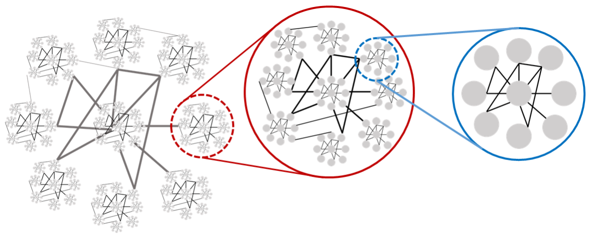

Fig. 1. In this cartoon, we demonstrate the recursive nature of a scale-free graph. At the highest view, we see that a few key vertices (hubs) contain the majority of the edges. We can partition the graph based on these key vertices. If we descend into a sub-graph created by this partitioning, we see the same behavior.

I. Introduction

The goal of this work is to provide an efficient data structure for storing and computing on real-world scale-free graphs. We target scale-free graphs because many real-world networks share that topology [1], [2]. In these graphs, there are a small number of nodes with many edges and many nodes with only a few edges. The nodes with many connections are referred to as hub nodes and they connect many neighboring nodes within their cluster. If we zoom into one of these neighborhoods we would see this behavior repeat, as illustrated by Figure 1. In this paper, we leverage the characteristics of scale-free graphs in order to create an efficient data structure that provides fast access to clusters of nodes in a hierarchical fashion.

Our data structure uses a hierarchical sparse data format to efficiently map scale-free graphs to modern computer architectures. This format captures the structure of the graph at multiple levels of resolution. By varying the resolution at a given level, we can fit that sub-graphs to the caches of the target system. This allows us to maximize cache reuse and minimize bandwidth requirements for Sparse Matrix-Vector (SPMV) like operations. Additionally, by using a sparse encoding at each level of the hierarchy, we can reduce the storage requirements of our format.

We recognize that the domain expert has extremely valuable knowledge on the structure of their graph and the details underlying the problem from which it came. In order to capture this structure, we rely on the domain expert to provide a mapping function between the structure of the graph and our data structure.

A. Our Contribution

This work can be broken into three key pieces.

- We provide a hierarchical sparse format called Recursive Matrix and Vector (RMV). This hierarchical format captures the structure of the graph using a tree of specialized containers.
- We provide an Application Programming Interface (API) that allows the domain expert to construct these RMV objects from domain knowledge. The expert can pass the structural information of the graph to our data structure.
- Lastly, we perform a performance analysis of our format using synthetic scale-free graphs.

II. Related Work

The purpose of our structure is to preserve the locality of the graph in the memory hierarchy. This is motivated by the ideas in [3]. The authors suggest that when distributing a matrix on a parallel machine, it is more important to distribute a matrix based on the elements of the vector rather than on the structure of the matrix. For example, if we discritize the body of an airplane for a simulation, we would want to keep the vertices from the wings near each other, the vertices on the fuselage clustered together and the vertices on the engine close to each other, because these neighboring vertices are more likely to interact with one another. Clustering these neighboring vertices in memory would insure that these interactions occur in cache.

In [4] the authors leveraged this idea of preserving the structure of the problem in memory for hp-FEM. They did this by using the original structure of the problem to guide its assembly in a hierarchical dense matrix storage format [5]. By utilizing both domain knowledge and a data structure that is amenable to hierarchical storage, the authors bypass the need for a graph partitioner. This is possible because they can insure that physical elements neighbor each other in memory.

In a similar vein, we want to use the structure to partition and store real-world graphs in a hierarchical fashion. Our work provides a hierarchical data structure and a mechanism that allows the expert to pass structural knowledge. This knowledge the user provides – using our API – guides the construction of the hierarchical data structure.

