Optimal Sparse Matrix Dense Vector Multiplication in the I/O-Model

Michael A. Bender*
Department of Computer Science,
Stony Brook University,
Stony Brook, NY 11794-4400, USA.
bender@cs.sunysb.edu

Gerth Stølting Brodal†
BRICS, Basic Research in
Computer Science,
University of Aarhus,
Aarhus, Denmark.
gerth@daimi.au.dk

Rolf Fagerberg‡
Department of Mathematics
and Computer Science,
University of Southern Denmark,
Odense M, Denmark.
rolf@imada.sdu.dk

Riko Jacob
ETH Zurich,
Institute of Theoretical
Computer Science,
8092 Zurich, Switzerland.
rjacob@inf.ethz.ch

Elias Vicari
ETH Zurich,
Institute of Theoretical
Computer Science,
8092 Zurich, Switzerland.
vicariel@inf.ethz.ch

ABSTRACT

We analyze the problem of sparse-matrix dense-vector multiplication (SpMV) in the I/O-model. The task of SpMV is to compute $y := Ax$, where A is a sparse $N \times N$ matrix and x and y are vectors. Here, sparsity is expressed by the parameter k that states that A has a total of at most kN nonzeros, i.e., an average number of k nonzeros per column. The extreme choices for parameter k are well studied special cases, namely for $k = 1$ permuting and for $k = N$ dense matrix-vector multiplication.

We study the worst-case complexity of this computational task, i.e., what is the best possible upper bound on the number of I/Os depending on k and N only. We determine this complexity up to a constant factor for large ranges of the parameters. By our arguments, we find that most matrices with kN nonzeros require this number of I/Os, even if the program may depend on the structure of the matrix. The model of computation for the lower bound is a combination of the I/O-models of Aggarwal and Vitter, and of Hong and Kung.

We study two variants of the problem, depending on the memory layout of A.
If A is stored in column major layout, SpMV has I/O complexity $\Theta\Big(\min\Big\{\frac{kN}{B}\Big(1+\log_{M/B}\frac{N}{\max\{M,k\}}\Big),\, kN\Big\}\Big)$ for $k \leq N^{1-\varepsilon}$ and any constant $1 > \varepsilon > 0$. If the algorithm can choose the memory layout, the I/O complexity of SpMV is $\Theta\Big(\min\Big\{\frac{kN}{B}\Big(1+\log_{M/B}\frac{N}{kM}\Big), kN\Big\}\Big)$ for $k \leq \sqrt[3]{N}$.

In the cache oblivious setting with tall cache assumption $M \geq B^{1+\varepsilon}$, the I/O complexity is $\mathcal{O}\left(\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{k}\right)\right)$ for A in column major layout.

Categories and Subject Descriptors: F.2 [Theory of Computation]: Analysis of Algorithms and Problem Complexity

General Terms: Algorithms, Theory

Keywords: I/O-Model, External Memory Algorithms, Lower Bound, Sparse Matrix Dense Vector Multiplication

1. INTRODUCTION

Sparse-matrix dense-vector multiplication (SpMV) is one of the core operations in the computational sciences. The task of SpMV is to compute $y := Ax$, where A is a *sparse matrix* (most of its entries are zero) and x and y are vectors. Applications abound in scientific computing, computer science, and engineering, including iterative linear-system solvers, least-squares problems, eigenvalue problems, data mining, and web search (e.g., computing page rank). In these and other applications, the same sparse matrix is used repeatedly; only the vectors x and y change.

From a traditional algorithmic point of view (e.g., the RAM model), the problem is easily solved with a number of operations proportional to the number of entries in the matrix, which is optimal. In contrast, empirical studies show that this naive algorithm does not use the hardware efficiently, for example [18] reports that CPU-utilization is typically as low as 10% for this algorithm. The explanation for this observation lies in the memory system of a modern computer, where an access to a data item happens without delay if the item is stored in the fast processor cache, whereas it takes a significant amount of time if the item needs to be fetched from memory, or even worse if the item resides on

*Supported in part by NSF grants CCF 0621439/0621425, CCF 0540897/05414009, CCF 0634793/0632838, and CNS 0627645.
†Partially supported by the Danish Research Agency.
‡Supported in part by the Danish Natural Science Research Council (SNF).

SPAA'07, June 9–11, 2007, San Diego, California, USA.

disk. Today the trend in hardware development is that cache has more levels, and that the speed of CPU and innermost caches increase faster than the speed of the slower memory, in particular the disk. Hence, the goal of taking algorithmic advantage of the memory hierarchy will also remain important for hardware in the foreseeable future.

Previous theoretical considerations.

The memory hierarchy of a computer is usually modeled by the **disk access machine (DAM)** [1] and **cache-oblivious (CO)** [7] models. The **disk access machine** model is a two-level abstraction of a memory hierarchy, modeling either cache and main memory or main memory and disk. The small memory level has limited size M, the large level is unbounded, and the block-transfer size is B. The objective is to minimize the number of block transfers between the two levels. The **cache-oblivious** model enables one to reason about a two-level model but to prove results about an unknown multilevel memory hierarchy. The CO model is essentially the DAM model, except that the block size B and main memory size M are unknown to the coder or algorithm designer. The main idea of the CO model is that if it can be shown that some algorithm performs well in the DAM model without being parameterized by B and M, then the algorithm also performs well on any unknown, multilevel memory hierarchy [7].

These models were successfully used to analyze the I/O complexity of permuting [1], which can easily be seen as a special case of sparse matrix-vector multiplication, namely where the matrix is a permutation matrix. There are known classes of permutation matrices that can be computed more efficiently [4]. In its classical formulation, the DAM model assumes that certain data items are atomic and can only be moved, copied or destroyed, but not split or analyzed further. Hence, the DAM model does not directly allow to consider an algebraic task as we consider here. Such tasks have been successfully analyzed by means of a red-blue pebble game [11], which captures the existence of two levels of memory, but does not assume that I/O operations consist of blocks, i.e., it assumes $B = 1$. There are other modification of the DAM model known that are geared towards computational geometry problems [2].

Previous practical considerations.

In many applications where sparse matrices arise, these matrices have a certain well-understood structure. Exploiting such structure to define a good memory layout of the matrix has been done successfully; examples of techniques applicable in several settings include "register blocking" and "cache blocking," which are designed to optimize register and cache use, respectively. See e.g., [5, 18] for excellent surveys of the dozens of papers on this topic; sparse matrix libraries include [6, 10, 13, 14, 17, 18]. In these papers, the metric is the running time on test instances and current hardware. This is in contrast to our considerations, where the block size B and the memory size M are parameters and the focus is on asymptotic performance.

Our Results.

In this paper, we consider an $N \times N$ matrix A and describe its sparsity solely by the number of nonzeros. More precisely, our sparsity parameter k states that A has a total of at most kN nonzeros, i.e., an average number of k nonzeros per column. For this parameter, we consider algorithms that work for all matrices with at most kN nonzeros, and we are interested in the worst-case I/O performance of such an algorithm. The bound on the number of I/Os performed by the algorithm may only depend on the dimension N of the matrix, the sparsity parameter k, and the parameters M and B of the DAM model. Here, we extend the DAM model to allow algebraic operations in a fairly controlled fashion.

In our discussion we assume that a sparse matrix is stored as a list of triples (i, j, x) describing that at position (row i, column j) the value of the nonzero entry is x. The order of this list corresponds to the layout of the matrix in memory. **Column major layout** means that the triples are sorted lexicographically according to j (primarily) and i (to break ties).

