

# I/O-Optimal Cache-Oblivious Sparse Matrix–Sparse Matrix Multiplication

Niels Gleinig  
ETH Zurich  
niels.gleinig@inf.ethz.ch

Maciej Besta  
ETH Zurich  
maciej.best@inf.ethz.ch

Torsten Hoefer  
ETH Zurich  
torsten.hoefer@inf.ethz.ch

**Abstract**—Data movements between different levels of the memory hierarchy (I/O-transitions, or simply I/Os) are a critical performance bottleneck in modern computing. Therefore it is a problem of high practical relevance to find algorithms that use a minimal number of I/Os. We present a cache-oblivious sparse matrix-sparse matrix multiplication algorithm that uses a worst-case number of I/Os that matches a previously established lower bound for this problem ( $O\left(\frac{N^2}{B \cdot M}\right)$  read-I/Os and  $O\left(\frac{N^2}{B}\right)$  write-I/Os, where  $N$  is the size of the problem instance,  $M$  is the size of the fast memory and  $B$  is the size of the cache lines). When the output does not need to be stored, also the number of write-I/Os can be reduced to  $O\left(\frac{N^2}{B \cdot M}\right)$ . This improves the worst-case I/O-complexity of the previously best known algorithm for this problem (which is cache-aware) by a logarithmic multiplicative factor. Compared to other cache-oblivious algorithms our algorithm improves the worst-case number of I/Os by a multiplicative factor of  $\Theta(M \cdot N)$ . We show how the algorithm can be applied to produce the first I/O-efficient solution for the sparse 2- vs 3-diameter problem on sparse directed graphs.

## I. INTRODUCTION

Data movements between slow and fast memories (referred to as I/O-transitions or simply I/Os) often dominate the time and energy consumption of computations [33]. Nowadays they are widely considered a principal bottleneck in high performance computing [52]. Simultaneously, algorithms that minimize I/Os have always been of theoretical interest [5], [30], [38], [40]. Therefore, developing such algorithms is a problem of high practical and theoretical value.

When developing an algorithm in the I/O setting, there is a fast internal memory of a size parametrized by  $M$  and a slow external memory of unlimited size (e.g., a cache and a DRAM, or a DRAM and a disk). One can move data between these two memories in blocks of size  $B$ . The number of I/Os performed by a given algorithm (its I/O-complexity) is then a function of  $M$ ,  $B$ , and  $N$ , where  $N$  is the input size. Some existing algorithms use the knowledge of  $M$  and  $B$  to achieve better results. However, in a particularly important class of algorithms called *cache-oblivious* [26] algorithms, the algorithm *cannot* know the values  $B$  and  $M$ . Specifically, the I/O-complexity of such an algorithm is still expressed using  $N$ ,  $M$ , and  $B$ , but its pseudocode must *not* explicitly use the knowledge of  $M$  or  $B$ . Efficient cache-oblivious algorithms are of particular value because they do not have to be tuned for different architectures, but instead work well “out of the box”, for different architectures.

Matrix-matrix multiplication is one of the most fundamental problems in computing. While many use cases focus on dense matrices (e.g., eigenvalue factorization [19], triangular solvers [21], machine learning [10], [11]), a plethora of applications use sparse matrix-sparse matrix multiplication (SpGEMM). Some examples include problems in engineering [29], general computational science [46], [54], graph processing [7], [13], [22], [32], [42], [51], and others. As matrices of interest can be prohibitively large, developing I/O-efficient SpGEMM algorithms is of great importance.

There is a long line of work dedicated to I/O-efficient SpGEMM algorithms and to the corresponding I/O lower bounds. Those include that of Amossen and Pagh [4], Pagh and Stöckel [43], Dusefante and Jakob [25], or Greiner [27]. However, none of them is I/O-optimal for general sparse matrices, and most are not cache-oblivious.

Addressing the above challenge, *we deliver the first SpGEMM algorithm that is I/O-optimal, cache-oblivious, and works for arbitrary sparse matrices*. The key idea is to appropriately reformulate a product of two sparse matrices of size  $N$  such that it is expressed as four products of matrices of size  $N/2$ , and two vector additions. We show that this decomposing of SpGEMM, combined with recursing this decomposition on each of the four partial products, yields an algorithm that is not only I/O-optimal (with respect to the worst case) and cache-oblivious, but also simple and deterministic.

When analyzing our algorithm, we explicitly distinguish between the complexity of I/O-reads and I/O-writes. We motivate this approach with prevalent differences in the demand for conducting reads and writes in algorithms. Other motivations are the different costs of reads and writes in the majority of architectures. With this we extend a current line of research on algorithms for “asymmetric read and write costs” [17].

Specifically, our work makes the following contributions:

- We develop a cache-oblivious I/O-optimal algorithm for SpGEMM *with* storing.
- We show how to extend our algorithm for SpGEMM to the setting *without* storing.
- We extensively compare our algorithm to state of the art, illustrating its superiority over best available baselines (both in cache-oblivious and in cache-aware settings) in terms of I/O complexity. Our design is also deterministic and substantially simpler than previous methods.

- We apply our algorithm to obtain the first I/O-efficient algorithm for the 2- vs 3-diameter graph problem.

## II. FUNDAMENTAL CONCEPTS

We first describe basic concepts and notation.

### A. The I/O-model

In this section we recall the I/O-model of Aggarwal and Vitter [1] as well as the definitions of the technical terms that we use in this paper. In the I/O-model we have an **internal memory** of size  $M$  and an **external memory** of unlimited size. We can move **blocks** of  $B < M$  contiguous data items between the internal and the external memory. Each of those data movements between the slow and the fast memory counts as one **I/O**. The **I/O-complexity of an algorithm** is defined as the maximal number of I/Os that the algorithm performs for an instance of size  $N$  using a fast-memory of size  $M$  and blocks of size  $B$ . I/O-complexities are typically given in  $O$ -notation and therefore two algorithms that use a number of I/Os that differs at most by a constant multiplicative factor are said to have the same I/O-complexity. The **I/O-complexity of a computational problem** is the minimal I/O-complexity among all algorithms that solve this problem.

The related literature does not typically distinguish between the *direction* of I/Os. In some cases it even simplifies the complexity of writing by not requiring the output to be stored in the end (although in real hardware settings, write-I/Os can be the more expensive ones). For many algorithms this may be justified as these two quantities are equal up to constant multiplicative factors or the number of writes is smaller (one can easily check this for sorting and for all algorithms that have the property that most of the values that we write will be read again). But when these quantities are different, it does make sense to distinguish them and minimize the maximum of them (especially when the number of writes is the larger one). As we show later, for sparse matrix products we need asymptotically different numbers of data movements in the two directions and thus we distinguish between **read-I/Os** (data movements from slow to fast memory) and **write-I/Os** (data movements from fast to slow memory). Corresponding to these different sorts of I/Os, we use the terms **read-I/O-complexity** and **write-I/O-complexity**.

Frigo, Leiserson, Prokop and Ramachandran [26] introduced the concept of cache-oblivious algorithms. An algorithm is called **cache-oblivious** if it does not need to *know* the hardware parameters  $M$  and  $B$  of the I/O-model. That is, those parameters still exist in the model and play a role in the analysis of the algorithm, but we can write the pseudo-code without specifying them. Interestingly, if a cache-oblivious algorithm performs a given computation I/O-optimally, it does this for all possible values of  $M$  and  $B$ , because we do not specify those values and hence they could be arbitrary. In particular, they are also optimal for the different values  $M$  and  $B$  of the different levels of the memory hierarchy of the given hardware (L1, L2, L3, DRAM, flash etc) and hence optimize the communication on all levels of the memory hierarchy *at the same time*.