In the airplane example, we partitioned the vertices based on the mechanical component that contained them. Those vertices that exist in the same component are more likely to communicate with each other than with distant neighbors. Like the mechanical components, the neighboring vertices around a well connected vertex hub form a well connected clusters in a scale-free graph. We would like to preserve this graph structure in the memory hierarchy, such that a cluster would take advantage of cache locality. To achieve this we would want to partition our graph about those hubs. The domain expert would need to provide our framework with a mapping function that captures this partitioning.

a) Existing Frameworks: If the user can provide us with domain knowledge of the graph's structure, then we need a data format that can mirror this structure. In this section, we discuss existing sparse data formats and how they compare and contrast to our format.

This simplest of the sparse formats is the coordinate storage format (COO). This format is composed of a list of edges, described as a tuple of: the starting vertex, the ending vertex and the weight. This format provides compression over a dense matrix by only storing the nonzero elements. The Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) format provide additional compression over the COO format. They do this by compressing all of the edges in the same row (or column) into a pair of lists containing the column (or row) element and the edge weight. While COO, CSC, and CSR provide compression over dense storage, they do not allow for blocking for locality. Thus, we cannot use these formats for hierarchical storage of our graph.

Hierarchically Tiled Arrays (HTA) [6] are a set of C++ objects for storing dense arrays in a hierarchical fashion. The goal of this project is to concisely describe parallelism using these hierarchical objects. Similarly, FLASH [5] provides a hierarchical representation for dense matrices. This API abstracts away the complex indexing associated with hierarchical matrices and allows the user to implement dense linear algebra routines over matrices that are stored as blocks. Both of these libraries provide an API for describing matrices hierarchically. A user can optimize their matrix operations for their target hardware by varying the depth of this hierarchy and by varying the dimensions of the blocks at each level. However, both formats target dense data, and we want to preserve sparsity at every depth of the hierarchy.

The Recursive Sparse Blocks (RSB) [7] store sparse matrices hierarchically. It does this by sub-dividing the matrix in a quad-tree fashion. It continues to sub-divide all blocks until each block only contains a small number of elements. These blocks are then small enough to fit in the cache of the target system. Their work imposes a specific partitioning of the matrix. Additionally, it fixes the access pattern for SPMV to a Z-Morton ordering. Our work differs in that the partitioning is determined by user knowledge, and the access pattern is also determined by the user.

The Optimized Sparse Kernel Interface (OSKI) [8] is a program generation system for implementing high performance SPMV kernels on a specialized data format. This specialized format can be viewed as a sparse collection of dense blocks. Their interface allows the user to embed domain knowledge as hints that affect the block size of their format. Compressed Sparse Block (CSB) [9] can be viewed as a dense collection of sparse blocks. This format gives no preference over computing on rows versus columns. In this way, one can consider it as a blocked version of the COO format. Their library provides efficient implementations of transposed and non-transposed SPMV. Their existing format only provides one level of tiling, whereas we want many levels of tiling. Both of these works show that tuned kernels are necessary for the data format being used. Our work leverages their ideas, to provide a hierarchical sparse collection of sparse blocks.

Lastly, the Spatio-Temporal Interaction Network and Graph Extensible Representation (STINGER) [10] is a graph library and data format for performing analytics on time varying graphs. They use a specialized data structure to accommodate frequent updates to the graph, and to facilitate parallel operations over the graph. Their interface allows the user to provide a mapping function between physical vertex ID in the graph and their logical ID in memory. Our interface also provides the user with the ability to define this mapping function. This allows the user to provide structural information to the constructor of our data format. In the following section we describe our format.

