RZ 3953 (#ZUR1906-036) 12/06/2019
Computer Science 11 pages

Research Report

Enhancing multi-threaded sparse matrix multiplication for knowledge graph oriented algorithms and analytics

Leonidas Georgopoulos, Aleksandros Sobczyk, Dimitrios Christofidellis,
Michele Dolfi, Christoph Auer, Peter W J Staar, Costas Bekas

IBM Research – Zurich
Säumerstrasse 4
CH-8803 Rüschlikon
Switzerland

Enhancing multi-threaded sparse matrix multiplication for knowledge graph oriented algorithms and analytics

Leonidas Georgopoulos, Aleksandros Sobczyk, Dimitrios Christofidellis,
Michele Dolfi, Christoph Auer, Peter W J Staar, Costas Bekas
{leg,obc,dic,dol,cau,taa,bek}@zurich.ibm.com
IBM Research
Rüschlikon, Switzerland

ABSTRACT

Graph algorithms can be implemented as a sequence of basic linear algebraic operations (BLAS) on a sparse adjacency matrix. Analytics, e.g. centralities, are computed fast through stochastic approaches employing a few SPMM operations. Edge traversals is an important operation for knowledge discovery, which can be implemented as a sequence of (SPMV) operations. Both, SPMV and SPMM, are notorious for being memory bound, cache demanding operations, and difficult to parallelize; (SPMM) suffering the least in terms of operations per byte. When the graph structure is fully known at runtime we can address in an off-line manner, memory overhead, and cache miss ratio by respectively using a smaller type for indexes, and a cache friendly sorting to reduce cache misses on the dense input vector. Parallelism is achieved by blocking the matrix. This work is the core of our HPC Knowledge Graph (HPC-KG) employed in production settings; among first such implementations.

1 INTRODUCTION

Representation of domain specific information as a graph enables to capture relational domain knowledge in a multitude of fields. This fact, has driven the development of several graph based solutions for relational representation (Knowledge Graph)[1]. Originally, most implementations have been targeted for *web-oriented social-media* like graphs, where the relations are easily extracted simply by mapping acquaintances to node relations. Queries typically are required to operate on a small part of the nodes in the graph, mainly through neighbor retrievals. The dominant approach is a *node-centric* implementation of a graph, which can be very successful for first order retrievals, e.g. to obtain a list of neighbors. However, in knowledge graphs deep-search capabilities and analytics are crucial for knowledge extraction. Therefore one can divide operations in three classes, (a) shallow, (b) depth, and (c) wide, depending on the way the operation interacts with the graph to reach the required result. For the implementation of a knowledge graph engine competing constraints are introduced. These are difficult to achieve in a node-centric approach. However, an *edge-centric* approach permits to tackle these simultaneously by focusing in efficient implementation of pure BLAS primitives; foremost SPMV and SPMM. We present our account on the ongoing work of building a High performance computing Knowledge Graph engine (HPC-KG) that permits to address both *depth-type* and *wide-type* of operations while at the same time maintaining a competitive edge on *shallow-type* operations.

Knowledge graphs are created by processing large volumes of diverse information, such as documents, diagrams, and pictures. First, the data ingestion phase extracts unstructured information, subsequently a knowledge extraction phase extracts corpus structure as relational mappings that finally permit to build a graph that encodes the knowledge extracted between first order terms, e.g. 'Mary-married-Jonathan' or 'copper-isa-metal'. Subsequently, these are augmented by a number of techniques that are not of interest for the purpose of this paper to higher order constructs, e.g. composites 'polyampholytes-isa-polyelectrolyte'. Given these, Knowledge Graphs can grow in size and complexity in an unbounded manner, depending on the method to populate and curate the graph.

Making this information available for searching, extracting dependencies, and hidden knowledge, can be achieved in a number of ways. Graph traversals, are the foremost operation to obtain such information. The former is an exemplary *depth-type* operation. On the other hand, extracting centrality information, retrieving the spectrum, or performing clustering, enables an integral approach to uncover node and subgraph importance on graphs. These are *wide-type* operations on the graph. Finally, *shallow-type* operations include getting the neighbors or the degree of a node or a set of nodes.

Given *shallow-type* operations, a straightforward approach is to store each vertex along with references to its outgoing connections; i.e. "node-centric". However, this does not work well for knowledge graphs where queries are complex and need to go in depth and breadth. There the number of traversed edges can grow in non-linear fashion, which turns out be difficult to handle in *node-centric* approach. However, when storing the graph in its dual form [10], the adjacency matrix, many standard graph algorithms are straightforward to apply by application of BLAS routines on a sparse matrix, [8] [4], [9]. Considering *wide-type* operations, the entire graph is required to be visited multiple times, e.g. centralities, spectral methods, etc.. These can be computed efficiently by means of BLAS operations [15].

Pure performance gains in terms of runtime over *node-centric* approach are only one side of the solution. The required development effort and maintenance are also important. In the

[1] Neo4j, IBM Titan, Google KG, Apache Giraph and Amazon Neptune, etc.

Conference'17, July 2017, Washington, DC, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

adjacency-list approach the complexity of maintaining a product code base grows exponentially, because performance and correctness needs to be achieved in every algorithm implementation. These are known to be hard to implement and software engineering skills are an important onset of the implementation effort. In contrary, taking a BLAS based graph approach operations can be decomposed in a sequence of easily maintainable BLAS primitives. Therefore, runtime performance improvements can be targeted on very specific parts of the product code base.

In this context, sparse matrix-vector multiplication (SPMV) and sparse matrix-matrix multiplication (SPMM) are crucial. The former plays a key role in *depth-type* and *breadth-type* operations, such as deep queries, whereas the latter is important *wide-type* operations, such as analytics employing the stochastic framework [1], [15] and concurrent processing of "query-like" retrievals.

Modern multi-core SMT / SMP architectures with high speed memory bus, and TB of memory are widely available. Processing power of these units is in the order of TFlops, and it is typical for implementations of SPMV and SPMM kernels to overhaul memory I/O capability. An efficient kernel implementation needs to tackle, thread contention on the output vector, cache miss on the input vector, and reduce memory bandwidth requirement to be efficient. The key observation that enables this work, is that knowledge graphs are considerably constant over time. That is due to the fact that knowledge creation phase is lengthy. The current state of the art is a create-store-n-load, since the load phase can be made efficient. Therefore, off-line optimizations can be made before loading the graph into memory.

In this regard, two main directions are of interest, blocking and reordering. Specifically, in this work, we consider blocked sparse matrix representations, based on the Coordinates format (COO). We achieve a 30% memory footprint reduction for large weighted graphs and up to 50% for unweighted graphs, both in the order of billion edges . Subsequently, we tune the structure of the blocked matrices by changing the element layout to reduce the cost of memory fetch per flop, and increase cache locality. Achieving high performance by targeting the memory-bound nature of the kernels and optimizing memory access patterns enables both CPU and GPU architectures to operate above the order of $10Gflops$ and $100Gflops$ respectively.