### B. Sparse Matrix-Sparse Matrix Multiplication

We assume that the sparse  $n \times k$  matrix  $U$  and the sparse  $k \times m$  matrix  $V$  are given by a list of their non-zero entries (“COO format”)

$$S_U = \{(i, j, U_{i,j}) | U_{i,j} \neq 0\} \quad (1)$$

and

$$S_V = \{(i, j, V_{i,j}) | V_{i,j} \neq 0\}. \quad (2)$$

We denote the number of non-zero entries  $N_U := |S_U|$ ,  $N_V := |S_V|$ , and define the problem size by the total number of non-zero entries  $N := |N_U| + |N_V|$ . We also assume that in the beginning the elements of  $S_U$  and  $S_V$  are stored in a contiguous block of the external memory. We do not need to assume that the entries  $S_U$  or  $S_V$  are given in any particular order (for example as a Peano curve or in row- or column-major order) since we can bring them into the necessary order with an additional amount of I/Os that does not increase the overall I/O-complexity of this algorithm. For the same reason our algorithm can also be extended to most other sparse matrix storage formats (LiL, CSR, CRS, etc).

We want to compute the entry list

$$S_C = \{(i, j, C_{i,j}) | C_{i,j} \neq 0\} \quad (3)$$

for the matrix  $C := UV$  and have this list stored in a contiguous block of the external memory in the end.

We also consider the case where  $C$  does not need to be stored, but we only need to “see” all its non-zero entries in internal memory. Here we demand that we see each non-zero entry of the product matrix exactly once (so that by solving this problem we can for example answer how many non-zero entries there are in the product matrix, or how often a particular value occurs). This is a slightly different problem from an algorithmic point of view and we call it **SpGEMM without storing**. About half of the previous work on SpGEMM in the I/O-model considers this version of the problem.

## III. PREVIOUS WORK

Hong and Kung [31] showed that multiplication of dense  $n \times n$  matrices needs at least  $\Omega\left(\frac{n^3}{\sqrt{M}}\right)$  I/Os in a model that is called the red-blue pebble game. The red-blue pebble game differs from the later developed I/O-model of Aggarwal and Vitter in two aspects: 1.) It does not take the size of the cache-line into account (this can be compared to the case  $B = 1$  of the I/O-model). 2.) It is always “played” on a particular cDAG (computation directed acyclic graph), which means that the algorithm that is used to do the computation needs to be fixed and the red-blue pebble game can only be used to answer what is the I/O-optimal way to run this particular algorithm, but it does not answer what is the I/O-optimal algorithm for the given problem. Recent works optimized the I/Os in this setting beyond the  $O$ -notation [35]–[37].

Bader and Zenger [8] presented a cache-oblivious algorithm for dense matrix multiplication in the I/O-model of Aggarwal

Reference	Worst-case read-I/Os	Worst-case write-I/Os	Overall I/Os	Cache-oblivious?	Deterministic?	Storing output?
[27] "sorting-based"	$O\left(\frac{N^3}{B} \log\left(\frac{N}{B}\right)\right)$	not considered	$O\left(\frac{N^3}{B} \log\left(\frac{N}{B}\right)\right)$	☞	☺	Not considered
[27] "tile-based"	$O\left(\frac{N^3}{B\sqrt{M}}\right)$	$O\left(\frac{N^2}{B}\right)$ ★	$O\left(\frac{N^3}{B\sqrt{M}}\right)$	☞	☺	Not considered
Dusefante and Jakob [25]	$\tilde{O}\left(\frac{N^3}{B}\right)$	not considered	$\tilde{O}\left(\frac{N^3}{B}\right)$	☺	☞	Not considered
Amossen and Pagh [4]	$O\left(\frac{N^2}{B \cdot M^{1/8}}\right)$	not considered	$O\left(\frac{N^2}{B \cdot M^{1/8}}\right)$	☞	☺	☞
Pagh and Stöckel [43]	$O\left(\frac{N^2}{B \cdot M} \min\left(\max\left(1, \log_{M/B}\left(\frac{N}{M}\right)\right), B\right)\right)$	not considered	$\tilde{O}\left(\frac{N^2}{B \cdot M}\right)$	☞	☞	☞
This work "with storing"	$O\left(\frac{N^2}{B \cdot M}\right)$ ★	$O\left(\frac{N^2}{B}\right)$ ★	$O\left(\frac{N^2}{B}\right)$ ★	☺	☺	☺
This work "without storing"	$O\left(\frac{N^2}{B \cdot M}\right)$ ★	$O\left(\frac{N^2}{B \cdot M}\right)$	$O\left(\frac{N^2}{B \cdot M}\right)$ ★	☺	☺	☞

TABLE I: Comparison of external memory algorithms for SpGEMM: "★" indicates optimality for the respective problem (*with* or *without storing*). We assume sparsity ( $n \in \Theta(N)$ ) to simplify the expressions and comparisons, although some of the methods also work for more general cases. For output sensitive algorithms we have used  $Z = N^2$  since we are interested in the worst-case behavior. The algorithms of Greiner [27] are very efficient for multiplication of regular matrices, but one could use them to perform multiplication of general sparse matrices (as explained later): The corresponding I/O-complexities that are listed in this table, are obtained by setting  $k_1 = k_2 = N$  in (4) and (5).

and Vitter. Their algorithm uses  $O\left(\frac{n^3}{B\sqrt{M}}\right)$  I/Os and is I/O-optimal among algorithms that follow a "standard matrix multiplication cDAG", because it uses  $\frac{1}{B}$  times the number of I/Os needed by Hong and Kungs lower bound (and with  $B'$  I/Os in the setting  $B = 1$  we can model one I/O in the setting  $B = B'$ ). Demaine et al [23] consider the I/O-complexity of dense matrix multiplication with other methods than standard matrix multiplication.

#### A. Sparse Matrices

Also SpGEMM has been widely studied from the perspective of I/O-complexity.

Amossen and Pagh [4] present a cache-aware algorithm for sparse matrix multiplication that uses  $O\left(\frac{N\sqrt{Z}}{B \cdot M^{1/8}}\right)$  I/Os, where  $Z$  is the number of non-zero entries of the product matrix. This is for worst-case instances  $O\left(\frac{N^2}{B \cdot M^{1/8}}\right)$ .

Dusefante and Jakob [25] present a cache-oblivious algorithm for sparse matrix multiplication that uses  $\tilde{O}\left(\frac{ZN}{B}\right)$  I/O-transitions, which is efficient when  $Z$  is small, but for worst-case instances this is  $\tilde{O}\left(\frac{N^3}{B}\right)$ . To the best of our knowledge, this was so far the only cache-oblivious algorithm for SpGEMM.

Greiner [27] considers in his PhD thesis the special case of multiplying sparse matrices that are row- or column-regular (that is, matrices that have a constant number of non-zero entries per row or column). He shows that a  $k_1$ -column regular  $n \times n$  matrix  $U$  can be multiplied with a  $k_2$ -row regular  $n \times n$  matrix  $V$  using

$$O\left(\frac{k_1 \cdot k_2 \cdot n}{B} \log(k_1 \cdot n/B)\right) \quad (4)$$

cache-aware I/Os. Notice that this low I/O-complexity is only possible because the product of a  $k_1$ -column regular matrix with a  $k_2$ -row regular matrix has at most  $k_1 \cdot k_2 \cdot n$  non-zero entries. This algorithm could also be used to perform multiplications of general non-regular sparse matrices by treating them as regular matrices (that is, viewing some of the zeros as

non-zeros until there are the same number of non-zero entries in each row or column). However, this would lead in worst-case instances to  $k_1, k_2 \in \Theta(n)$  and an I/O-complexity that is cubic in  $n$ .