III. Proposed Mechanism

In this section, we describe our hierarchical sparse data format and its constructor function. If the user provides our framework with domain knowledge about the structure of the graph, then our framework can construct both a sparse matrix and dense vector in our format that preserves this structure in memory.

b) Data format: The key component to our storage scheme is the hierachical Recursive Matrix or Vector (RMV) element. The graph is stored recursively by blocks inside this container. The graph is represented at multiple granularities (Figure 2). At the very top this block contains the entire graph below it and at the very bottom the blocks contain pointers to the weights of the edges or values of the vertices.

```
struct recursive_dense_vector{
 type; size;
 values; };
```

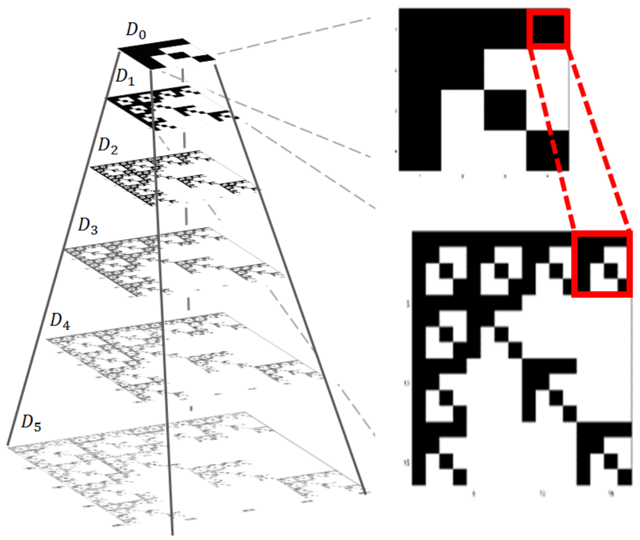

Fig. 2. In this figure, we show how the incidence matrix of the graph is stored hierarchically using our format. At the depth d_5, we have a view of what the original incidence matrix. We store the graph hierarchically, so as we move up this pyramid, the graph is coarsened. Each element in the levels above d_5 is a pointer to the elements in the level below it.

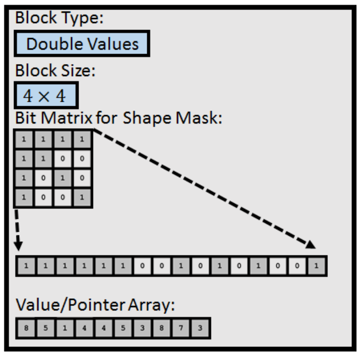

Fig. 3. This figure represents an instantiated Recursive Sparse Matrix. It contains the shape of a sparse graph as a bit matrix and the corresponding values stored as a dense list. Note that the bit matrix can be treated as an integer, which can be used to select a kernel specialized to that specific shape.

```
struct recursive_sparse_matrix{
 type; size;
 bitmask_matrix;
 values; };
```

In each Recursive Matrix or Vector container, there is a field that determines if its elements point to containers at a finer level of the graph, or if they point to the actual values. Next, it contains the number of rows and columns of blocks that it can access. Additionally, there is a bit matrix that describes the pattern of the non-zero elements. Lastly, there is an array that either containers pointers to the next level of blocks or values if we are the final level. In Figure 3 we show an instantiation of this object. We then show in Figure 4 a fully constructed Recursive Sparse Matrix of height 2. The top level provides a coarse view of the matrix, and each of its elements point to a Recursive Sparse Matrix on the bottom level.

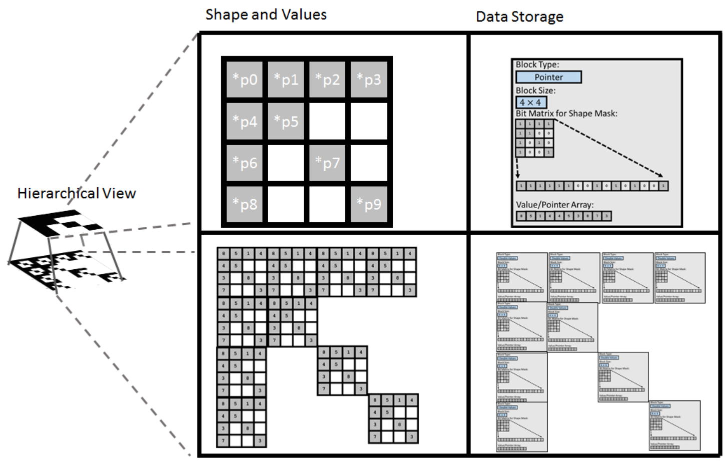

Fig. 4. We have a two level Recursive Sparse Matrix object. The top level captures the shape of the bottom layer. Additionally, each of its elements points to the Recursive Sparse Matrix objects on the bottom level.

