Community-based Matrix Reordering for Sparse Linear Algebra Optimization

Vignesh Balaji
NVIDIA
vbalaji@nvidia.com

Neal C. Crago
NVIDIA
ncrago@nvidia.com

Aamer Jaleel
NVIDIA
ajaleel@nvidia.com

Stephen W. Keckler
NVIDIA
skeckler@nvidia.com

***Abstract*—Sparse linear algebra kernels achieve sub-optimal performance due to their poor cache locality. Matrix reordering is an effective pre-processing optimization that improves cache locality and performance of these kernels. While many reordering techniques have been proposed, most prior work on matrix reordering suffer from two key limitations: (1) they evaluate their reordering proposal on a small set of arbitrarily-selected inputs and (2) they do not quantify the additional headroom for improvement after reordering is applied. To address these two limitations, we perform a detailed characterization of reordering techniques across a broad set of 50 input matrices where we quantify the ability of matrix reordering techniques to bring sparse linear algebra kernels close to hardware limits. Our analysis reveals that community-based matrix reordering is most effective at optimizing the execution of sparse linear algebra kernels, bringing the cuSPARSE SpMV kernel to within 54% of ideal run time on an NVIDIA A6000 GPU on average. However, community-based reordering is not uniformly effective across all 50 input matrices. We investigate the reasons when community-based reordering falls short and propose an enhanced version of community-based reordering that provides up to 1.57× additional performance improvements for the SpMV kernel.**

I. Introduction

As sparse linear algebra kernels are typically memory-bound [42], they are well-suited to GPU execution due to high GPU DRAM bandwidth. However, these kernels achieve sub-optimal performance on GPUs. We characterized the DRAM bandwidth utilization for cuSPARSE's SpMV (Sparse Matrix times Dense Vector) kernel on NVIDIA's A6000 GPU across a broad range of matrices and observed that the SpMV kernel achieves only 63% of the theoretical peak bandwidth. The primary reason for this sub-optimal performance is poor cache locality which stems from the characteristic irregular memory access pattern of sparse linear algebra kernels. The cuSPARSE SpMV kernel achieves an average L2 cache hit rate of only 34% (maximum of 62%) which leads to many expensive main memory accesses that end up hurting performance. To improve the performance of sparse linear algebra kernels on GPUs requires improving the cache locality of these kernels.

Matrix reordering is a software-based cache locality optimization that has been shown to be highly effective for irregular memory access workloads such as graph analytics and sparse linear algebra kernels [1], [2], [17], [41], [43]. Since matrix reordering is a pre-processing based solution, it can be easily applied to a broad range of workloads without requiring any application-level changes. The effectiveness and versatility of matrix reordering has inspired extensive research in developing solutions with varying levels of effectiveness and sophistication [2], [5], [10], [28], [41]. However, most prior work on matrix reordering techniques suffer from two key limitations. First, most techniques have been evaluated on a small set of arbitrarily-selected inputs. Since the locality and performance improvements offered by matrix reordering are a function of the structural properties of the input, the benefits of the reordering technique may be overstated if the set of chosen inputs are biased towards matrices exhibiting specific properties. Second, to the best of our knowledge, all prior reordering proposals have only compared their performance improvements against prior reordering strategies. Absolute performance measurements that quantify the remaining distance to peak performance on a given architecture after reordering is applied are absent from the vast literature on matrix reordering. Addressing these two shortcomings of prior matrix reordering research is an important step for our ultimate goal of developing a universally effective matrix reordering solution that can bring the performance of sparse linear algebra kernels close to the peak GPU performance.

To address the first shortcoming, we clearly define our process for curating the matrices used in our evaluation so as to avoid any selection bias. Our final dataset consists of 50 matrices from 3 different matrix/graph repositories [13], [27], [32]. To address the second shortcoming, we compare the performance achieved by sparse linear algebra kernels running on reordered matrices against *ideal* DRAM traffic and performance that can be achieved on our evaluation platform. We evaluated the locality and performance improvements offered by multiple, state-of-the-art matrix reordering techniques across our broad data set of 50 matrices and observed that a *community-based* matrix reordering technique called RABBIT [1] is the most effective reordering technique overall (bringing cuSPARSE SpMV's performance to 1.54× of the ideal value on average). The benefits of RABBIT, however, are not uniform across all our input matrices. For some inputs, RABBIT brings the DRAM traffic of cuSPARSE's [33] SpMV kernel within 10% of ideal traffic, corresponding to when the last level cache only incurs compulsory misses [22]. For other inputs, the DRAM traffic of the SpMV kernel is multiple factors above the ideal value. We define metrics to analyze the quality of communities detected by RABBIT and its effect on the locality and performance improvements provided by

RABBIT. We make two interesting observations from our analysis. First, as the skew in the degree distribution of the matrix increases (due to the presence of a small number of "hub" nodes with disproportionate connectivity compared to most of the nodes in the matrix), the quality of communities detected by RABBIT (and, consequently, the quality of matrix reordering) decreases. Second, there is significant community structure even in input matrices where RABBIT is unable to bring the performance of sparse linear algebra kernels close to hardware limits. We use these two observations to develop enhancements to RABBIT . Our new reordering proposal (called RABBIT++) is more effective than RABBIT in bringing sparse linear algebra kernels closer to ideal locality and performance on our evaluation platform. The performance of cuSPARSE SpMV on RABBIT++ ordered matrices is within 46% of the ideal run time on average.

In summary, this paper makes the following contributions:

- We clearly define our input matrix selection process to find a bias-free evaluation data set (Section III).
- We characterize the locality and performance of state-of-the-art matrix reordering techniques on cuSPARSE's SpMV kernel (Section IV).
- We explain the link between quality of community detection and the performance achieved with RABBIT (Section V) and provide the intuition for developing RABBIT++ (Section VI-A).
- We demonstrate the efficacy of RABBIT++ using cache simulations (Section VI-B) and show its effectiveness across other cuSPARSE kernels (Section VI-D).

II. Background on Matrix Reordering

This section highlights the main reason why sparse linear algebra kernels typically suffer from poor cache locality. We also develop the intuition for how matrix reordering techniques leverage structural properties of real-world matrices to improve locality and performance of sparse linear algebra kernels.

Source of poor cache locality: The primary cause for poor cache locality in sparse linear algebra kernels is the representation format for input data. Since sparse linear algebra kernels typically operate on inputs where more than 99% of the entries in the adjacency matrix are zero, using compressed formats to represent the matrices is a necessity. While compressed representation formats such as the Compressed Sparse Row (CSR) are great for memory efficiency, they cause irregular memory accesses in sparse linear algebra kernels when dereferencing the matrix's non-zeros, which leads to poor cache locality.

