# 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
MADALGO[*], Department of Computer Science,
Aarhus University,
Aarhus, Denmark.
gerth@cs.au.dk

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

Riko Jacob
Technische Universität München,
Department of Computer Science,
Munich, Germany.
jacob@in.tum.de

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

August 23, 2010

[*]Center for Massive Data Algorithmics - a Center of the Danish National Research Foundation

## Abstract

We study the problem of sparse-matrix dense-vector multiplication (SpMV) in external memory. The task of SpMV is to compute $y := Ax$, where $A$ is a sparse $N \times N$ matrix and $x$ is a vector. We express sparsity by a parameter $k$, and for each choice of $k$ consider the class of matrices where the number of nonzero entries is $kN$, i.e., where the average number of nonzero entries per column is $k$.

We investigate what is the external worst-case complexity, i.e., the best possible upper bound on the number of I/Os, as a function of $k$, $N$ and the parameters $M$ (memory size) and $B$ (track size) of the I/O-model. We determine this complexity up to a constant factor for all meaningful choices of these parameters, as long as $k \leq N^{1-\varepsilon}$, where $\varepsilon$ depends on the problem variant. Our 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 variants of the problem, differing in the memory layout of $A$. If $A$ is stored in column major layout, we prove that SpMV has I/O complexity $\Theta\Big(\min\Big\{\frac{kN}{B}\max\Big\{1, \log_{M/B}\frac{N}{\max\{k,M\}}\Big\}, kN\Big\}\Big)$ for $k \leq N^{1-\varepsilon}$ and any constant $0 < \varepsilon < 1$. If the algorithm can choose the memory layout, the I/O complexity reduces to $\Theta\Big(\min\Big\{\frac{kN}{B}\max\Big\{1, \log_{M/B}\frac{N}{kM}\Big\}, kN\Big\}\Big)$ for $k \leq \sqrt[3]{N}$. In contrast, if the algorithm must be able to handle an arbitrary layout of the matrix, the I/O complexity is $\Theta\Big(\min\Big\{\frac{kN}{B}\max\Big\{1, \log_{M/B}\frac{N}{M}\Big\}, kN\Big\}\Big)$ for $k \leq N/2$.

In the cache oblivious setting we prove that with tall cache assumption $M \geq B^{1+\varepsilon}$, the I/O complexity is $\mathcal{O}\left(\frac{kN}{B}\max\Big\{1, \log_{M/B}\frac{N}{\max\{k,M\}}\Big\}\right)$ 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 $x$ 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.

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 nonzero entries in the matrix, which is optimal. In contrast, empirical studies show that this naïve 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 lies in the memory system of modern computers, where access to a data item takes a few CPU cycles when the item is currently stored in cache memory, but significantly more if the item needs to be fetched from main memory, and much more if the item resides on disk. Hence, optimizing for the memory hierarchy is important for achieving efficiency in many computational tasks, including the SpMV problem.

**Previous theoretical considerations** The memory hierarchy of a computer is usually modeled by the **I/O-model** [1] (also known as the DAM-model) and the **cache-oblivious model** [8]. The I/O-model is a two-level abstraction of a memory hierarchy, modeling either cache and main memory, or main memory and disk. The inner memory level has limited size $M$, the outer level is unbounded, and transfers between the two levels take place in tracks of size $B$. Computation can only take place on data residing in the inner memory level, and the cost of an algorithm is the number of track transfers, or I/Os, performed between the two memory levels. The cache-oblivious model is essentially the I/O-model, except that the track size $B$ and main memory size $M$ are unknown to the algorithm designer. More precisely, algorithms are expressed in the RAM model, but analyzed in the I/O-model (assuming an optimal cache replacement policy). The main virtue of the model is that an algorithm proven efficient in the I/O-model for all values of $B$ and $M$ is automatically efficient on an unknown, multilevel memory hierarchy [8]. Thus, the cache-oblivious model enables one to reason about a two-level model while proving results about an unknown multilevel memory hierarchy.

These models were successfully used to analyze the I/O complexity of sorting and permuting. In the I/O-model it was shown that the optimal bound for comparison based sorting $N$ data items is $\mathcal{O}\left(\frac{N}{B}\overline{\log}_{M/B}\frac{N}{M}\right)$ I/Os and for permuting $N$ data items it is $\mathcal{O}\left(\min\{N, \frac{N}{B}\overline{\log}_{M/B}\frac{N}{M}\}\right)$ I/Os [1].[1]

[1]Throughout the paper log stands for the binary logarithm and $\overline{\log}_b x := \max\{1, \log_b x\}$. We also use notation $\ell = \mathcal{O}(f(N,k,M,B))$ with the established meaning that there exists a constant $c > 0$ such that $\ell \leq c \cdot f(N,k,M,B)$ for all valid $N, k, M$, and $B$.

In the cache-oblivious model permuting and sorting can both be performed in $\mathcal{O}\left(\frac{N}{B}\overline{\log}_{M/B}\frac{N}{M}\right)$ I/Os, provided $M \geq B^{1+\varepsilon}$ for some constant $\varepsilon > 0$ (the so-called tall cache assumption) [8], and provably no algorithm can achieve the I/O bounds from the I/O-model in the cache-oblivious model [3]. Permuting is a special case of sparse matrix-vector multiplication (with sparsity parameter $k = 1$), namely where the matrix is a permutation matrix. There are classes of permutation matrices for which the complexity of the problem is known [5]. In its classical formulation [1], the I/O-model assumes that data items are atomic records which can only be moved, copied, or destroyed. Hence, the I/O-model does not directly allow to consider algebraic tasks, as we do here. In particular, for lower bounds a specification of the algebraic capabilities of the model is needed. One specification was introduced in the red-blue pebble game of Hong and Kung [9] (which captures the existence of two levels of memory, but does not assume that I/O operations consist of tracks, i.e., it assumes $B = 1$). Another is implicit in the lower bound proof for FFT in [1]. There are other modifications of the I/O-model known which 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., [18, 6] for excellent surveys of the dozens of papers on this topic, and [18, 17, 10, 13, 12, 7] for sparse matrix libraries. In this line of work, the metric is the running time on test instances and current hardware. This is in contrast to our considerations, where the track size $B$ and the memory size $M$ are the parameters and the focus is on asymptotic I/O performance.

**Problem definition and new results** In this paper, we consider $N \times N$ matrices $A$ and quantify their sparsity by their number of nonzero entries. More precisely, a sparsity parameter $k$ bounds the total number of nonzero entries of $A$ by $kN$, i.e., the average number of nonzero entries per column is at most $k$. For a given value of $k$, we consider algorithms that work for all matrices with at most $kN$ nonzero entries, and we are interested in the worst-case I/O performance of such algorithms as a function of the dimension $N$ of the matrix, the sparsity parameter $k$, and the parameters $M$ and $B$ of the I/O-model. When we need to refer to the actual positions of the nonzero entries of a given matrix, we denote these the **conformation of the matrix**.