The same PhD thesis presents a cache-aware "tile-based algorithm" that solves this special problem with  $O(n^2/B)$  write-I/Os and

$$O\left(\frac{n^2}{B} \sqrt{\frac{k_1 \cdot k_2}{M}}\right) \quad (5)$$

read-I/Os. Both algorithms of this PhD thesis are cache-aware.

Pagh and Stöckel [43] show that the I/O-complexity of multiplication without storing can be lower bounded by

$$\Omega\left(\frac{N}{B} \min\left(\sqrt{\frac{Z}{M}}, \frac{N}{M}\right)\right), \quad (6)$$

where  $Z$  is the number of non-zero entries of  $U \cdot V$ . They also present a randomized algorithm that solves this problem with

$$O\left(\min\left(\max\left(1, \log_{\frac{M}{B}}\left(\frac{N}{M}\right)\right), B\right) \cdot \frac{N}{B} \min\left(\sqrt{\frac{Z}{M}}, \frac{N}{M}\right)\right) \quad (7)$$

I/Os and hence is optimal up to a multiplicative factor of

$$\min\left(\max\left(1, \log_{\frac{M}{B}}\left(\frac{N}{M}\right)\right), B\right). \quad (8)$$

These I/O-complexities are achieved with an algorithm that does not store the result, but the algorithm can be adapted to store the result. Notice however that the product of two sparse matrices can have up to  $N^2$  non-zero entries and therefore requires in the worst case  $\Omega\left(\frac{N^2}{B}\right)$  write-I/Os to be stored. Consequently, by storing the result, the lower bound on the worst-case number of write-I/Os increases to  $\Omega\left(\frac{N^2}{B}\right)$ . Table I gives an overview of SpGEMM algorithms in the I/O-model.

Finally, there is more distantly related work that can be mentioned. Greiner and Jacob [28] analyze the I/O-complexity of sparse matrix dense matrix multiplication. Bender et al [12]

present cache-aware and cache-oblivious algorithms for sparse matrix dense vector multiplication. Ballard et al [9] study sparse matrix multiplication in a parallel setting. The doctoral thesis of Scott [48] proves I/O-lower bounds for general recursive matrix multiplication. However, those bounds do not directly apply to the problem of our paper, because they are derived in a different model (in particular, not considering the size of the cache lines).

Overall, there is extensive work on matrix multiplication from the perspective of performance. This is not surprising given the high importance of this computational problem in essentially all areas of modern science and engineering. However, for sparse matrix multiplication, the best known algorithm requires in the worst case a number of read-I/Os that exceeds the best worst-case I/O-complexity of this problem by a logarithmic multiplicative factor. Among the cache-oblivious algorithms this multiplicative factor is even of the order of  $\Theta(M \cdot N)$ . Furthermore, in the previous work the write-I/O-complexity was usually not considered separately from the read-I/O-complexity.

Some of the existing algorithms work only on restricted classes of sparse matrices (for example matrices with a bounded or even constant number of entries in each column).

Many of the existing algorithms have a runtime that strongly depends on other parameters than the input size. These algorithms perform well when the given instance of the problem lies in a suitable parameter range, but can perform very poorly on the worst-case instances of a given size. Particularly disadvantageous is that these parameters are often initially unknown for a given problem instance and not trivial to determine. For example the so called **output-sensitive** algorithms, depend on the number of non-zero entries of the output.

The previous work often did not require the output to be written back into external memory (SpGEMM without storing). This can be motivated by the fact that for many applications of matrix multiplications we just want to verify if the product matrix has a certain property (for example if the square of adjacency matrices has some entry that is larger than 10, which is equivalent to asking if there is a pair of vertices that has more than 10 neighbors in common), but we do not look at the product matrix again. Yet for those cases where we want to store the output, it is non-trivial to do the writing efficiently. In particular, since the output can be much larger than the input (up to size  $N^2$  for input size  $N$ ) it is important that we write the output values immediately into the right place whenever possible, to avoid expensive relocations at the end (we do not require the output entries to be sorted, but they need to be stored in a contiguous block of memory). Compressing a scattered output into a contiguous block could require  $\Omega(N^2/B)$  read- and write-I/Os, which is  $\Theta(M)$  times larger than the optimal worst-case read-I/O-complexity of sparse matrix multiplication.

### B. Our Results

We present a cache-oblivious algorithm for SpGEMM *with* storing. Our algorithm has optimal I/O-complexity. It has a

worst-case read-I/O-complexity of  $O\left(\frac{N^2}{B \cdot M}\right)$ , which significantly improves the worst-case read-I/O-complexity  $\tilde{O}\left(\frac{N^3}{B}\right)$  of the cache-oblivious algorithm of Dusefante and Jakob. It even improves the worst-case read-I/O-complexity of the cache-aware algorithm of Pagh and Stöckel [43] by a logarithmic multiplicative factor and makes it I/O-optimal. Furthermore, it uses  $O\left(\frac{N^2}{B}\right)$  write-I/Os, which is also optimal for this problem. Our algorithm is deterministic and substantially simpler than the algorithm of Pagh and Stöckel, which is randomized and cache-aware. We also show how our approach can be adapted to perform SpGEMM *without* storing, using  $O\left(\frac{N^2}{B \cdot M}\right)$  I/O-writes and the same number of I/O-reads. Our algorithms perform optimally on five of the six different cases that correspond to the two problems (SpGEMM with and without storing) and three types of I/Os (read-I/Os, write-I/Os and overall I/Os).

By being optimized with respect to worst-case-performance on general instances rather than being optimized for some subclass of instances corresponding to a special range of parameters or special non-zero structures (as most of the previous algorithms), our algorithm can be applied to obtain I/O-efficient solutions for other theoretical problems such as combinatorial and (sparse) graph problems, which are notoriously difficult in the I/O-setting. As an example, we show how to solve the 2- vs 3-diameter problem (defined later) on sparse, directed graphs with  $O\left(\frac{|V|^2}{B \cdot M}\right)$  cache-oblivious I/Os.

## IV. THE ALGORITHM

We begin by sorting the elements of  $S_U$  in row-major order:  $(i, j, U_{i,j}) \in S_U$  is placed before  $(g, h, U_{g,h}) \in S_U$  if and only if  $i < g$  or  $(i = g) \wedge (j < h)$ . Likewise, we sort the elements of  $S_V$  in column-major order:  $(i, j, V_{i,j}) \in S_V$  is placed before  $(g, h, V_{g,h}) \in S_V$  if and only if  $j < h$  or  $(j = h) \wedge (i < g)$ . Then we split the entry list of  $U$  in the middle (giving one more entry to one of the halves if  $|S_U|$  is odd) and define the matrix  $U^\wedge$  to be the matrix (of the same dimensions as  $U$ ) that has the non-zero entries of the first half of  $S_U$  and define  $U^\vee$  to be the matrix (also of the same dimensions as  $U$ ) that has the non-zero entries of the second half of  $S_U$ . Since  $S_U$  is in row-major order, there exists some row index  $split_U \in \{1, \dots, n\}$  such that  $U^\wedge$  is equal to  $U$  in the rows  $i < split_U$  and zero in the rows  $i > split_U$  and the matrix  $U^\vee$  is equal to  $U$  in the rows  $i > split_U$  and zero in the rows  $i < split_U$ . On the  $split_U$ -th row, both matrices  $U^\wedge$  and  $U^\vee$  may have non-zero entries. Furthermore, the matrices  $U^\wedge$  and  $U^\vee$  have half the number of elements of  $U$  and we have  $U = U^\wedge + U^\vee$ . Likewise, we define the matrices  $V^\wedge$  and  $V^\vee$  as the matrices that correspond to the first and second half of  $S_V$  and have the same dimension as  $V$ . Since  $S_V$  is in column-major order, there exists now a column index  $split_V \in \{1, \dots, m\}$  such that  $V^\wedge$  is equal to  $V$  in the columns  $j < split_V$  and zero in the columns  $j > split_V$  and the matrix  $V^\vee$  is equal to  $V$  in the columns  $j > split_V$  and zero in the columns  $j < split_V$ . Also, the matrices  $V^\wedge$  and  $V^\vee$  have half the number of elements of  $V$  and we have  $V = V^\wedge + V^\vee$ . It follows that