Summarizing, In this paper we take a high performance computing approach to graph processing and design kernels for multi-threaded in memory processing of knowledge graphs. The SPMM and SPMV kernels serves as an intrinsic building block for operations on Graph induced sparse matrices. We present our motivation and justification for making significant design choices, such as choosing BCOO over BCSC/BSCR. We contribute our approach to alleviate main performance hurdle of the SPMM/SPMV kernels, the memory bandwidth. Parallelism is achieved by grinding the matrix in many small blocks such that it is easily distributed across threads. Moreover, we tackle in a greedy on-line manner thread contention on the output vector, and address cache misses by reordering the sparse matrix in cache-line sized blocks off-line. Performance results in standard graphs results are presented to show improvements achieved by our approach. Our CPU and GPU kernels achieve at least 2x the performance of industry standard solutions.

2 DISCUSSION

The basic notions and solutions to the introductory problems already presented Section 1 are detailed subsequently. The first four parts of this section introduce graph formalism, grouping of operations, and detail the applications of interest. The rest of this sections is dedicated to presenting our implementation solutions that lead to the performance detailed in the subsequent sections.

2.1 Graphs and adjacency matrices

Briefly, a graph $G(V, E)$ is a tuple of sets V the vertex set , and E the edge set, formally and an incidence function that we omit here for simplicity. The former contains a numbering of nodes v_i where i is in $\{1, 2, ..., n\}$, whereas the latter is a set of tuples $e_{ij} : (v_i, v_j)$ denoting a connection between vertices v_i and v_j. Therefore the adjacent matrix A can represent the graph by setting the value of A_{ij} to unity. The latter frequently is a scalar value assigning weight related to some underlying represented quantity. In knowledge graphs this results in a sparse matrix. The latter is well represented in COO format. Taking advantage of multi-threaded architectures requires partitioning the matrix; such is blocking the COO format.

2.2 Graph operations

The field of graph algorithms is fundamental to computer science, and representation of a graph has always been conceived either through an adjacency-list or and adjacency-matrix approach. Glossing over the implementation details the former is very similar to a list of lists. For each node a list of its connected neighbors is stored; a *node-centric* approach. In contrast, the adjacency-matrix approach encodes the edges, almost as they are represented in the edge set; an *edge-centric* way.

In context of knowledge graphs, we can differentiate graph operations with respect to the portion of the graph that needs be addressed at each operation. Foremost, and actually what most graph engines currently provide are *shallow-type*. Given a node or a small subset of the graph one is interested to retrieve first-order information. For example a list of neighbors (get-neighbors) or a list of node-degrees. The second type of operations *wide-type* operations concern the class of algorithms where the entirety of the graph is used at each step of the algorithm. These are from example centrality algorithms, spectral algorithms, and matrix functions. Finally, *depth-type* operations, can be grouped as those that starting from a specific node or a set of nodes, an operation is applied incrementally, such as graph-traversals.

All three cases can be effectively be decomposed to sequence of matrix product operations, along with some auxiliary operations that we gloss over here for sake of brevity. The first type *shallow-type* operations be conceived equivalent to

$$\mathbf{1}_{i_1, i_2, \ldots, i_{d_j}} = \mathbf{A}\mathbf{1}_j \tag{1}$$

where $\mathbf{1_j}$ is an indicator vector where all elements except the subscript indices are zero, and d_j is the degree of the j-th vertex. Once, having the appropriate indices, it is straightforward to obtain associated matrix meta-data by simple lookup. One can also extend to multi-index operations by using a fat vector on the right side of equation (1).

For second type of operations, *wide-type*, we consider computation of centralities as in [15]. There the prevalent operation is a multiplication of the Chebychev polynomial expansion associated with the matrix functional with a fat matrix of Hadamard vectors as in [1].

$$f(\mathbf{A})\mathbf{V} = \sum_m c_m P_m(A)\mathbf{V} \tag{2}$$

Where f is a matrix function in the Cauchy sense, and $\mathbf{V}$ is a Hadamard matrix, P_m the m-th Chebychev polynomial of the first kind and c_m the associated coefficients. For a detailed description the reader is referred to [15], [1], [7]. From (2) we can observe that at each operation requires the entire adjacency matrix and therefore the entire graph is processed. Finally, *depth-type* operations, such as graph traversals can be seen as a sequence of mappings of the following kind

$$x(k+1) = \mathbf{A}x(k) \tag{3}$$

where now $x(k)$ is the required from the graph traversal operation k-order neighborhood; evidently $x(k) = \mathbf{A}^k x(0)$. Summarizing, for *wide-type* operation the SPMM primitive is crucial, whereas for the *depth-type* and *shallow-type* operations SPMV is more important.

2.3 Graph algorithms

Between the many graph algorithms and their respectively high number of applications, we detail here two which are important for knowledge graphs. Edge traversals in knowledge graphs are a very frequent to search, given a set of vertices, for the most related nodes taking into account the edge matrix weights. One to this is to start from the set of previously selected nodes and accumulate edge weights as one traverses the graph. The final outcome will be a weighted vector that provides relative importance. This is a crucial and very frequent *depth-type* operation for extracting relational importance between nodes.

For subgraph membership information, breadth first search, alternatively level-order traversal, can be used. The difference from edge traversal is that one is interested to obtain the k-th level set starting from a given root node. One traverses the graph by visiting all the neighboring nodes and marking them as visited, continuing iteratively on the child nodes until all the nodes have been visited up to a specified depth. It can be used for a multitude of applications, such as finding the shortest path between two nodes, or to provide a k-th order neighborhood node membership. Most *node-centric* implementations exhibit poor performance due to the continuously growing queue of visited nodes. Executing BFS as a sequence of linear algebraic operations has constant "level-traversal" cost [9].

Starting from a given vertex, the algorithm runs iteratively by applying an SPMV operation at each loop to perform a "get-neighbors" operation. The starting frontier is represented as a vector, having all elements equal to infinity except the element corresponding to the starting vertex index, which is equal to one. The traversal step is executed for all the vertices in the frontier simultaneously. This (SPMV) call is the heaviest computational part per iteration. A filtering step follows to omit vertices already visited in previous steps, see Algorithm 1.

Algorithm 1 GrB BFS(A, s)

Require: graph A, starting vertex s
Ensure: level vector p

Initialize $\delta \leftarrow 0$, $p \leftarrow \mathbf{0}$, $\psi \leftarrow 1$, $f(i) \leftarrow \begin{cases} 1, \text{if } i = s \\ 0, \text{else} \end{cases}$
while $\psi > 0$ **do**
  Update $p \leftarrow \delta \times f + p$
  Update $f \leftarrow A^\top f$
  Update $f \leftarrow f \odot \neg p$
  Set $\psi \leftarrow \mathbf{1}^\top f$
  Update $\delta \leftarrow \delta + 1$
return p

2.4 Analytics

Having information represented as a graph, one is generally interested to uncover importance of vertices (these are mapped to actual data) having taken into account the entire graph structure. This is achieved by computation of centrality measures. There are many different measures, each baring their importance. We steer away from the discussion of centrality optimality and detail those that are most commonly used, *eigenvector-centrality*, *subgraph-centrality*, and *katz-centrality*,[3], [5], [14]. Each has its merits, and in the HPC-KG we have made all them available.