Algorithm 1 SpMV kernel with sparse matrix in CSR format

```
1: Y ← 0
2: parfor row in [0, |Rows|) do
3:     rowStart ← A.rowOffsets[row]
4:     rowEnd ← A.rowOffsets[row + 1]
5:     for i in [rowStart, rowEnd) do
6:         Y[row] += A.values[i] * X[A.coords[i]]
```

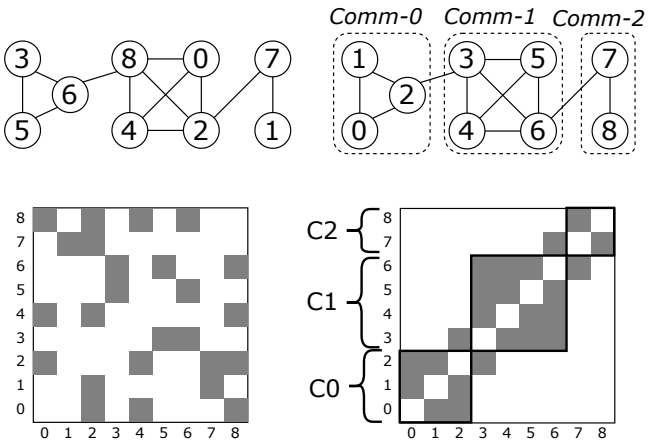

Fig. 1: **Community-based matrix reordering:** *Kernel executions on a community-ordered matrix will have a smaller cache footprint. (Matrix non-zeros are represented in gray)*

Algorithm 1 lists the pseudo code for the SpMV (Sparse Matrix multiplied by Dense Vector) kernel where the sparse matrix is represented in the CSR format. The algorithm shows that the output vector (Y) and the components of the CSR format (*rowOffsets*, *coords*, and *values*) are accessed in a streaming fashion. These regular accesses are a great fit for the high streaming DRAM bandwidth offered by GPUs. However, the access pattern for the input vector (X) depends on the contents of the $coords$ array of the CSR. Since the contents of a CSR can be arbitrarily ordered, input vector memory accesses are irregular. Furthermore, since sparse linear algebra kernels such as SpMV typically operate on large matrices with millions of rows and columns, the worst-case cache footprint requirements of the input vector ($|Rows| * elemSize$) can be orders of magnitude higher than the available last level cache capacity on GPUs. Therefore, SpMV and other sparse linear algebra kernels perform expensive fine-grained accesses from DRAM due to their poor cache locality.

Leveraging community-structure to improve locality: While sparse linear algebra kernels all suffer from irregular memory accesses, the non-zero patterns of real-world matrices are not entirely random. Instead, real-world matrices exhibit strong structural properties in their non-zero distribution such as power-law degree distribution [4], small-world behavior [30], and community structure [15]. Matrix reordering techniques leverage these structural properties to change the order of rows and columns in the matrix to make the memory accesses more regular. Figure 1 shows an example of community-based matrix reordering. As the graph[1] on the top left is randomly ordered, the corresponding adjacency matrix representation of the graph (bottom left panel) has non-zeros scattered across the matrix. In contrast, the reordered graph on the top right panel groups the nodes into three communities and orders them consecutively. Consequently, the

[1]We use the terms graph/matrix interchangeably throughout the text and also mix terminologies (nodes ↔ rows/cols and edges ↔ non-zeros).

corresponding adjacency matrix (bottom right panel) has significant structure with most of the non-zeros close to the main diagonal. While SpMV execution on the randomly-ordered matrix may require all nine elements of the input vector to be present in the cache throughout the execution, SpMV on the community-ordered matrix only needs to cache four elements of the input vector at any point in the execution. In summary, matrix reordering techniques can significantly improve the cache locality of sparse linear algebra kernels. Section V provides details on how communities can be detected.

III. Process for Selecting Input Matrices

The previous section showed how leveraging structural properties of the matrix (such as community structure) can improve cache locality of sparse linear algebra kernels. However, not all matrices necessarily exhibit the structural property exploited by a specific reordering technique. Therefore, to fulfill our goal of developing a universally effective reordering solution that improves locality across many different kinds of matrices, it is imperative that we evaluate our matrix reordering solutions across a broad set of diverse inputs.

Prior work on matrix reordering techniques have used an arbitrary input selection process, and most reordering proposals were evaluated on fewer than 10 inputs [1]–[3], [16], [17], [23], [28], [41], [43]. In this work, we clearly specify our input matrix selection process and take steps to avoid any input selection bias. Our main repository for matrices is the SuiteSparse matrix collection (formerly the University of Florida Sparse Matrix Collection) [13]. We use the following criteria for selecting inputs:

- We only consider square matrices with more than 1.5M nodes. For smaller matrices, the worst-case cache footprint for the input vector ($1.5M * 4B$ as described in Section II) would be smaller than the 6MB L2 cache capacity of the GPU used in our evaluation (Table I).
- We restrict the number of non-zeros to 2.5B because of memory capacity limitations on our GPU (Table I).
- SuiteSparse organizes matrices into *groups* such that matrices uploaded by the same publisher are assigned to the same group. To restrict the pool of matrices to a reasonable number, when there are multiple matrices in the same group we select the largest matrix in that group, since all matrices within a group tend to have similar properties. The exceptions to this rule are the SNAP and DIMACS10 groups for which we run all matrices since they are aggregated from different sources.

The above selection process yields 41 input matrices from SuiteSparse. We apply the same constraints to two other graph repositories, Konect [27] and Web Data Commons [32], where we obtain seven and two matrices respectively from each repository. In total, our final data set consists of 50 input matrices spanning a wide range of matrix sizes – 1.5M to 226M rows, 5M to 2B non-zeros, and average row-lengths (i.e. average degree) ranging from 2 to 139. The input matrices come from diverse range of sources including social networks, hyperlink graphs, circuit simulation matrices, non-linear optimization problems, computational fluid dynamics, road networks, protein k-mer graphs, knowledge databases, electro-magnetics problems, and DNA electrophoresis models. Evaluation of matrix reordering techniques across such a diverse range of inputs provides more robust results than employing narrower data sets.

IV. Analysis of Existing Reordering Techniques

We now evaluate the effectiveness of different matrix reordering solutions across the broad range of matrices introduced in Section III. To quantify the headroom for improvement after reordering, we compare the different reordering techniques based on their ability to bring sparse kernel performance close to hardware limits.