The precise contributions are as follows:

- We give a precise model of computation that combines the models of [1] and [11].
- We give an upper bound parameterized by k on the cost for the SpMV when the (nonzero) elements of the matrices are stored in column major layout. Specifically, the cost for the SpMV is
$$\mathcal{O}\left(\min\left\{\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{\max\{M,k\}}\right),\ kN\right\}\right).$$
This bound generalizes the permutation bound, where the first term describes a generalization of sorting by destination, and the second term describes moving each element directly to its final destination.
- We also give an upper bound parameterized by k on the cost for the SpMV when the (nonzero) elements of the matrices can be stored in arbitrary order. The cost for the SpMV now reduces to
$$\mathcal{O}\left(\min\left\{\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{kM}\right),\ kN\right\}\right).$$
- We give a lower bound parameterized by k on the cost for the SpMV when the nonzero elements of the matrices are stored in column major order of $\Omega\Big(\min\left\{\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{\max\{k,M\}}\right),\ kN\right\}\Big)$ I/Os. This result applies for $k \le N^{1-\varepsilon}$, for all constant $1 > \varepsilon > 0$ (and the trivial conditions that $B > 2$ and $M \ge 4B$). This shows that our algorithm is optimal up to a constant factor.
- We conclude with a lower bound parameterized by k on the cost for the SpMV when the nonzero elements of the matrices can be stored in any order, and, for $k > 5$, even if the layout of the input and output vector can be chosen by the algorithm. Then $\Omega\Big(\min\left\{\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{kM}\right),\ kN\right\}\Big)$ I/Os are required. This result applies for $k \le \sqrt[3]{N}$ (and the trivial conditions that $B > 6$ and $M \ge 3B$) and shows that our corresponding algorithm is optimal up to a constant factor.
- In the cache oblivious setting with the tall cache assumption $M \ge B^{1+\varepsilon}$, the I/O complexity of SpMV is $\mathcal{O}\left(\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{k}\right)\right)$ for column major layout.

In this the lower bound takes most of the effort and relies upon the following counting argument. We consider on one hand data-flow descriptions that can arise from ℓ I/O operations. On the other hand, we determine how many matrices (of a specific family) can be dealt with by the same data flow. Comparing the numbers (including the size of the family), we can conclude that a certain number of I/Os are necessary to handle all matrices of the family, or in other words that some matrices in the family cannot be handled if the number of I/Os is too small.

As a side result, our arguments show that a uniformly chosen random sparse matrix will almost surely require half the I/Os claimed by the worst-case lower bounds. This is due to the fact that the number of programs grows exponentially with the number of I/O operations.

Limits of the Results, Extensions, and Future Work.

Ideally, what we would like to have is some kind of compiler that takes the matrix A, analyzes it, and then efficiently produces a program and a good layout of the matrix, such that the multiplication Ax with any input vector x is as quick as possible. To complete the wish list, this compiler should produce the program more quickly than the program takes to run. Today, this goal seems very ambitious, both from a practical and theoretical viewpoint. Our investigations are a theoretical step toward this goal.

In this process, we work with a sparsity parameter that merely counts the number of nonzeros. This is not the perfect parameter. It is easy to specify matrices of arbitrary sparsity that can be multiplied with in scanning time. Nevertheless it is a natural parameter because kN is also the number of multiplications and the input size. Additionally, it is a good parameter to express the I/O complexity of the task because it allows to show the existence of many sparse matrices for which we now know the data flow I/O complexity up to a constant factor.

Another interesting question that remains open is to pinpoint which matrices require many I/Os. This problem is not even solved for permutation matrices (i.e., the problem of permuting) because the counting argument only shows that many difficult matrices exist. This question is perhaps the aspect of the mentioned matrix compiler that is least understood, namely what are strong lower bounds for the necessary I/O of a concrete matrix.

Moreover, the DAM model only counts memory transfers rather than computation. Thus, the algorithm with the best data locality may not be the algorithm with the fastest running times.

Finally, our lower bounds rely upon fairly strict algebraic assumptions. These assumptions are standard, it seems natural to separate the algebraic aspects from the data flow aspects if this allows to show non-trivial lower bounds. Still, it would be nice to relax these assumptions. We conjecture that even if an algorithm is allowed to exploit the full structure of the real numbers, this will not increase its efficiency.

Map.

This paper is organized as follows: In Section 2 we describe the computational model in which our lower bounds hold. In Section 3 we present upper bounds on the SpMV. We prove the upper bound for column major layout, and then for free layout. We conclude with a description of an upper bound in the cache-oblivious model. Section 4 presents our lower bound for column major layouts. Section 5 presents our lower bound for free layouts.

We use the established $\ell = \mathcal{O}(f(N,k,M,B))$ notation, which here means that there exists a constant $c > 0$ such that $\ell \leq c \cdot f(N,k,M,B)$ for all $N > 1, k, M, B$, unless otherwise stated. Throughout the paper log stands for the binary logarithm.

2. MODEL OF COMPUTATION

Our aim is to analyze the I/O cost of computing a matrix-vector product. I/Os are generated by movement of data, so our real object of study is the data flow of algorithms for matrix-vector multiplication, and the interaction of this data flow with the memory hierarchy. Our modeling is a combination of the two models that were used to analyze the special cases of permuting [1] and matrix-matrix multiplication [11]. We assume, as is standard, a two-level memory hierarchy of Aggarwal and Vitter with limited main memory and data transfer organized in blocks [1]. Furthermore, we restrict the algorithm to the classical modeling of [11], and assume what they call "independent evaluation of multivariate expressions", as explained in detail below.

Our model is based on the notion of a **commutative semiring** $\mathbb{S}$, i.e., a set of numbers with addition and multiplication, where operations are assumed to be associative and commutative, and are distributive. There is a neutral element 0 for addition, 1 for multiplication, and multiplication with 0 yields 0. In contrast to a field, there are no inverse elements guaranteed, neither for addition nor for multiplication.

Definitions.

The **semiring I/O-machine** has an infinite size **disk** D, organized in **tracks** of B numbers each, and main **memory** containing M numbers from a semiring $\mathbb{S}$. Accordingly, a **configuration** can be described by a vector of M numbers $\mathcal{M} = (m_1, \ldots, m_M)$, and an infinite sequence D of tracks modeled by vectors $\mathbf{t}_i \in \mathbb{S}^B$. A step of the computation leads to a new configuration according to the following allowed **operations**:

- **Computation** on numbers in main memory: algebraic evaluation $m_i := m_j + m_k$, $m_i := m_j \times m_k$, setting $m_i := 0$, setting $m_i := 1$, and assigning $m_i := m_j$.
- **Input operations**, each of which moves an arbitrary track of the disk into the first B cells of memory, $(m_1, \ldots, m_B) := \mathbf{t}_i$, $\mathbf{t}_i := \mathbf{0}$.
- **Output operations**, each of which copies the first B cells of memory to a previously empty ($\mathbf{t}_i = \mathbf{0}$) arbitrary track, and sets $\mathbf{t}_i := (m_1, \ldots, m_B)$.

Input and output operations are collectively called I/O operations.

A program is a finite sequence of operations allowed in the model, and an algorithm is a family of programs. For the sparse matrix-vector multiplication, we allow the algorithm to choose the program based on N and the **conformation of the matrix**, i.e., the positions of the nonzeros. More precisely, we say that an algorithm solves the sparse matrix-vector multiplication problem with $\ell(k,N)$ I/Os if for all dimensions N, all k, and all conformations of $N \times N$ matrices

with kN non-zero coefficients, the program chosen by the algorithm performs at most $\ell(k, N)$ I/Os.

