Optimal sparse matrix dense vector multiplication in the I/O-model

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

ETH Technical Report 523

Michael A. Bender* Gerth Stølting Brodal† Rolf Fagerberg‡ Riko Jacob§
Elias Vicari§

June 28, 2006

Abstract

We analyze the problem of sparse-matrix dense-vector multiplication (SpMV) in the I/O model. In the SpMV, the objective is to compute $y = Ax$, where A is a sparse matrix and x and y are vectors. We give tight upper and lower bounds on the number of block transfers as a function of the sparsity k, the number of nonzeros in a column of A.

Parameter k is a knob that bridges the problems of permuting ($k = 1$) and dense matrix multiplication ($k = N$). When the nonzero elements of A are stored in column-major order, SpMV takes $\mathcal{O}\left(\min\left\{\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{\max\{M,k\}}\right), kN\right\}\right)$ memory transfers and has a lower bound of $\Omega\left(\min\left\{\kappa(\varepsilon)\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{\max\{M,k\}}\right), \kappa'(\varepsilon)kN\right\}\right)$, for $k \leq N^{\varepsilon}$, $0 < \varepsilon < 1$. If $N \leq M$ the problem is trivially $\Theta(N/B)$. Thus, these bounds are tight. When A's layout can be optimized, SpMV takes $\mathcal{O}\left(\min\left\{\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{kM}\right), kN\right\}\right)$ memory transfers and has a lower bound of $\Omega\left(\min\left\{\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{kM}\right), kN\right\}\right)$ memory transfers, for $k \leq \sqrt[3]{N}$. As before, these bounds are tight.

*Department of Computer Science, Stony Brook, NY 11794, USA. E-mail: bender@cs.sunysb.edu. Supported in part by NSF Grant CCR-0208670.

†BRICS, Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation, University of Aarhus, Arhus, Denmark. gerth@daimi.au.dk. Partially supported by the Danish Research Agency.

‡Department of Mathematics and Computer Science, University of Southern Denmark, DK-5230 Odense M, Denmark. Supported in part by the Danish Natural Science Research Council (SNF). E-mail: rolf@imada.sdu.dk.

§ETH Zurich, Institute of Theoretical Computer Science, 8092 Zurich, Switzerland. E-mail: {rjacob,vicariel}@inf.ethz.ch.

1 Introduction

Sparse-matrix dense-vector multiplication (SpMV) is one of the core operations in the computational sciences. The idea of SpMV is compute $y = Ax$, where A is a *sparse matrix* (most of its entries are zero) and x is a vector. 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 vector x changes.

It has long been known that SpMV is memory limited, typically exhibiting poor cache (and I/O) performance. According to [16], untuned code runs at below 10% of machine peak, and tuned code runs with some improved percentage of machine peak. Untuned code is likely to run even more inefficiently, as memory hierarchies grow "steeper." In contrast, dense matrix-vector multiplication (e.g., in dense linear-system solvers) does not suffer from this memory bottleneck. Because the same matrix is used repeatedly in these SpMV applications, it is worth the effort to lay out and encode the matrices to optimize performance. Examples of techniques include "register blocking" and "cache blocking," which are designed to optimize register and cache use, respectively. See e.g., [3, 16] for excellent surveys of the dozens of papers on this topic; sparse matrix libraries include [4, 7, 10, 11, 15, 16]. In these papers, the metric is the running time on test instances and current hardware.

In this paper we analyze the SpMV problem in the *disk access machine* [1] and *cache-oblivious* [5] models. Our objective is to analyze worst-case instances of matrices and to gain asymptotic insight into running times on current and future hardware. The *disk access machine (DAM)* 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 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 (CO)* model enables one to reason about a two-level model but proves 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 proved that some algorithm performs a nearly optimal number of memory transfers in the DAM model without parameterizing by B and M, then the algorithm also performs a nearly optimal number of memory transfers on any unknown, multilevel memory hierarchy.