A. Reordering techniques evaluated

For our evaluation, we used four recent matrix reordering techniques that leverage the power-law degree distribution and community structure commonly found in real-world matrices. The four reordering techniques are degree-sorting (DEGSORT), degree-based grouping (DBG), rabbit (RABBIT), and gorder (GORDER). DEGSORT is a simple reordering technique that assigns vertex IDs in decreasing order of degree so as to pack highly connected vertices into the fewest number of cache lines. DBG [17] divides a matrix's vertices into different buckets based on specific degree ranges and then groups together IDs for members within a group while maintaining the same relative ordering as the original matrix. Unlike DEGSORT, which completely reassigns all the IDs of a matrix, DBG attempts to pack highly accessed vertices into fewer cache lines while also retaining some of the locality benefits of the original ordering (if any). DBG was shown to be more effective than other popular degree-based reordering techniques such as Frequency-based clustering [43] and HubCluster [2]. Both DEGSORT and DBG are designed to leverage the power-law degree distribution in matrices. We use in-degrees for both DEGSORT and DBG based on the observations of prior work [2], [17] for push-style workloads [6], [9]. RABBIT [1] is a community-based matrix reordering technique that first performs community detection on the matrices and then assigns community members consecutive IDs. RABBIT was shown to match or exceed the performance of many popular reordering techniques including SlashBurn [31], Reverse Cuthill-Mckee [23], and METIS [24]. GORDER [41] uses an approximation algorithm to find an ordering that maximizes a *locality score*. A high locality score is achieved when nodes in a sliding window have a large number of overlapping in-neighbors. We refer interested readers to the original publications for more details on the four reordering techniques. Finally, in addition to the above four reordering techniques, we also evaluate the original node ordering of matrices as found in the public datasets (ORIGINAL) and when nodes are randomly assigned IDs (RANDOM).

TABLE I: **NVIDIA A6000 GPU Specifications [35], [36].**

Peak Compute Throughput (SP)	38.7 TFLOPS	**L2 Cache Capacity**	6 MB
Peak DRAM Bandwidth	768 GB/s	**Main Memory Capacity**	48 GB

B. Ideal SpMV Locality and Performance

To the best of our knowledge, all prior work on matrix reordering only compare themselves to other reordering techniques. While comparisons against prior reordering techniques are certainly insightful, comparing the locality and performance improvements offered by matrix reordering against hardware limits is useful for guiding future development efforts of better reordering techniques. We primarily focus our analysis on the SpMV (Sparse Matrix multiplied by a Dense Vector) kernel, but we also present results for other kernels in Section VI-D. The minimum DRAM traffic (or compulsory traffic) for the SpMV kernel (Algorithm 1) is achieved when the last level cache only incurs compulsory cache misses [22] (i.e. data is brought into the cache only once and reused completely). Therefore, assuming 4 bytes for matrix values and the CSR coordinates and an $|N|$ x $|N|$ sparse matrix with $|NZ|$ non-zeros, the compulsory traffic for SpMV is $(2{*}|N|{*}4B){+}((|N|{+}1{+}|NZ|{+}|NZ|){*}4B)$ where the first part comes from moving the input (`X`) and output (`Y`) vectors and the second part comes from the CSR (`A.rowOffsets`, `A.coords`, and `A.values` respectively).

Based on the roofline analysis [42], SpMV is typically a memory-bound kernel. The theoretical upper-bound on arithmetic intensity (i.e. floating point operations per byte transferred from DRAM) for SpMV is 0.25 which is much less than the arithmetic intensity required to become computation-bound on most architectures (typically $\sim$10). Specifically, for the NVIDIA A6000 GPU, on which we perform all our experiments, applications need an arithmetic intensity of at least 50 to be limited by the computation units. Table I lists the specifications of our evaluation platform. Therefore, SpMV performance is limited by the DRAM bandwidth, and ideal SpMV performance corresponds to moving compulsory traffic at peak DRAM Bandwidth. To compute the minimum (ideal) execution time for SpMV on a given input matrix, we divide the compulsory DRAM traffic (listed above) by the peak *achievable* DRAM bandwidth on the NVIDIA A6000 GPU (672GB/s as determined using BabelStream [14]).

C. Observations on existing reordering techniques

Figure 2 shows the DRAM traffic (normalized to compulsory traffic) for cuSPARSE's SpMV kernel (compiled with cuda-11.7 and performance counter values collected using NVIDIA Nsight Compute [37]). The results reveal 5 interesting trends:

Observation 1: Matrix reordering can bring SpMV's DRAM traffic very close to ideal. As shown in Figure 1, matrix reordering techniques can improve the structure of non-zeros in a matrix and reduce the dynamic working set so as to fit in the available on-chip cache. Out of the 50 matrices used in our evaluation, matrix reordering helps bring the DRAM traffic of the SpMV kernel to within 10% of ideal traffic for 22 matrices which indicates that reordering maximizes the reuse in the 6MB L2 cache of the NVIDIA A6000 GPU.

Observation 2: The ability to reach ideal traffic is unrelated to the matrix size. For example, we are able to achieve close to ideal traffic for the `sk-2005` matrix (50.91M rows/cols and 1.93B non-zeros) whose total cache footprint is much larger than 6MB. Conversely, we are only able to bring SpMV's DRAM traffic to about 1.5$\times$ compulsory traffic for the smaller `com-LiveJournal1` matrix (4M rows/cols and 69.36M non-zeros). These results indicate that the ability to reach compulsory traffic is a function of the structural properties of the matrix instead of their shape or size.

Observation 3: ORIGINAL ordering can be a misleading baseline. The ORIGINAL ordering of matrices exhibit a wide range of DRAM traffic with some inputs such as `sk-2005` already achieving compulsory traffic whereas other inputs such as `pld-arc` achieve DRAM traffic close to a RANDOM ordering. The primary problem with ORIGINAL is that the ordering does not stem from any inherent property of the matrix generation process. Instead the ordering reflects an arbitrary choice made by the dataset publisher. For example, both `sk-2005` and `pld-arc` are matrices representing web-crawls, but they achieve very different DRAM traffic with their respective ORIGINAL ordering because the publisher of the `sk-2005` dataset applied a sophisticated reordering algorithm [10] whereas the publishers of `pld-arc` did not. Additionally, as public datasets often do not track any information on how the matrices were generated or ordered, ORIGINAL ordering is an ill-defined concept.

Observation 4: RABBIT is consistently the most effective reordering technique. The DEGSORT and DBG matrix reordering techniques very rarely bring SpMV's DRAM traffic close to ideal and are only effective for matrices exhibiting very strong power-law degree distribution. In contrast, community-based matrix reordering (RABBIT) is broadly effective across all the 50 matrices. In addition to achieving the lowest mean DRAM traffic, RABBIT is the best reordering technique for 26 out of 50 matrices and is on average only 11% away from the best reordering technique for the remaining 24 matrices. While GORDER is also a broadly effective matrix reordering technique, prior work has showed that it can impose extreme reordering overheads that overshadow its benefits [2]. We corroborate this finding in Section VI-C.

Observation 5: Matrix reordering techniques can bring SpMV's execution close to ideal run time. We do not include the run time (normalized to ideal) across reordering techniques in the interest of space. However, as noted in the caption of Figure 2, SpMV's performance trends across different matrix reordering techniques largely tracks the DRAM traffic numbers. Across all reordering techniques, RABBIT offers the best performance for SpMV, bringing the mean run time to within 54% of the ideal run time as calculated in Section IV-B).