Note that this model is non-uniform, such that an algorithm always knows what an intermediate result stands for, for example which row and column a coefficient of the matrix belongs to. In particular, this reduces an input-triple (i, j, x) to a single number, whereas the indices i and j are implicit and known to the algorithm and reflected in the program. Because the underlying algebraic structure is an arbitrary semiring, by Lemma A.1, we may assume that all intermediate values are a "subset" of the final results and can be classified as follows: every intermediate results is what we call a **canonical partial results**, i.e., of the form x_j, a_{ij}, $a_{ij}x_j$, or $s_i = \sum_{j \in S} a_{ij}x_j$, where $1 \leq i, j \leq N$, $S \subseteq R_i \subseteq \{1, \ldots, N\}$, and R_i is the conformation of row i of the matrix. The various forms above are called **input variable**, **coefficient**, **elementary product**, and **partial sum**, respectively, with i being the **row** of the partial sum. If $S = R_i$, we simply denote the partial result a **result**. Hence, every intermediate result is either part of one column or part of one row. Our algebraic assumption that an algorithm must work for any semiring imply that all useful intermediate results are canonical partial results, see the Appendix A for details. This coincides precisely with the notion of "independent evaluation of a multivariate expressions" of [11].

One partial result p is a **predecessor** of another p' if it is an input to the addition or multiplication leading to p', and is said to be **used** for calculating p' if p is related to p' through the closure of the predecessor relation.

For the algorithm to be fully specified, we still need to decide on the layout in memory of the input matrix and vector, and the output vector. One option is to consider fixed layouts (such as column major layout for the matrix and sequential layout for the vectors). Another is to allow the algorithm to use a different layout for each program. We consider both possibilities in the rest of the paper.

Discussion of the model.

This model is natural because all efficient algorithms for matrix-vector multiplication known to us work in it. As do many natural algorithms for related problems, in particular matrix multiplication, but not all—the most notable exception is the algorithm for general dense matrix multiplication by Strassen. It is unclear if ideas similar to Strassen's can be useful for the matrix-vector multiplication problem.

Models similar to the semiring I/O-machine above have been considered before, mainly in the context of matrix-matrix multiplication, in [16], implicitly in [15], and also in [11]. Another backing for the model is that all known efficient circuits to compute multilinear results (as we do here) use only multilinear intermediate results, as described in [12].

We note that the model is non-uniform, not only allowing an arbitrary dependence on the size of the input, but even on the conformation of the input matrix. In this, there is similarity to algebraic circuits, comparison trees for sorting, and lower bounds in the standard I/O-model of [1]. In contrast, the algorithms are uniform, only relying on the comparison of indices.

One natural idea is to extend the algorithm to compare numbers, along the lines of comparison trees. In the plain semiring model, this would be binary equality tests. As it turns out, this does not allow for more efficient algorithms, see Appendix A for details.

3. ALGORITHMS

A number of standard algorithms for sparse-matrix dense-vector multiplication exist. One is the naive algorithm, which for increasing i computes the sums $c_i = \sum_{j \in R_i} a_{ij}x_j$ by summing for increasing j. If the matrix is stored in row major layout, this takes $\mathcal{O}(kN)$ I/Os, just for accessing the variables, if it is stored in column major for accessing the coefficients. A blocked version (see [9]) uses a blocked row major mode—think of the matrix as N/M matrices of size $N \times M$, store each of these in row major mode, and calculate the matrix-vector product as the pointwise sum of the N/M products of these matrices with the appropriate size M segments of the vector. This amounts to $\mathcal{O}\left(\min\{kN, N^2/(BM)\} + kN/B\right)$ I/Os. A different approach that works independently of the matrix layout is to create all products $a_{ij}x_j$ by sorting the coefficients and variables on j, and then create the sums by sorting the products on i. Using the sorting algorithm of [1], this takes $\mathcal{O}\left((kN/B)\left(1 + \log_{M/B}(kN/M)\right)\right)$ I/Os.

In this section, we propose improved algorithms based on the sorting approach. Essentially, the resulting improvement in I/O cost consists of exchanging Nk with N/k inside the logarithm in the last bound above. In particular, the new algorithm asymptotically beats the above algorithm for $k = \omega(1)$, has the same performance for $k = 1$, meets the dense matrix multiplication algorithm for $k = N$, and provides a smooth transition between these two cases. Furthermore, in the remaining sections we prove that the algorithms are optimal for large fractions of the parameter space. In the following, we assume $k \leq N$, and $M \geq 3B$.

3.1 Column Major Layout

For matrices stored in column major layout, any algorithm computing the product of the all-ones vector with a sparse matrix can be used to compute a matrix-vector product with the same matrix: perform an initial simultaneous scan of the coefficients and the input variables, where the coefficients are replaced by the elementary products $a_{ij}x_j$, and then apply the algorithm. This does not change the asymptotic performance of the algorithm.

The task of computing the product of the all-ones vector with a matrix can be understood as having to sort the coefficients according to rows (after which the result vector can be computed by scanning the matrix once more). The idea to gain efficiency over plain sorting is to immediately perform additions as soon as two partial sums for the same output value meet, i.e., reside in memory simultaneously during the sorting. The resulting gradual decrease in data size during the sorting gives the speedup.

In more detail: If $k < M$, we start by sorting the coefficients (elementary products) in chunks of M consecutive elements in internal memory, leading to $kN/M < N$ many runs. Otherwise, the input itself constitutes N runs of average length k. These runs are merged in a bottom-up fashion as in the M/B-multiway merge sort—performing immediate additions of partial results meeting during the merging—until there are at most k runs. This marks the end of phase one, which costs at most $\mathcal{O}\left(\frac{kN}{B}\log_{M/B} r/k\right)$ I/Os, where r is the initial number of runs. Due to the merging, no run can

ever become longer than N, as this is the number of output values, so at the start of phase two, we have at most k runs of length at most N. The algorithm finishes phase two by simply merging (again with immediate additions) each run into the first, at a total I/O cost of $\mathcal{O}(kN/B)$ for phase two.

For $k < M$ we get $\mathcal{O}\left(\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{M}\right)\right)$ I/Os and otherwise $\mathcal{O}\left(\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{k}\right)\right)$, hence the overall number of I/Os is

$$\mathcal{O}\left(\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{\max\{M,k\}}\right)\right).$$

3.2 Free Layout of the Matrix

For general layouts, we can get a further speed-up by using a blocked row major layout. More precisely, we store the coefficients in blocks of $M-2B$ full columns, where each such block is stored in row major layout.

In an outer loop, we load the variables of the input vector into main memory, $M-2B$ at a time. In an inner loop, we produce elementary products by loading the appropriate coefficients (accessed in scanning time due to the layout), and multiplying them with the corresponding variables. The elementary products of the same row are merged immediately by summing them up. In kN/B I/Os, this produces $r = N/(M-2B)$ runs. Then, we continue as in the previous algorithm. By the analysis of Section 3.1, the overall number of I/Os of this algorithm is

$$\mathcal{O}\left(\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{kM}\right)\right).$$

3.3 Cache Oblivious Algorithm

We can execute the algorithm of Section 3.1 in a cache-oblivious setting (under the tall cache assumption $M \geq B^{1+\varepsilon}$, as is needed for optimal sorting in this model). Recall that the algorithm uses column major layout. We first do all multiplications of coefficients and variables in a single scan, as in Section 3.1. We then group the columns into k groups of N/k columns each. In memory, each group will form a file consisting of N/k sorted runs, which by the cache-oblivious adaptive sorting algorithm of [3] can be sorted using $\mathcal{O}\left(n/B\log_{M/B}N/k\right)$ I/Os, where n is the number of coefficients in the group. Summed over all groups we get $\mathcal{O}\left(kN/B\log_{M/B}N/k\right)$ I/Os for this. The first phase ends by compressing each group to size at most N by doing additions during a scan of each group. The second phase proceeds exactly as in Section 3.1. The total I/O cost is $\mathcal{O}\left(kN/B(1+\log_{M/B}N/k)\right)$, which under a tall cache assumption is asymptotically equal to the bounds in Section 3.1 and 3.2.