We give upper and lower bounds on the I/O complexity of the SpMV in both the DAM and CO models. Our analyses are parameterized by the degree of sparsity of the matrix. Specifically, we let parameter k be the number of nonzeros, in each column of the N by N matrix. We show how the upper bound depends on matrix layout and we give lower bounds that apply for all layouts. Our results apply to worse-case instances. To best of our knowledge, our results represent the first upper and lower I/O bounds for this important computational problem.

One appealing aspect of the SpMV with k as the sparsity parameter is that the problem acts as a bridge between two well studied problems in the DAM model, *dense matrix multiplication* and *permuting*. When $k = \Theta(N)$, the matrix is dense. In this case, matrix multiplication requires $\Theta(N^2/B)$ memory transfers. This analysis is tight since it matches the scan bound, i.e., the cost to scan the matrix. When $k = 1$, i.e., the matrix is sparse, the SpMV requires $\Theta\left(\min\left\{(N/B)\log_{M/B}(N/B), N\right\}\right)$ memory transfers [1]. This is because when $k = 1$, SpMV is a minor generalization of external-memory permutating.

We now explain this permutation bound and its connection to the SpMV; see, e.g., [14]. Permutation matrices are a particular kind of sparse matrix for $k = 1$: the non-zeros are 1's and there is exactly one 1 in each row and column. To permute N elements, either sort by final destination, which requires $\Theta\Big((N/B)\log_{M/B}(N/B)\Big)$ memory transfers, or put each element in its final destination, which requires $O(N)$ memory transfers. The permutation cost is the minimum of these two quantities. A counting argument shows that this bound is tight, and holds for any layout of the permutation matrix and vectors. The same strategy applies for any sparse matrix with $k = 1$. For nonzeros that are not 1's, multiply the elements of the source vector before permuting. If some column or row has two nonzeros, then one element of the source vector x may have several final destinations or one element of the target vector y may be composed

of several elements of x.

Results. In this paper we give upper and lower bounds parameterized by k on the number of memory transfers to solve the SpMV. Our bounds show how the permutation bound gradually transforms into the dense matrix-multiplication bound as k increases. Specifically, we prove the following:

- 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 order. 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 above, where the first term measures a generalization of sorting by destination, and the second term measures 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 next define a model of computation to prove lower bounds on SpMV.

- 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. Thus, the cost for the SpMV is

$$\Omega\left(\min\left\{\kappa(\varepsilon)\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{\max\{k,M\}}\right),\ \kappa'(\varepsilon)kN\right\}\right).$$

  This result applies for $k \le N^{\varepsilon}$, $0<\varepsilon<1$ (and the trivial conditions that $B>2$, $M\ge 4B$). 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 $k>21$, even if the layout of the input and output vector can be chosen by the algorithm. Thus, the cost for the SpMV is

$$\Omega\left(\min\left\{\frac{kN}{B}\left(1+\log_{M/B}\frac{N}{kM}\right),\ kN\right\}\right)$$

  for the former case. This result applies for $k \le \sqrt[3]{N}$ (and the trivial conditions that $B>6$, $M\ge 3B$).

Map. This paper is organized as follows: 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. In Section 2 we describe the computational model in which our lower bounds hold. Section 4 presents our lower bound for column-major layouts. Section 5 presents our lower bound for free layouts.

We use the established $\ell = \mathcal{O}\left(f(N,k,M,B)\right)$ notation, which here means that there exists a constant $c>0$ such that $\ell \le c\cdot f(N,k,M,B)$ for all N,k,M,B, unless otherwise stated.

1.1 Roadmap of the Lower Bound

Our lower bound is the main technical contribution of this paper, and we here give a high-level outline of it.

The overall reasoning is to prove that there are many different inputs that all require different program executions, and then bound the number of essentially different program executions with at most ℓ I/Os, thereby yielding a lower bound on ℓ. For this, we first normalize and simplify the program execution. Then, we bound the number of program executions by encoding the program with a short string over a small alphabet.

In general (for upper and lower bounds), we consider $N\times N$ matrices with kN non-zero entries. For the lower bound, we in particular focus on a special case of sparse matrices, namely the **k-regular matrices**,

characterized by exactly k non-zero entries in each column. We define the **conformation** of a matrix as the locations of its non-zero entries.