In summary, the results in Figure 2 show the pervasiveness of community-structure in real-world matrices and the broad

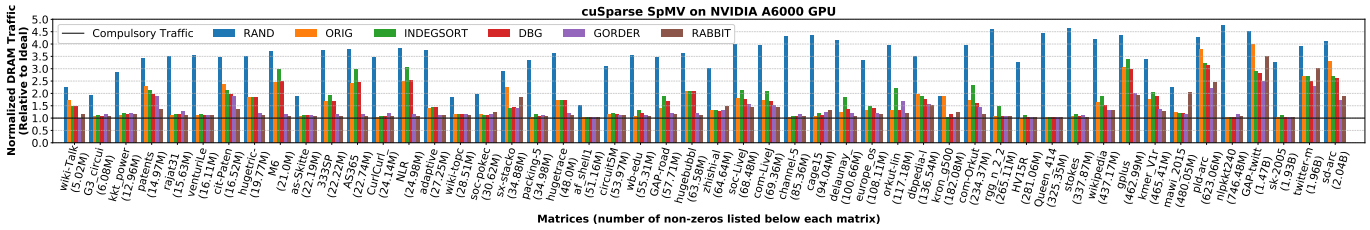

Fig. 2: **SpMV DRAM traffic (normalized to compulsory traffic) with different matrix reordering techniques:** *Mean DRAM traffic numbers across different reordering techniques are – RANDOM (3.36×), ORIGINAL (1.54×), DEGSORT (1.61×), DBG (1.48×), GORDER (1.29×), and RABBIT (1.27×). The mean run time values (not shown) normalized to ideal run time are – RANDOM (6.21×), ORIGINAL (1.96×), DEGSORT (2.17×), DBG (1.94×), GORDER (1.56×), and RABBIT (1.54×).*

effectiveness of the community-based matrix reordering technique RABBIT. Therefore, the main focus of our work is to understand how RABBIT improves locality and performance for sparse linear algebra kernels and identify opportunities to further improve RABBIT where it falls short.

V. Explaining Improvements with RABBIT

The results in Figure 2 show that RABBIT is effective at improving the locality and performance of cuSPARSE's SpMV kernel. However, RABBIT's benefits are not uniform. While for some matrices RABBIT is able to achieve close to ideal DRAM traffic, for other inputs the traffic for RABBIT is much further away from ideal. In this section, we first present a high-level overview of how RABBIT works and develop metrics to help explain when RABBIT is able to bring the SpMV kernel close to hardware limits versus when it cannot.

A. Metrics for analyzing quality of RABBIT's communities

RABBIT [1] was developed to leverage the *hierarchical* community structure present in many real-world networks, such as people organized into cliques based on shared interests and, within each group, sub-groups based on more niche interests. The authors of RABBIT sought to map the hierarchical communities onto the multi-level, hierarchical caches available on server-class CPUs, with the most tightly-knit innermost communities mapped to the small, fast cache closest to the processor and the looser, higher-level communities assigned to the larger, on-chip cache closer to main-memory. The core of RABBIT, community detection, is a well-studied problem [10], [15], [19], [29], and RABBIT is based on a popular class of community detection algorithms that maximize *modularity* [34]. Simply put, modularity measures the fraction of edges in a graph that only connect vertices of the same community minus the expected fraction if edges were randomly distributed. Higher values of modularity indicate that the communities discovered in the graph are of high quality and are able to effectively partition the graph. However, as a quality measure for community detection, modularity is difficult to visualize. Thus we introduce a simpler metric called *insularity* to measure community detection quality.

Figure 1 showed how community-based matrix reordering created more structure within the matrix's non-zeros and, therefore, reduced cache misses. Insularity is the fraction of edges that only connect members within the same community (i.e. intra-community edges) divided by the total number of edges in the graph. In the example shown in Figure 1, the insularity value of the graph after community-based matrix reordering is 0.83 ($\frac{20}{24}$). A high insularity value (insularity ranges from 0 to 1) is desirable because it indicates that, at any given point, most of the irregular accesses are to nodes of a single community, which maximizes cache locality. As we show later, high insularity values typically correspond to small community sizes, and smaller communities are more likely to fit within the on-chip cache and incur fewer DRAM accesses.

To show the impact of insularity on the performance achieved with RABBIT, we plot the run time of the cuSPARSE SpMV kernel (normalized to the ideal run time) with input matrices arranged in increasing order of insularity (Figure 3). Figure 3 shows that as the matrix insularity increases, RABBIT becomes more effective at bringing the performance of the SpMV kernel closer to ideal run time. Specifically, for matrices with insularity ≥ 0.95, RABBIT brings SpMV's performance to within 26% of the ideal run time. For the remaining matrices with insularity values < 0.95, the average run time achieved by RABBIT is 1.81× the ideal run time.

B. Impact of matrix structure on insularity

RABBIT brings the SpMV performance very close to hardware limits for high-insularity matrices (Figure 3). However, the `mawi` matrix is an exception; despite having a high insularity value of 0.988, the run time with RABBIT for `mawi` is 4.18× the ideal value. The reason for this anomalous behavior is that while for most inputs a high insularity value corresponds to a small average community size, structural properties of the `mawi` matrix force RABBIT's community detection algorithm to terminate early and not detect many communities. Specifically, the largest community detected by RABBIT for the `mawi` matrix corresponds to nearly 98% of the matrix size. The `mawi` matrix represents a corner case where the insularity metric is not a useful indicator of performance improvements with RABBIT since calling the entire

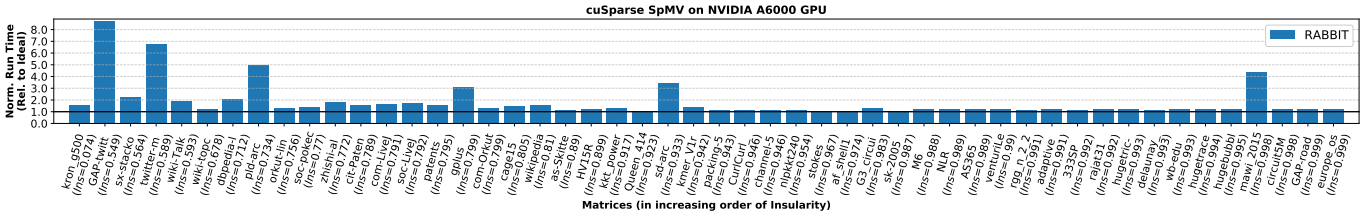

Fig. 3: **SpMV run time (normalized to Ideal) with RABBIT:** *RABBIT offers close to ideal performance for high-insularity matrices (on the right) where the community sizes are also typically small. For low-insularity matrices (on the left), the average communities are larger in size and RABBIT is unable to offer close to ideal SpMV performance for these matrices.*

matrix a single community maximizes insularity but does not provide any benefits from a cache locality or performance perspective. However, for other matrices besides `mawi`, the Pearson correlation between insularity and average community size normalized to the number of nodes is -0.472, meaning that higher insularity is correlated with smaller community sizes. High insularity is an indicator of achieving peak performance with RABBIT (Figure 3).