The *eigenvector-centrality* is used to represent the stationary distribution of a random walk in the graph. This model has been very successful in uncovering importance of nodes when the underlying plant is actually a random walk. In large graphs represented as an adjacency list it is difficult to compute. However, in the adjacency-matrix format its computation can be achieved by employing power iteration to get the dominant eigenvector; thus employing a number of SPMV operations. Contrasting, *subgraph-centrality* is difficult to compute in large graphs. This is achieved by means of stochastic matrix function estimators based on equation (2), with $f(\mathbf{A}) = e^{\mathbf{A}}$. Finally, *katz-centrality* $(\mathbf{I} - c\mathbf{A})^{-1}$ is an alternative to degree centrality and is computed by means of Jacobi iteration [14].

Another important aspect of analytics is obtain quantitative information about a graph, a subgraph, or composition of graphs. Standard approach for characterization of the irregular domains represented by graphs, given the graph-matrix duality [10], is to compute spectral properties of the weighted adjacency matrix. The *spectrum-range* can be effectively computed with straightforward sequence of matrix vector multiplications, as well. Spectrum density estimation techniques have also been devised [15]. These and the aforementioned operations, all three types, are effectively based on efficient (SPMV) and (SPMM) implementations. We dedicate the rest of the section to detail our approach in improving them given the fact that for knowledge graphs graph processing can be performed in an off-line manner.

2.5 Blocked Sparse matrix formats

The three prevalent sparse matrix formats are Coordinate list (COO), Compressed sparse row (CSR), and Compressed sparse column (CSC). The COO matrix format stores three arrays, I, J, V such that I and J store the row and column indices respectively of the non zero elements and V stores the values. In the case of graph adjacency matrices the edge weights between nodes I and J are stored in V . The memory requirements for the COO format are linear with respect to the number of edges in the graph. On the other hand CSR format uses three arrays to store the nonzero values, the start and end of rows, and the column indices, requiring as well linear storage but at a smaller constant factor than COO format. There are two drawbacks in using CSR in this specific context.

First, when considering blocking as the number of blocks increases the memory requirement is smaller for blocked COO (BCOO). The (BCSR) formats space requirement is bounded from below order of

$$g(\mathsf{B_n}^2\mathsf{B_{nnz}}) \approx g(\mathsf{M}^2) \tag{4}$$

where M is the vertex set size, $\mathsf{B_{nnz}}$ is the number of (non-zero) blocks and $\mathsf{B_n}$ are non-zero elements per block side. In knowledge graphs it is common that zero blocks are scarce, thus for simplicity we assume all blocks have non-zero elements. Furthermore, strong diagonal structures exist, which makes $\mathsf{B_n} \approx \mathsf{M/K}$. Thus a quadratic growth rate dependency on the vertex set size for space requirements. For example nearly (22GB) are required for storing the uk-2005 graph [12] in (BCSR/BCSC), compared to (7.5GB) (BCOO) for a grid of 512×512. In contrast BCOO requires storage in the order of

$$O(\mathsf{E} + \mathsf{K}^2) = O(\mathsf{CM} + \mathsf{K}^2) \tag{5}$$

Where C is a constant factor much smaller than M, E are the non-elements of the matrix; equal to the edge set cardinality. Second, (BCSR) is difficult to work during the edge ingestion phase, because the column index holding array will need to constantly be updated with insertions at random positions. Whereas in BCOO new edges can easily be appended directly at the block. Finally beyond these reasons for choosing BCOO format as we are going to see below, this choice further facilitates memory bandwidth reduction.

Detailing the blocking, the matrix is split in a two dimensional grid; illustrated in Figure 1. Blocking a matrix in COO format is achieved by storing the blocks as individual COO matrices. The block coordinates are stored in a pair two dimensional arrays BI and BL which hold respectively the starting coordinates of each block and their lengths.

- `BI`: (Block index start) Element i, j holds a pair of coordinates denoting the row-offset and column-offset respectively of the block i, j in the original matrix.
- `BL`: (Block length) The pair in position i, j denotes the number of row and columns of the respective block i, j.

The second array is only needed for the general case of non square matrices and uneven blocking. Since adjacency matrices are square we do not need to consider lengths explicitly.

A key aspect for storage requirements of the adjacency matrix is the integral type used for the row and column indexing arrays. In the pure COO format two thirds are spent on the

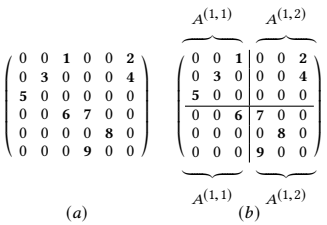

Figure 1: Different ways of blocking a sparse matrix. (a) **is the original matrix.** (b) **is the matrix split in a** 2×2 **grid of equal size blocks. Each block is stored as a COO.**

indices.Take for example a COO matrix of size $n \times n$ and `nnz` number of nonzero elements, an `index_type` of size $s_i$bytes for the elements of I, J and `scalar_type` of size s_s bytes for elements of V. The memory footprint of this matrix is

$$\mathsf{nnz} \times (2s_i + s_s) \tag{6}$$

For the blocked version of the same matrix we observe that the size is given by

$$\mathsf{nnz} \times (2s_{bi} + s_s) + B_n^2(s_i + s_{bi}) \tag{7}$$

Where we have assumed a grid of $B_n \times B_n$ blocks and introduced a `block_index_type` of size s_{bi} for the elements of the blocked I, and J. The blocks hold values in the range of the $2^{s_{bi}}$. The elements of `BI` are `index_type` and the elements of `BL` are `block_index_type`.

Memory footprint can be reduced for the blocked case by choosing a `block_index_type` of smaller size than the `index_type`. This can be achieved by restricting the block dimensions to be smaller than the maximum number that can be represented by the `block_index_type`. We applied this blocking strategy on the twitter_rv dataset [11], with dimension $m = 41.7$ millions and `nnz`=1.47 billions. The edges are not weighted and only the indices need be stored. Thus the values arrays can be ignored. For I and J, $s_i = 4$ bytes per index are required, i.e. a 32-bit unsigned integer. Then the total memory footprint of the COO representation would be roughly 11.76GB. Taking the blocked COO representation, and enforcing block dimensions no larger than 65535, the range of a short, enables us to use as a `block_index_type` a 16-bit unsigned short integer. This results to a memory footprint of 5.88GB, per equation (7), achieving a 49.98% compression. In Figure 2 we demonstrate the blocked vs non-blocked versions of six matrices, all having more than a billion non-zero elements. Matrices from the LAW [2] graph collection have been included for comparison. In this plot we assume that edge weights are stored as float data types to highlight the size compression even for demanding weighted graphs. This enables to utilize only a fraction of the memory required by industry standard graph solutions such as Neo4j which requires at least 32GB [17].