We observe that in the machine model we use (defined in Section 2), we may assume that the algorithm is normalized in the sense that only "canonical" intermediate results can be produced. This again allows us to uniquely assign to every intermediate result either the one variable x_j it is representing, or the result c_i it contributes to. Using this, we then define three **traces** of the computation. These are the compact encodings describing the actions of the normalized algorithm. There is a time-forward trace of the movement of the x_j, and a time-backward trace of the movement of partial results. Additionally, based on these two traces describing the movements, there is a third trace that gives a compact representation of the algebraic operations. We show that these traces uniquely determine the normalized algorithm, and hence the conformation of the matrix it is computing.

Finally, we calculate the number of different traces of computations with ℓ I/Os, and we compare it to the number of different k-regular conformations of $N \times N$-matrices. This shows in particular that there are many sparse matrices that require many I/Os.

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 dataflow of matrix-vector product algorithms, and the interaction of this dataflow with the memory hierarchy. Hence, we need a model of computation defining the allowed forms of intermediate results and their possible movements in the memory hierarchy.

In this section, we define our model. Intuitively, it contains all algorithms which compute the values $c_i = \sum_j a_{ij} x_j$ of the output vector through multiplications and additions, starting from the coefficients and variables of the input matrix and vector. The model encompasses all proposed algorithms that we are aware of. We discuss the model further in Section 2.1.

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. 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. The semiring model is established in the context of matrix multiplication, see Section 2.1.

We now define our model, which we call the **semiring I/O-machine**. It has an infinite size **disk** D, organized in **tracks** of B numbers each, and main **memory** containing M numbers. 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 an arbitrary track, $\mathbf{t}_i := (m_1, \ldots, m_B)$ and assume $\mathbf{t}_i = \mathbf{0}$.

Input and output operations are collectively called I/O operations. Note that in a input operation we *move* a track. This may cause at most a factor two loss with respect to a *copy* operation, but it is important as it preserves a time symmetry, as it will be clear later.

A program is a finite sequence of operations allowed in the model, and an algorithm is a set 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. 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.

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.

Clearly, 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 $c_i = 1 \cdot \sum a_{ij}x_j$, where the sum is taken over the conformation (i.e., the non-zero entries) of row i of the matrix.

We below observe that all intermediate results can be assumed to be a "subset" of the final results. This will later allow us to classify every intermediate result as either part of one column or part of one row. First some nomenclature: We call intermediate results of one of the forms x_j, a_{ij}, or $s_i = \sum_{j \in S} a_{ij}x_j$, where $1 \leq i, j \leq N$, $S \subseteq R_i \subseteq \{1, \dots, N\}$, and R_i is the conformation of row i of the matrix, for **canonical partial results**. The various forms above are called **input variable**, **coefficient**, 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**. 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.

Lemma 1. *If there is a program for matrix-vector multiplication for 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.*

Proof. First, intermediate results that are computed but are not used for a result can as well not be computed. Second, if two partial sums are in memory, they can be merged 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, 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 > 1, the result cannot be used for a final result, because in all successor results there will remain a term with poly-coefficient > 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. □

2.1 Discussion of the model

As noted, all efficient algorithms for matrix-vector multiplication known to us work in the model. 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 Strassens 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 multiplication, in [13], implicitly in [12], and also in [8]. Another backing for the model is that

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

all known efficient circuits to compute multilinear results (as we do here) use only multilinear intermediate results, as described in [9].

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].

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. Or one could assume an additional ordering, giving ternary inequality tests. A program for the machine then consists of a 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. However, adding comparisons cannot really help the algorithm:

Lemma 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 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 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. □

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 $c_i = \sum_{j=1}^{N} a_{ij}x_j$ by summing for increasing j. This takes $\mathcal{O}(kN)$ I/Os. A blocked version (see [6]) 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 takes $\mathcal{O}\left(\min\{kN, N^2/(BM)\} + kN/B\right)$ I/Os. A different appoach is to create all products $a_{ij}x_j$ by sorting the input values on j, and then create the sums by sorting the products on i. This takes $\mathcal{O}\left((kN/B)\log_{M/B}(kN/B)\right)$ I/Os independently of the matrix layout.