RABBIT is less effective at optimizing SpMV's performance for low-insularity matrices. Low insularity implies that the community assignment produced by RABBIT results in a large number of inter-community links. The most likely cause for low insularity is the power-law degree distribution where highly connected hub vertices make it difficult to find good community assignments with few links crossing community boundaries. To confirm this hypothesis, we calculated the correlation between insularity and the *skew* in the degree distribution. We define skew as the percentage of non-zeros connected to the top 10% most connected rows, with high skew values indicating a stronger power-law behavior where the hub vertices are even more disproportionately connected to the rest of the graph. For matrices with insularity ≥ 0.95, the average skew was 16.37% meaning that the top 10% most connected rows account for only 16.37% of all non-zeros. For the remaining matrices (insularity < 0.95), the average skew was 41.74%. The Pearson correlation between insularity and skew for matrices was -0.721 which confirms our hypothesis that hub nodes with disproportionately high connectivity impede the ability of RABBIT's community detection algorithm to form good-quality, isolated communities.

In the next section, we use these two findings (the importance of high insularity and skew in the degree distribution in explaining performance achieved with RABBIT) to develop an enhanced version of RABBIT.

VI. Modifying RABBIT to Improve Performance

Based on the analysis in the previous section, we propose an enhanced version of RABBIT (called RABBIT++) that brings the SpMV kernel closer to peak performance. We first build the intuition for RABBIT++ and then, using cache simulations, show that RABBIT++ achieves close to optimal locality on the NVIDIA A6000 GPU. Finally, we show that RABBIT++ is a versatile and low-overhead matrix reordering technique that can improve performance in many different settings.

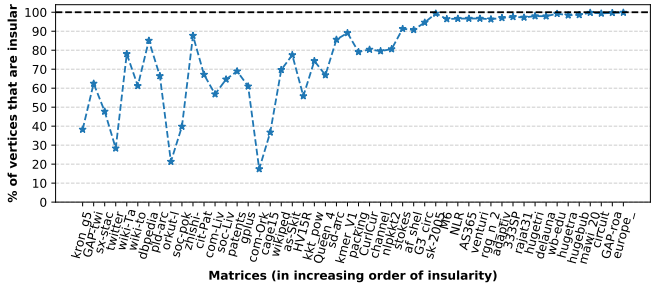

Fig. 4: **Percentage of insular nodes in matrices:** *Even for low insularity matrices (on the left), a large fraction of the nodes are only referenced from within their community.*

A. Design space of RABBIT modifications

While low insularity matrices do not achieve peak performance with RABBIT, this does not mean that community structure is entirely absent from these matrices. Even low insularity matrices may have strong community structure but it may be restricted to a subset of the matrix. To illustrate this point, we measure the number of *insular* nodes (i.e. nodes that are only connected to other nodes in their own community). Figure 4 shows the percentage of insular nodes for each matrix. As expected, high insularity matrices are almost entirely composed of insular nodes. However, even for low insularity matrices, a substantial portion of the matrix is insular, indicating the presence of some community structure.

To make the best use of the community structure in low insularity matrices, our first modification to RABBIT is to group all the insular nodes together. Specifically, for the first modification, we visit every node in the matrix and determine whether or not it is an insular node. Next, we group the vertex IDs for insular and non-insular nodes while maintaining RABBIT's relative ordering for both the types of nodes (Figure 5). Grouping insular nodes is akin to removing all nodes that contribute to inter-community traffic from each community. Compared to the RABBIT ordered matrix, the average community size drops by 27% (41% for matrices with Insularity < 0.95) for the insular portion of matrices

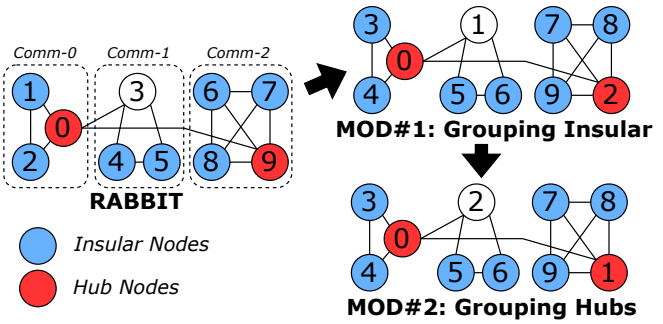

Fig. 5: **Modifications applied to RABBIT.**

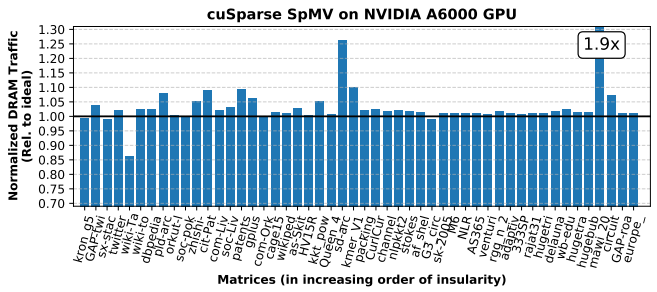

Fig. 6: **Normalized DRAM traffic for the insular sub-matrix:** *When insular nodes are grouped after applying the first modification to RABBIT (Figure 5), the insular portion of the matrix achieves ideal traffic. (y-axis values start from 0.7).*

TABLE II: **Design space of RABBIT modifications:** *SpMV run time (normalized to ideal) when applying different combinations of RABBIT modifications (Figure 5).*

	Without Insular Nodes Grouped			With Insular Nodes Grouped		
	ALL-MATS	*INS <0.95*	*INS >= 0.95*	*ALL-MATS*	*INS <0.95*	*INS >= 0.95*
RABBIT	1.54×	1.81×	1.25×	1.49×	1.70×	1.25×
RABBIT+HUBSORT	1.63×	1.89×	1.35×	1.57×	1.86×	1.26×
RABBIT+HUBGROUP	1.48×	1.65×	1.29×	**1.46×**	1.65×	1.25×

after the first modification is applied (Figure 5). Consequently, the DRAM traffic for SpMV on just the insular sub-matrix (evaluated by masking all non-zeros that do not connect to insular nodes) is very close to compulsory traffic[2] (Figure 6).