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 $a_{ij}$ is $x$. The order of this list corresponds to the layout of the matrix in memory. In this paper, we consider the following types of layouts. **Column major layout** means that the triples are sorted lexicographically according to $j$ (primarily)

Table 1: Summary of the I/O performance of the algorithms in Section 3 (*cache-oblivious).

| | | |
|---|---|---|
| Sect. 3.1 | Naïve algorithm, row and column major | $\mathcal{O}\left(kN\right)^*$ |
| Sect. 3.2 | Worst case (all layouts) | $\mathcal{O}\left(\frac{kN}{B}\,\overline{\log}_{M/B}\,\frac{N}{M}\right)^*$ |
| Sect. 3.3 | Column major | $\mathcal{O}\left(\frac{kN}{B}\,\overline{\log}_{M/B}\,\frac{N}{\max\{k,M\}}\right)^*$ |
| Sect. 3.4 | Best case (blocked row major) | $\mathcal{O}\left(\frac{kN}{B}\,\overline{\log}_{M/B}\,\frac{N}{kM}\right)$ |

and $i$ (to break ties). Symmetrically, **row major layout** means that the triples are sorted lexicographically according to $i$ (primarily) and $j$ (to break ties). If the algorithm is allowed to choose any contiguous layout of the matrix, we refer to this situation as the **best case layout**. If an arbitrary contiguous layout is part of the problem specification, it is called **worst case layout**. Regarding the layout of the input and output vectors, we most of the time assume the elements to be stored contiguous in the order $x_1, x_2, \ldots, x_N$, but we also consider the case where the algorithm is allowed to choose a best case contiguous layout of these vectors.

Our contributions in this paper are as follows and summarized in Table 1:

- We extend the I/O-model Aggarwal and Vitter [1] by specific algebraic capabilities, giving a model of computation which can be seen as a combination of the I/O-model and the model of Hong and Kung [9].
- We give an upper bound parameterized by $k$ on the cost for SpMV when the (nonzero) entries of the matrices are stored in column major layout. Specifically, the I/O cost for SpMV is $\mathcal{O}\left(\min\left\{\frac{kN}{B}\,\overline{\log}_{M/B}\,\frac{N}{\max\{k,M\}}\,,\,kN\right\}\right)$. This bound generalizes the permuting 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 show that when the (nonzero) entries of the matrices are stored in an order specified by the algorithm (best case layout), the I/O cost for SpMV reduces to $\mathcal{O}\left(\min\left\{\frac{kN}{B}\,\overline{\log}_{M/B}\,\frac{N}{kM}\,,\,kN\right\}\right)$.
- We give a lower bound parameterized by $k$ on the cost for SpMV when the nonzero entries of the matrices are stored in column major layout of $\Omega\left(\min\left\{\frac{kN}{B}\,\overline{\log}_{M/B}\,\frac{N}{\max\{k,M\}},\,kN\right\}\right)$ I/Os. This result applies for $k \leq N^{1-\varepsilon}$, for all constant $0 < \varepsilon < 1$. This shows that our corresponding algorithm is optimal up to a constant factor.
- We determine the I/O-complexity parameterized by $k$ on the cost for SpMV when the layout of the matrix is part of the problem specifica-

tion (worst case layout) for $k \leq \sqrt[3]{N}$ to be $\Theta\Big(\min\left\{\frac{kN}{B}\,\overline{\log}_{M/B}\frac{N}{M},\, kN\right\}\Big)$ I/Os.

- We show a lower bound parameterized by $k$ on the cost for SpMV when the layout of the matrix and the vectors is chosen by the algorithm: for $k$ with $8 \leq k \leq \sqrt[3]{N}$ we show that $\Omega\Big(\min\left\{\frac{kN}{B}\,\overline{\log}_{M/B}\frac{N}{kM},\, kN\right\}\Big)$ I/Os are required. This result shows that in this case the choice of the layout of the vector does not allow asymptotically faster algorithms.

- Finally, we show that in the cache oblivious setting with the tall cache assumption $M \geq B^{1+\varepsilon}$, SpMV can be done with $\mathcal{O}\left(\frac{kN}{B}\,\overline{\log}_{M/B}\frac{N}{\max\{k,M\}}\right)$ I/Os if the matrix is stored in column major layout.

In this, the lower bounds take most of the effort. They rely upon a counting argument already used in [1] which we outline here: We define the notion of a semiring I/O-program (details in Section 2.2) that captures the I/O-data flow of an algorithm. We use this to compare the number of different programs using $\ell$ I/O operations with the number of inputs that are possible for the different tasks. From this we can conclude that some inputs (matrices) require a certain number of I/O-operations. 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 (see Section 9).

**Road map of the paper** In Section 2, we describe the computational models in which our lower bounds are proved. In Section 3, we present our upper bounds. The presentation of the lower bound results starts in Section 4 with a lower bound for a simple data copying problem. In Section 5 we use the lower bound for the data copying problem to derive a lower bound for SpMV for the case of worst case layout. In Section 6 we present the lower bound for the case of column major layout, and in Section 7 we presents the lower bound for best case layouts. We may also consider allowing the algorithm to choose the layout of the input and output vectors. In Section 8, we show that this essentially does not help algorithms already allowed to choose the layout of the matrix. Section 9 discusses additional variations on the results, and possible future work.

## 2 Models of Computation

Our aim is to analyze the I/O cost of SpMV. As I/Os are generated by movement of data, we effectively are studying what data flows are necessary for performing the computations involved in SpMV, and how these data flows interact with the memory hierarchy. The final result, i.e., the entries of the output vector, is a set of algebraic expressions, so we need a model specifying both the memory hierarchy and the algebraic capabilities of our machine.

In this section, we define two such models, one to formulate algorithms, and another one to prove lower bounds. By **I/O-model** we mean in this paper the model used to formulate algorithms as explained in Section 2.1 and used in Section 3. In contrast, we prove the lower bounds for so called **semiring I/O-programs**, as explained in detail in Section 2.2. The relation between the two models is explained in Section 2.3, where we argue that a lower bound for semiring I/O-programs also applies to the I/O-model used to formulate algorithms.

Both models combine ideas from the I/O-model of Aggarwal and Vitter [1] previously used to analyze the problem of permuting (a special case of SpMV), with the algebraic capabilities used by Hong and Kung [9] under the name "independent evaluation of multivariate expressions" for analyzing matrix-matrix multiplication. They are based on the notion of a **commutative semiring** $\mathbb{S}$. This is a set of elements with addition ($x + y$) and multiplication ($x \times y$ or $xy$) operations. These operations are associative and commutative, and distributive. There is a neutral element 0 for addition, a neutral element 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), in which matrix multiplication can be used, for example, to compute shortest paths with negative edge length. Other well known semirings are the natural, rational, real or complex numbers with the usual addition and multiplication operations. In the following, we denote the elements of the semiring as numbers.

## 2.1 The I/O-model

The **I/O-model** is the natural extension of the I/O-model of Aggarwal and Vitter [1] to the situation of matrix-vector multiplication. It models main memory to consist of $M$ data atoms, and disk tracks of $B$ data atoms. We count the number of I/O-operations that transfer such a track between main memory and the disk. Here, data atoms are either integers (e.g., used to specify rows or columns of the matrix and positions in a layout) or they are elements from a semiring. In particular, elements of the semiring cannot be compared with each other. All computation operations require that the used data items are currently stored in main memory.