Only the sorting based version exploits the sparsity of the matrix efficently. However, for dense matrices ($k = \theta(N)$), even the naive method is better by a logarithmic factor. This indicates that for non-sparse matrices, the sorting method can be improved.

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 particular, the algorithms beat the basic sorting based algorithm for $k = \omega(1)$, meets the algorithms above for $k = N$, and provides a smooth transition in running time between $k = 1$ and $k = N$. Furthermore, in the remaining sections we prove that the algorithms are optimal for large fractions of 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 (with one additional scan) to compute a matrix-vector product with the same matrix. To do so, an initial simultaneous scan of the coefficients and the input variables is performed,

where the coefficients are replaced by the elementary products. This scan can also be incorporated without really changing the number of I/Os when the algorithm reads the coefficients for the first time.

The task of computing the product of the all ones vector with a matrix can be understood as having to sort the coefficients (here stored in column major mode) according to rows. The idea used here to gain efficiency over plain sorting is to add partial sums used for the same output value as soon as they meet (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 input 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 with an M/B-multiway merge sort—with online additions of partial results meeting during the merging—until there are $\leq k$ runs. This marks the end of phase one, which costs at most $\mathcal{O}\left(\frac{kN}{B}\log_{M/B} r/k\right)$, 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 online 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 the blocked row major layout described above. More precisely, we store the coefficients in blocks of $M-2B$ full columns, where each such block is stored in row-major layout.

We first, in each of N/M mergings, load the next $M-2B$ variables of the input vector into main memory, and then produce the next B elements of the merged run by loading the appropriate coefficients (accessed in scanning time due to the layout), and multiplying them with the corresponding variable. This way, with kN/B I/Os, we get N/M runs of total length kN. Then, we continue as in the previous algorithm. For $M > 4B$, the number of merging steps until the (average) length of a run is N, i.e., until there are k runs, is $\mathcal{O}\left(\log_{M/B} N/(kM)\right)$.

Hence, 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 [2] 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 A Lower Bound

For all the algorithms of Section 3, there are parameter ranges in which they are optimal up to a constant factor. Because the number of I/Os of the algorithms differ by more than that, we have different lower bound proofs. All of these proofs are building on the same assumptions about the algorithms.

4.1 Producing k-Regular Conformations

We consider a computational task, which is the time-backward version of computing the row-sums of a k-regular matrix in column major layout. This time-duality is an important ingredient to our analysis. The process considered here, reflects the possibilities how the variables of the vector can be distributed by the computation.

Consider the problem where some numbers $y_1, \ldots, y_N$ are initially on disk, in this order. The algorithm produces a list of length kN of copies of these variables, in N blocks of length k. Now, the i-th block consists of k variables (of the N input variables), stored in ascending order of index. This block corresponds to the i-th column of a k-regular $N \times N$-matrix. We assume that variables can only be copied, moved, or destroyed. We normalize the program to comply with the following rules: In the final configuration, non-output tracks of the disk and the memory have an arbitrary content. Variables are only destroyed if they are overwritten by a write operation (on disk) or by a read operation (in memory). Tracks are used consecutively, such that no track address is bigger than the number of I/Os performed.

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 $i \geq 1$).

In this setting, we can ask how many different k-regular $N \times N$-matrices can be "produced" in time ℓ. We give an answer by counting the number of different descriptions of the program flow.

For an output operation, it is sufficient to state which subset of the variables in main memory are written, and to which track they are written. For an input operation, we denote the subset of variables in memory that get destroyed (overwritten), and which track is read. With this encoding of the program flow, we can recreate the sequence of configurations.

By counting the number of matrix-conformations that can result from the same final configuration in the separated model and by counting the number of different "instructions" per input or output, we can relate the running time to the number of different matrices that can be created. These calculations are done in the next section.