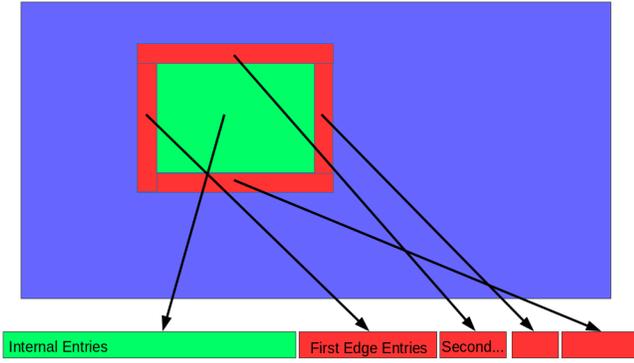


Fig. 2: The layout of internal entries and edge-entries in memory

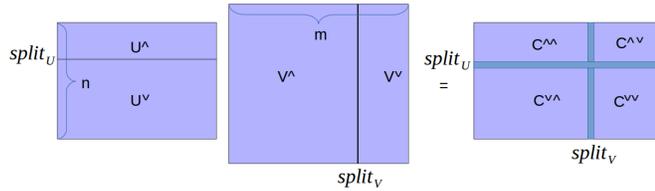


Fig. 1: The reduction that gives rise to our recursive algorithm. The four corners of the matrix on the right (the product  $C := U \cdot V$ ) coincide with the matrices  $C^{AA}$ ,  $C^{AV}$ ,  $C^{VA}$  and  $C^{VV}$ , while – in the  $split_U$ -th row and the  $split_V$ -th column –  $C$  equals the sum of those four matrices.

$$C = (U^A + U^V)(V^A + V^V) = U^A V^A + U^V V^A + U^A V^V + U^V V^V \quad (9)$$

and, for each entry with  $(i \neq split_U)$  and  $(j \neq split_V)$ , at most one of the four matrices  $C^{AA} := U^A V^A$ ,  $C^{VA} := U^V V^A$ ,  $C^{AV} := U^A V^V$  or  $C^{VV} := U^V V^V$  will be non-zero. Thus, those entries are given as the output of one of those four smaller matrix products (in the upper left corner,  $C$  coincides with  $C^{AA}$ ; in the lower left, it coincides with  $C^{VA}$ , and so on) and the value of  $C$  on the remaining entries (those that are either in the  $split_U$ -th row or the  $split_V$ -th column) is given by the sum of the values of the matrices  $C^{AA}$ ,  $C^{VA}$ ,  $C^{AV}$  and  $C^{VV}$  in those entries. Hence, it suffices to sum the  $split_U$ -th rows and the  $split_V$ -th columns of those four matrices.

---

**ALGORITHM 1:** Input: Sparse  $n \times k$  matrix  $U$  and sparse  $k \times m$  matrix  $V$  given by lists of non-zero entries  $S_U$  and  $S_V$ . Output: Matrix  $C = UV$  given by its list of non-zero entries  $S_C$ .

---

Order the entries in  $S_U$  row-wise;  
Order the entries in  $S_V$  column-wise;  
Compute  $C^{AA} := U^A V^A$  using this algorithm;  
Compute  $C^{AV} := U^A V^V$  using this algorithm;  
Compute  $C^{VA} := U^V V^A$  using this algorithm;  
Compute  $C^{VV} := U^V V^V$  using this algorithm;  
Set  $C = C^{AA} + C^{AV} + C^{VA} + C^{VV}$ ;

---

In other words, the multiplication of two sparse matrices can be reduced to four multiplications of sparse matrices with half the number of non-zero entries and two sparse

vector additions. This gives rise to our recursive Algorithm 1. Figure 1 shows the reduction that underlies the recursive step.

The pseudocode of Algorithm 1 does not specify how we perform the I/Os because we do not have direct control over this in cache-oblivious algorithms (after all we do not know where the blocks start and end because we do not know their size). Frigo et al. [26] showed that we can assume WLOG that values are evicted from fast memory according to an optimal eviction policy. But even with an optimal eviction policy, in order to achieve I/O-efficiency we need to store the values in a way that ensures spatial and temporal locality. In the following section we show how to achieve this.

### A. Storing Values

In order to explain how we store the matrices of the recursive subcomputations, we need some formal definitions.

**Definition 1.** A non-zero entry of a sparse matrix is an **edge-entry** if

- its row-index is minimal or maximal among the row-indices of all non-zero entries, or
- its column index is minimal or maximal among the column-indices of all non-zero entries.

**Definition 2.** We define the **internal entries** to be all non-zero entries that are not edge-entries.

**Definition 3.** A sparse matrix is stored in **internal-edge form** if all its non-zero entries are stored as visualized by Figure 2. That is, they are stored in a contiguous block of memory, such that each non-zero entry is stored exactly once and

- the internal entries are stored in the first segment,
- the edge-entries with minimal column-index are stored in the next segment and are sorted with respect to their row index,
- the edge-entries with minimal row-index are stored in the next segment and are sorted with respect to their column index,
- the edge-entries with maximal column-index are stored in the next segment and are sorted with respect to their row index,
- the edge-entries with maximal row-index are stored in the final segment and are sorted with respect to their column index.

For example, the edge-entries of the matrix  $C^{AA}$  are all the non-zero entries that are either:

- In the first column (assuming  $C^{AA}$  has non-zero entries in the first column).
- In the first row (assuming  $C^{AA}$  has non-zero entries in the first row).
- In the first  $split_V$ -th column (assuming  $C^{AA}$  has non-zero entries in the  $split_V$ -th column).
- Or in the first  $split_U$ -th row (assuming  $C^{AA}$  has non-zero entries in the  $split_U$ -th row).

To keep track of the I/O-costs of the transformations that we present in this section, we need the following lemma to estimate how many edge-entries are in the four matrices.

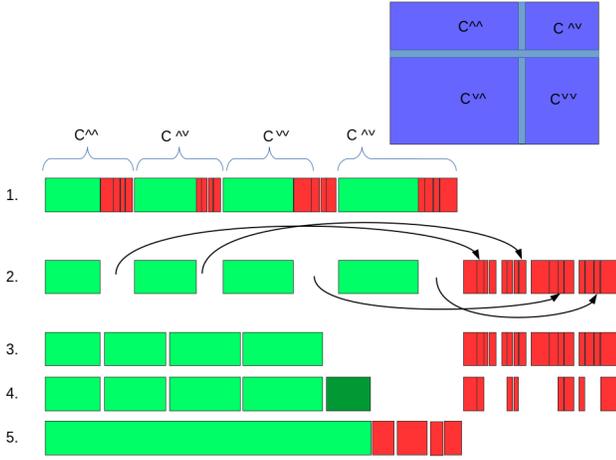


Fig. 3: The process of transforming submatrices into larger matrices. 1. The four submatrices stored in a contiguous block 2. Extracting the edge-entries 3. “Restocking” 4. Computing and appending the entries of the  $split_U$ -th row and  $split_V$ -th column to  $C$  5. Concatenating and appending the edge-entries of  $C$ .