## 2.2 Semiring I/O-programs

In contrast to the algorithmic model, for the lower bound we consider a non-uniform setting. This means that an algorithm is given by a family of **semiring I/O-programs**, one program for each conformation (including $N$ and $k$) of the matrix, irrespective of the underlying semiring and numbers. The layout of the matrix can be implied by the conformation (e.g., column major layout), part of the conformation (worst-case layout), or part of the program (best-case layout). Such a semiring I/O-program is a finite sequence of operations executed sequentially (without branches or loops) on the following abstract machine: there is a **disk** $D$ of infinite size, organized in **tracks** of $B$ numbers each, and a main **memory** containing $M$ numbers, with all numbers belonging to a commutative semiring $\mathbb{S}$. A **configuration** of the model is a vector of $M$ numbers $\mathcal{M} = (m_1, \dots, m_M) \in \mathbb{S}^M$, and an infinite sequence $D$ of track vectors $\mathbf{t}_i \in \mathbb{S}^B$. Initially, all numbers not part of the input have the value 0. A step of a computation leads to a new configuration according to the following allowed **operations**:

- **Computation** on semiring numbers in main memory: algebraic operations $m_i := m_j + m_k$, $m_i := m_j \times m_k$, setting $m_i := 0$, setting $m_i := 1$, and assigning $m_i := m_j$.
- **Read**: move an arbitrary track of the disk into the first $B$ cells of memory, $(m_1, \dots, m_B) := \mathbf{t}_i$; $\mathbf{t}_i := \mathbf{0}$. The cells are required to be empty prior to the read ($(m_1, \dots, m_B) = \mathbf{0}$).
- **Write**: move the first $B$ cells of memory to an arbitrary track, $\mathbf{t}_i := (m_1, \dots, m_B)$; $(m_1, \dots, m_B) := \mathbf{0}$. The track is required to be empty prior to the write ($\mathbf{t}_i = \mathbf{0}$).

Read and write operations are collectively called I/O-operations. We say that an algorithm for SpMV runs in (worst-case) $\ell(k, N)$ I/Os if for all $N$, all $k$, and all conformations for $N \times N$ matrices with $kN$ nonzero coefficients, the program chosen by the algorithm performs at most $\ell(k, N)$ I/Os.

**Intermediate results** During the execution of a program, the disk and the main memory contain numbers, which are either part of the input (i.e., matrix and input vector entries), are one of the two neutral elements 0 and 1 of the semiring, or are the result of a series of semiring operations. We call the numbers appearing during the run of a program for **intermediate results**. One intermediate 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. By the commutativity of addition and multiplication, every intermediate result may be written in the form of a polynomial, with integer coefficients, in the $x_j$'s of the input vector and the nonzero $a_{ij}$'s of the matrix—that is, as a sum of monomials where each monomial is a product of $x_j$'s and $a_{ij}$'s, each raised to some power, and one element from $\{1, 1+1, 1+1+1, \dots\}$. We call this the **canonical form** of an intermediate result.

The degree of a canonical form is the largest sum of powers that occur in one of the monomials. If two canonical forms $f$ and $g$ are multiplied or added, then the set of variables in the resulting canonical form is the union of the variables of $f$ and $g$. For multiplication, the degree is equal to the sum of the degrees. For addition, the number of monomials and the degree cannot decrease.

Note that the canonical forms themselves form a semiring, called the free commutative semiring over the input numbers. Because of this it is impossible that two different canonical forms stand for the same element in all possible semirings. Hence, the final results $y_i$ of SpMV are the unique canonical forms

$y_i = \sum_{j \in R_i} 1 \times a_{ij} \times x_j$ of degree at most two. Here $R_i$ is the set of columns in which row $i$ has nonzero entries, which depends only on the conformation of the matrix. By a similar reasoning, as detailed in Lemma 2.1 below, we may assume that each intermediate value is a "subset" of a final result, i.e., it has one of the following forms: $x_j$, $a_{ij}$, $a_{ij}x_j$, or $s_i = \sum_{j \in S} a_{ij}x_j$, where $1 \leq i, j \leq N$, $S \subseteq R_i \subseteq \{1, \ldots, N\}$. These forms are called, respectively, **input variable**, **coefficient**, **elementary product**, and **partial sum**. We call $i$ the **row** of the partial sum. If $S = R_i$, we simply denote the partial sum a **result**. Collectively, we call these forms **canonical partial results**. Every canonical partial result belongs in a natural way to either a column ($j$ for $x_j$) or a row ($i$ for the rest of the canonical partial results).

**Lemma 2.1** *If there is a program which multiplies a given matrix $A$ with any vector in the semiring I/O-model using $\ell$ I/Os, then there is also a program performing the same computation using $\ell$ I/Os, but computing only canonical partial results and never having two canonical partial sums of the same row in memory immediately before an I/O is performed.*

**Proof.** Assume that the canonical form $f$ represents an intermediate result. If $f$ contains a monomial of degree one (i.e., single variables or coefficients), it needs to be multiplied with a non-constant canonical form before it can become a final result. Hence, if it contains two different monomials of degree one, there will be products of two variables or two coefficients, or a product of a variable with a coefficient of the wrong column. Such an $f$ cannot be used for a final result. The following four properties of $f$ will be inherited by anything derived from $f$ and hence prevent $f$ from being used for any final result.

1. $f$ has degree of three or more,
2. $f$ has a poly-coefficient of $1 + 1$ or larger,
3. $f$ contains matrix entries from different rows,
4. $f$ contains the product of two vector variables or two matrix entries.

From this we can conclude that a useful intermediate result $f$ must be a canonical partial result (by case analysis of the degree of $f$).

Clearly, intermediate results that are computed but are not used for a final result may as well not be computed. Similarly, the 0-canonical form is equivalent to the constant 0 and any occurrence of it as intermediate result can be replaced by the constant 0.

Finally, observe that partial sums can only be used in summations that derive new partial sums or final results. Hence, if two partial sums $p_i$ and $q_i$ that both are used to eventually derive $y_i$ are simultaneously in memory, they can be replaced by $p'_i = p_i + q_i$ and the constant $q'_i = 0$ (not a partial sum) without changing the final result. $\square$

## 2.3 Connection between algorithmic and lower bound model

The two introduced models only make sense if the complexities in the two models are essentially the same:

**Lemma 2.2** *Every algorithm $\mathcal{A}$ for SpMV in the I/O-model with I/O-complexity $\ell(k, N)$ gives rise to a family of semiring I/O-programs using at most $\ell(k, N)$ I/Os.*

**Proof.** Fix $k$, $N$, a matrix conformation and its layout. Because the branches and loops of the algorithm do not depend on the semiring values, we can symbolically execute the algorithm $\mathcal{A}$ and finally ignore all computations on non-semiring variables. This leads to a family of semiring I/O-programs that use no more I/Os than $\mathcal{A}$. Here the layout of the matrix can either be fixed, part of the input or part of the program, precisely in the same way as for $\mathcal{A}$. □

It follows that lower bounds for semiring I/O-programs translate into lower bounds for algorithms in the I/O-model. For the case of worst-case layout, it is noteworthy that the algorithms we propose in Section 3 do not need to know the layout in advance.

# 3 Algorithms

In this section we describe several algorithms for solving SpMV. All algorithms compute the necessary elementary products $a_{ij}x_j$ (in some order) and then compute the sums $y_i = \sum_{j=1}^{N} a_{ij}x_j$ (in some order). We describe different algorithms for each of the possible layouts: row major, column major, worst case, and best case. The bounds achieved are summarized in Table 1. Throughout this section we assume that $x$ is given in an array such that we can scan the sequence $x_1, x_2, \ldots, x_N$ using $\mathcal{O}(N/B)$ I/Os. We assume $1 \leq k \leq N$ and $M \geq 3B$.