2.6 Kernels

Both SPMV and SPMM kernels are known to be memory bound, since only a few floating point operations are executed per memory transaction. Consider the description assuming a

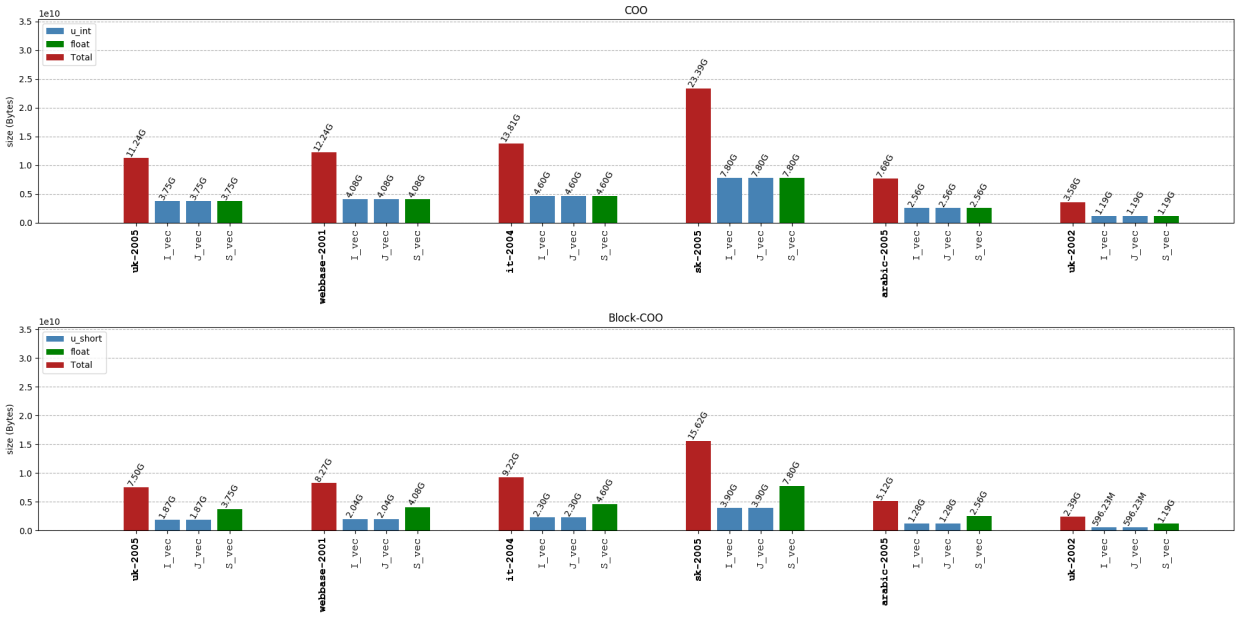

Figure 2: Matrix memory footprints with (bottom) and without (top) blocking.

COO storage format for the sparse matrix A. SPMV operates by holding a single counter which iterates over the elements of the arrays I, J, V and fetches from x indirectly the corresponding J indexed element; made explicit in Algorithm 2. Evidently, elements are accessed sequentially for the arrays and exhibit good cache locality as no element will be fetched more than once. The same, however, does not hold for x and y. The access pattern on these two vectors heavily depends not only on the sparsity structure of the input, but also on the ordering of the arrays. Ordering heavily affects access patterns, and can lead to many redundant transactions between cache and global memory; a low cache-hit ratio. This impacts the GPU architectures as well, where un-coalesced memory accesses impact performance.

Algorithm 2 SPMV-COO

Require: A in COO format (arrays I, J, V)
Ensure: $y \leftarrow \beta y + \alpha Ax$
  for $i = 1 : M$ **do**
    $\mathrm{y}[i] \leftarrow \beta \mathrm{y}[i]$
  for $k = 1 : \mathrm{nnz}$ **do**
    $\mathrm{y}[\mathrm{I}[k]] \leftarrow \mathrm{y}[\mathrm{I}[k]] + \alpha \mathbf{V}[k]\mathbf{x}[\mathrm{J}[k]]$

Respectively, the SPMM kernel, shown in Algorithm 3, operates in a similar manner but accessing an entire row of the right hand-side $\mathbf{X}$ for each increment of the counter. However, for each iteration more vectorized computations can be performed, which increase the operation per fetch ratio. Nevertheless, the same effect as in the SPMV kernel is primarily at hand.

We are interested in taking advantage of modern SMP/SMT enabled architectures. In order to distribute work efficiently among threads a blocking strategy is straightforward. As shown in Algorithm 4 employing a block-row-wise strategy

Algorithm 3 SPMM-COO

Require: A in COO format (arrays I, J, V)
Ensure: $Y \leftarrow \beta Y + \alpha AX$
  for $i = 1 : M$ **do**
    for $j = 1 : K$ **do**
      $\mathrm{Y}[i, j] \leftarrow \beta \mathrm{Y}[i, j]$
  for $k = 1 : \mathrm{nnz}$ **do**
    for $j = 1 : K$ **do**
      $\mathrm{Y}[\mathrm{I}[k], j] \leftarrow \mathrm{Y}[\mathrm{I}[k], j] + \alpha \mathbf{V}[k]\mathbf{X}[\mathrm{J}[k], j]$

employed avoids synchronization between threads, i.e. every thread works on a single row of blocks each time (Figure 3). Once processing of a block row is complete, it can continue to process another. A single atomic counter suffices to avoid race condition between threads operating at the same row. Therefore elements in y are accessed in sequence within the context of each thread, since the same block of y is used for the entire block-row computation of A. The same does not hold for x where for every block of the block-row, a new block of x has to be used. As a consequence, column-wise oriented orderings tend to improve performance since it improves cache locality on x, while locality on y is already guaranteed.

Algorithm 4 Block SPMV

Require: Block sparse matrix A, m_b number of block-rows, n_b number of block-columns. x, y are block dense vectors.
Ensure: $y \leftarrow \beta y + \alpha Ax$
  Spawn $p \leq m_b$ processes. Each process k:
  repeat
    pick an unvisited block-row index $i \in [0, m_b - 1]$
    Scale $y^{(i)} \Leftarrow \beta y^{(i)}$
    for $j = 1 \ldots n_b$ **do**
      Update $y^{(i)} \Leftarrow \mathrm{SPMV}(\alpha, A^{(i,j)}, x^{(j)}, 1, y^{(i)})$
  until all block-rows of y are updated

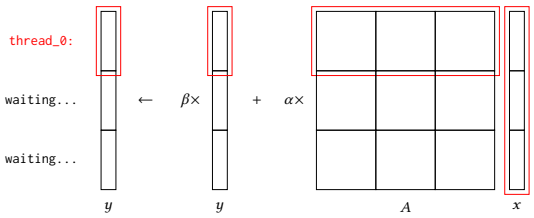

Figure 3: Task-based parallel SPMV (in this case having only a single thread-worker). Matrix A is blocked 3-by-3. `thread_0` (red) is assigned to the first block-row, computing $y^{(1)} \leftarrow \beta y^{(1)} + \alpha \sum_{k=1}^{3} A^{(1,k)} x^{(k)}$, where $y^{(i)}$ is the i-th block of y, $x^{(i)}$ is the i-th block of x and $A^{(i,j)}$ is the block of A in row i, column j. The other two block rows are waiting for a worker to finish its task and be assigned to them.