**Lemma 1.** *Each of the matrices  $C^{AA}$ ,  $C^{VA}$ ,  $C^{AV}$ , and  $C^{VV}$  has at most  $4 \cdot N$  edge-entries.*

*Proof.* We will show this for  $C^{AA}$ , the proofs for the other matrices follow the identical line of arguments.

$C^{AA} := U^A V^A$  has only non-zero entries in those rows in which  $U^A$  has non-zero entries and in those columns in which  $V^A$  has non-zero entries. However, the matrices  $U^A$  and  $V^A$  have together at most  $\lceil N/2 \rceil + 1 \leq N$  non-zero entries (by construction) and therefore they have at most  $N$  different rows and columns with non-zero entries. Consequently, there are at most  $N$  different rows and columns in which  $C^{AA}$  has non-zero entries. In particular, in each of the 4 rows and columns that have edge-entries, there are at most  $N$  non-zero entries.  $\square$

Now we are in a position to describe how we store recursively computed submatrices and transform them into larger matrices. This process is visualized in Figure 3 and we describe it now in more detail. We store the matrices  $C^{AA}$ ,  $C^{VA}$ ,  $C^{AV}$ , and  $C^{VV}$  of Algorithm 1 in internal-edge form in a contiguous block of memory, one matrix right after the other. Thus, this block starts with the internal entries of  $C^{AA}$ , followed by the edge entries of  $C^{AA}$ , followed by the internal entries of  $C^{VA}$ , and continuing in this fashion, ending on the edge-entries of  $C^{VV}$ . In the remainder of this section, we will show that using only a few I/Os can transform this layout of entries to become an internal-edge form of the matrix  $C$  that we want to compute.

We start extracting the edge-entries from between the internal entries and move them to a separate block of memory, so that between the four blocks of internal entries there are three gaps. We eliminate these gaps by “restocking” the blocks of internal entries to form a new contiguous block that contains all internal entries of the four matrices. Since we do not care

about the order in which the internal entries are stored, we do not need to shift the blocks of internal entries (which would be expensive) but can simply pick entries from the end of the block and fill the gaps. From Lemma 1 we know that each of the gaps has at most size  $4N$  and therefore the steps of extracting the edge-entries and restocking the internal entries can be done with  $O(N/B)$  I/Os.

Now, this new contiguous block of internal entries of the four matrices already consists of internal entries of  $C$ . To compute the remaining internal entries of  $C$  (these are the internal entries in the  $split_U$ -th row or  $split_V$ -th column) we need to add the corresponding rows and columns of these four matrices. Since the non-zero entries of these rows and columns are edge-entries of the four matrices, they are already index-sorted and we can efficiently access and add them. In particular, this step and the step of appending these newly computed internal entries to the other internal entries, can also be done with  $O(N/B)$  I/Os.

Thus, we have created a contiguous block that contains internal entries of  $C$  and it only remains to compute the edge-entries of  $C$ .

Notice that the first edge-entries of the matrix  $C$  (those with minimal column-index) are given by the first edge-entries of  $C^{AA}$ , followed by the first edge-entries of  $C^{VA}$  (if these two matrices have their first edge-entries in different columns, then some of those entries are not edge-entries of  $C$  and we append them to the contiguous block of internal entries of  $C$ ; if both  $C^{AA}$  and  $C^{VA}$  have a non-zero edge-entry in the  $split_U$ -th column, we add both values). Likewise, the other edge-entries of  $C$  can be obtained by adequately concatenating the remaining edge-entries of  $C^{AA}$ ,  $C^{VA}$ ,  $C^{AV}$ , and  $C^{VV}$ . Since there are in total  $O(N)$  edge-entries, all these steps of concatenating and appending can be done with  $O(N/B)$  I/Os. In the following lemma, we summarize the properties of the transformation that we just described.

**Lemma 2.** *If the matrices  $C^{AA}$ ,  $C^{VA}$ ,  $C^{AV}$ , and  $C^{VV}$  are stored in internal-edge form on a contiguous block of memory, we can obtain  $C$  in internal-edge form with  $O(N/B)$  I/Os.*

1) *Storing Values for SpGEMM without Storing:* Notice that the internal entries of  $C^{AA}$ ,  $C^{VA}$ ,  $C^{AV}$ , and  $C^{VV}$  are entries of the matrix  $C$  and no further computations need to be done with them. Therefore, as soon as we see them during one of the computations of the four submatrices, we can already discard them. Thus, the difference to SpGEMM *with* storing is that we do not need to store the internal entries. Yet, we do need to store the edge-entries to compute the submatrices recursively. It is easy to see that the transformations that we do at each recursive step to compute the edge-entries, can also be done with  $O(N/B)$  I/Os.

### B. The I/O-complexity of SpGEMM with Storing

In this section, we will prove the following theorem using the techniques of Pagh and Stöckel [43].

**Theorem 1.** *For all  $N \in \mathbb{N}$ :*

- 1) There exist matrices  $U$  and  $V$  with  $O(N)$  non-zero entries, such that to compute and store the matrix  $C = U \cdot V$ , we need  $\Omega\left(\frac{N^2}{B \cdot M}\right)$  read-I/Os.
- 2) There exist matrices  $U$  and  $V$  with  $O(N)$  non-zero entries, such that to compute and store the matrix  $C = U \cdot V$ , we need  $\Omega\left(\frac{N^2}{B}\right)$  write-I/Os.
- 3) For any matrices  $U$  and  $V$  with  $O(N)$  non-zero entries, Algorithm 1 uses  $O\left(\frac{N^2}{B \cdot M}\right)$  read-I/Os and  $O\left(\frac{N^2}{B}\right)$  write-I/Os and matches therefore the lower bounds established in 1) and 2).

We first prove (1) and (2) and leave (3) for the following section.

**Note:** There are many practically relevant instances that can be solved with I/O-complexity below the lower bounds of 1) and 2). Indeed, as we mentioned in section III there exist algorithms that do exploit these possibilities (of achieving I/O-complexities below these bounds) by using a more fine-grained analysis of their I/O-complexity. Hence, these algorithms can be of great practical value. On the other hand, we present the first algorithm for which statement 3) of this theorem has been proven, making it the first truly I/O-optimal solution.

*Proof of (1) and (2) of the theorem.* For any  $N \in \mathbb{N}$ , let  $U$  and  $V$  be  $N \times N$ -matrices with  $N$  non-zero entries: the non-zero entries of  $U$  are exactly those in the first column and the non-zero entries of  $V$  are exactly those in the first row. Then, to compute  $U \cdot V$  we need to perform  $N^2$  many multiplications because each non-zero entry of  $U$  needs to be multiplied with each non-zero entry of  $V$ . However, with each additional read-I/O we can only perform  $O(M \cdot B)$  new multiplications because each of the  $O(B)$  values that are being brought into memory can be multiplied at most with the  $O(M)$  values that are in memory. Therefore, to perform all  $N^2$  multiplications we need to perform  $\Omega\left(\frac{N^2}{B \cdot M}\right)$  read-I/Os. Further, since  $U \cdot V$  has  $N^2$  non-zeros we need  $\Omega\left(\frac{N^2}{B}\right)$  write-I/Os to store them.  $\square$

1) *The overall I/O-complexity of our Algorithm:* In this section, we prove the claims about the I/O-complexity of our algorithm to conclude the proof of the theorem. Before considering the I/O-complexity of general instance sizes we look at the *base case*: instances small enough to fit entirely into fast memory.