## 3.1 Naïve Algorithm

We first describe how the layout of a matrix influences the I/O performance if we simply perform a sequential scan over all triples $(i, j, a_{ij})$ and repeatedly update $y_i := y_i + a_{ij}x_j$ to solve the SpMV problem. In internal memory this approach yields an optimal $\mathcal{O}(kN)$ time algorithm, but for both row major and column major layouts we get poor I/O performance—but for different reasons.

**Row major layout** In row major layout we finish computing $y_i$ before advancing to compute $y_{i+1}$. The nonzero entries in a row $i$ of $A$ are in the worst case sparsely distributed such that each $x_j$ accessed while computing $y_i$ is in a new track, in the worst case causing an I/O to fetch each $x_j$. While scanning all triples in total costs $\mathcal{O}(kN/B)$ I/Os and updating $y$ costs $\mathcal{O}(N/B)$ I/Os, the bottleneck becomes accessing $x$, causing in the worst case a total of $\mathcal{O}(kN)$ I/Os.

**Column major layout** The algorithm performs a single sequential scan of the list of the $kN$ triples for the nonzero entries of $A$ while simultaneously performing a single scan of all entries in $x$, i.e. we compute all products $y_{ij}x_j$ involving $x_j$ before advancing to $x_{j+1}$. The cost of accessing $A$ and $x$ is $\mathcal{O}(kN/B)$ I/Os. While updating $y_i := y_i + a_{ij}x_j$ we potentially skip many entries of $y$ between consecutive updates, such that each of the $kN$ updates to $y$ in the worst case causes one read and one write I/O. It follows that in the worst-case the total number of I/Os is $\mathcal{O}(kN)$.

Note that while for row major layout the I/O bottleneck was to access $x$ it for column major layout are the updates to $y$ that become the bottleneck.

## 3.2 Worst Case Layout

A generic algorithm to multiply $A$ with $x$, without any assumption on the layout of the nonzero entries of $A$, is the following.

1. Sort all triples $(i, j, a_{ij})$ by $j$.
2. Scan the list of all triples $(i, j, a_{ij})$ (in increasing $j$ order) while simultaneous scanning $x$ and generating all triples $(i, j, a_{ij}x_j)$.
3. Sort all triples $(i, j, a_{ij}x_j)$ by $i$.
4. Scan the list of all triples $(i, j, a_{ij})$ (in increasing $i$ order) while adding consecutive triples generating the sums $y_i = \sum_{j=1}^{N} a_{ij}x_j$.

Steps 1 and 3 sort $kN$ elements using $\mathcal{O}\left(\frac{kN}{B}\overline{\log}_{M/B}\frac{kN}{M}\right)$ I/Os [1], while Steps 2 and 4 scan $\mathcal{O}(kN)$ elements using $\mathcal{O}(kN/B)$ I/Os. In total $\mathcal{O}\left(\frac{kN}{B}\overline{\log}_{M/B}\frac{kN}{M}\right)$ I/Os are used.

The logarithm in the above bound can be reduced by partitioning the $kN$ coefficients into $k$ groups each of size $N$ (using an arbitrary partitioning). By using the above algorithm for each of the $k$ groups we use $\mathcal{O}\left(\frac{N}{B}\overline{\log}_{M/B}\frac{N}{M}\right)$ I/Os per group. The results are $k$ vectors of size $N$ with partial results. By adding the $k$ vectors using $\mathcal{O}(kN/B)$ I/Os we compute the final output vector $y$. In total we use $\mathcal{O}\left(\frac{kN}{B}\overline{\log}_{M/B}\frac{N}{M}\right)$ I/Os.

## 3.3 Column Major Layout

If $A$ is given in column major layout the algorithm in Section 3.2 can be improved. Step 1 can be omitted since the triples are already sorted by increasing $j$. Step 2 remains unchanged. After Step 2 we have for each column $j$ all triples $(i, j, a_{ij}x_j)$ sorted by $i$. We split the columns into at most $2k$ groups, each group consisting of at most $N/k$ columns and at most $N$ triples. For each group we have at most $N/k$ sequences of triples sorted by $i$. For each group we compute the merged sequence of triples using an I/O-optimal merger. Since $t$ sequences with a total of $s$ elements can be merged using $\mathcal{O}\left(\frac{s}{B}\,\overline{\log}_{M/B}\min\{t, \frac{s}{M}\}\right)$ I/Os [4], the merging in total for all groups requires $\mathcal{O}\left(\frac{kN}{B}\,\overline{\log}_{M/B}\frac{N}{\max\{k,M\}}\right)$ I/Os. By scanning the output of each group we can compute a vector of $N$ partial results for each group using a total of $\mathcal{O}(kN/B)$ I/Os, since groups are sorted by $i$. Finally we add the $k$ vectors using $\mathcal{O}(kN/B)$ I/Os. The total I/O cost becomes $\mathcal{O}\left(\frac{kN}{B}\,\overline{\log}_{M/B}\frac{N}{\max\{k,M\}}\right)$.

## 3.4 Best Case Layout

By adopting a different input layout to the algorithm in Section 3.3 we can achieve a further speedup. Assume we store the coefficients of $A$ in slabs of $M - 2B$ consecutive full columns, where each such slab is stored in row major layout. This layout allows a sequentially scan of the triples $(i, j, a_{ij})$ to generate the triples $(i, j, a_{ij}x_j)$ in total $\mathcal{O}(kN/B)$ I/Os, by having the window of the $M-2B$ elements from $x$ in memory corresponding to the columns of the current slab. The resulting sequence of triples consists of $N/(M - 2B)$ subsequences sorted by $i$. As in Section 3.3 we split the generated triples into $2k$ groups, but now each group only consists of $\frac{N}{k(M-2B)} = \mathcal{O}\left(\frac{N}{kM}\right)$ sorted sequences to merge. The rest of the algorithm and analysis remains unchanged. The total I/O cost becomes $\mathcal{O}\left(\frac{kN}{B}\,\overline{\log}_{M/B}\frac{N}{kM}\right)$.

## 3.5 Cache Oblivious Algorithms

The algorithms in Section 3.2 and Section 3.3 can be implemented cache obliviously (under the tall cache assumption $M \geq B^{1+\varepsilon}$, as is needed for optimal sorting in this model [3]), by using a cache-oblivious sorting algorithm [8] for sorting steps and using the *Run*-optimal cache-oblivious adaptive sorting algorithm of [4, Section 5] for merging steps. Under the tall cache assumption the asymptotic I/O complexities remain unchanged.

# 4 Lower Bound for a Copying Problem

Central to our lower bound arguments for the SpMV problem will be the analysis of a copying problem which distributes $N$ input values into $H$ specified output positions in memory. In this section, we consider that problem on its own, in a way which will allow us to reuse the central part of the analysis in different settings in later sections. Specifically, Lemma 4.2 will be referenced repeatedly (Section 5 can reuse a bit more of the analysis, and therefore references later lemmas of Section 4).

In the $N$-$H$ **copy problem**, an instance is specified by an index $i_j \in [N]$ for each $j \in [H]$, where $[N] = \{1, \ldots, N\}$, and the task is to transform an input vector $(x_1, \ldots, x_N)$ of $N$ variables into the vector $(x_{i_1}, x_{i_2}, x_{i_3}, \ldots, x_{i_H})$. In this section, we show a lower bound on $\ell(H, N)$, the worst case number of I/Os necessary to perform any such task, seen as a function of $H$ and $N$. We for this problem assume the I/O-model of Aggarwal and Vitter [1], i.e., the entries $x_i$ of the input vector are atomic and can only be moved, copied, or deleted.