2.7 Element layout

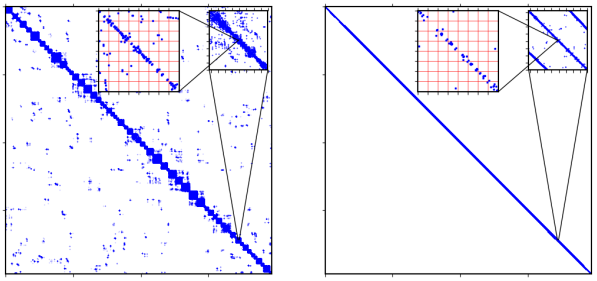

Figure 4: `roadNet-CA` matrix before (left) and after (right) Reverse-Cuthill-McKee reordering. Red grid represents cache-line sized blocks (elements in the same block access elements from the same cache-line of x, y).

Different ordering of the arrays I, J, V produce different access patterns to x and y, which in turn heavily affects cache locality on large graphs. When the size of the node set crosses the million boundary, most modern processors will not be able to fit it into the size of the processor cache. Ordering arbitrarily can yield poor cache locality even if the matrix exhibits a cache-friendly sparsity structure (i.e. diagonal matrices). This observation becomes crucial in the case of multi-threaded execution of the kernel onto blocked sparse matrices. There, how the scheduler selects the thread to execute is unpredictable which further deteriorates cache locality and introduces thread contention for the cache. We observe that the more scattered the non-zeros the larger that effect is. Therefore we devise a strategy for confining scarcity.

The sparsity structure of a matrix can be restructured by applying row/column permutations. A standard approach is the Reverse-Cuthill-McKee algorithm [6]. The latter, produces a matrix with a smaller band, which makes it appealing as the graph becomes larger and sparser. Originally employed for matrix fill-in reduction during decompositions like the LU, it is natural baseline comparison. Although being far from providing an optimal band reduction. Specifically, this kind of reordering forms bands of non-zeros and clustering around the main diagonal. In principle, this can increase the average number of elements per cache-block, by reducing the number of blocks containing non-zeros, effectively leading to a decreased number of kernel invocations by the Block-SPMV/SPMM algorithm. However, we have observed that obtaining a sparsity structure that has reduced fill-in is not sufficient to improve cache-miss ratio per operation. Ideally, elements of the matrix should be grouped into dense blocks.

Consider Figure 4 presenting the `roadNet-CA` dataset from the SNAP matrix collection. A first impression from this "spy-plot" is that most of the non-zeros are nicely concentrated close to the diagonal, expecting a good cache behavior. Zooming in the structure of the matrix, specifically around the main diagonal, structure is arbitrary. The red-colored grid depicts blocks of 64-byte size (i.e., each row can store up to 8 floats, which is the size of the cache-line of a POWER8 core). It is easy to see in this case that sorting by row indices, a different cache line from x has to be accessed in almost every iteration during execution of the (SPMV) kernel. A second observation is that sorting elements based on their "cache-block index" is preferential. The latter, implies placing elements which belong to the same block in consequent positions in the arrays I and J.

2.8 Cache friendly blocking

In one cache line of size L bytes, $c = L/\texttt{scalar\_type}$ elements of x are fetched. We split the matrix in hypothetical cache-blocks of dimension $c \times c$. Each non-zero element k belongs to a single cache-block, indexed by $\lfloor I[k]/c \rfloor$ and $\lfloor J[k]/c \rfloor$. The sorting algorithm scans through the arrays I and J and computes the respective cache-block for each element, assigning a unique index for each block. The arrays I, J are then sorted based on the block index. As a result elements belonging to the same cache-block are grouped together in the sorted arrays. In a second phase, elements within the same cache-block are ultimately sorted based on their column index, $J[i]$.

This formulation of the problem reduces to a simpler I/O model. Viewing the cache-blocks as individual elements, we now need to optimize the ordering of the blocks rather than individual elements. This is an operation that can be run off-line and has less cost than operating on the entire matrix. Recall that each cache block $A^{(c_i, c_j)}$ operates with a cache-block of x and y, namely $x^{(c_j)}$ and $y^{(c_i)}$. The iterations of SPMV now perform on a cache-block level, and a new iteration starts when a new cache-block is executed. Row-wise/column-wise sorting of the cache-blocks can be expected to boost the block-cache locality, just like in the element-wise case. We opt for a hybrid approach: Starting from an arbitrary block, choose as a next block one of the neighboring blocks, prioritizing column-wise neighbors rather than row-wise. If no neighbor is available, pick a new arbitrary block. Proceed until there are no more blocks to process.

2.9 Graphics Processing Units

Graphics processing units have introduced prevalent paradigm of parallel computation. Efficient utilization of the available compute capabilities of such systems beyond simple batching is not a straightforward task. Herein we detail current approach for enabling single-GPU and multi-GPU processing

for knowledge graph operations. Once more the main task at hand is to appropriately map the SPMV and SPMM kernels.

2.9.1 CUDA kernels. The key component to enable the SPMV-BCOO and SPMM-BCOO kernels for the GPU architecture is the ability to directly control the small, fast memory called `shared memory` on the GPU. We provide an algorithmic description in Algorithm 5 and a visual representation in Figure 5.

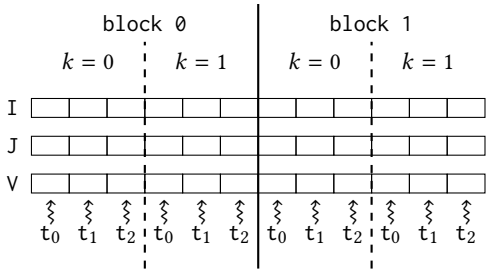

Figure 5: SPMV-BCOO-GPU. Kernel with two CUDA blocks and three threads per block. At iteration $k = 0$, the threads of both blocks read consecutive elements from the arrays I, J, V. In the next iteration $k = 1$, they stride "block-size"$= 3$ steps further from their original position, ensuring coalesced accesses on the COO arrays in every step.

In our setup, we fix n the number of threads per CUDA block, typically 512, and k the number of nonzero elements of A that will be processed per thread, typically $16 - 32$. Having these fixed, we invoke the SPMV CUDA kernel with $\lceil \mathtt{nnz}/(nk) \rceil$ CUDA blocks. Inside the CUDA kernel, k iterations are performed. During each iteration, each thread fetches the elements of I, J, V, indexed by its global thread id, `g_id = block_id` $\times$ `threads_per_block + local_thread_id` such that threads belonging to the same warps will access consequent elements in the arrays. As a second step each element reads the element of x required for the multiplication and atomically adds the result on the appropriate position in y. This last step is the unavoidable performance bottleneck of this approach, as the accesses in x and y will not be coalesced in their majority. Nevertheless this observation, our experimental results in latter sections indicate roughly 3× speedup compared to the CPU counterparts for the SPMV, and 2× speedup for the SPMM. The element layout as discussed in 2.7 is important in order to reduce the number of un-coalesced memory accesses.

Algorithm 5 SPMV-COO-GPU

```
start = t_id + b_id*B*k
for i=1...k do
    ind = start + i*B
    val = α*V[ind]*x[J[ind]]
    atomicAdd(&(y[I[ind]]), val)
```

2.9.2 Multiple devices and large graphs streaming. Knowledge graphs can grow in an unbounded manner both in size and complexity. Social graphs, as well, tend to reach the order of billions of edges and will not fit in the memory of a single GPU device. Therefore, we have extended our kernels to handle this case (see Algorithm 6).