Grouping insular nodes alone is insufficient. We also need to find a good ordering for the non-insular nodes in the matrix because they can constitute a non-trivial portion of the matrix and account for a large percentage of the matrix's non-zeros (especially for low insularity matrices). In Section V-B, we saw that low insularity matrices have a very skewed degree distribution. Prior work [2], [17], [43] observed that graphs with skewed degree distributions benefit from ordering the highly-accessed hub nodes (typically defined as nodes with degree greater than the average degree of the graph) with contiguous IDs. Ordering the hubs contiguously ensures that the highly-accessed hub nodes are mapped across the smallest number of cache lines, which improves both spatial and temporal locality. For example, in Figure 5, as the matrix has 9 rows with 28 non-zeros, there are two hub nodes with degree greater than the average degree (3). With only the first modification applied to RABBIT, the hub nodes are assigned IDs 0 and 2, which could be mapped to different cache lines depending on the line size. However, applying the second modification (of grouping hub nodes) to RABBIT assigns the hub nodes IDs 0 and 1 which makes it very likely that the highly-reused nodes would be placed on the same cache line.

The two modifications described in Figure 5 reveal a design space for potential RABBIT++ solutions. We can choose whether or not to group insular nodes and/or use different options for contiguously ordering the hub nodes. Specifically, we evaluate two options for contiguously ordering hub nodes: HUBSORT (i.e. ordering hub in decreasing order of their in-degrees) and HUBGROUP (i.e. simply grouping the hub nodes maintaining the same relative ordering among hubs as in RABBIT). Table II compares the normalized run time (relative to ideal) for the SpMV kernel on RABBIT ordered matrices and 5 other matrix orderings derived from different combinations of the two modifications in Figure 5. The result shows that simply grouping insular nodes (i.e. applying only the first modification) in RABBIT improves performance, particularly for the low-insularity matrices (defined as Insularity < 0.95). Grouping insular nodes is especially important to ensure good performance for high-insularity matrices (Insularity ≥ 0.95) when the hub nodes are ordered contiguously with RABBIT+HUBSORT or RABBIT+HUBGROUP. Prior work has shown that ordering hub nodes in decreasing order of degrees is an effective reordering technique when the matrix is in ORIGINAL or RANDOM order [2], [3], [43]. However, we find that RABBIT+HUBSORT consistently *increases* the run time compared to RABBIT which suggests that there is *some community structure even among the hub nodes*. By maintaining the same relative ordering among hubs as discovered by RABBIT, RABBIT+HUBGROUP is able to preserve this community structure and improve performance over RABBIT. Across the different combinations of RABBIT modifications, grouping insular *and* hub nodes brings the kernel run time closest to hardware limits. Thus our RABBIT++ solution starts with RABBIT ordered matrices and modifies them by first grouping the insular nodes and then grouping the hub nodes.

While Table II shows aggregate performance results, Figure 7 shows the reduction in DRAM Traffic for the SpMV kernel achieved with RABBIT++ for individual matrices. RABBIT++ provides a maximum DRAM traffic reduction of 1.56× over RABBIT and a mean traffic reduction of 4.1%. Across low-insularity matrices (insularity < 0.95), RABBIT++ provides a mean traffic reduction of 7.7% compared to RABBIT. The DRAM traffic improvements with RABBIT++ translate into a maximum speedup of 1.57× over RABBIT (with a mean speedup of 5.3% across all inputs and 9.7% across inputs with insularity < 0.95). Locality and performance improvements of RABBIT++ over RABBIT are a function of both the percentage of insular nodes in the matrix as well as the reduction in the cache footprint of hub nodes. For

[2]The `wiki-Talk` matrix achieves lower than ideal traffic because 93% of its rows are empty, causing us to overestimate the ideal traffic (Section IV-B).

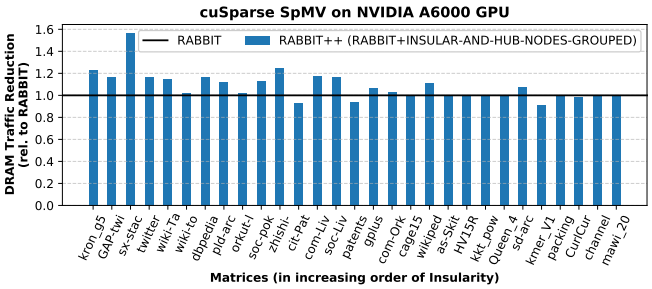

Fig. 7: **Reduction in SpMV's DRAM traffic with RABBIT++:** *In the interest of space, we only include results for matrices with Insularity < 0.95). For matrices with Insularity ≥ 0.95, RABBIT++'s DRAM traffic is within 1% of RABBIT.*

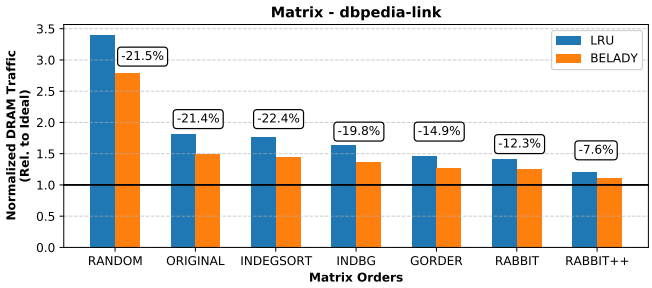

Fig. 8: **Headroom for additional DRAM Traffic reduction:** *Among all the matrix reordering techniques, DRAM traffic of SpMV on a RABBIT++ ordered matrix is closest to the DRAM traffic achieved with Belady's optimal replacement policy.*

TABLE III: **Effect of matrix reordering on the average % of dead lines inserted into the cache for the SpMV kernel.**

RANDOM	ORIGINAL	DEGSORT	DBG	GORDER	RABBIT	RABBIT++
63.31%	25.08%	26.88%	25.23%	17.73%	22.25%	**16.37%**

example, the highest DRAM traffic improvements in Figure 7 are achieved for the `sx-stackoverflow` matrix which has 47.74% insular nodes. Grouping insular nodes provides perfect locality for 47.74% of the `sx-stackoverflow` matrix (Figure 6). Among the remaining non-insular nodes, `sx-stackoverflow` has 446K hubs which are sparsely distributed across multiple cache lines with RABBIT. After grouping the hub nodes, RABBIT++ shrinks the cache footprint from 5.5MB to 1.7MB which significantly improves locality. Across all the reordering techniques evaluated (Figure 2), RABBIT++ brings the DRAM traffic and performance of cuSPARSE's SpMV kernel closest to hardware limits, at 1.22× the compulsory DRAM traffic and 1.46× the ideal run time on the NVIDIA A6000 GPU.