The remaining of this section is devoted to prove the following lower bound theorem:

**Theorem 4.1** *For every $N$ and $H$ where $N/B \leq H \leq N^2$, there exists an $N$-$H$ copy problem $(i_1, \ldots, i_H) \in [N]^H$ that requires at least*

$$\min\left\{\frac{H}{6}, \frac{H}{4B} \overline{\log}_{\frac{M}{B}} \frac{N}{M}\right\}$$

*I/Os in the I/O-model of Aggarwal and Vitter [1].*

In our analysis, we will during the execution of the algorithm consider the contents of the main memory, and of each track on disk, as *sets*. That is, inside each of these, we will disregard order and multiplicities of elements. This view allows us to analyze tightly the development in combinatorial possibilities as the number of I/Os grows.

We call the values of these sets the *state* at a given point in time, and the sequence of states we call the *trace* of the program. For a state, the content of the main memory is described by a subset $\mathcal{M} \subseteq [N]$, $|\mathcal{M}| \leq M$ (the indices of the variables currently in memory). Similarly, the tracks of the disk are described by a list of subsets $\mathcal{T}_i \subseteq [N]$, $|\mathcal{T}_i| \leq B$ (for $i \geq 1$).

Note that between I/Os, the contents of main memory and of each track on disk do not change seen as sets, if the program only copies and moves elements. For the $N$-$H$ copy problem, any element created which is not used for the output may as well not be created. If there is an input element which does not appear in the output, we may as well assume it is deleted immediately when it enters main memory the first time (which gives rise to no combinatorial choices in the analysis). Hence, we can during the analysis assume deletions do not take place.

**Lemma 4.2** *Consider a family of programs in the I/O-model starting from the unique (set-based) state $\{1, \ldots, B\}, \{B+1, \ldots, 2B\}, \ldots, \{\ldots, N\}$ and using at most $\ell$ I/Os, assuming $\ell \geq (N+H)/B$ and $M \geq 4B$. The number of different (set-based) traces, and hence output (set-based) states, is at most*

$$\left(\left(\frac{4M}{B}\right)^B 4\ell\right)^\ell .$$

**Proof.** We assume that the program complies with the following rules: The program executes precisely $\ell$ I/Os (if fewer I/Os are executed we pad the program with repeatedly writing and reading an empty track). Tracks used for intermediate results are numbered consecutively and hence require an address space of at most $\ell$ positions, such that together with the at most $(N+H)/B$ I/Os to read input and write output, we get an address space of at most

$(N+H)/B+\ell \leq 2\ell$ tracks. Any correct program can easily be transformed to comply with the above rules without increasing the number of I/Os.

We can now bound the number of traces (equivalently, number of final states) that can be created with $\ell$ I/Os, by bounding the number of choices for generating a new state when performing an I/O.

For a write operation, there is the choice of the up to $B$ elements out of $M$ when creating the contents of the track. There is a difference in resulting state if an element moved out of main memory by a write is the last copy in main memory, or if it is not—in the former case, the content of main memory, seen as a set value, changes, in the latter case, it does not. In the analysis, we move this combinatorial choice to the reads (since the combinatorial choices of writes already dominate in the argument), and we do this by never changing the set value of the main memory when performing a write, but instead leave the elements of the former case as dead elements of the set. To avoid the set size growing above $M$, reads now must remove enough such dead elements to make room for the up to $B$ elements entering main memory. Because in the actual program there is room for the elements entering main memory, there are in the analysis always enough dead elements for such a removal to be feasible. Thus, in our analysis reads also have a choice of up to $B$ elements out of $M$ when choosing which dead elements to remove.

For writes, the number of elements written is not necessarily $B$, since we are considering set-based traces. Rather, a write may write out sets of sizes $1, 2, \ldots, B$. Hence, the number of different sets which a write can choose is $\sum_{i=1}^{B}\binom{M}{i}$. From the standard identity $\sum_{i=0}^{n}\binom{r}{i}\binom{s}{n-i} = \binom{r+s}{n}$ it follows that $\sum_{i=0}^{B}\binom{M}{i} \leq \sum_{i=0}^{B}\binom{M}{i}\binom{B}{B-i} = \binom{M+B}{B}$. With $M \geq 4B$, implying $M+B \leq 5M/4$, and $e5M/4 < 4M$ we get by $\binom{x}{y} \leq (xe/y)^y$ the bound $\binom{M+B}{B} \leq \left(\frac{4M}{B}\right)^B$.

For each I/O, there is also the choice of which of the at most $2\ell$ tracks on disk it pertains to, and there is a factor of 2 to choose between read and write operations. □

The above finishes our analysis of algorithms for the $N$-$H$ copy problem in terms of set-based traces. To account for the actual ordering and multiplicities of elements of the output when the algorithm stops, we must add a final step in the analysis where for each non-empty track (main memory must be empty by the definition of the output), a multiset is created from its set-based representation, to give the actual output of the algorithm. Multiplying with the combinatorial possibilities for that step gives a bound on the total number of different outputs which can be generated by $\ell$ I/Os. This number can then be compared to the number of different outputs the algorithm is required to be able to create. We do this in the following lemma.

**Lemma 4.3** *Assume there is a family of programs in the I/O-model computing all copying tasks using $N$ variables and producing $H$ outputs with $\ell = \ell(H,N)$ I/Os for memory size $M$ and track size $B$. Then, for $H \geq N/B$ and $M \geq 4B$*

*it must hold that*

$$N^H \leq \left( \left( \frac{4M}{B} \right)^B 4\ell \right)^\ell \cdot B^H \, .$$

**Proof.** During execution, we consider the contents of the main memory, and of each track on disk, as *sets*, and we now have to count the number of ways in which a set-based state describing the tracks output can be interpreted as a list of copies. Given a specific subset of $[N]$ residing on an output track, a variable in the final (ordered, multiset) version of the track can be expressed by the rank (a number in $[B]$) of the index of the variable within the subset of the track. Hence, one (set-based) trace can solve at most $B^H$ different copying tasks when adding order and multiplicity in the end. Because there are $N^H$ copying tasks, the lemma follows from Lemma 4.2, provided $\ell \geq (N+H)/B$. Since $H \geq N/B$ there exists an $N$-$H$ copy problem that outputs $H/B$ blocks all different from an input block, and all $N/B$ input blocks contain at least one element that is part of the output, i.e., we have $\ell \geq (N+H)/B$. $\square$

To finish the proof of Theorem 4.1, we need to solve the equation of Lemma 4.3 for $\ell$. We do this in the following lemma. Note that $H \geq N/B$ implies $\ell \geq (N+H)/B$, as shown in the proof of Lemma 4.3, so the assumption $\ell \geq H/B$ below is justified.

Comparing to Lemma 4.3, one notes that the assumption (1) of the lemma is slightly weakened. This is done in order to match conditions in Section 5, such that we there can refer directly to Lemma 4.4.