c) Construction: The user provides the structural information of the graph to our framework. In order to enable the assembly of these RMV objects based on the user's knowledge, we provide a tree based descriptor. This tree captures the recursive partitioning of the vertices in the graph as a nesting of partitionings. The nodes of this tree are shape descriptors and they are generated by a user provided `get_child` function.

```
struct shape_desc{
 depth; nnz; rows; cols;
 void *user_data
 void *get_child;
};
```

Each of these descriptors represents the shape of the graph at its assigned depth. The first four parameters give the coarse structure of the sub-graph being viewed. The user data field allows the user to pass bookkeeping information to the `get_child` function during the formation of this tree.

```
desc_child = get_child(i,j, desc_parent )
```

Using these descriptors and functions, we can assemble a RMV object according to a user defined partitioning. We have used this tree based approach for assembling the Recursive Matrix object from synthetic data, as well as, from COO formatted sparse matrix data.

```
assemble_recursive_matrix(
      recursive_sparse_matrix *head,
      shape_desc              *parent )
{
 head->values = malloc( parent-> nnz )

 for i,j in parent->rows,cols
  if child = parent->get_child(i,j,parent)
     != NULL
   mark_bit_mask(i,j, head )
   assemble_recursive_matrix
       ( head->values[p++], child )
}
```

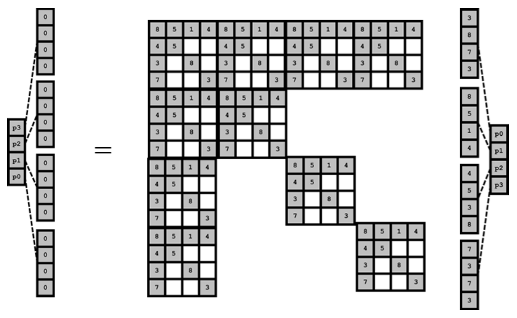

Fig. 5. Here we illustrate a blocked Sparse Matrix-Vector Multiply (`spmv`). The output and input vector are stored as Recursive Dense Vectors with a depth of 2. The top level of the vectors are 1×4 vectors of pointers, that point to 1×4 sub-vectors of scalar elements. The matrix is stored as a Recursive Sparse Matrix, also with a depth of 2. The top level is a 4×4 sparse matrix, where each non-zero points to a 4×4 sparse matrix of scalars.

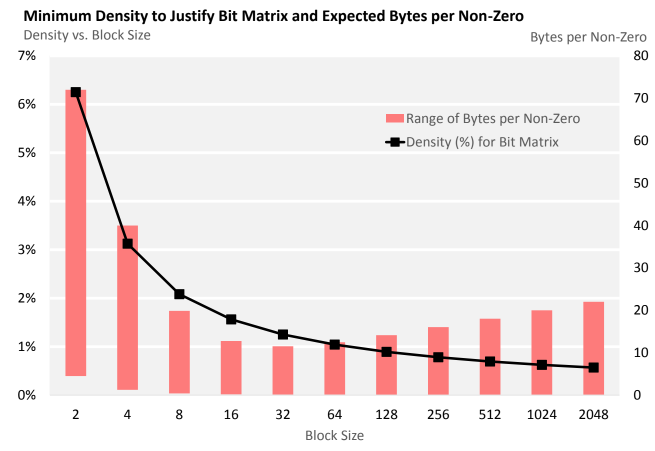

Fig. 6. In this analysis, we determine the minimum block density needed to justify the use of a bit matrix over a COO style list for a range of block sizes. If the user can partition their graph into blocks that maintain these densities, then the graph will reap the benefits of our storage mechanism. On the secondary axis we show the possible range of overhead of the RMV structure, measured in Bytes per Non-Zero, for each given block size. The denser the block, the lower the overhead.