**Lemma 3.** *When  $N \in O(M)$ , Algorithm 1 uses  $O(M/B)$  read-I/Os.*

*Proof.* Recall that the number of I/Os used with a LRU-eviction policy differs only by constant factors from the number of I/Os used by an optimal eviction policy [26]. Therefore we can assume WLOG that whenever we remove a block of data from fast memory, this block contains values that will be reused farthest in the future. In particular, if we have a block of data that will not be needed again in the computation

(for example entries of  $U \cdot V$  that we finished to compute) this block will be evicted before we evict any block of data that contains values that are still needed. Hence, if we can show that Algorithm 1 has at no moment more than  $O(M)$  values that will be needed again in the computation, the result would follow: we would spend  $O(M/B)$  I/Os to read each of the entries of  $U$  and  $V$  once and after that we would never need to write (and later re-read) temporary values to external memory because at all times all values that are still needed fit into fast memory.

At each moment during the execution of Algorithm 1, the set of values that are needed again can be partitioned as follows:

- 1) entries of  $U$  and  $V$ ,
- 2) edge-entries of  $C^{\wedge\wedge}$ ,  $C^{\wedge\vee}$ ,  $C^{\vee\wedge}$ , and  $C^{\vee\vee}$
- 3) values that will be needed again during whichever sub-computation the algorithm is currently performing (computation of  $C^{\wedge\wedge}$ ,  $C^{\wedge\vee}$ ,  $C^{\vee\wedge}$ , or  $C^{\vee\vee}$ ).

The total number of values in (1) is in  $O(M)$ . Further, Lemma 1 implies that the total number of values in (2) is  $\leq 4 \cdot 4 \cdot N$ . Hence, letting  $A(N)$  denote the total number of values in (2) and (3) this quantity satisfies the recursive relation

$$A(N) \leq 16 \cdot N + A(N/2). \quad (10)$$

Therefore  $A(N) \in O(N)$  and hence  $A(N) \in O(M)$ .  $\square$

**Note:** In order to obtain the I/O-complexity from the previous lemma we did not need to switch to another routine (which of course would be invalid for a cache-oblivious algorithm). Only the analysis depends on  $M$  and  $B$  but the algorithm does not.

**Lemma 4.** *Algorithm 1 uses in the worst case  $O\left(\frac{N^2}{B \cdot M}\right)$  read-I/Os.*

*Proof.* The sorting step in the beginning of our algorithm can be done with [1]

$$O\left(\frac{N}{B} \log_{M/B}(N/B)\right) \quad (11)$$

read- and write-I/Os and is therefore not a bottleneck of our algorithm. Furthermore, once we sorted the input, we do not need to sort it again in the recursive calls of our algorithm. Hence, we can analyze the I/O-complexity of our algorithm with a recursive relation that does not take the I/O-complexity of sorting into account.

If we let  $R(N)$  denote the read-I/O-complexity of our algorithm for input size  $N = N_U + N_V$ , we have the recursive relation

$$R(N) \leq 4 \cdot R\left(\frac{N}{2}\right) + c \frac{N}{B}, \quad (12)$$

where  $4 \cdot R\left(\frac{N}{2}\right)$  corresponds to the computations of  $U^{\wedge\wedge}V^{\wedge\wedge}$ ,  $U^{\vee\vee}V^{\wedge\wedge}$ ,  $U^{\wedge\wedge}V^{\vee\vee}$  and  $U^{\vee\vee}V^{\vee\vee}$  and  $c \frac{N}{B}$ , where  $c$  is some constant, corresponds (recall Lemma 2) to the transformation of the internal-edge form of these submatrices to the internal-edge form of the larger matrix  $C$ .

“Telescoping” the recurrence relation (12), we obtain

$$\begin{aligned}
R(N) &\leq 4^2 \cdot R\left(\frac{N}{2^2}\right) + \left(\frac{4}{2} + 1\right) c \frac{N}{B} \\
&\dots \\
&\leq 4^{\log_2(\frac{N}{M})} \cdot R\left(\frac{N}{2^{\log_2(\frac{N}{M})}}\right) \\
&+ \left(\left(\frac{4}{2}\right)^{\log_2(\frac{N}{M})-1} + \dots + \frac{4}{2} + 1\right) c \frac{N}{B} \quad (13) \\
&= 4^{\log_2(\frac{N}{M})} \cdot R(M) + (2^{\log_2(\frac{N}{M})} - 1) c \frac{N}{B} \\
&= \frac{N^2}{M^2} \cdot R(M) + \left(\frac{N}{M} - 1\right) c \frac{N}{B}.
\end{aligned}$$

Furthermore, according to Lemma 3 we have the base case  $R(M) \in O(\frac{M}{B})$ . Therefore,

$$R(N) \leq \left(\frac{N}{M}\right)^2 \cdot O\left(\frac{M}{B}\right) + c \frac{N^2}{B \cdot M} \in O\left(\frac{N^2}{B \cdot M}\right). \quad (14)$$

□

**Lemma 5.** *Algorithm 1 has a worst-case write-I/O-complexity of  $O(N^2/B)$ .*

*Proof.* Let  $W(N)$  denote the write-I/O-complexity for input size  $N$ . Notice that  $W(N)$  satisfies the same recursive relation (12) as the read-I/O-complexity  $R(N)$ . However, the base case differs, because the product of two matrices with totally  $M$  non-zero entries can have up to  $M^2$  non-zero entries and hence the required number of write-I/Os can be up to

$$W(M) = \frac{M^2}{B} \neq R(M) = \frac{M}{B}. \quad (15)$$

Plugging this different base case into the “telescoped” recursive relation (13), we obtain

$$W(N) \in O(N^2/B). \quad (16)$$

□

### C. The I/O-complexity of SpGEMM without Storing

**Theorem 2.** *SpGEMM without storing can be done with  $O(\frac{N^2}{B \cdot M})$  read- and write-I/Os.*

*Proof.* The recursive relation (12) for read-I/Os and its base case (13) are the same as for SpGEMM *with* storing. Hence, the read-I/O-complexity is the same.

Also the write-I/Os satisfy again the same recursive relation, but the base case is

$$W(M) = O\left(\frac{M}{B}\right), \quad (17)$$

because we only have to store the  $O(M)$  edge-entries of the product matrix. Plugging this base case into the inequality (13), we obtain again

$$W(N) \in O\left(\frac{N^2}{B \cdot M}\right). \quad (18)$$

□

**Note:** The overall I/O-complexity (defined as the sum of read- and write-I/Os) of our algorithm is optimal for SpGEMM without storing, but the write-I/O-complexity  $O(\frac{N^2}{B \cdot M})$  is not optimal. One could solve this problem with 0 write-I/Os by iterating over all  $m \times n$  (possibly zero-valued) entries of the product matrix and compute each one of them with  $O(\frac{N^2}{B \cdot M})$  read-I/Os (this high number of read-I/Os is needed because the input matrices are unsorted and sorting them would require write-I/Os). Hence, this approach with 0 write-I/Os would require  $O(\frac{N^4}{B \cdot M})$  read-I/Os and lead to an overall I/O-complexity that is far from optimal. It remains an open question if this problem can be solved with  $O(\frac{N^2}{B \cdot M})$  read-I/Os and 0 write-I/Os or if there is an inherent trade-off between read-I/Os and write-I/Os.

## V. APPLICATION TO GRAPH ALGORITHMS

It is well known that sparse graph problems require particularly many I/Os (as noted for example by [23]). For some of those problems the state-of-the-art solutions still produce asymptotically one cache-miss per instruction, independently of the size of the internal memory or the cache-line.