**Lemma 4.4** *Assume $M \geq 4B$, $H \leq N^2$, and $\ell \geq H/B$. Then*

$$\left(\frac{N}{2}\right)^H \leq \left( \left( \frac{4M}{B} \right)^B 4\ell \right)^\ell \cdot B^H \tag{1}$$

*implies*

$$\ell \geq \min \left\{ \frac{H}{4B} \overline{\log}_{M/B} \frac{N}{M} \, , \, \frac{H}{6} \right\} \, .$$

**Proof.** For $B \leq 6$ the bound $\ell \geq H/B$ justifies the theorem, hence we assume $B \geq 6$. For $N \leq 2^{10}$ the lemma holds trivially: Because of $B \geq 6$ we have $M \geq 24$ and $N/M < 2^6$. Using $M/B \geq 4$ we get $\log_{M/B} \frac{N}{M} < 3$, and hence again the bound $\ell \geq H/B$ justifies the theorem. Therefore, we assume $N \geq 2^{10}$. If $\ell > H/6$, the lemma holds trivially. Hence, in the following we assume $\ell \leq H/6$, implying $4\ell \leq H$.

Taking logs on the main assumption (1), we get

$$H \log \frac{N}{2} \leq \ell \left( \log H + B \log \frac{4M}{B} \right) + H \log B \, .$$

which rewrites as

$$\ell \geq H \frac{\log \frac{N}{2B}}{\log H + B \log \frac{4M}{B}} \, .$$

Now, we distinguish by the leading term of the denominator: If $\log H < B \log \frac{4M}{B}$, we get (using $\log \frac{M}{B} \geq 2$ and $2B \leq M$)

$$\ell \geq H \frac{\log \frac{N}{2B}}{2B\left(2 + \log \frac{M}{B}\right)} \geq \frac{H}{4B} \log_{\frac{M}{B}} \frac{N}{M} .$$

If $\log H \geq B \log \frac{4M}{B}$, we use $\log \frac{4M}{B} \geq \log 16 = 4$, which leads (using $H \leq N^2$) to $B \leq \frac{\log H}{\log \frac{4M}{B}} \leq \frac{2}{4} \log N \leq \frac{1}{2} \sqrt[3]{N}$ for $N \geq 2^{10}$. We conclude with

$$\ell \geq H \frac{\log\left(\frac{N}{2 \cdot \frac{1}{2} \cdot \sqrt[3]{N}}\right)}{2\log(N^2)} = H \frac{\frac{2}{3}\log N}{4 \log N} = \frac{H}{6} .$$

□

## 5 Lower Bound for Worst-Case Layout

In this section, we give a lower bound for the SpMV problem with worst case layout of the matrix. We consider the special case where the input vector is the all ones vector ($x_i = 1$), i.e., we consider programs computing the row-sums of the matrix.

The idea of the proof is to trace backwards in time the computation of each of the $N$ output results $s_i = \sum_{j \in R_i} a_{ij}$ to the $kN$ coefficients $a_{ij}$ of the matrix (recall that $R_i$ is the set of column indexes of the nonzero entries of row $i$ of the matrix). We will show how to interpret this time-reversed execution of a semiring I/O-program for computing row-sums as solving an instance of the $N$-$H$ copy problem. This will allow us to reuse the analysis of Section 4.

To trace the program execution backwards in time, we annotate each result $s_i$ by its row index $i$, and propagate these annotations backwards in time to other values appearing during the execution of the program. By Lemma 2.1 and the discussion preceding it, we may assume that all values appearing are either 0, 1, coefficients, or partial sums. We may also assume that additions by 0 and multiplication by 1 do not take place, as such operations can be omitted without changing the output. By a case analysis of possible value types and the operations allowed in the model (and the uniqueness of the canonical form of a value, cf. text preceding Lemma 2.1), one sees that a partial sum $m_t = \sum_{j \in S} a_{ij} x_j$ with $|S| \geq 2$ can only arise as the result of either the addition $m_t := \sum_{j \in S_1} a_{ij} x_j + \sum_{j \in S_2} a_{ij} x_j$, where $S_1, S_2$ is a nontrivial partitioning of $S$ ($S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$, $|S_1|, |S_2| \geq 1$), or the assignment $m_t := m_s$, where $m_s = \sum_{j \in S} a_{ij}$. In the former case, we propagate the annotation $i$ backwards in time to both operands of the addition. In the latter case, we propagate the annotation $i$ backwards in time to $m_s$. Similarly, we propagate the annotation of an elementary product $a_{ij} x_j$ back to $a_{ij}$ at the multiplication creating this elementary product. For read and write operations, note that these are time reverses of each other, due to the definitions of reads and writes in our model (which are symmetrical, and involve moving rather than copying). For these operations, annotations just follow the values moved.

By induction on the number of time steps backwards in time, it follows that at all times, all entries $a_{ij}$ are annotated exactly once (by their row index $i$) if the annotation of a partial sum $\sum_{j \in S} a_{ij} x_j$ is seen as pertaining to all coefficients $a_{ij}$ appearing in the sum.

Given a vector $(x_1, \ldots, x_N)$ of $N$ variables, we may change the annotation to use the variable $x_i$ instead of the integer $i$. This annotation, when viewed backwards in time, defines a program for copying a vector $(x_1, \ldots, x_N)$ of $N$ variables into $kN$ positions in memory, with $x_i$ being copied to the positions on disk where the nonzero coefficients of row $i$ of the matrix reside in the given layout of the matrix. In other words, it is a program solving an instance of the $N$-$H$ copy problem for $H = kN$, with the output vector $(x_{i_1}, x_{i_2}, x_{i_3}, \ldots, x_{i_H})$ specified by the matrix and its layout. It works in the I/O-model of Aggarwal and Vitter [1], and uses copying (the time reverse of addition of two partial sums), movement (the time reverse of assignment), and reads and writes (the time reverses of each other). The number of I/Os performed is the same as the original program for computing row-sums.

However, not all of the $N^{kN}$ instances of the $N$-$H$ copy problem with $H = kN$ can be specified by a matrix in this way, since no row has more than $N$ nonzero entries. More precisely, the possible instances are exactly those where the output vector $(x_{i_1}, x_{i_2}, x_{i_3}, \ldots, x_{i_{kN}})$ has no input variable $x_i$ appearing more than $N$ times. We claim that for $k \leq N/2$, the number of such output vectors is at least $(N/2)^{kN}$: Construct the vector by choosing $x_{i_j}$ for increasing $j$. At some point, a given variable $x_i$ may no longer be available because it has been used $N$ times already. However, if this is true for more than $N/2$ of the $N$ variables, more than $N^2/2$ entries in the vector have been chosen, which is a contradiction to the assumption $k \leq N/2$. Hence, at each of the $kN$ positions of the input vector, we have at least $N/2$ choices.

Therefore, by adjusting the last line of the proof of Lemma 4.3, we arrive at the following variant of that lemma.

**Lemma 5.1** *Assume there is a family of semiring I/O-programs, one for each $kN$-dense $N \times N$ matrix and each layout of it, computing the matrix vector product. Assume all programs use $\ell = \ell(H, N)$ I/Os. Then, for $M \geq 4B$ and $1 \leq k \leq N/2$ it holds that*

$$\left(\frac{N}{2}\right)^{kN} \leq \left(\left(\frac{4M}{B}\right)^{B} 4\ell\right)^{\ell} \cdot B^{kN} .$$

Using Lemma 4.4, we get the following theorem.

**Theorem 5.2** *Let $\ell = \ell(k, N)$ be an upper bound on the number of I/O-operations necessary to solve the SpMV problem with worst case layout in the semiring I/O-model. Then, for $M \geq 4B$ and $1 \leq k \leq N/2$ it holds that*

$$\ell \geq \min\left\{\frac{kN}{4B} \overline{\log}_{M/B} \frac{N}{M}, \frac{kN}{6}\right\} .$$

## 6 Lower Bound for Column Major Layout

In this section, we consider the case of matrices stored in column major layout. Here, it turns out to be sufficient to consider only $k$-regular sparse matrices, i.e., matrices with precisely $k$ nonzero entries per column.