In the pseudo code snippet listed above, we capture the essence of how a Recursive Matrix is constructed recursively from the shape tree. In the snippet we show that at each descent into the recursive matrix we simultaneously descend into the user provided shape tree that describes the graph. The construction of the Recursive Vector follows a similar approach based on these tree shape descriptors.

d) Computation: Computing a Sparse Matrix-Vector Multiply (`spmv`) using the RMV object is equivalent to performing a blocked Matrix-Vector Multiply. Functionally, it entails a recursively blocked `spmv`, that descends down the matrix and vector objects until it reaches the actual data values. This is illustrated in Figure 5. Note that the sub-vectors are reused across many elements in the matrix, so they are kept in the cache. The amount of reuse is dependent on how the user partitioned the vertices of the graph.

An additional feature of our structures is the ability to dispatch to specialized functions. In each container for the sparse matrices, there is a bit matrix which determines the shape at that level. This matrix can be treated as a single integral type and used to dispatch the children to a specialized version of that function. The specialized function can be optimized to include the indirect indexing directly in the code.

```
spmv_dispatch( y, mat, x){
  switch(mat->bit_matrix)
  ..
  case 65:
    spmv_65(y,mat->values, x)
}
```

The corresponding spmv_65 function can be fully unrolled and optimized to avoid indirect address computation. As long as the size of the instruction cache permits, we can specialize to all potential matrix shapes for a given block size.

e) Data Structure Analysis:

$$s_{\text{rmv}}(n) = s_{\text{type}}(n) + s_{\text{size}}(n) + s_{\text{nnz}}(n) + s_{\text{mask}}(n) + s_{\text{ptr\_mask}} + s_{\text{ptr\_vals}} \quad (1)$$

Where s_{type} is the block type, s_{size} are the dimensions of the block, s_{nnz} is the number of non zeros in this block, s_{mask} is the actual bit mask matrix, and $s_{\text{ptr\_mask}}$ and $s_{\text{ptr\_vals}}$ are the bit mask and value pointers, respectively.

These blocks only store a small local sub-graph, so the size of the data types used can be minimized to the number of bits needed to encode a block of that size. We can represent these sizes s as functions of n, where n represents the size of an $n \times n$ block that we would like to store in our RMV Structure. We can replace the size s terms with the following functions:

$$\begin{aligned} s_{\text{type}}(n) &= 8b \\ s_{\text{size}}(n) &= 2log_2(n) \\ s_{\text{nnz}}(n) &= 2log_2(n) \\ s_{\text{mask}}(n) &= n^2 \\ s_{\text{ptr\_mask}}(n) &= 64b \\ s_{\text{ptr\_vals}}(n) &= 64b \end{aligned} \quad (2)$$

This overhead formula is only for a single block. If we assemble a graph in a hierarchical fashion using blocks of size b (fixed size blocks are not a requirement of our structure), then our overhead becomes:

$$s(n,b) = \frac{1 - \frac{1}{r^k}}{1 - \frac{1}{r}} s_{\text{rmv}}(b) \quad (3)$$

Where $k = \log_b n$, $r = db^2$ and d is the average density of the graph in each block.

If we make n arbitrarily large, then $s_{\text{mask}}(n)$ becomes the dominant factor. If we store a large graph as a single block in our storage scheme, then the overhead would be unnecessarily large compared to Coordinate Storage (COO). However, our scheme is designed to store a scale-free graph hierarchically such that the density of the leaves is greater than the entire graph. Thus, if the density of the blocks is sufficiently large, then this overhead is amortized over many elements. Otherwise, a COO list should be used for that block, a feature that our format allows. Thus, we need to balance the size of our blocks with the density of the sub-graph that they will store.