Algorithm 6 Block-stream SPMV-GPU

Require: Block sparse matrix A, m_b number of block-rows, n_b number of block-columns, p number of devices. x, y are block dense vectors.
Ensure: $y \leftarrow \beta y + \alpha A x$
1: Spawn $p \leq m_b$ processes, each assigned to a cuda device.
2: **for** each process k on device k **do**
3: Allocate A_{gpu}, x_{gpu}, y_{gpu}
4: Allocate A_{tmp}, x_{tmp} for temp storage
5: Pick a unique block-row i
6: **Sync** copy $y_{gpu} \leftarrow y^{(i)}$ to the device
7: **Async** execute $y_{gpu} \leftarrow \beta y_{gpu}$
8: **Async** copy $\begin{cases} A_{gpu} \leftarrow A^{(i,0)} \\ x_{gpu} \leftarrow x^{(0)} \end{cases}$
9: **repeat**
10: **for** $j = 1 \ldots n_b$ **do**
11: **Synchronize**
12: **Async** execute:
13: $y_{gpu} \Leftarrow \text{SPMV}(\alpha, A_{gpu}, x_{gpu}, 1, y_{gpu})$
14: Copy $A_{tmp} \leftarrow A^{(i,j+1)}$
15: Copy $x_{tmp} \leftarrow x^{(j+1)}$
16: Swap pointers $\begin{cases} A_{tmp} \leftrightarrow A_{gpu} \\ x_{tmp} \leftrightarrow x_{gpu} \end{cases}$
17: Pick a new i
18: **until** all block-rows of y are updated

Having allocated enough space to store a single block in the device, we work in an asynchronous fashion, overlapping cuda kernel invocations with hosted to device memory copy. Taking advantage of the thread-safe context switching mechanisms of the CUDA API it is possible to initialize an ad hoc system with multiple devices. The algorithm is designed to efficiently handle this case.

Each CPU thread receives a block-row to execute in a similar fashion with the CPU based SPMV-BCOO Algorithm 4. At the initialization step, the algorithm allocates on the respective devices a COO matrix and two vectors. These will contain in each iteration's step the respective blocks of A, x and y. Additionally, one extra block is allocate for temporary storage, namely A_{tmp}, x_{tmp} and y_{tmp}. Assuming block row i is assigned to the thread at some iteration. As a first step, it copies from host to device the block $y^{(i)}$, and once copied it is scaled by β with a cuda kernel invocation. Concurrently with the scale kernel execution, another CUDA stream copies from host to device the block $x^{(0)}$ and then $A^{(i,0)}$. Then the iteration over the block-row of A begins. At each iteration, the first step is a synchronization barrier, ensuring all asynchronous calls have finished. The main part of the iteration consists of three asynchronous steps:

(1) Execution of SPMV on the current block.
(2) Copy the next block in A_{tmp}, x_{tmp}, y_{tmp} from host to device.
(3) Swap the pointers between the main and the temporary variables.

Once all iterations have finished, $y^{(i)}$ is added back to the CPU, and a new block row is fetched, until all block rows have been executed.

3 RESULTS

We validate our approach on CPU of the IBM POWER architecture, single-GPU and multi-GPU results are presented as well. An important focal point of this work is processing large graph in a single fat-node equipped with memory in the order of a. Such a configuration can fit a graph nearing the order of a hundred billion edges when using our blocking strategy. Given these, a single-fat-node configuration is a fitting test-bed for the application at hand. Selection between processing on CPU, single-GPU, and multi-GPU mainly depends on the size of the graph.

Summarizing our results, the SPMM-BCOO kernel reached **87**Gflops running only on a P8-CPU and **251**GFlops on the P100-GPU. The SPMV kernel performed at a peak rate of **12.1**Gflops on the P8-CPU, Figure 6; cache-friendly-sorting performs best. We observe running time is reduced by nearly half and better scaling is exhibited with respect to the number of real cores in the system. The kernel reached a rate of **34** Gflops on the P100-GPU. To our knowledge the baseline state-of-the-art CSR based SPMV provided by *cuSPARSE* does not cross above the 60Gflops mark [16], and this seems to be achieved for small well banded matrices. On average performance is around **30** Gflops. Having performed our own baseline comparison with *cuSPARSE* similar results have been exhibited; Figure 9. Our kernels are on average **10**% above the state-of-the-art GPU mark, especially on larger graphs where difference is up to an order of magnitude.

In subsequent sections, a baseline comparison of the single-threaded COO-SPMV implementation versus the graphBlas is provided [4]. Then we present results for BCOO-SPMV/SPMM kernel performance. Subsequently, we evaluate on single-GPU and demonstrate consistent performance across differing sparse matrices, along with speedup results on large graphs for our multi-GPU approach. Subsequently, we evaluate the efficiency of our graph algorithms against the industry standard graph implementation *Neo4j*. Specifically, *edge-traversals*, *level-order-traversal*, and analytics (*eigenvector-centrality*, *subgraph-centrality*, *katz-centrality*, *spectrum-range*. Finally, linear scaling for our BFS and edge traversal implementations is achieved. Notably a graph in the order of billion edges on a single node is used for the final test.

3.1 Experimental setup

We evaluate on the POWER8 nodes, each with 512 GB of main memory, 2 CPU sockets with 10 physical cores per socket and an 8-way Simultaneous-multi-threading (SMT) technology. Each core is associated with a segment of 8MB of L3 cache, while being to access other core's shared L3 cache. Each POWER8 module has two memory controllers, each with two memory channels, reaching a total peak memory bandwidth between L3 and main memory of 105 GBps able to operate 2B reads and 1B write per clock cycle. Finally, each socket is connected with 2 Nvidia P100 GPU devices, totaling 4 devices with 16GB of main memory each. The nodes are also powered with NVLink, NVIDIA's high-speed interconnect technology which accelerates both host to device and device to device data transactions. The compiler used for these tests was g++ 6.4

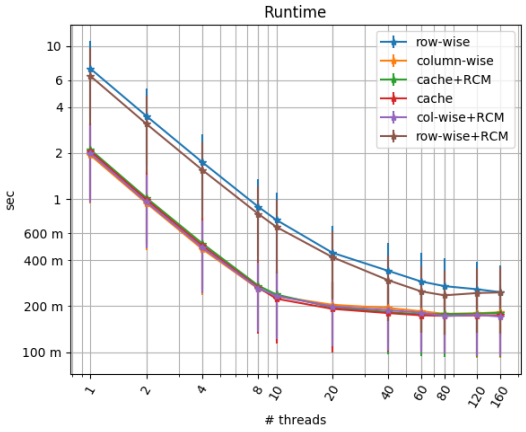

Figure 6: BCOO-SPMV on the LAW collection of matrices [2], comparing the effect of different sorting approaches. The block index type is set to `u_int16` and the block dimension is $64K$.

version with optimization level -O3. All our tests have been executed the RHEL-7.5 Linux distribution for POWER.

For our tests we have used (a) synthetic randomly generated sparse matrices, (b) graphs from SNAP [12], (c) LAW [2] collections, and (d) graph500 generated graphs [13]. All of the graphs chosen are in the order tenths of million edges. For each of the experiments, unless stated otherwise, we execute ten runs and we present the mean and standard deviation. For the SPMV, we evaluate the floating point operations per second (flops) as $2(\mathtt{nnz} + m)/t$ where t is the running time of the operation, `nnz` the number of non-zeros in the matrix (i.e. equal to the number of edges in the graph), and m is the matrix dimension.