4. AN EASY LOWER BOUND: COLUMN MAJOR LAYOUT

All the algorithms of Section 3 are asymptotically optimal for wide ranges of the parameters. This section and Section 5 are devoted to the proofs of these lower bounds. In this section, we consider the somewhat easier case that matrices are stored in column major layout, the natural layout to compute the elementary products. Here, it is sufficient to consider only the k-regular sparse matrices, i.e., matrices with precisely k nonzeros per column.

As described later in more detail, we here use the assumptions about the allowed intermediate results to describe the numbers constituting the configuration of the machine by an index, either standing for a column (position in the input vector) or a row (position in the output vector). Further, we abstract from the order in which numbers are stored in main memory and on disk. To describe the content of the main memory of the machine, we only record the subset $\mathcal{M} \subseteq [N]$, $|\mathcal{M}| \leq M$ of variables currently in memory. This is an important difference to the vector of numbers that describe the main memory in the general model. Similarly, the tracks of the disk are described by the list D as subsets $\mathcal{T}_i \subseteq [N]$, $|\mathcal{T}_i| \leq B$ (for $i \geq 1$). As we will see, the same formalism can describe all intermediate results.

Additionally, we assume that the program complies with the following rules: In the initial and final configuration, unused tracks of the disk and the memory are empty. Recall that read operations leave the track of the disk empty, and write operations require the track of the disk to be empty. Tracks are used consecutively, such that no track address is bigger than the total number of I/Os performed. Any program that correctly computes matrix-vector products (or achieves one of the studied tasks) can easily be transformed to comply with these rules, changing the number of I/Os only by a constant factor.

4.1 Producing k-Regular Conformations

We start with a simple computational task, whose importance will become clear later in this section.

The precise job is to transform an initial list of atomic numbers $y_1, \ldots, y_N$ into a list of kN numbers organized in N blocks. This output is understood as the representation of a k-regular matrix by assuming that every block contains copies of different variables that are sorted according to the index of the variables. With this representation, the i-th block describes the i-th column of a k-regular $N \times N$-matrix. The algorithm should be able to produce *any* k-regular matrix conformation.

In this setting, the model of computation allows variables only to be copied, moved, or destroyed. Hence the abstraction to use the index of the variable to describe a configuration of the machine is certainly justified here.

Suppose that an algorithm is given and look at its execution on an instance. By analyzing the changes of the configurations of memory and tracks on the disk, one can characterize the I/O data-flow and, thus, the program execution uniquely.

Imagine to revert the configuration sequences and to replace the *copy* operation by a *sum* operation. Along with some details discussed in the next section, we get an algorithm that computes the row-sums of a matrix initially

$$(1,2,3) \rightsquigarrow ([1,3],[1,2],[2,3]) \simeq \begin{pmatrix} y_1 & y_1 & \\ & y_2 & y_2 \\ y_3 & & y_3 \end{pmatrix}$$

Figure 1: The representation for an example of producing a matrix in column major layout ($N = 3$ and $k = 2$).

stored in column major layout. Thus we say that the task of producing conformations is *time dual* to the task of computing the row-sums of a matrix in column major layout.

By comparing the number of conformations that can be created by programs with ℓ I/Os with the number of possible conformations, we get a lower bound on the I/Os that some matrix requires to calculate its row-sum.

Note that in Section 5, we argue about the sequence of configurations of collecting *results* using the same time direction as discussed here. Hence, the time duality gains even more interest.

4.2 Column Major Layout

The argument of Section 4.1 and time duality directly imply a lower bound that we will calculate in this section. Still, we describe the process of computing row-sums in a time forward manner, because it is the basis for the data-flow analysis of Section 5, together with the process described in Section 4.1.

Set up.

We define a trace of the algorithm when running on the semiring I/O-machine. The values are replaced by row-indices describing the row of the matrix a value stems from and, by this, the index of the variable in the result vector it is a predecessor of. This representation assumes in particular, that the program immediately merges in memory results with the same index by summing them up. This assumption can always be achieved without inducing additional I/Os.

Furthermore, we work with **stubs** of partial results, i.e., zeros that are a predecessor of a particular output value. At first sight, this is perhaps a strange concept, but it is actually quite natural if we want to analyze time-backward. Then, having stubs is dual to the policy of destroying elements only when it is necessary because the memory locations are needed again. Here, as soon as a memory location becomes available, a stub for some other output variable is created. Note that creating a stub for an output variable already residing in memory would violate the policy of immediately merging.

Initial and final configurations.

The final configuration of the machine is uniquely determined, it has $\mathcal{M} = \emptyset$, and $\mathcal{T}_1 = \{1, \ldots, B\}$, $\mathcal{T}_2 = \{B + 1, \ldots, 2B\}$, $\ldots$, $\mathcal{T}_{\lceil N/B \rceil} = \{(\lceil N/B \rceil - 1)B + 1, \ldots, N\}$. In general, the initial configuration of the disk does not identify the conformation of the k-regular matrix uniquely. We denote by $\tau = \tau(N, k, B)$ the maximal number of different conformations that any initial configuration can be used for.

If $k = B$, then there is a one-to-one correspondence between the precisely k entries in a column and the content of a track. Since the column major layout dictates that the entries of the matrix are stored in order of increasing row, the set $\mathcal{T}_i$ uniquely identifies the vector $\mathbf{t}_i$. For other values of $B < k$, some tracks belong completely to a certain column. The other tracks are shared between two neighboring columns. Every element of the track can hence belong either to the left column, the right column, or both columns, i.e., there are at most 3 choices for at most kN elements. Once it is clear to which column an entry belongs to, the order within the track is prescribed. For $B > k$ we describe these choices per column of the resulting matrix. Such a column has to draw its k entries from one or two tracks of the disk, these are at most $\binom{2B}{k}$ choices. Over all columns, this results in up to $\binom{2B}{k}^N \leq (2eB/k)^{kN}$ different matrices that are handled by the same initial configuration. Summarizing we have:

$$\tau(N, k, B) \leq \begin{cases} 3^{kN}, & \text{if } B < k \\ 1, & \text{if } B = k \\ (2eB/k)^{kN}, & \text{if } B > k \end{cases}$$

Identifying the program execution.

The sequence of configurations $(\mathcal{M}, D)$ is described according to the I/Os performed, starting from the unique final configuration, "backward" in time. Accordingly, we will reconstruct the sequence starting from the unique final configuration. In any case, we specify the track of the disk that is touched by the I/O. Remember that this track is empty after reading or before writing. Consider the transition from $(\mathcal{M}, D)$ (earlier in time) to $(\mathcal{M}', D')$ (later in time). Since we want to reconstruct backwards, $(\mathcal{M}', D')$ is assumed to be known already, and we want to specify $(\mathcal{M}, D)$ succinctly. For an input operation we specify the up to B row-indices that come into memory as a subset of $\mathcal{M}'$. Knowing the track number of the input, this specifies D. For the memory we have $\mathcal{M} = \mathcal{M}'$ because we assume that at least stubs of partial sums are already there. For an output operation, the specified track must be empty, and the content of the track in D' must have been in the memory $\mathcal{M}$. To this end, we have to describe which variables of $\mathcal{M}'$ are not in $\mathcal{M}$, i.e., which stubs are created to fill positions of partial results that are moved to disk. With this encoding we can reconstruct the sequence of configurations. There are less than $\ell\binom{M+B}{B}$ choices for each step, where ℓ is a bound of the considered tracks on the disk and the term $\binom{M+B}{B}$ allows the possibility to read or write incomplete tracks.