Achieving this desired density for larger block sizes may not be practical for real world scale-free graphs. Fortunately, we do not need a high density of the overall graph, only high densities in tightly clustered sub-graphs. Thus, the question becomes: given a sub-graph of size n, what minimum density is needed to overcome the overhead of our storage format, RMV, compared to COO? In Figure 6 we compare necessary density to break even in overhead for a given block size. This is computed using by solving for density d in $\text{mask}(n) = \text{coo}(n)\text{nnz}$, where $\text{nnz} = dn^2$. For a range of block sizes, this plot shows what density is needed to justify the use of the bit matrix over COO inside the RMV structure. In that plot, we also calculate the range of the overhead of our RMV mechanism for each block size.

IV. Experimental Setup and Analysis

In this section, we evaluate the performance of our data format for synthetic scale-free graphs. We want to show that we can achieve reasonable spmv performance for a single threaded, scalar, and un-optimized implementation of our framework. Additionally, we show that we achieve competitive performance with the state of the art. We chose spmv as a proxy for graph operations for the following reasons: First, many graph operations can be represented in terms of iterative spmv-like operations. Second, this operation is typically expertly tuned, so it sets a high bar for performance.

Synthetic Dataset: For our datasets, we generate synthetic Discrete Kronecker Graphs [11] of various sizes. We chose these graphs because they approximate scale-free graphs. Using Kronecker graphs we can control the sparsity, number of non-zeros and graph size in a predictable fashion. The construction of the Kronecker Graphs used in our experiments is as follows:

We start with an initiator matrix B_i, which in our case is the arrowhead pattern.

$$B_1 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad (4)$$

We construct larger Kronecker Graphs B_i for $i > 1$ using the Kronecker Tensor $\otimes$ in this formula $B_i = B_1 \otimes B_{i-1}$. The term on the left-hand side describes the coarse structure of the matrix, where the term on the right-hand side describes the fine grain structure. For our arrowhead initiator B_1, we can visualize this matrix as:

$$B_{i+1} = \begin{bmatrix} B_i & B_i & B_i & B_i \\ B_i & B_i & 0 & 0 \\ B_i & 0 & B_i & 0 \\ B_i & 0 & 0 & B_i \end{bmatrix} \quad (5)$$

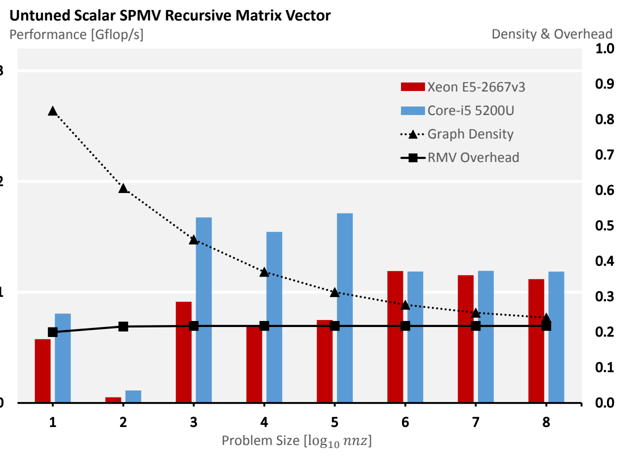

Fig. 7. On the primary y-axis we measure the performance of an untuned scalar spmv on our RMV data structure for graph sizes ranging from 10 to 10^8 non-zero edges. For the problems larger than 10^6 the graph data exceeds the size of the caches on both systems, with the largest graph containing 800MB of graph data. The untuned scalar implementation achieves between 10% to 40% of the target systems' double precision floating point peak performance for synthetic data. On the secondary axis, we compare the graph's density relative to the overhead imposed by our data structure as the problem size grows. For comparison the overhead for COO is 66%.