B. Cache Locality Analysis of RABBIT++

To better understand how RABBIT++ achieves the best locality for the SpMV kernel, we built a cache simulator to model the L2 cache of the A6000 GPU (Table I). We validated the DRAM traffic reported by our simulator for the SpMV kernel and found that the numbers reported are within 4% of the real-GPU numbers collected using NVIDIA Nsight Compute [37]. Based on cache simulation across all matrices and matrix reordering techniques, one of the primary reasons why RABBIT++ offers the lowest DRAM traffic is because it reduces waste of the available L2 cache capacity. Specifically, as Table III shows, RABBIT++ reduces the percentage of "dead" lines [18], [25] (i.e. cache lines which are brought into the cache but never reused). By reducing the percentage of dead lines, RABBIT++ makes the best use of the scarce L2 cache capacity available.

To quantify the additional headroom for improvements over RABBIT++, we compared the DRAM traffic achieved by our cache model for the A6000 GPU with an L2 cache with Belady's optimal cache replacement policy. Belady's replacement policy is an idealized, oracular policy that makes the best cache replacement decisions by looking at future reuse of contents in the cache. Figure 8 compares the DRAM traffic (normalized to compulsory traffic) achieved by an L2 cache with LRU replacement policy (which closely models A6000's L2 cache) with the traffic achieved by an idealized L2 cache equipped with Belady's optimal cache replacement policy [8]. The result show that, across matrix reordering techniques, the idealized L2 cache further reduces DRAM traffic compared to the L2 cache with LRU replacement. However, the gap between Belady's and LRU replacement policy is the smallest for RABBIT++ (7.6%). While estimating the optimal matrix reordering is NP-hard [41], the limited gap between DRAM traffic with realistic and idealized L2 caches suggests that RABBIT++ is providing close to the best cache locality achievable for the SpMV kernel on the A6000 GPU.

C. Pre-processing costs of RABBIT++

In this work, we focused on developing the best matrix reordering technique that can bring the performance of sparse linear algebra kernels on a broad range of matrices closest to hardware limits. The pre-processing overhead of matrix reordering is not a critical cost because it can be amortized across multiple iterations of the same kernel on the reordered input and/or multiple kernels running on the same reordered matrix (Table IV). Figure 9 shows the pre-processing time for GORDER, RABBIT, and RABBIT++. The result shows that while GORDER is an effective matrix reordering technique (Figure 2), the reordering costs of GORDER scale poorly as the matrix size increases. If we consider matrices to be in the RANDOM order at the beginning, then GORDER requires 7467 iterations (on average) of cuSPARSE's SpMV kernel to amortize the pre-processing cost of generating a new matrix order with GORDER. In contrast, RABBIT and RABBIT++ both incur significantly lower reordering costs. RABBIT amortizes its pre-processing costs in 741 SpMV iterations while RABBIT++ (which adds a small additional pre-processing overhead on top of RABBIT) requires 1047 iterations to amortize its pre-processing overhead.

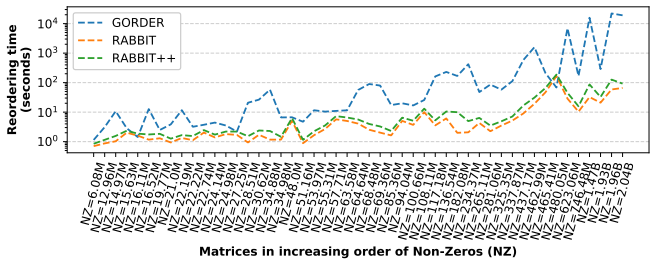

Fig. 9: **Matrix reordering time as the matrix size increases:** *Compared to GORDER, both RABBIT and RABBIT++ are more practical matrix reordering solutions.*

TABLE IV: **Run time (normalized to ideal) across different cuSPARSE kernels:** *RABBIT++ consistently improves upon RABBIT's ability to bring kernels closer to peak performance.*

	SpMV-COO			SpMM-CSR-4			SpMM-CSR-256		
	ALL	*I <0.95*	*I >= 0.95*	*ALL*	*I <0.95*	*I >= 0.95*	*ALL*	*I <0.95*	*I >= 0.95*
RANDOM	5.37×	4.94×	5.97×	29.33×	32.17×	26.07×	139.3×	196.6×	75.13×
ORIGINAL	1.84×	2.1×	1.55×	5.97×	8.92×	3.58×	26.81×	43.79×	10.99×
RABBIT	1.49×	1.73×	**1.23×**	4.31×	7.39×	**2.18×**	20.32×	50.3×	**3.91×**
RABBIT++	**1.4×**	**1.55×**	**1.23×**	**3.79×**	**5.85×**	**2.18×**	**18.7×**	**43.97×**	3.95×

D. Effectiveness on other cuSPARSE kernels

Our results show the effectiveness of matrix reordering techniques on improving the locality and performance of the SpMV kernel when the sparse matrix is stored in the CSR format. However, a major appeal of matrix reordering is that it can be easily applied to many sparse linear algebra kernels. We evaluated RABBIT++'s effectiveness across cuSPARSE's SpMV kernel when the matrix is stored in the Coordinates (COO) format. We also evaluated it on cuSPARSE SpMM (Sparse Matrix multiplied by a dense matrix) kernel for two different dense matrix dimensions, $|N|$ x 4 and $|N|$ x 256 where $|N|$ is the number of rows/columns of the sparse matrix. Table IV shows the execution time for these three kernels across a several different matrix orders. As with the SpMV results (Table II), the execution times are normalized to the ideal execution time where the compulsory traffic is updated according to the kernel. The results show that across matrices of different insularities, RABBIT++ is consistently better than RABBIT, regardless of the compressed representation (SpMV-COO and SpMV-CSR from Table II) or the dimensions of the sparse problem (SpMM on $|N|$ x 4 dense matrices as well as $|N|$ x 256 matrices). Finally, in addition to RABBIT++ generalizing to other irregular kernels on GPUs, we also expect RABBIT++ to be equally effective in achieving peak performance on other platforms such as multi-core CPUs and heterogeneous systems.

VII. Related Work

In addition to the related work discussed in Section IV, we present three other categories of related work:

Community-based matrix reordering: RABBIT uses a modularity-maximization based community detection algorithm. Prior work has proposed many community-based matrix reordering techniques using other types of community detection algorithms such as nested dissection [29], layered label propagation [10], shingle [12], and slashburn [31]. RABBIT was shown to either match or outperform the these other community-based matrix reordering techniques in addition to incurring low pre-processing overheads [1]. While we focused on RABBIT as our main matrix reordering technique and proposed improvements to it (RABBIT++), we believe the insights of grouping insular and hub nodes should extend to community-based reordering in general as well as matrix reordering techniques based on graph partitioning [24], [39]

Analysis of matrix reordering: Barik et al. [5] performed an extensive analysis of multiple reordering techniques and proposed various "gap" measures to estimate the quality of reordering solutions. They also study the performance implications of different reordering techniques on CPU-based graph analytics. Esfahani et al. [16] performed a detailed analysis of the locality benefits offered by slashburn, RABBIT, and GORDER. The authors proposed metrics to estimate the spatial locality improvements offered by the three reordering techniques and provided useful recommendations for improving each technique. However, neither work quantified the remaining headroom for additional performance improvement after matrix reordering. Therefore, our work is complimentary to these prior analyses of graph reordering techniques.