Because the number of distinct conformations of a $N \times N$ k-regular matrix is $\binom{N}{k}^N$, we can summarize this discussion in the following Lemma:

LEMMA 4.1. *If an algorithm computes the row-sums for all k-regular $N \times N$ matrices stored in column major layout in the semiring I/O-model with at most $\ell = \ell(k, N)$ I/Os then it holds that*

$$\binom{N}{k}^N \leq \left(\binom{M+B}{B}\ell\right)^\ell \cdot \tau$$

where $\tau = \tau(N, k, B)$ is the maximal number of different matrix conformations that any initial configuration can specify.

Proof. By exploiting the aforementioned time duality, we argue in terms of the matrix producing task. $\binom{N}{k}^N$ is the number of distinct conformations of a $N \times N$ k-regular matrix. Therefore $\binom{N}{k}^N/\tau$ is a lower bound on the number of distinct final configurations. As showed above, any program execution generates at most $\binom{M+B}{B}\ell$ different configurations by every I/O. Since the algorithm is required to deal with any final conformation

$$\binom{N}{k}^N/\tau \leq \left(\binom{M+B}{B}\ell\right)^\ell$$

is a necessary condition and the claim is proved. □

Doing the math.

Algebraic manipulations of the inequality in Lemma 4.1 lead to the following Theorem. Note that for all values of N, k, M, and B the scanning bound (i.e., the cost of scanning the coefficients once) holds, that is, $\ell(k,N) \geq kN/B$.

THEOREM 4.2. *Assume that an algorithm computes the row-sums for all k-regular $N \times N$ matrices stored in column major layout in the semiring I/O-model with at most $\ell(k,N)$ I/Os. Then, for $B > 2$, $M \geq 4B$, and $k \leq N^{1-\epsilon}$, $0 < \epsilon < 1$, there is the lower bound*

$$\ell(k,N) \geq \min\left\{\kappa \cdot \frac{kN}{B}\log_{M/B}\frac{N}{\max\{k,M\}}\,,\ \frac{1}{8}\cdot\frac{\epsilon}{2-\epsilon}kN\right\}$$

for $\kappa = \min\left\{\frac{\epsilon}{3}, \frac{(1-\epsilon)^2}{2}, \frac{1}{16}\right\}$.

Comparing this lower bound to the number of I/Os of the algorithm in Section 3.1 shows that the algorithm for fixed layout is optimal up to a constant factor as long as $k \leq N^{1-\epsilon}$.

PROOF OF THEOREM 4.2: We perform some algebraic manipulation on the claim of Lemma 4.1.

Using $\ell \leq kN + \lceil N/B \rceil \leq 2kN$ and taking into account the different cases established for τ, we get the following inequalities:

For $k < B$: $\binom{N}{k}^N \leq \left(\binom{M+B}{B}2kN\right)^\ell \cdot (2eB/k)^{kN}$. Using $(x/y)^y \leq \binom{x}{y} \leq (xe/y)^y$ and taking logs we get $kN\log\frac{N}{k} \leq \ell\left(\log(2kN) + B\log\frac{e(M+B)}{B}\right) + kN\log(2eB/k)$. With $M \geq 4B$, implying $M + B < 4M/3$, and $e4M/3 < 4M$ we get

$$\ell \geq kN\frac{\log\frac{N}{2eB}}{\log(2kN) + B\log\frac{4M}{B}}\,. \tag{1}$$

For $k \geq B$: $\binom{N}{k}^N \leq \left(\binom{M+B}{B}2kN\right)^\ell \cdot 3^{kN}$. Taking logs: $kN\log\frac{N}{k} \leq \ell\left(\log(2kN) + B\log\frac{e(M+B)}{B}\right) + kN\log 3$, i.e.,

$$\ell \geq kN\frac{\log\frac{N}{3k}}{\log(2kN) + B\log\frac{4M}{B}}\,. \tag{2}$$

Combining (1) and (2) we get

$$\ell \geq kN\frac{\log\frac{N}{\max\{3k,2eB\}}}{\log(2kN) + B\log\frac{4M}{B}}\,. \tag{3}$$

For small N, this bound is weaker than the scanning bound for accessing all kN entries of the matrix: If $N \leq \max\{2^{2/(1-\epsilon)^2}, 16^{1/\epsilon}, 9^{1/\epsilon}, 2^{10}\}$, then $\log_{M/B} N \leq 2/(1-\epsilon)^2$ and $\log_{M/B} N \leq 3/\epsilon$. Similarly $N \leq 16B$ gives $\log_{M/B} N \leq -1 + \log_2 16 = 3$.

Otherwise, for large N, distinguish between the dominating term in the denominator. Fix an $1 > \epsilon > 0$ and assume that $k \leq N^{1-\epsilon}$; if $\log 2kN \geq B\log(4M/B)$ we get

$$\ell \geq kN\frac{\log\frac{N}{\max\{3k,2eB\}}}{2\log(2kN)}$$

Now, Lemma B.3 gives $B \leq 3N^{1-\epsilon}/2e$, and using $k \leq N^{1-\epsilon}$, $N > 3^{2/\epsilon}$ we get $\ell \geq kN\frac{\log\frac{N}{3N^{1-\epsilon}}}{2\log(2kN)} \geq kN\frac{\epsilon\log N - \log 3}{2(2-\epsilon)\log N + 2} \geq kN\frac{\epsilon/2\log N}{2\cdot 2(2-\epsilon)\log N} \geq \frac{\epsilon kN}{16-8\epsilon}$.

Otherwise ($\log 2kN < B\log(4M/B)$), by using $\log M/B \geq 2$, we get $\ell \geq kN\frac{\log\frac{N}{\max\{3k,2eB\}}}{2B\log\frac{4M}{B}} \geq \frac{kN}{2B}\cdot\frac{\log\frac{N}{2e\max\{k,B\}}}{\log\frac{M}{B}+2} \geq \frac{kN}{2B}\cdot\frac{-3+\log\frac{N}{\max\{k,B\}}}{2\log\frac{M}{B}}$. Now, assume further $N \geq 16B$, which, together with $N/k \geq N^\epsilon \geq 16$ for $N \geq 16^{1/\epsilon}$, implies $\log\frac{N}{\max\{k,B\}} > 4$. Hence,

$$\ell \geq \frac{kN}{2B}\cdot\frac{\log\frac{N}{\max\{k,B\}}}{8\log\frac{M}{B}} \geq \frac{kN}{16B}\cdot\log_{\frac{M}{B}}\frac{N}{\max\{k,M\}}\,.$$

Here, we estimate $M > B$ which is certainly weakening the lower bound, but since this estimation is inside a $\log_{M/B}$ it merely loses an additive constant. □

5. LOWER BOUND: FREE LAYOUT - THE SEPARATED MODEL

In this section, we consider algorithms that are free to choose the layout of the matrix on the disk; we refer to this case as the free matrix layout. In this setting, computing row-sums can be done by one single scan if the matrix is stored in row major layout. Hence, the input vector becomes an important part of the input, whereas the coefficients of the matrix are no longer important. Accordingly, in our argument, we do not even count the accesses to coefficients as I/Os. As already hinted at in the introduction, we trace both the movements of variables and the movements of canonical partial sums while the algorithm is executing.