This process is interesting because it is time-inverse to computing row-sums of a k-regular matrix in column major layout: The produced matrix can be understood as an input matrix in column major layout, and following the data-flow backward, we arrive at a single copy of each variable, which is to be understood as the row sum. More precisely, we exchange input with output and the copying of a variable becomes the merging of two partial results. Hence, whenever we can produce a matrix with ℓ I/Os, we can also compute its row-sums with ℓ I/Os.

4.2 Column Major Layout

For the lower bound proof, we consider the special situation where the vector has all entries equal to one, and the algorithm we consider does not even need to read this vector. Accordingly, the algorithm computes the sum of the rows of a matrix stored in column major layout.

We define a trace of the algorithm when running on the algebraic I/O-machine, using only canonical partial results. First, we extract a symbolic configurations: we replace the real values by the index of the row they belong to, i.e., are a partial sum. Now, to describe the contents of the main memory of the machine, we only record the subset $\mathcal{M} \subseteq [N]$, $|\mathcal{M}| \leq M$ of partial results currently in memory. Similarly, the tracks of the disk can be described by subsets $\mathcal{T}_i \subseteq [N]$, $|\mathcal{T}_i| \leq B$. Here, we in particular allow **stubs** of partial results, i.e., zeros that are already marked with the row for which they will become partial results. 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. The possibility that we keep an element in memory without using it again corresponds to the possibility to have a stub.

The final configuration of the machine is determined uniquely, it has $\mathcal{M} = \emptyset$, and $\mathcal{T}_1 = \{1, \ldots, B\}$, $\mathcal{T}_2 = \{B+1, \ldots, 2B\}$, and so on, until $\mathcal{T}_{N/B} = \{N-B+1, \ldots, N\}$. Additionally, the initial configuration

($\mathcal{M} = \emptyset$) almost identifies which of the k-regular conformations the algorithm works on: 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 a track is given. 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 $\binom{2B}{k}^N$ different matrices that are handled by the same initial configuration. Hence, an initial configuration can be used for at most 3^{kN} conformations if $B \leq k$, and $(2eB/k)^{kN}$ otherwise.

The sequence of the $\mathcal{M}, D$ is described according to the I/Os, 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. Consider the transition from $\mathcal{M}, D$ (earlier in time) to $\mathcal{M}'_R, D'_R$ (later in time). Since we want to reconstruct backwards, $\mathcal{M}', D'$ is assumed to be known already, and we want to specify $\mathcal{M}, D$ compactly. For an input operation we specify the up to B row-indices that come into memory by a subset of $\mathcal{M}'$. With this information we can reconstruct in particular D merging back the right memory positions into the considered track. Memory itself does not change since we assume at least stubs of partial sums to be already there, that is remain there if thinking backward. For an output operation, it is sufficient to denote the change in memory, which amounts to stating which stubs got created, again a subset of $\mathcal{M}'$. With this encoding we can reconstruct the sequence of result 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 $\binom{M+B}{B}$ keeps in consideration the possibility to read incomplete tracks.

In total, using $\ell \leq kN + N \leq 2kN$, we get the following inequalities for the different cases noting that $\binom{N}{k}^N$ is the number of distinct conformations of a $N \times N$ k-regular matrix:

For $k < B$: $\binom{N}{k}^N \leq \left(\binom{M+B}{B}2kN\right)^\ell \cdot (2eB/k)^{kN}$ taking logs and using $k < B$ 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 hence $e4M/3 < 4M$ 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: $kN\log\frac{N}{k} \leq \ell\left(\log(2kN) + B\log\frac{e(M+B)}{B}\right) + kN\log 3$

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

Now assume $M \geq 4B$. Combining (1) and (2) we get

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

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

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

For N big enough, Lemma 11 gives $B \leq 3N^{1-\varepsilon}/2e$, and using $k \leq N^{1-\varepsilon}$, $N > 3^{2/\varepsilon}$ we get $\ell \geq kN\frac{\log\frac{N}{3N^{1-\varepsilon}}}{2\log(2kN)} \geq kN\frac{\varepsilon\log N - log 3}{2(2-\varepsilon)\log N + 2} \geq kN\frac{\varepsilon/2\log N}{2\cdot 2(2-\varepsilon)\log N} \geq \frac{\varepsilon kN}{16-8\varepsilon}$.