While our format does not require the graph to be a Kronecker Graph, using it allows us to quickly compute the number of non-zeroes (or edges), the number of vertices and the density. We can then relate these features to the performance of our implementation.

$$\begin{aligned} \text{nnz}(B_i) &= 10^i \\ \text{num\_verts}(B_i) &= 4^i \\ \text{density}(B_i) &= \frac{\text{nnz}(B_i)}{\text{num\_verts}(B_i)^2} \\ &= \left(\tfrac{2}{5}\right)^i \end{aligned} \quad (6)$$

We selected these graphs because they represent the ideal case for our data format, because we can base our partitioning on the mathematical representation of the Kronecker Graph. However, our data format is not limited only to Kronecker Graphs. It can store arbitrary sparse graphs, but we only expect to see performance benefits if the user can map the structure of their graph to our Recursive Matrix data structure.

Test Bench: Our target systems include: an Intel Core i5-5200U running at 2.20 GHz with a memory bandwidth of 25.6 GB/s and an Intel Xeon E5-2667 v3 running at 3.2 GHz with 68 GB/s of memory bandwidth. The theoretical peak spmv performance on these machines are 6.4 GFLOP/s and 12.75 GFLOP/s respectively. This assumes that the vectors are resident in the cache and that for every 8B consumed, 2 FLOPs are performed.

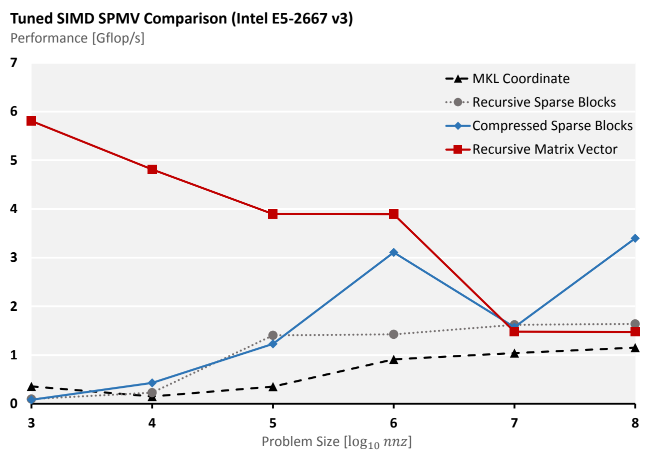

Fig. 8. In this experiment, we compare the performance of a tuned SIMD `spmv` implementation on our RMV data structure against state of the art implementations on synthetic scale-free data. This data set exceeds the cache and is 800MB, excluding overhead. Our RMV implementation outperforms the other implementations until 10^7 non-zero elements. We suspect that by rearranging our data layout we can make more effective use of the large number of channels on this system.

A. *Performance Analysis*

In order to demonstrate the effectiveness of our data structure for matrix-vector like operations on scale-free graphs, we evaluated two variants of our framework: a scalar and Single Instruction Multiple Data (SIMD) implementation.

The goal of our scalar experiment (Figure 7) is to show the performance of a minimally tuned scalar `spmv` implementation using our RMV data structure over synthetic scale-free data. We do this to establish a baseline of what is achievable by using our data-structure and we relate this to the density of the graph and the overhead of our format. What we see in Figure 7 is that as the problem size grows larger, the density of the synthetic scale free graph decreases, but the fraction of overhead introduced by our structure converges to 25%. For comparison this value is 66% for COO. We also see that for our untuned scalar implementation, the RMV `spmv` sustains 10% and 40% peak floating point performance on the Xeon E5 and Core i5, respectively.

In Figure 8 we compare an optimized version of our data-structure and `spmv` framework against state of the art implementations. We use SIMD short vector instructions to compute the inner-most 4×4 blocks of the `spmv`. These kernels specialize to the shape of sub-graph, which dispatch on the bit-matrix and only compute on the non-zero elements for that particular mask. Additionally, we chose block sizes that fit the sub-graphs into the various caches. When constructing the graph, we lay out the elements in a contiguous order that matches how they will be computed (this is done by passing a user defined `malloc` routine during the RMV construction). Lastly, we use prefetching to load the next sub-graph in its respective cache. The application of these optimizations parallel the implementation of a high performance dense linear algebra routine.