Set up.

At this point, we introduce a slight structural modification of the semiring I/O-machine that formalizes this classification of intermediate results without changing the I/O-behavior. Each configuration in the separated model encapsulates the sequence of internal operations between two I/Os. In particular, it abstracts away the order of multiplications and summations, and thus focuses on the resulting movement of partial results and variables.

The input variable memory $\mathcal{M}_V \subseteq [N]$, $|\mathcal{M}_V| \leq M$ is the set of variables in memory right after the preceding I/O, before the internal computation at hand starts. The result memory $\mathcal{M}_R \subseteq [N]$, $|\mathcal{M}_R| \leq M$ gives the row index of the matrix (or the result vector) for which some partial sum is in memory after the internal computation, before the succeeding I/O.

We also separate the disk into D_V containing copies of variables, and D_R containing partial results. Each track of such a disk contains at most B items.

The multiplication step $\mathcal{P} \subseteq \mathcal{M}_R \times \mathcal{M}_V$ represents which coefficients of the matrix A are used in this internal computation. The sequence of multiplication steps is called the **multiplication trace**. We have $(i,j) \in \mathcal{P}$ if the partial result for row i contains the term $a_{ij}x_j$ after the internal computation, but not before. Note that this can only be the case if x_j is in the variable memory at the start of this sequence of internal computations. Additionally, we can assume that every coefficient a_{ij} is only used once, and that this use is as early as possible, i.e., at the earliest memory configuration where a copy of variable j and a partial sum of row i are simultaneously in memory.

Identification of the conformation.

By the assumption that all intermediate values are a predecessor of some output value, the multiplication trace specifies the conformation of the matrix uniquely. For this reason we do not need to argue about the parameter τ as in

Section 4.2. This proves the following Lemma.

LEMMA 5.1. *The conformation of the computed matrix is determined uniquely by the sequence of configurations in the separated model.*

Identifying the program execution.

For a computation with ℓ I/O operations, consider the sequences (over time) of $\mathcal{M}_V, D_V, \mathcal{P}$, and $\mathcal{M}_R, D_R$ separately. Observe that for the purpose of a lower bound, there is no harm in overestimating $\mathcal{M}_V$ or $\mathcal{M}_R$ with a superset, just as we could assume this for the variables y_i in the example setting of Section 4.1. Hence, we assume that variables are only deleted when this is necessary to provide space for other variables to be loaded into $\mathcal{M}_V$. That is, as long as there are less than $M-B$ variables in memory, no variable is deleted. Then, only at the moments when new variables are loaded into memory, variables previously residing in memory are deleted. This is the only occasion where we trace numbers that are no longer used in the computation. This is dual to the concept of a stub. It is useful to separate the time of using a variable from its movement in memory and on disk. It is not important how we decide upon which variables to keep, but for the sake of concreteness we can, for example, deterministically always delete the variable that is not in the memory of the original algorithm and has not been used for the most number of I/Os, breaking ties by index (variable name). Symmetrically, we can assume that $\mathcal{M}_R$ is always filled by allowing empty partial results (stubs) of a specific row. Then, immediately after writing some partial results to disk, the new stubs can be created. For concreteness, we can create the empty stub for which there is no partial result in memory, which is used next (time forward) by the algorithm. Note that the possibility to create stubs and delete variables differently only weakens the lower bound because it allows several descriptions of the same program.

Memory configurations.

Now, we can compactly describe the sequence of separated memory and disk configurations as in the previous sections. More precisely, as in Section 4.1, the $\mathcal{M}_V, D_V$ can be described by a time forward trace of input and output operations, whereas, by the argument of Section 4.2, the sequence of the $\mathcal{M}_R, D_R$ is described backward in time, starting from the unique final configuration. For all four cases, there are at most $\ell\binom{M+B}{B}$ choices for each I/O.

Multiplication trace.

It remains to specify the multiplication trace compactly. Every multiplication is specified by an index into the current subset of variables and partial results present in the respective memories. Remember that we assume that every coefficient a_{ij} is used only once when either its corresponding variable or the partial sum are freshly loaded into memory (is not present in the predecessor configuration). Because the differences in memory-configurations stem from I/Os of at most B elements, there are at most B such variables and at most B such partial sums. Now, over the run of the algorithm there are at most $2\ell B$ such elements, each can combine (get multiplied) with at most M other elements. Out of these possibilities, we have to identify a subset of precisely kN positions where actually a multiplication happens. Hence the number of (codes for) multiplication traces for the complete computation is at most $\binom{2\ell MB}{kN}$. We summarize this discussion in the following lemma:

LEMMA 5.2. *Any computation of a matrix-vector product taking ℓ I/Os that uses only canonical partial results, can be characterized by two sequences of length ℓ of $\ell\binom{M+B}{B}$ codes, and a specification of the multiplication trace by a subset of size kN out of a universe of size $2\ell BM$. This code uniquely determines the conformation of the matrix.*

It remains to compare the compact coding of Lemma 5.2 with the number of different conformations of a matrix.

LEMMA 5.3. *If an algorithm computes the matrix-vector product for all k-regular $N \times N$ matrices in the semiring I/O-model with free matrix layout with at most $\ell(k,N)$ I/Os. Then, for $M \geq 3B$ and $B \geq 2$, it holds:*

$$\binom{N}{k}^N \leq \left(\binom{M+B}{B}\ell\right)^{2\ell} \cdot \binom{2\ell BM}{kN}.$$

Now, we can conclude the following theorem that shows that the algorithm presented in Section 3 is optimal up to a constant factor for large ranges of the parameters. In the proof we use the upper bound $\ell \leq 2kN$ ($B \geq 2, M \geq 3B$): The entries of the matrix are read in blocks of size B, for each entry the corresponding variable is read and the product is added to the current partial result. As soon as B results are ready, they are output with one I/O and the memory is freed for the next B results.

THEOREM 5.4. *Assume that an algorithm computes the matrix-vector product for all k-regular $N \times N$ matrices in the semiring I/O-model with free matrix layout with at most $\ell(k,N)$ I/Os. Then for $M \geq 3B$, $B \geq 6$, and $k \leq \sqrt[3]{N}$ there is the lower bound*

$$\ell(k,N) \geq \min\left\{\frac{1}{84}kN, \frac{1}{24}\cdot\frac{kN}{B}\log_{\frac{M}{B}}\frac{N}{kM}\right\}$$

PROOF OF THEOREM 5.4: Consider the inequality claimed in Lemma 5.3. Because of the naive algorithm we estimate $\ell \leq 2kN$ such that we get: $\binom{N}{k}^N \leq \left(\binom{M+B}{B}2kN\right)^{2\ell}\cdot\binom{2\ell BM}{kN}$. Using $(x/y)^y \leq \binom{x}{y} \leq (xe/y)^y$ we arrive at $kN\log\frac{N}{k} \leq 2\ell\left(\log(2kN) + B\log\frac{e(M+B)}{B}\right) + kN\log\frac{2e\ell BM}{kN}$ by taking logarithms. Rearranging terms and using $e(M+B) \leq 4M$ yields $2\ell \geq kN\frac{\log\frac{N}{k}\frac{kN}{2e\ell BM}}{\log(2kN)+B\log\frac{4M}{B}}$.