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 $N/k \geq N^{\varepsilon} \geq 16$ for $N \geq 16^{1/\varepsilon}$, 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. We summarize this discussion in the following theorem, that does not need the assumed lower bound on N. If $N \leq \max\{16^{1/\varepsilon}, 9^{1/\varepsilon}, 2^{2/(1-\varepsilon)^2}\}$, then $\log_{M/B} N \leq \max\{3/\varepsilon, 2/(1-\varepsilon)^2\}$ ($M \geq 4B$), and hence the stated formula is smaller than the scanning bound for accessing all kN entries of the matrix, being also correct. Similarly, if $N \leq 16B$, then $\log_{M/B} N \leq -1 + \log_2 16 = 3$, and the claimed formula is again weaker than the scanning bound.

Theorem 3. *Assume an algorithm computes the row sums for all k-regular $N \times N$ matrices stored in column major layout in the algebraic I/O-model using only canonical partial results with at most $\ell(k,N)$ I/Os. Then, for $B > 2$, $M \geq 4B$,and $k \leq N^{1/\varepsilon}$, $0 < \varepsilon < 1$, there is the lower bound*

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

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

For all values of N, k, M, and B the scanning bound holds $\ell(k,N) \geq kN/B$.

For example, taking $\varepsilon = 1/2$, previous theorem provides following lower bound:

$$\ell(k,N) \geq \min\left\{\frac{1}{16}\frac{kN}{B}\log_{M/B}\frac{N}{\max\{k,M\}}, \frac{1}{24}kN\right\}.$$

Comparing these lower bounds to the algorithms of Section 3.1 that achieve a performance of

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

it becomes clear that the algorithms for fixed layout are optimal up to a constant factor as long as $k \leq N^{\varepsilon}$. Indeed the missing "1+" term in the lower bound can be safely added because of the scanning bound.

5 Lower Bound: Free Layout

In this section, we consider algorithms that are free to choose the layout of the matrix; 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, 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.

5.1 The separated model

At this point, we introduce a slight structural modification of the algebraic 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 succeding I/O. The multiplication table $\mathcal{P} \subseteq \mathcal{M}_R \times \mathcal{M}_V$ represents which coefficients of the matrix A are used in this internal computation. 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 variable memory at the start of this sequence of internal computations. We also separate the disks into D_V containing copies of variables, and D_R containing partial results. Each track of such a disk contains at most B items.

This way of describing the computation separates the distribution of the input variables from the collection of the results. This allows us to analyse the distribution process time forward, whereas we can analyse the collection process time backward, such that in both cases the trees of information flow can be analyzed from the root towards the leaves. Importantly, there is a strong symmetry between a read operation time-forward and a write operation time-backward. In fact a configuration A can be the predecessor of a configuration B by a write operation if and only if B can be the predecessor of A for a read operation, and the other way round with respect of the read/write operation. Hence, counting the number of allowed distribution sequences (time forward) also gives the number of allowed collection sequences (time backward).

Lemma 4. *The conformation of the computed matrix is determined by the sequence of configuration in the separated model.*

Proof. Because all intermediate results are used, the conformation is the union of the multiplication tables. If a product $a_{ij}x_j$ is computed, it is in one of the multiplication tables. Because all intermediate results are assumed to become part of some result, if a product $a_{ij}x_j$ is in one of the multiplication tables, it is part of the conformation. □

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 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.

With this normalization, we can compactly describe the sequence of the $\mathcal{M}_V, D_V$, by following the performed I/Os. For an output operation, it is sufficient to specify the track of the disk and the subset of at most B variables currently in memory that are copied to this track. For an input operation, it is sufficient to specify the track of the disk and the subset of at most B variables currently in memory that are deleted from memory to make space for the new variables. Because there is no choice for the initial configuration, this information uniquely determines the sequence of configurations $\mathcal{M}_V, D_V$ (forward in time)

Symmetrically, similar to 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 step.