3.2 Kernel evaluation

In order to evaluate our kernels we take a bottom up approach. First, we demonstrate better performance for the singe-threaded implementation over the graphBLAS counterpart [4]. Subsequently, we show strong scaling for for the multi-threaded CPU based kernels.

3.2.1 Single threaded baseline. We compare with GraphBLAS implementation of the (SPMV) operation. The latter employs sparse vector instead of a dense right vector. To evaluate we have generated random sparse matrices and varied their density structure. Similarly for the right hand side vector. Figure 7 demonstrates a clear advantage of our implementation.

3.2.2 Multi-threaded performance. In our experiments a a maximum block size with $64K$ elements was chosen, in order to reduce memory requirements. Gain from each discussed improvement is depicted in Figure 8 and 8.

Examining results we observe that reducing fill-in by applying RCM reduces running time. The RCM pre-processing step is intensive both in memory and time, but for the application at hand where the graphs are not dynamic, it is a viable choice.

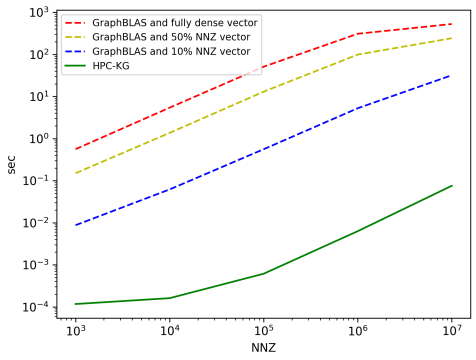

Figure 7: Single-threaded SPMV versus GraphBLAS. Our kernel achieves at least two orders of magnitude improvement.

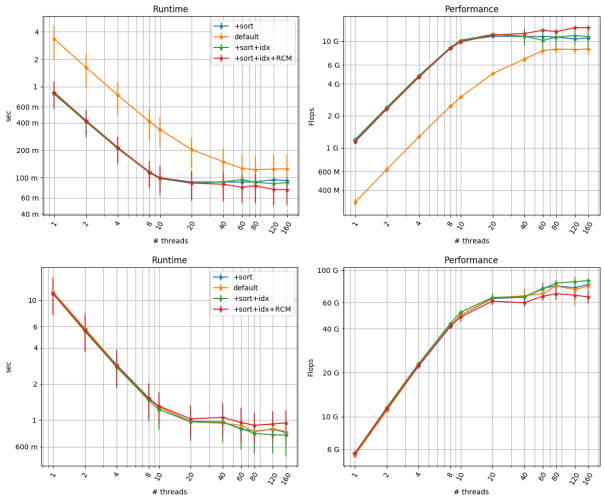

Figure 8: (Top) SPMV kernel on `arabic-2005` ($m = 22.7M$, nnz $= 640M$) **and** uk-2002 ($m = 18.5M$, nnz $= 298M$) **datasets from the LAW collection. (Bottom) SPMM kernel. Block dimension is** $64K$ **and the number of columns of** X **and** Y **for SPMM is** 64.

matrix	1 GPU	2 GPUs	3 GPUs	4 GPUs
com-Orkut	0.132	0.08	0.063	0.060
	3.58	5.94	7.52	7.82
soc-LiveJournal1	0.055	0.036	0.0311	0.0314
	2.68	4.10	4.75	4.71

Table 1: SPMV scaling with four P100 Nvidia GPU devices evaluated on `com-Orkut` **and** `soc-LiveJournal1`**; SNAP collection. Results in seconds (top) and estimated Gflops (bottom) of each line.**

Cache-friendly sorting achieves similar performance, while discounting off-line optimization cost. Inspecting 6, the minimum running time of the kernel is reduced by a factor of two, reaching below $100ms$ on graphs in the order of nearly half a billion edges and a performance of 12.1 Gflops on average. Peak performance reached 13.8 Gflops on the selected set of

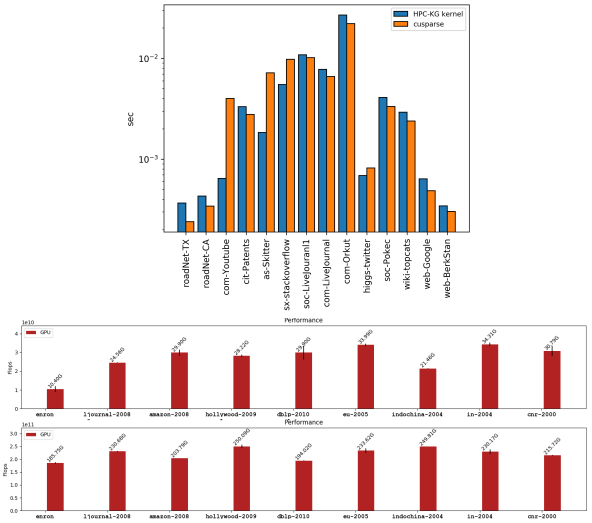

Figure 9: (Top) Comparison with cuSparse exhibiting 10% **performance improvement over CSR-SPMV. Initialization time for cuSparse has been discounted. (Bottom) Performance of the single-GPU SPMM and SPMV kernels on the LAW collection.**

graphs; Figure 8. Nearly linear speedup is observed up to the number of physical cores available in the system.

Theoretically for each SPMV operation on a COO matrix we are required to perform a read of one element from the I and J arrays and an additional read from the scalar array V. Considering 32bit long integers and single precision floats, this results in having to read 12 bytes from memory. Additionally from the x vector we need to read another float and write one float on the y result vector. This totals short of 20 bytes that can be performed with 8 read operations and 4 write operations given the specifics of the system; i.e. 8 memory operations. Using our compressed block storage format results in a total of 16 bytes that can be performed in a total of 6 memory operations. To evaluate the effective bandwidth utilization we naively assume that for each edge two floating point operations are executed. The scale operation is ignored, along with cache locality. We arrive at the following simplified model to compute the effective bandwidth based on performance for the SPMV kernel.

$$b_e = 3max(\lceil b_r/2 \rceil, b_w)s_c/2 \tag{8}$$

Where b_e is the effective bandwidth, b_r are the bytes read per iteration and b_w are the bytes written. Finally, s_c is measured floating point operations per second. Given the observed 12.1 Gflops we arrive at an effective bandwidth of 108.9 GBps. Taking into account a nominal peak bandwidth of 115GBps, our kernels achieve full utilization of the peak bandwidth per memory controller, indicating efficient utilization of the available resources.

The performance of the SPMM reaches 87Gflops; Figure 8. Evaluating results from for both SPMM and SPMV we come to the conclusion that the cache-friendly sorting approach provides good performance for both SPMM and SPMV operations.