The outline of the proof in this section follows closely the structure from Sections 4 and 5. Again, we will trace the execution backwards in time to arrive at an instance of the $N$-$H$ copy problem, again we consider set-based states during this backwards execution, adding order and multiplicity of elements in the output as a final step, and again, we consider the special case of computing row-sums (i.e., we consider an input vector with $x_i = 1$ for all $i$). Compared to Section 5, we now restrict ourselves to $k$-regular sparse matrices in column major layout, which means that the input matrix constitutes a list of indices of length $kN$, where $k$ consecutive positions encode one column of the matrix.

When adding order and multiplicity of elements as a final step, we could certainly again bound the choices for a given set-based state by $B^{kN}$. As it turns out, this would asymptotically weaken our result. Instead, we have to more carefully count the number of different matrix conformations corresponding to the same set-based state. We denote this number by $\tau = \tau(N, k, B)$, and now proceed to give bounds on it.

If $k = B$, then there is a one-to-one correspondence between the precisely $k$ entries in a column and the contents of a track. Since in column major layout, the entries of one column are stored in order of increasing row, the set $\mathcal{T}_i$ of a track in this case uniquely identifies the vector $\mathbf{t}_i$ of a track, making $\tau = 1$. For $B < k$, some tracks belong completely to a certain column, and other tracks are shared between two neighboring columns. Every element of the track can hence belong either to one column, the other 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, giving at most $\binom{2B}{k}$ choices per column. Over all columns, this results in up to $\binom{2B}{k}^N \le (2eB/k)^{kN}$ different matrix conformations corresponding to the same set-based state. Summarizing we have:

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

Because the number of distinct conformations of an $N \times N$ $k$-regular matrix is $\binom{N}{k}^N$, we can summarize this discussion (using Lemma 4.2) in the following statement:

**Lemma 6.1** *If an algorithm computes the row-sums for all $k$-regular $N \times N$ matrices stored in column major layout using at most $\ell = \ell(k, N)$ I/Os then it* *holds that*

$$\binom{N}{k}^N \leq \left(\left(\frac{4M}{B}\right)^B 4\ell\right)^\ell \cdot \tau \,, \tag{2}$$

*where $\tau = \tau(N,k,B)$ is the maximal number of different matrix conformations corresponding to any set-based state.*

Algebraic manipulations of the inequality in Lemma 6.1 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 6.2** *Let $\ell = \ell(k,N)$ be an upper bound on the number of I/O-operations necessary to solve the SpMV problem for matrices stored in column major layout in the semiring I/O-model. Then, for $B > 2$, $M \geq 4B$, and $k \leq N^{1-\varepsilon}$, where $\varepsilon$ is any constant with $0 < \varepsilon < 1$, it holds that*

$$\ell(k,N) \geq \min\left\{\kappa \cdot \frac{kN}{B} \overline{\log}_{M/B} \frac{N}{\max\{k,M\}} \,, \, \frac{1}{4} \cdot \frac{\varepsilon}{2-\varepsilon} kN\right\}$$

*for $\kappa = \min\left\{\frac{\varepsilon}{3}, \frac{(1-\varepsilon)^2}{2}, \frac{1}{7}\right\}$.*

For example, taking $\varepsilon = 1/2$, Theorem 6.2 provides the following lower bound:

$$\ell(k,N) \geq \min\left\{\frac{1}{8}\frac{kN}{B} \overline{\log}_{M/B} \frac{N}{\max\{k,M\}} \,, \, \frac{1}{12} kN\right\} .$$

Comparing this lower bound to the algorithm of Section 3.3 which achieves a performance of

$$\mathcal{O}\left(\min\left\{\frac{kN}{B} \overline{\log}_{M/B} \frac{N}{\max\{k,M\}}, kN\right\}\right) ,$$

we see that this algorithm for column major layout is optimal up to a constant factor, as long as $k \leq N^{1-\varepsilon}$.

Proof of Theorem 6.2: Fix $k$, $N$, and $\ell = \ell(k,N)$, and assume the inequality (2) of Lemma 6.1 holds. If $\ell > kN/4$, the theorem is trivially true. Hence, we assume $\ell \leq kN/4$. Taking into account the different cases for $\tau$, we get the following inequalities:

For $k < B$: The inequality is $\binom{N}{k}^N \leq \left(\left(\frac{4M}{B}\right)^B kN\right)^\ell \cdot (2eB/k)^{kN}$. Using $(x/y)^y \leq \binom{x}{y}$ and taking logs we get $kN \log \frac{N}{k} \leq \ell \left(\log(kN) + B \log \frac{4M}{B}\right) + kN \log(2eB/k)$, leading to

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

For $k \geq B$: From $\binom{N}{k}^N \leq \left(\left(\frac{4M}{B}\right)^B kN\right)^\ell \cdot 3^{kN}$ by taking logs we get $kN \log \frac{N}{k} \leq \ell \left(\log(kN) + B \log \frac{4M}{B}\right) + kN \log 3$, i.e.,

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

Combining (3) and (4) we get

$$\ell \geq kN \frac{\log \frac{N}{\max\{3k, 2eB\}}}{\log(kN) + B \log \frac{4M}{B}} \,. \tag{5}$$

For small $N \leq \max\{2^{2/(1-\varepsilon)^2}, 16^{1/\varepsilon}, 9^{1/\varepsilon}, 2^{10}, 16B\}$, this bound is perhaps weak, but the statement of the theorem is justified by the scanning bound $\ell \geq kN/B$ for accessing all $kN$ entries of the matrix: If $N \leq 2^{2/(1-\varepsilon)^2}$, then $\log_{M/B} N \leq 2/(1-\varepsilon)^2$. If $N \leq 16^{1/\varepsilon}$ then $\log_{M/B} N \leq 2/\varepsilon$. If $N \leq 16B$ then $\log_{M/B} \frac{N}{M} \leq -1 + \log_2 16 = 3$. Similarly, $N \leq 2^{10}$ implies $\log_{M/B} \frac{N}{M} \leq 7$.

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

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

Now, Lemma A.3 gives $B \leq 3N^{1-\varepsilon}/2e$, and using $k \leq N^{1-\varepsilon}$, $N > 3^{2/\varepsilon}$ we get $\ell \geq kN \frac{\log \frac{N}{3N^{1-\varepsilon}}}{2\log(kN)} \geq kN \frac{\varepsilon \log N - \log 3}{2(2-\varepsilon)\log N} \geq kN \frac{\varepsilon/2 \log N}{2(2-\varepsilon)\log N} \geq \frac{\varepsilon kN}{8-4\varepsilon}$.