Since many sparse graph problems can be translated into sparse matrix multiplication problems (either of the adjacency matrix, incidence matrix or the Laplacian matrix, each of which is sparse when the graph is sparse), it is worthwhile to consider the applicability of our results in this context of algebraic graph theory. Examples of graph computations that involve matrix multiplications include finding cycles [2], [3], [53], subgraphs [41], shortest paths [49], [50], [55], solving reachability [24], [45], matchings [16], [20], [39], [44], and others [18], [34]. So for each of these problems our algorithm could be used to save I/Os. For some graph problems our improved worst-case I/O-complexity may even lead to new state-of-the-art solutions. This is the case with the 2- vs. 3-Sparse-Diameter problem, which we will discuss as a motivating example in the remainder of this section.

### A. Use Case: 2- vs. 3-Diameter Problem

The 2- vs 3-diameter problem consists in determining if the diameter of a given graph is greater than 2. [23] showed that this problem can be solved with  $O(\frac{|V|^2}{B \cdot M})$  I/Os on a sparse (meaning  $|E| \in O(|V|)$ ), **undirected** graph. This was an improvement over the previously best-known  $O(|V|^2 \log(|V|)/B)$  I/O-complexity, that one obtained by solving this problem with the exact diameter computation of [6]. However, for sparse, **directed** graphs we are not aware of any I/O-efficient solutions of the 2- vs 3-diameter problem. To solve this problem again by computing the exact diameter, not even  $O(|V|^2 \log(|V|)/B)$  I/Os are enough, as the previously used method would not be applicable anymore (only for sparse, directed, **planar** graphs this would be enough).

Notice however that this problem can be phrased elegantly in terms of sparse matrix multiplication: When  $G$  is a directed graph with adjacency matrix  $A_G$ , its diameter is greater than 2 if and only if  $A_G \cdot (A_G + I)$  has a non-diagonal entry that is zero. Furthermore, when  $G$  is sparse and given by its adjacency

list, we can efficiently transform this list into the entry lists of the sparse matrices  $A_G$  and  $A_G + I$  (for both matrices each edge  $(a, b)$  is transformed into an entry  $(a, b, 1)$  and to the latter matrix, diagonal ones are added). Hence, we can apply our sparse matrix multiplication algorithm to compute  $A_G \cdot (A_G + I)$  and then verify that the product matrix has no non-diagonal entries that are zero. Notice that the step of verifying that  $A_G \cdot (A_G + I)$  has no non-diagonal entries that are zero, can be done without scanning through all its elements (which could take up to  $\Omega(|V|^2/B)$  I/Os), by directly counting how many non-diagonal non-zero entries we write. And since we do not read the entries of  $A_G \cdot (A_G + I)$  again, we can avoid writing them all together and simply count how many entries we *would* write. Therefore, we can apply here our algorithm for SpGEMM without storing. In this way we solve this problem with  $O(\frac{|V|^2}{B \cdot M})$  I/Os.

## VI. DISCUSSION

1) *Our Notion of Optimality*: In this paper we optimize the worst-case-I/O-complexity in terms of  $N$ . The reason for this choice is that this is the most relevant measure when it comes to applying matrix multiplication to derive solutions for other theoretical problems. For example, when we use matrix multiplication to tackle graph problems as in the previous section,  $N$  equals the size  $|E|$  of the corresponding graph and since for graph algorithms “optimality” typically refers to worst-case-optimality in terms of the size of the graph, we need matrix multiplication algorithms that are optimal in terms of  $N$ . Clearly, there are matrix multiplication algorithms that are more efficient than ours on special classes of matrices. For example, with the algorithm proposed by Greiner [27] we can get better I/O-complexity when we multiply  $O(1)$ -regular matrices and the algorithm of Pagh and Stöckel [43] performs better when the output matrix is again sparse. Also when the input matrices are diagonal or dense (which corresponds to  $N \sim n^2$ ), we can perform multiplication more efficiently. However, these are only special subclasses of matrices, and hence these algorithms cannot be used to derive I/O-optimal algorithms for general graphs (they would only result in efficient solutions for graphs with adjacency matrices that happen to belong to one of these subclasses). Due to the worst-case-optimality in terms of  $N$  achieved by our algorithm, we believe that it has the potential to help produce I/O-efficient solutions for other sparse graph problems.

2) *Matrices with Non-zero Entries in only one Row or Column*: Notice that our algorithm *splits* the matrices  $U$  and  $V$  recursively along “parallel lines” (the matrices from the left are split along a horizontal line and the matrices from the right along a vertical line) until the whole multiplication fits into the fast memory. Therefore, these matrices become always “thinner” and may still be further divided even when one of the matrices has all its non-zero entries in only one row or column. In our pseudo-code we did not consider switching to a different method when we are in this situation, because it would not have improved the worst-case I/O-complexities and

would only have made the pseudo-code more complex. Yet in practice one could save here I/Os by solving this as a sparse matrix vector multiplication problem.

3) *Parallelism*: The computations of the matrix multiplications  $U^\wedge \cdot V^\wedge$ ,  $U^\wedge \cdot V^\vee$ ,  $U^\vee \cdot V^\wedge$ , and  $U^\vee \cdot V^\vee$  are independent of each other and can be performed in parallel. In fact, in the analysis of our algorithms, when we compute for example  $U^\wedge \cdot V^\wedge$  and then  $U^\wedge \cdot V^\vee$ , we count twice the I/Os for reading  $U^\wedge$ . That is, we do not rely on keeping  $U^\wedge$  in memory. Hence, if we assign the computations of the matrices  $U^\wedge \cdot V^\wedge$  and  $U^\wedge \cdot V^\vee$  to different machines, and consequently have to read the matrix  $U^\wedge$  twice, this does not affect the I/O-optimality of our algorithm.

## VII. CONCLUSION

Sparse matrix-matrix multiplication (SpGEMM) is a fundamental problem with a plethora of applications in engineering [29], general computational science [46], [54], graph processing [7], [13], [14], [32], [47], [51], and others [15]. Developing algorithms for solving SpGEMM that minimize the number of I/Os has been of significant interest in both theoretical and practical domains [32], [43]. However, unlike the multiplication of dense matrices, it is challenging to provide a fast SpGEMM due to its inherent lack of locality.

Addressing this challenge, we introduce the first *I/O-optimal* and *cache-oblivious* algorithm for SpGEMM. The worst-case I/O-complexity of our algorithm ( $O(\frac{N^2}{MB})$  read-I/Os and  $O(N^2/B)$  write-I/Os) is not only better than that of the cache-oblivious algorithm of Dusefante and Jakob ( $\tilde{O}(N^3/B)$ ) [25], but it even outperforms the *cache-aware* algorithm of Pagh and Stöckel by a logarithmic multiplicative factor [43].

A key observation and the basis of our algorithm is that the multiplication of two sparse matrices can be reduced to four multiplications of (appropriately preprocessed) half-sized sparse matrices and two vector additions. A recursion on this form of decomposition is the core part of our algorithm. Contrary to several existing schemes, for example a randomized algorithm by Pagh and Stöckel [43], our approach is simple and deterministic. We believe that our recursive decomposition of SpGEMM may be used to tackle other problems related to minimizing I/O-communication.

In the analysis of our algorithm, we distinguish between the complexity of I/O-reads and I/O-writes. With this we extend a current line of research on “asymmetric read and write costs” [17], that is motivated by the inherent differences in the demand for conducting reads and writes in algorithms, as well as the differences of cost of reads and writes in the majority of architectures.