For a single thread, our implementation outperforms the state of the art for synthetic scale-free graphs up to the size of 10^7 non-zero edges. We suspect that for larger sizes our blocking dimensions are not optimal. Furthermore, we suspect that for those larger sizes our data layout does not efficiently use the memory subsystem. This could be addressed by using a layout that maximizes memory level parallelism. We leave these two adjustments for future work.

V. Summary

In this paper, we provide an API for a sparse and hierarchical data format for storing scale-free graphs. Additionally, we provide an API that allows the domain expert to pass the structural information of the graph to our framework. This structural information is represented as a vertex partitioning stored as a tree, and it determines the shape of our Recursive Matrix Vector (RMV) objects.

For graph problems that generate their own data, or when the topology of the graph is known, this allows the data structure to be constructed in a way that fits both the graph and the memory hierarchy. Thus, we can map real-world scale-free graphs to modern computer architectures. In future work, we will investigate the optimal construction of our data-structure from real-world graphs. Particularly, we will investigate how to automatically map the graph to the memory hierarchy of the target system.

Acknowledgment

This work was sponsored by the DARPA PERFECT program under agreement HR0011-13-2-0007. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government. No official endorsement should be inferred.

References

[1] A.-L. Barabsi, R. Albert, and H. Jeong, "Mean-field theory for scale-free random networks," *Physica A: Statistical Mechanics and its Applications*, vol. 272, no. 12, pp. 173 – 187, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378437199002915

[2] ——, "Scale-free characteristics of random networks: the topology of the world-wide web," *Physica A: Statistical Mechanics and its Applications*, vol. 281, no. 14, pp. 69 – 77, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378437100000182

[3] C. Edwards, P. Geng, A. Patra, and R. van de Geijn, "Parallel matrix distributions: have we been doing it all wrong?" Department of Computer Sciences, The University of Texas at Austin, Tech. Rep. TR-95-40, 1995.

[4] P. Bientinesi, V. Eijkhout, K. Kim, J. Kurtz, and R. van de Geijn, "Sparse direct factorizations through unassembled hyper-matrices," *Computer Methods in Applied Mechanics and Engineering*, vol. 199, pp. 430–438, 2010.

[5] T. M. Low and R. van de Geijn, "An API for manipulating matrices stored by blocks," Department of Computer Sciences, The University of Texas at Austin, Tech. Rep. TR-2004-15, May 2004.

[6] B. B. Fraguela, J. Guo, G. Bikshandi, M. J. Garzarán, G. Almási, J. Moreira, and D. Padua, “The hierarchically tiled arrays programming approach,” in *Proceedings of the 7th Workshop on Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, ser. LCR ’04. New York, NY, USA: ACM, 2004, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1066650.1066657

[7] M. Martone, S. Tucci, M. Paprzycki, and M. Ganzha, “Utilizing recursive storage in sparse matrix-vector multiplication - preliminary considerations,” in *Proceedings of the ISCA 25th International Conference on Computers and Their Applications (CATA*. ISCA, 2010, pp. 300–305.

[8] R. Vuduc, J. W. Demmel, and K. A. Yelick, “Oski: A library of automatically tuned sparse matrix kernels,” *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 521, 2005. [Online]. Available: http://stacks.iop.org/1742-6596/16/i=1/a=071

[9] A. Bulu, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, “Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks,” in *IN SPAA*, 2009, pp. 233–244.

[10] D. Ediger, R. McColl, E. J. Riedy, and D. A. Bader, “STINGER: high performance data structure for streaming graphs,” in *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*, 2012, pp. 1–5. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2012.6408680

[11] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, “Kronecker graphs: An approach to modeling networks,” *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, March 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1756039