Assume $\frac{N}{kM} > 5$ and $N \geq 2^{10}$. Otherwise the logarithmic term is inferior to the scanning bound and the theorem is true.

We take the statement of Lemma B.2 with $s := \frac{N}{keBM}$, $t := \log(2kN) + B\log\frac{4M}{B}$, and $x := 2\ell/kN$. Because $s > 5/eB$ we have $st > \frac{5}{e}\log\frac{4M}{B} > \frac{5}{e}\log 4 > 1$, such that the assumption of Lemma B.2 is satisfied. Hence, we conclude $2\ell/kN \geq \log(st)/2t$. Now, we distinguish according to the leading term of t. If $\log(2kN) < B\log\frac{4M}{B}$, then we get $2\ell/kN \geq \frac{\log(\frac{N}{keBM}B\log\frac{4M}{B})}{4B\log\frac{4M}{B}}$.

Using $\log\frac{4M}{B} \geq 3 \geq e$ (recall that $4M \geq 8B$): $\ell \geq \frac{kN}{8B}\cdot\frac{\log\frac{N}{kM}}{2+\log\frac{M}{B}}$. Using $\log\frac{M}{B} \geq 1$: $\ell \geq \frac{kN}{24B}\frac{\log\frac{N}{kM}}{\log\frac{M}{B}}$, which is the statement of the theorem.

Otherwise, if $\log(2kN) > B\log(4M/B)$, we use, $N \geq 2^{10}$, $t \geq 1$ and Lemma B.3 (iii), yielding $st \geq \frac{N}{keBM}B\log\frac{4M}{B} \geq$

$\frac{N}{keM} \geq \sqrt[9]{N}$. Now, $\frac{\log st}{\log 2kN} \geq \frac{\frac{1}{9}\log N}{\frac{4}{3}\log N+1} \geq \frac{1}{9}\cdot\frac{3}{7} = \frac{1}{21}$. Hence, $\ell \geq kN/84$. □

THEOREM 5.5. *Assume that an algorithm computes the matrix-vector product for all k-regular $N \times N$ matrices in the semiring I/O-model with free matrix layout with at most $\ell(k,N)$ I/Os. Here, the algorithm can chose the memory layout (order) of the input vector and the result vector. Then for $M \geq 3B$, $B \geq 6$, and $18 < k \leq \sqrt[3]{N}$ there is the lower bound:*

$$\ell(k,N) \geq \min\left\{\frac{1}{420}kN, \frac{1}{120}\cdot\frac{kN}{B}\log_{\frac{M}{B}}\frac{N}{kM}\right\}$$

PROOF OF THEOREM 5.5: We proceed as in the proof of Lemma 5.3 and Theorem 5.4. The input vector can be stored in an arbitrary order, generating $N!$ different initial configurations. This arguments applies also for the end configurations.

Summarizing, ℓ has to be as least so large, that

$$\binom{N}{k}^N \leq \left(\binom{M+B}{B}2kN\right)^{2\ell}\cdot\binom{2\ell BM}{kN}(N!)^2$$

is satisfied.

By Lemma B.4 we have that $\binom{N}{k}^N/(N!)^2 \geq \binom{N}{\lfloor k/5\rfloor}$. So ℓ has to fulfill: $\binom{N}{\lfloor k/5\rfloor}^N \leq \left(\binom{M+B}{B}2kN\right)^{2\ell}\cdot\binom{2\ell BM}{kN}$.

The remaining computations follows as easy as in Theorem 5.4. □

APPENDIX

A. DETAILS OF THE MODEL OF COMPUTATION

Our model is based on the notion of a **commutative semiring** $\mathbb{S}$. One well investigated example of such a semiring (actually having multiplicative inverses) is the max-plus algebra (tropical algebra), where matrix multiplication can be used to for example compute shortest paths with negative edge length. Another semiring obviously is $\mathbb{R}$ with usual addition and multiplication.

In any matrix-vector multiplication algorithm in our model, every intermediate result can be represented as a polynomial in the input numbers (the x_js of the vector and the non-zero a_{ij}s of the matrix) with natural coefficients[1] (i.e. coefficients in $\{0, 1, 1+1, 1+1+1, \dots\}$). As the algorithm must work over any commutative semiring, in particular the free commutative semiring over the input numbers where this representation is unique, the result values can only be represented as the polynomials $c_i = 1\cdot\sum_{R_i} a_{ij}x_j$, where the sum is taken over the conformation (i.e., the non-zero entries) of row i of the matrix.

LEMMA A.1. *If there is a program that multiplies a given matrix A with any vector in the semiring I/O-machine that performs ℓ I/Os, then there is also a program with ℓ I/Os that computes only canonical partial results, that does only compute partial results which are used for some output value, and that has never two partial sums of the same row in memory when an I/O is performed.*

[1]We denote these poly-coefficients, to distinguish them from the input coefficients a_{ij} of the matrix.

Proof. Note that for the sake of this proof, we can analyze and modify the given program without any uniformity constraint. First, intermediate results that are computed but are not used for a result may as well not be computed. Second, if two partial sums are in memory, they can be merged (summed) into one of them.

Multiplication by 0 can be replaced by setting the result to 0. Hence, the constant 0 is useless to the algorithm, and we can assume that intermediate results are not the 0-polynomial.

If two such polynomials f and g are multiplied or added, then the set of variables of the resulting polynomial is the union of the variables of f and g. For multiplication, the degree is equal to the sum of the (non-negative) degrees. For addition, the number of terms cannot decrease.

If an intermediate result has a poly-coefficient bigger than 1, the result cannot be used for a final result, because in all successor results there will remain a term with poly-coefficient bigger than 1. If an intermediate result contains (matrix) coefficients from different rows, it can never be used for a final result. If an intermediate result contains the product of two variables or two coefficients, this product remains in all successors and it is not useful for the final results. If an intermediate result contains two terms of total degree one (i.e., single variables or coefficients), then it needs to be multiplied with a non-constant polynomial before it can become a result. But then, there will be products of two variables or two coefficients, or a product of a variable with a coefficient of the wrong column. Hence, such a result cannot be useful for a result. □

Allowing Comparisons

Alternatively, one could assume an additional ordering, giving ternary inequality tests. A program for the machine then consists of a finite tree, with binary (ternary) comparison nodes, and all other nodes being unary. The input is given as the initial configuration of the disk and all memory cells empty. The root node of the tree is labeled with this initial configuration, and the label of a child of a unary node is given by applying the operation the node is marked with. For branching nodes, only the child corresponding to the outcome of the comparison is labeled. The final configuration is given by the only labeled leaf node. It is assumed that main memory is again all zero.

An algorithm with comparisons is allowed to branch according to the output of an equality test. The equality test is performed by checking whether a non-trivial polynomial of the data present in the memory equals zero.

LEMMA A.2. *Consider the conformation of a matrix A. If there is a program P computing the matrix-vector product $c = Ax$ for all inputs in $\mathbb{R}^d$ (for appropriate d) with at most ℓ I/Os, then there is a program Q without comparisons (equality tests) computing the same product with at most ℓ I/Os.*

Proof. Consider the tree of the program P. Since the number of leaves is finite, there must be a leaf which is reached exactly when all tests on the internal nodes have been negative; define T to be the path from the root to that leaf. By definition the set of inputs of $\mathbb{R}^d$ reaching that leaf (also following T) is an open set in the Zariski topology [8] and thus dense in the usual metric topology.