Tiling/blocking optimizations: Poor cache locality of sparse linear algebra and graph analytics kernels has prompted the development of many software-based cache locality optimizations. Tiling optimizations [21], [38], [40], [43] typically divide the matrix into smaller sub-matrices (also stored using compressed formats) so as to reduce the range of irregular accesses (Algorithm 1). Blocking optimizations [7], [11], [20], [26] distribute the irregular updates across many bins to make the access stream more regular. While these optimizations offer significant locality and performance improvements, they require modifying the application. In contrast, matrix reordering techniques are completely pre-processing based and, therefore, a more versatile optimization. Finally, RABBIT++ can potentially improve the efficiency of tiling and blocking optimizations; we leave this exploration to future work.

VIII. Conclusion

We presented two methodological improvements in the evaluation of matrix reordering techniques in this work. First, we follow a well-defined, bias-free selection process to curate our pool of input matrices (Section III). Second, we compared the performance of sparse linear algebra kernels against the hardware limits of our evaluation platform. Using the above evaluation methodology across 50 large input matrices, we observed that community-based matrix reordering (RABBIT) is already quite an effective reordering technique. We analyzed the shortcomings of RABBIT and proposed an enhanced version of RABBIT that bring sparse linear algebra kernels closer to peak performance on our GPU. Our insights and contributions are also likely to extend to other compute platforms and we plan to explore this in future work.

REFERENCES

[1] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time Parallel Reordering for Fast Graph Analysis," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 22–31.

[2] V. Balaji and B. Lucia, "When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 203–214.

[3] ——, "Combining Data Duplication and Graph Reordering to Accelerate Parallel Graph Processing," in *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019, p. 133–144.

[4] A.-L. Barabási, "Scale-free Networks: A Decade and Beyond," *Science*, vol. 325, no. 5939, pp. 412–413, 2009.

[5] R. Barik, M. Minutoli, M. Halappanavar, N. R. Tallent, and A. Kalyanaraman, "Vertex Reordering for Real-world Graphs and Applications: An Empirical Evaluation," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 240–251.

[6] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing Breadth-first Search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[7] ——, "Reducing Pagerank Communication via Propagation Blocking," in *Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 820–831.

[8] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-storage Computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[9] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations," in *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2017.

[10] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered Label Propagation: A Multiresolution Coordinate-free Ordering for Compressing Social Networks," in *International Conference on World Wide Web*, 2011, pp. 587–596.

[11] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing Sparse Matrix-vector Multiplication for Large-scale Data Analytics," in *International Conference on Supercomputing (ICS)*, 2016, pp. 1–12.

[12] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On Compressing Social Networks," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009, pp. 219–228.

[13] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[14] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating Attainable Memory Bandwidth of Parallel Programming Models via BabelStream," *International Journal of Computational Science and Engineering*, vol. 17, no. 3, pp. 247–262, 2018.

[15] D. Easley and J. Kleinberg, *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, 2010.

[16] M. K. Esfahani, P. Kilpatrick, and H. Vandierendonck, "Locality Analysis of Graph Reordering Algorithms," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 101–112.

[17] P. Faldu, J. Diamond, and B. Grot, "A Closer Look at Lightweight Graph Reordering," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2019.

[18] ——, "Domain-specialized Cache Management for Graph Analytics," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 234–248.

[19] M. Girvan and M. E. Newman, "Community Structure in Social and Biological Networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[20] Z. Gu, J. Moreira, D. Edelsohn, and A. Azad, "Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication using Propagation Blocking," in *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020, pp. 293–303.

[21] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A High-performance and Energy-efficient Accelerator for Graph Analytics," in *International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[22] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, 1989.

[23] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzarán, and K. Pingali, "Parallelization of Reordering Algorithms for Bandwidth and Wavefront Reduction," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014, pp. 921–932.

[24] G. Karypis and V. Kumar, "METIS–Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0," University of Minnesota, Tech. Rep., 1995.

[25] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling Dead Block Prediction for Last-level Caches," in *International Symposium on Microarchitecture (MICRO)*, 2010, pp. 175–186.

[26] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing Indirect Memory References with Milk," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2016, pp. 299—312.

[27] J. Kunegis, "KONECT – The Koblenz Network Collection," in *International Conference on World Wide Web*, 2013, pp. 1343–1350.

[28] K. Lakhotia, S. Singapura, R. Kannan, and V. Prasanna, "Recall: Reordered Cache Aware Locality Based Graph Processing," in *International Conference on High Performance Computing (HiPC)*, 2017, pp. 273–282.

[29] D. Lasalle and G. Karypis, "Multi-threaded Graph Partitioning," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013, pp. 225–236.

[30] V. Latora and M. Marchiori, "Efficient Behavior of Small-world Networks," *Physical Review Letters*, vol. 87, no. 19, 2001.

[31] Y. Lim, U. Kang, and C. Faloutsos, "Slashburn: Graph Compression and Mining Beyond Caveman Communities," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 12, pp. 3077–3089, 2014.

[32] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Graph Structure in the Web—Revisited: A Trick of the Heavy Tail," in *International Conference on World Wide Web*, 2014, pp. 427–432.

[33] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse Library," in *GPU Technology Conference*, 2010.

[34] M. E. Newman and M. Girvan, "Finding and Evaluating Community Structure in Networks," *Physical Review E*, vol. 69, no. 2, 2004.

[35] "NVIDIA A6000 Datasheet," https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/proviz-print-nvidia-rtx-a6000-datasheet-us-nvidia-1454980-r9-web%20(1).pdf, accessed: 2022-12-10.

[36] "NVIDIA A6000 White Paper," https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, accessed: 2022-12-10.

[37] "NVIDIA nsight compute," https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html, accessed: 2022-12-10.

[38] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric Graph Processing Using Streaming Partitions," in *Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 472–488.

[39] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "GraphGrind: Addressing Load Imbalance of Graph Partitioning," in *International Conference on Supercomputing (ICS)*, 2017.

[40] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High Performance Graph Analytics Made Productive," *Proceedings of the VLDB Endowment (VLDB)*, vol. 8, no. 11, pp. 1214–1225, 2015.

[41] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup Graph Processing by Graph Ordering," in *International Conference on Management of Data (SIGMOD)*, 2016, pp. 1813–1828.

[42] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[43] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making Caches Work for Graph Analytics," in *International Conference on Big Data (Big Data)*, December 2017, pp. 293–302.