3.2.3 single-GPU and Multi-GPU. Evaluation of the kernels in a single GPU arrives at a maximum performance of 252 Gflops; Figure 9. Results for our multi-GPU implementation are in Table 1 exhibit performance of 7.82Gflops obtained by executing our multi-GPU SPMV implementation on two large graphs. These are graphs under-perform on CPU execution, and in the multi-GPU implementation exhibiting near double the CPU performance. Specifically, soc-LiveJournal1 on CPU reached a maximum of 4.8 Gflops, which results in a 1.5× speedup when being streamed in a multi-GPU configuration.

3.3 Applications

We present results for applications of graph-operations and analytics on IBM POWER-8 CPU as described already. We compare with the industry standard graph database *Neo4j*; subject to algorithm availability. Furthermore, we demonstrate linear scaling for *level-order* traversals.

3.3.1 Analytics. Four different centralities, detailed in Section 2.4 are presented. We compare with neo4j provided *eigenvector-centrality* which runs orders of magnitude slower. Notably, difficult to compute *subgraph-centrality* can be evaluated in a practical time-frame.

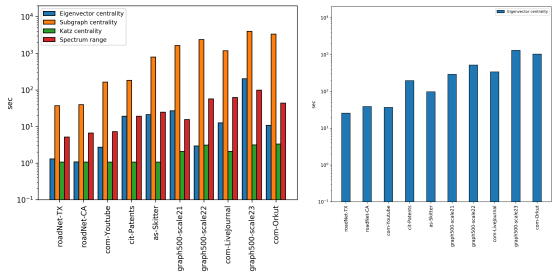

Figure 10: (Left) Runtime of *eigenvector*, *subgraph*, *katz* and *spectrum-range*, enabling on large analytics on graphs. (Right) *eigenvector* centrality for *Neo4j* (other algorithms not provided).

3.3.2 Traversals. To evaluate deep-search capabilities in knowledge graphs with multi-threaded optimized kernels on CPU we have measured algorithm time on a range of graphs from the SNAP and graph500 collections. These have been selected to be representative of different sizes and sparsity. We used a sampling policy to select the starting node for the traversal among the top thousand central nodes. This setup was executed multiple times (10) to achieve statistical importance. Comparing edge traversals with *Neo4j* it is evident how *node-centric* implementation impacts deep-search capabilities. Further demonstrating *level-order* traversal with near constant time traversal per level further justifies our approach.

BFS traversal has been evaluated on (uk-2005) which is in the order of billion edges and tenths of million of vertices. Figure 11. Average running time per level is 0.22 seconds. This graph is not a fully connected graph, therefore obtaining a good measurement can be achieved only by a statistical approach. Specifically, root was chosen randomly and measured

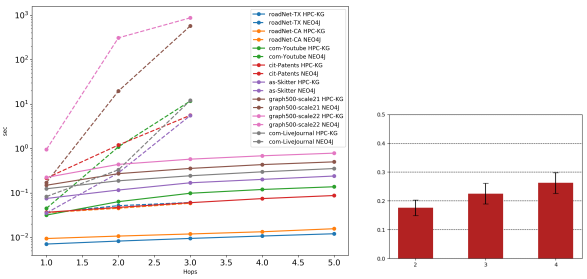

Figure 11: (Left) Runtime of *edge-traversals* for *HPC-KG*, depicted with *dashed-lines*, versus industry standard *Neo4j*. The former enables deep queries. (Right) Level-order traversal (BFS) time for 2, 3 and 4 levels of BFS for the datasets uk-2005 (m = 39.5M, nnz = 936M), showing an average traversal time of 0.22seconds for the first three levels.

time only until a depth four was reached. Here only results after the algorithm had converged before reaching max depth are reported.

4 SUMMARY

We enable fast-deep-n-wide graph operations and analytics on a fat-n-fast single node. That is achieved by selecting a combination of techniques to enhance SPMV and SPMM kernels for graph operations. Mainly, our approach favors a highly blocked matrix COO, along with row-oriented multi-threaded operations. Cache-friendly sorting on the actual blocks further improves performance in the SMT region. We achieved from 1.5× memory footprint compression rate for medium sized weighted graphs, up to 4× for huge un-weighted (binary) graphs. From a performance perspective, we demonstrate near linear speedup up to the number of cores available in the system, and memory bandwidth saturation. Our improvements shows a 1.2 factor improvement when compared to a naive approach. We have also extended to single-GPU and multi-GPU cases, where we show similar speedup results versus state-of-the-art comparison. Finally, we exhibit our techniques or real application problems such as graph-operations and analytics. There we achieve double the performance compared to industry standard adjacency-list oriented implementations. The HPC-KG, among the first BLAS based graph engines, enables deep-search single node in memory processing. Our big next step is to extend to distributed memory processing for HPC cluster processing, to tackle huge knowledge graphs which are becoming available.

REFERENCES

[1] Bekas, C., Kokiopoulou, E., and Saad, Y. An estimator for the diagonal of a matrix. *Applied numerical mathematics 57*, 11-12 (2007), 1214–1229.

[2] Boldi, P., and Vigna, S. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)* (Manhattan, USA, 2004), ACM Press, pp. 595–601.

[3] Bonacich, P. Some unique properties of eigenvector centrality. *Social networks 29*, 4 (2007), 555–564.

[4] Buluc, A., Mattson, T., McMillan, S., Moreira, J., and Yang, C. Design of the GraphBLAS API for C. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International* (2017), IEEE, pp. 643–652.

[5] Estrada, E., and Rodriguez-Velazquez, J. A. Subgraph centrality in complex networks. *Physical Review E 71*, 5 (2005), 056103.

[6] George, A., and Liu, J. W. Computer solution of large sparse positive definite.

[7] Hutchinson, M. F. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics-Simulation and Computation 19*, 2 (1990), 433–450.

[8] Kepner, J., Aaltonen, P., Bader, D., Buluç, A., Franchetti, F., Gilbert, J., Hutchison, D., Kumar, M., Lumsdaine, A., Meyerhenke, H., et al. Mathematical foundations of the GraphBLAS. *arXiv preprint arXiv:1606.05790* (2016).

[9] Kepner, J., and Gilbert, J. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[10] Konig, D. Graphen und matrizen (graphs and matrices). *Matematekai Lapok 38* (1931), 116–119.

[11] Kwak, H., Lee, C., Park, H., and Moon, S. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), ACM, pp. 591–600.

[12] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[13] Murphy, R. C., Wheeler, K. B., Barrett, B. W., and Ang, J. A. Introducing the graph 500. *Cray User's Group (CUG) 19* (2010), 45–74.

[14] Nathan, E., and Bader, D. A. A dynamic algorithm for updating katz centrality in graphs. In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017* (2017), ACM, pp. 149–154.

[15] Staar, P. W., Barkoutsos, P. K., Istrate, R., Malossi, A. C. I., Tavernelli, I., Moll, N., Giefers, H., Hagleitner, C., Bekas, C., and Curioni, A. Stochastic matrix-function estimators: Scalable big-data kernels with high performance. In *Parallel and Distributed Processing Symposium, 2016 IEEE International* (2016), IEEE, pp. 812–821.

[16] Steinberger, M., Zayer, R., and Seidel, H.-P. Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the gpu. In *Proceedings of the International Conference on Supercomputing* (New York, NY, USA, 2017), ICS '17, ACM, pp. 13:1–13:11.

[17] TigerGraph. Real-time deep link analytics, 2018.