It remains to specify the multiplication tables compactly. Every multiplication is specified by an index into the current subset of variables and rows present in the respective memories. Additionally, we can assume that every coefficient a_{ij} is only used once, say as early as possible. Then, at the time of multiplication, either the variable or the partial sum needs to be just 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 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 tables for the complete computation is at most $\binom{2\ell MB}{kN}$.

We summarize this discussion in the following lemma:

Lemma 5. *Any computation of a matrix-vector product, taking $\ell \leq kN + N \leq 2kN$ 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 tables 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 with the number of different conformations of a matrix.

Lemma 6. *Assume an algorithm computes the matrix-vector product for all k-regular $N \times N$ matrices in the algebraic I/O-model with free matrix layout using only canonical partial results with at most $\ell(k, N)$ I/Os. Then, for $M \geq 3B$ and $B \geq 2$, there is the lower bound $\frac{2\ell}{kN} \geq \frac{\log\left(\frac{N}{keBM} \cdot \frac{kN}{2\ell}\right)}{\log(2kN)+B\log\frac{4M}{B}}$.*

Proof. The conformation of a k-regular $N \times N$ matrix is given by choosing N times k-subset of N positions. Because of the naive algorithm we can estimate $\ell \leq kN + N \leq 2kN$ such that Lemma 5 gives $\binom{N}{k}^N \leq \left(\binom{M+B}{B}2kN\right)^{2\ell} \cdot \binom{2\ell BM}{kN}$. Taking logarithms and using $(x/y)^y \leq \binom{x}{y} \leq (xe/y)^y$, we get $kN\log\frac{N}{k} \leq 2\ell\left(\log(2kN) + B\log\frac{e(M+B)}{B}\right) + kN\log\frac{2e\ell BM}{kN}$ rearranging terms and using $e(M+B) \leq 4M$ we get $2\ell \geq kN\frac{\log\frac{N}{k}\frac{kN}{2e\ell BM}}{\log(2kN)+B\log\frac{4M}{B}}$ □

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.

Theorem 7. *Assume an algorithm computes the matrix-vector product for all k-regular $N \times N$ matrices in the algebraic I/O-model with free matrix layout using only canonical partial results 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. 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 combine the statement of Lemma 6 and of Lemma 10 with $s := \frac{N}{keBM}$ and $t := \log(2kN) + B\log\frac{4M}{B}$, 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 10 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 $\log\frac{M}{B} \geq 1$, we get $\ell \geq \frac{kN}{24B}\frac{\log\frac{N}{kM}}{\log\frac{M}{B}}$ which is the statement of the theorem.

Otherwise, if $\log(2kN) > B\log(4M/B)$, then have to compare $\log st$ with $\log kN$. In this case, $N \geq 2^{10}$, $t \geq 1$ and Lemma 11 (iv) yields $st \geq \frac{N}{keBM}B\log\frac{4M}{B} \geq \frac{N}{keM} \geq 2\sqrt[9]{N}$. 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, $\ell \geq kN/84$. □

Corollary 8. *Assume an algorithm computes the matrix-vector product for all k-regular $N \times N$ matrices in the algebraic I/O-model with free matrix layout using only canonical partial results with at most $\ell(k, N)$ I/Os. Furthermore, we assume that the input vector and the result vector might be stored in an arbitrarly order. Then for $M \geq 3B$, $B \geq 6$, and $21 < k \leq \sqrt[3]{N}$ there is the lower bound: $\ell(k, N) \geq \min\left\{\left(\frac{1}{96} - \frac{3}{16k}\right)kN, \frac{1}{24} \cdot \frac{kN}{B}\log_{\frac{M}{B}}\frac{N^{1-2/k}}{kM}\right\}$*

Proof. We proceed as in the proof of Lemma 6 and Theorem 7. The input vector can be stored in an arbitrarily 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.

The remaining computations follows as easy as in Theorem 7 with help of Lemma 11 (v). □

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] 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*, volume 3580 of *Lecture Notes in Computer Science*, pages 576–588. Springer Verlag, Berlin, 2005.