The program Q is induced by following T and deleting the comparison nodes from the path. Every result of Q is given by a polynomial q on the input and it is equal to the multilinear result p of the computation for an open set C of inputs. Thus it follows that $p = q$ on the closure of C, hence on the whole space. □

B. TECHNICAL LEMMAS

LEMMA B.1. *For every $x > 1$ and $b \geq 2$, the following inequality holds:* $\log_b 2x \geq 2 \log_b \log_b x$.

Proof. By exponentiating both sides of the inequality we get $2x \geq \log_b^2 x$. Define $g(x) := 2x - \log_b^2 x$, then, $g(1) = 2 > 0$ and $g'(x) = 2 - \frac{2}{\ln^2 b} \frac{\ln x}{x} \geq 2 - \frac{2}{\ln^2 b} \cdot \frac{1}{e} \geq 2 - \frac{2}{\ln^2 2} \cdot \frac{1}{e} > 0$ for all $x \geq 1$, since $\ln(x)/x \leq 1/e$. Thus g is always positive and the claim follows. □

LEMMA B.2. *Let $b \geq 2$ and $s, t > 0$ such that $s \cdot t > 1$. For all positive real numbers x, we have $x \geq \frac{\log_b(s/x)}{t} \Rightarrow x \geq \frac{1}{2} \frac{\log_b(s \cdot t)}{t}$*

Proof. Assume $x \geq \log_b(s/x)/t$ and, for a contradiction, also $x < 1/2 \log_b(s \cdot t)/t$. Then we get $x \geq \frac{\log_b(s/x)}{t} > \frac{\log_b \frac{2s \cdot t}{\log_b(s \cdot t)}}{t} = \frac{\log_b(2s \cdot t) - \log_b \log_b(s \cdot t)}{t} \geq \frac{\log_b(2s \cdot t) - \frac{1}{2} \log_b(2s \cdot t)}{t} = \frac{1}{2} \frac{\log_b(2s \cdot t)}{t}$, where the last inequality stems from Lemma B.1. This contradiction to the assumed upper bound on x establishes the lemma. □

LEMMA B.3. *Assume $\log kN \geq B \log(4M/B)$, $k \leq N/2$, $B \geq 2$, and $M \geq 2B$. Then*

(i) $N \geq 2^{10}$ implies $B \leq \frac{2}{3} \sqrt[3]{N}$.
(ii) $0 < \epsilon < 1$, $N \geq 2^{2/(1-\epsilon)^2}$ implies $B \leq \frac{3}{2\epsilon} N^{1-\epsilon}$.
(iii) $N \geq 2^{10}$, $B \geq 6$, and $k \leq \sqrt[3]{N}$ implies $\frac{N}{keM} \geq \sqrt[9]{N}$.

Proof. By $M \geq 2B$, we have $\log(4M/B) \geq \log 8 = 3$. Thus $B \leq \frac{\log 2kN}{\log \frac{4M}{B}} \leq \frac{2}{3} \log 1/\sqrt{2} N \leq \frac{2}{3} \sqrt[3]{N}$ for $N \geq 2^{10}$, which proves (i). Similarly follows statement (ii). To see (iii), we rewrite the main assumption as $(2kN)^{1/B} > 4M/B$. With $B \geq 6$, $k \leq \sqrt[3]{N}$, and (i) we get: $M \leq \frac{1}{4} B (2kN)^{1/B} \leq \frac{1}{6} \sqrt[3]{N} (2kN)^{1/6} \leq \frac{2^{1/6}}{6} N^{\frac{1}{3} + \frac{4}{3} \cdot \frac{1}{6}} = \frac{2^{1/6}}{6} N^{\frac{5}{9}}$.

Hence $\frac{N}{keM} \geq \frac{6N}{2^{1/6} \sqrt[3]{N} e N^{\frac{5}{9}}} \geq N^{1 - \frac{1}{3} - \frac{5}{9}} = \sqrt[9]{N}$. □

LEMMA B.4. *For any $N \geq 5$, $5 \leq k \leq \sqrt[3]{N}$ it holds $\binom{N}{k} / (N!)^2 \geq \binom{N}{\lfloor k/5 \rfloor}$.*

Proof. By applying the usual estimates for the binomial factor and the factorial function we get $N^{N(k-2) - kN/2} \cdot 2^{2N} \cdot 1/k^{kN} \geq N^{kN/5} \cdot e^{kN/5} \cdot 5^{kN/5} \cdot 1/k^{kN/5}$. By taking the logarithm, dividing by N and using that $k \leq \sqrt[3]{N}$ we get $(8/15 \log N - \log(5e)/5)k - 2 \log N + 2 \geq 0$ which is implied by $k \geq \frac{2 \log N - 2}{8/15 \log N - \log(5e)/5}$. The function on the right side never exceeds 4. Since the binomial coefficient estimates are only valid if the lower term is positive, by choosing $k \geq 5$ the claim is proved. □

C. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, September 1988.

[2] L. Arge and P. B. Miltersen. On showing lower bounds for external-memory computational geometry problems. In J. M. Abello and J. S. Vitter, editors, *External Memory Algorithms*, vol. 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 139–159. AMS Press 1999.

[3] G. S. Brodal, R. Fagerberg, and G. Moruz. Cache-aware and cache-oblivious adaptive sorting. In *Proc. 32nd International Colloquium on Automata, Languages, and Programming*, vol. 3580 of *Lecture Notes in Computer Science*, pages 576–588. Springer Verlag, Berlin, 2005.

[4] T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM J. Comput.*, 28(1):105–136, 1999.

[5] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, R. V. Antoine Petitet, R. C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proc. of the IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), February 2005.

[6] S. Filippone and M. Colajanni. PSBLAS: A library for parallel linear algebra computation on sparse matrices. *ACM Trans. on Math. Software*, 26(4):527–550, Dec. 2000.

[7] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symp. on Foundations of Computer Science (FOCS)*, pages 285–297, New York, NY, Oct. 17–19 1999.

[8] R. Hartshorne. *Algebraic Geometry*. Springer, 1977.

[9] T. Haveliwala. Efficient computation of pagerank. Technical Report 1999-31, Database Group, Computer Science Department, Stanford University, Feb. 1999. Available at http://dbpubs.stanford.edu/pub/1999-31.

[10] E. J. Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, University of California, Berkeley, May 2000.

[11] H. Jia-Wei and H. T. Kung. I/O complexity: The red-blue pebble game. In *STOC '81: Proc. 13th annual ACM symposium on Theory of computing*, pages 326–333, New York, NY, USA, 1981. ACM Press.

[12] R. Raz. Multi-linear formulas for permanent and determinant are of super-polynomial size. In *Proc. 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 633–641, Chicago, IL, USA, June 2004.

[13] K. Remington and R. Pozo. NIST sparse BLAS user's guide. Technical report, National Institute of Standards and Technology, Gaithersburg, Maryland, 1996.

[14] Y. Saad. Sparsekit: a basic tool kit for sparse matrix computations. Technical report, Computer Science Department, University of Minnesota, June 1994.

[15] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In J. M. Abello and J. S. Vitter, editors, *External Memory Algorithms*, vol. 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 161–179. AMS Press 1999.

[16] J. S. Vitter. External memory algorithms and data structures. In J. M. Abello and J. S. Vitter, editors, *External Memory Algorithms*, vol. 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–38. AMS Press 1999.

[17] R. Vudac, J. W. Demmel, and K. A. Yelick. *The Optimized Sparse Kernel Interface (OSKI) Library: User's Guide for Version 1.0.1b*. Berkeley Benchmarking and OPtimization (BeBOP) Group, March 15 2006.

[18] R. W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, Fall 2003.