Finally, our algorithm comes with many direct use cases. We offer a broad discussion of its applicability to the domain of graph computations, and one concrete example, in which we apply the algorithm to improve the state-of-the-art solution to the 2- vs 3-sparse-diameter problem on sparse and directed graphs.

## ACKNOWLEDGEMENT

This project has received funding from the European Research Council (ERC ) under the European Union's Horizon 2020 research and innovation programme grant agreement No 101002047 and from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No.101034126.

## REFERENCES

- [1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, Sept. 1988.
- [2] N. Alon, R. Yuster, and U. Zwick. Color-coding. *JACM*, 42(4):844–856, 1995.
- [3] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.
- [4] R. R. Amossen and R. Pagh. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory, ICDT '09*, page 121–126, New York, NY, USA, 2009. Association for Computing Machinery.
- [5] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms. In *WADS*, pages 334–345. Springer, 1995.
- [6] L. Arge, U. Meyer, and L. Toma. External memory algorithms for diameter and all-pairs shortest-paths on sparse graphs. In *ICALP*, pages 146–157, 2004.
- [7] A. Azad et al. Parallel triangle counting and enumeration using matrix algebra. In *IPDPSW*. IEEE, 2015.
- [8] M. Bader and C. Zenger. Cache oblivious matrix multiplication using an element ordering based on a peano curve. *Linear Algebra and its Applications*, 417(2):301 – 313, 2006.
- [9] G. Ballard et al. Brief announcement: Hypergraph partitioning for parallel sparse matrix-matrix multiplication. In *SPAA*, pages 86–88, New York, NY, USA, 2015. ACM.
- [10] T. Ben-Nun et al. A modular benchmarking infrastructure for high-performance and reproducible deep learning. In *IPDPS*, pages 66–77. IEEE, 2019.
- [11] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *CoRR*, abs/1802.09941, 2018.
- [12] M. Bender, G. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the i/o-model. volume 47, pages 61–70, 06 2007.
- [13] M. Besta et al. Slimsell: A vectorizable graph representation for breadth-first search. In *IPDPS*, volume 17, 2017.
- [14] M. Besta et al. To push or to pull: On reducing communication and synchronization in graph computations. In *ACM HPDC*, pages 93–104, 2017.
- [15] M. Besta et al. Communication-efficient jaccard similarity for high-performance distributed genome comparisons. In *IPDPS*, pages 1122–1132. IEEE, 2020.
- [16] M. Besta et al. Substream-centric maximum matchings on fpga. *ACM TRETS*, 13(2):1–33, 2020.
- [17] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Sorting with asymmetric read and write costs. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '15*, page 1–12, New York, NY, USA, 2015. Association for Computing Machinery.
- [18] T. M. Chan. Dynamic subgraph connectivity with geometric applications. *SIAM Journal on Computing*, 36(3):681–694, 2006.
- [19] F. Chatelin. *Eigenvalues of Matrices: Revised Edition*, volume 71. Society for Industrial and Applied Mathematics (SIAM), 2012.
- [20] J. Cheriyan. Randomized  $o(m(-v-))$  algorithms for problems in matching theory. *SIAM Journal on Computing*, 26(6):1635–1655, 1997.
- [21] J. Choi et al. Design and Implementation of the ScalAPACK LU, QR, and Cholesky Factorization Routines. *Sci. Program.*, 5(3), Aug. 1996.
- [22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.
- [23] E. D. Demaine et al. Fine-grained I/O Complexity via Reductions: New Lower Bounds, Faster Algorithms, and a Time Hierarchy. In A. R. Karlin, editor, *ITCS*, volume 94, pages 34:1–34:23, 2018.
- [24] C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: breaking through the  $o(n/\sup 2l)$  barrier. In *FOCS*, pages 381–389. IEEE, 2000.
- [25] M. Dusefante and R. Jacob. Cache oblivious sparse matrix multiplication. In M. A. Bender, M. Farach-Colton, and M. A. Mosteiro, editors, *LATIN 2018: Theoretical Informatics*, pages 437–447, Cham, 2018. Springer International Publishing.
- [26] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1):4:1–4:22, Jan. 2012.
- [27] G. Greiner. *Sparse Matrix Computations and Their I/O Complexity*. PhD thesis, Technische Universität München, München, 2012.
- [28] G. Greiner and R. Jacob. The i/o complexity of sparse matrix dense matrix multiplication. In A. López-Ortiz, editor, *LATIN: Theoretical Informatics*, 2010.
- [29] A. Jennings. *Matrix computation for engineers and scientists(book). London and New York, Wiley-Interscience, 1977. 340 p, 1977.*
- [30] H. Jia-Wei and H.-T. Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333, 1981.
- [31] H. Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. pages 326–333, 1981.
- [32] J. Kepner et al. Mathematical foundations of the GraphBLAS. *arXiv:1606.05790*, 2016.
- [33] G. Kestor et al. Quantifying the energy cost of data movement in scientific applications. In *IISWC*, pages 56–65. IEEE, 2013.
- [34] D. Kratsch and J. Spinrad. Between  $o(nm)$  and  $o(n\alpha)$ . *SIAM Journal on Computing*, 36(2):310–325, 2006.
- [35] G. Kwasniewski et al. Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication. In *ACM/IEEE Supercomputing*, 2019.
- [36] G. Kwasniewski et al. On the parallel i/o optimality of linear algebra kernels: near-optimal matrix factorizations. In *ACM/IEEE Supercomputing*, pages 1–15, 2021.
- [37] G. Kwasniewski et al. Pebbles, graphs, and a pinch of combinatorics: Towards tight i/o lower bounds for statically analyzable programs. In *ACM SPAA*, pages 328–339, 2021.
- [38] J. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear i/o. In *European Symposium on Algorithms*, pages 723–735. Springer, 2002.
- [39] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 345–354, 1987.
- [40] K. Munagala and A. Ranade. I/o-complexity of graph algorithms. In *SODA*, volume 99, pages 687–694. Citeseer, 1999.
- [41] J. Nešetřil and S. Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985.
- [42] A. Y. Ng et al. On spectral clustering: Analysis and an algorithm. In *NIPS*, pages 849–856, 2002.
- [43] R. Pagh and M. Stöckel. The input/output complexity of sparse matrix multiplication. In A. S. Schulz and D. Wagner, editors, *Algorithms - ESA 2014*, pages 750–761, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- [44] M. O. Rabin and V. V. Vazirani. Maximum matchings in general graphs through randomization. *Journal of Algorithms*, 10(4):557–567, 1989.
- [45] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 679–688. IEEE, 2002.
- [46] D. Rose. *Sparse Matrices and their Applications: Proceedings of a Symposium on Sparse Matrices and Their Applications*. Springer Science & Business Media, 2012.
- [47] S. Sakr et al. The future is big graphs: a community view on graph processing systems. *Communications of the ACM*, 64(9):62–71, 2021.
- [48] J. N. Scott. *An I/O-Complexity Lower Bound for All Recursive Matrix Multiplication Algorithms by Path-Routing*. PhD thesis, UC Berkeley, 2015.
- [49] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, 1995.
- [50] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 605–614. IEEE, 1999.
- [51] E. Solomonik et al. Scaling Betweenness Centrality using Communication-Efficient Sparse Matrix Multiplication. In *SC*, 2017.
- [52] D. Unat et al. Trends in Data Locality Abstractions for HPC Systems. *IEEE TPDS*, 28(10), Oct. 2017.

- [53] R. Yuster and U. Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 254–260. Society for Industrial and Applied Mathematics, 2004.
- [54] Z. Zlatev. *Computational methods for general sparse matrices*, volume 65. Springer Science & Business Media, 1991.
- [55] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM (JACM)*, 49(3):289–317, 2002.