[3] 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. *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), February 2005.

[4] 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, December 2000.

[5] 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.

[6] 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.

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

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

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

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

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

[12] 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*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 161–179. American Mathematical Society, 1999.

[13] J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.

[14] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, 2001.

[15] 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.

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

A Technical Lemmas

Here comes a list of technical Lemmas that we make use of during the exposition of the lower bounds.

Lemma 9. *For every $x > 1$ and $b \geq 2$, 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, since $g(1) = 2 > 0$ 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 10. *Let $b \geq 2$ and $s, t > 0$ such that $s \cdot t > 1$. For all positive real numbers x, we have* $x \geq \frac{\log_b(s/x)}{t} \quad \Rightarrow \quad x \geq \frac{1}{2} \frac{\log_b(s \cdot t)}{t}$

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

$$\begin{aligned} x &\geq \frac{\log_b(s/x)}{t} > \frac{\log_b \frac{2s \cdot t}{\log_b(s \cdot t)}}{t} = \frac{\log_b(2s \cdot t) - \log_b \log_b(s \cdot t)}{t} \\ &\geq \frac{\log_b(2s \cdot t) - \frac{1}{2} \log_b(2s \cdot t)}{t} = \frac{1}{2} \frac{\log_b(2s \cdot t)}{t}, \end{aligned}$$

where the last inequality stems from Lemma 9. This contradiction to the assumed upper bound on x establishes the lemma. □

Lemma 11. *Assume $\log kN \geq B \log(4M/B)$, $k \leq N$, $B \geq 2$, and $M \geq 2B$. Then*

(i) $N \geq 2^{10}$ implies $B \leq \frac{2}{3} \sqrt[3]{N}$.

(ii) $0 < \varepsilon < 1$, $N \geq 2^{2/(1-\varepsilon)^2}$ implies $B \leq \frac{3}{2e} N^{1-\varepsilon}$.

(iii) $N \geq 2^{10}$ and $B \geq 6$ implies $M \leq \frac{1}{6} N^{\frac{5}{9}}$

(iv) $N \geq 2^{10}$, $B \geq 6$, and $k \leq \sqrt[3]{N}$ implies $\frac{N}{keM} \geq 2 \sqrt[9]{N}$.

(v) $N \geq 2^{10}$, $B \geq 6$, and $k \leq \sqrt[3]{N}$ implies $\frac{N^{1-2/k}}{ke^{1+2/k}M} \geq \frac{1}{4} N^{1/9-2/k}$.

Proof. By $M \geq 2B$, we have $\log(4M/B) \geq \log 8 = 3$. Thus $B \leq \frac{\log kN}{\log \frac{4M}{B}} \leq \frac{2}{3} \log N \leq \frac{2}{3} \sqrt[3]{N}$ for $N \geq 2^{10}$, which proves (i).

For statement (ii) one can easily check, that N is big enough to fulfill the inequality similarly to (i).

To see (iii), we rewrite the main assumption as $(kN)^{1/B} > 4M/B$, with $B \geq 6$, $k \leq \sqrt[3]{N}$, and (i) we get $M \leq \frac{1}{4} B (kN)^{1/B} \leq \frac{1}{6} \sqrt[3]{N} (kN)^{1/6} \leq \frac{1}{6} N^{\frac{1}{3} + \frac{4}{3} \cdot \frac{1}{6}} = \frac{1}{6} N^{\frac{5}{9}}$.

(iv) stems from (iii) and $\frac{N}{keM} \geq \frac{6N}{\sqrt[3]{N} e N^{\frac{5}{9}}} \geq 2N^{1-\frac{1}{3}-\frac{5}{9}} = 2\sqrt[9]{N}$. The proof of (v) is similar as that of (iv). □

Note that statements like Lemma 11 (iv) are true for all restrictions $k \leq N^{\varepsilon}$, for sufficiently large bounds on N and B, and with a deterioration of the exponent in the achieved bound. With this, the lower bound holds asymptotically for all $k \leq N^{\varepsilon}$, we have optimal algorithms.