Otherwise ($\log(kN) < B \log \frac{4M}{B}$), by using $\log \frac{M}{B} \geq 2$ and $e < 4$ (in $eB \leq M$), we get $\ell \geq kN \frac{\log \frac{N}{\max\{3k, 2eB\}}}{2B \log \frac{4M}{B}} \geq \frac{kN}{2B} \cdot \frac{\log \frac{N}{3\max\{k,M\}}}{\log \frac{M}{B} + 2} \geq \frac{kN}{2B} \cdot \frac{-\frac{8}{5} + \log \frac{N}{\max\{k,M\}}}{2 \log \frac{M}{B}}$. Now, we use $N \geq 16B$, 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,

$$\ell \geq \frac{kN}{2B} \cdot \frac{\frac{3}{5} \log \frac{N}{\max\{k,M\}}}{2 \log \frac{M}{B}} \geq \frac{kN}{7B} \cdot \log_{\frac{M}{B}} \frac{N}{\max\{k, M\}} \,.$$

The $\overline{\log}$ in the statement of the theorem is justified by the scanning bound. $\square$

## 7 Lower Bound for Best Case Layout

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 best case matrix layout. In this setting, computing row-sums can be done by a single scan if the matrix is stored in row major layout. Hence, the input vector needs to become part of the lower bound argument, and we will now trace both the movements of input variables and the movements of partial sums while the algorithm is executing.

Intuitively, each input variable $x_j$ needs to disperse to meet and join a partial result $\sum_{j \in S} a_{ij} x_j$ for all $i$ with $a_{ij} \neq 0$, while the partial results created during the run of the program need to be combined to the final output results. The first is a copying process in the forward direction, while the latter may be seen as a copying process in the backwards direction (similar to the analysis in Section 5 and 6). Note that once the algorithm has chosen how to copy the input variables, and at what points in time they should join partial results, the access to the relevant matrix coefficient $a_{ij}$ (for creating the elementary product $a_{ij}x_j$ to be added to a partial result, or form one itself) is basically for free: each matrix coefficient $a_{ij}$ is needed exactly once, and for best case layout, the algorithm can choose to lay out the matrix in access order, making the entire access to matrix coefficients become a single scan.

**Set up** To formalize the intuition above, we use an accounting scheme which encapsulates the internal operations between each pair of consecutive I/Os and abstracts away the specific multiplications and additions, while capturing the creation and movement of partial results and variables. We call the semiring I/O-model extended with this accounting scheme the **separated model**. We now describe the accounting scheme.

In between two I/Os, we define the input variable memory $\mathcal{M}_V \subseteq [N]$, $|\mathcal{M}_V| \leq M$ to be the set of indices of input variables contained in memory right after the preceding I/O, before the internal computation between the I/Os starts, and we define the result memory $\mathcal{M}_R \subseteq [N]$, $|\mathcal{M}_R| \leq M$ to be the set of row indices of the matrix for which some partial sum or coefficient is in memory after the internal computation has taken place, right before the succeeding I/O. We define the **multiplication step** $\mathcal{P} \subseteq \mathcal{M}_R \times \mathcal{M}_V$, representing which coefficients of the matrix $A$ are used in this internal computation, by $(i, j) \in \mathcal{P}$ iff the partial result (only one can be present, cf. Lemma 2.1) 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 memory at the start of this sequence of internal computations. Additionally, we can assume that every coefficient $a_{ij}$ is only used once (cf. Lemma 2.1), and that this use is as early as possible, i.e., at the earliest memory configuration where a copy of variable $x_j$ and the coefficient $a_{ij}$ are simultaneously in memory (if not, another program fulfilling this and using the same number of I/Os is easily constructed). We call the total sequence of such multiplication steps the **multiplication trace**.

Additionally, the separated model splits the disk $D$ into $D_V$ containing the variables present on $D$, and $D_R$ containing the partial results and coefficients present on $D$.

**Identification of the conformation** Since we may assume that all intermediate values are used for some output value (if not, another program fulfilling this and using the same number of I/Os is easily constructed), the multiplication trace of a correct program specifies the conformation of the matrix uniquely. Indeed, the pairs $(i, j)$ in the multiplication trace are exactly the nonzero elements of the matrix. In other words, we have the following lemma.

**Lemma 7.1** *The conformation of the input matrix is determined uniquely by the multiplication trace.*

**Counting traces** We now bound the number of multiplication traces as a function of $\ell$, the number of I/Os performed.

In our accounting scheme, the separated memories $\mathcal{M}_V$ and $\mathcal{M}_R$ are sets. Hence, the possible numbers of different traces during execution for these can by Lemma 4.2 each be bounded by $\left(\left(\frac{4M}{B}\right)^B 4\ell\right)^\ell$, if we think of $\mathcal{M}_V, D_V$ and $\mathcal{M}_R, D_R$ as separate copying computations (the latter when seen as running backwards in time, as follows from the arguments in Section 5).

We analyze the algorithm as choosing a trace for each of these, and for each fixed choice specifying when the multiplications take place. Each fixed choice of traces for $\mathcal{M}_V$ and $\mathcal{M}_R$ makes some set of multiplications $(i, j)$ possible, from which the algorithm during its execution chooses $kN$, thereby determining the multiplication trace and hence the conformation of the matrix, cf. Lemma 7.1. We now bound the number of ways the algorithm can make this choice.

Remember that we assume that every coefficient $a_{ij}$ is used only once, and as early as possible, i.e., when either its corresponding variable or itself has just been loaded into memory and was not present in the predecessor configuration. That is, a multiplication $(i, j)$ can only be specified when both $j \in \mathcal{M}_V$ and $i \in \mathcal{M}_R$ hold, and additionally at least one of them did not hold before the last I/O. Differences in memory configurations stem from I/Os, each of which moves at most $B$ elements, leading to at most $B$ new members of $\mathcal{M}_V$ and at most $B$ new members of $\mathcal{M}_V$ after each I/O. Each of these new members potentially gives $M$ new possible multiplications. Thus, in total over the run of the algorithm, there arise at most $2\ell MB$ possibilities for multiplications. Out of these, the algorithm by its chosen multiplications identifies a subset of size precisely $kN$. Hence the number of possible multiplication traces for the complete computation is at most $\binom{2\ell MB}{kN}$, given a fixed choice of traces for $\mathcal{M}_V$ and $\mathcal{M}_R$.

Combining the above with the fact that the number of distinct conformations of an $N \times N$ $k$-regular matrix is $\binom{N}{k}^N$, we arrive at the following statement:

**Lemma 7.2** *If an algorithm computes the matrix-vector product for all $k$-regular $N \times N$ matrices in the semiring I/O-model with best case matrix layout with at most $\ell = \ell(k, N)$ I/Os, then for $M \geq 4B$ it holds:*

$$\binom{N}{k}^N \leq \left(\left(\frac{4M}{B}\right)^B 4\ell\right)^{2\ell} \cdot \binom{2\ell BM}{kN}.$$

From this, we can conclude the following theorem, which shows that the algorithm presented in Section 3 is optimal up to a constant factor.

**Theorem 7.3** *Assume that an algorithm computes the matrix-vector product for all $k$-regular $N\times N$ matrices in the semiring I/O-model with best case matrix layout with at most $\ell(k,N)$ I/Os. Then for $M \geq 4B$, and $k \leq \sqrt[3]{N}$ there is the lower bound*

$$\ell(k,N) \geq \min\left\{\frac{1}{14}\cdot kN, \frac{1}{8}\cdot\frac{kN}{B}\,\overline{\log}_{\frac{M}{B}}\frac{N}{kM}\right\}$$

**Proof.** The inequality claimed in Lemma 7.2 allows the application of Lemma A.4 as follows: Fix $k$, $N$, and $\ell = \ell(k,N)$. If $\ell > kN/4$, the theorem is trivially true. Hence, we assume $\ell \leq kN/4$. Assume $B \geq 12$, otherwise the first term of the statement is weaker than the scanning lower bound and the theorem is true. This implies $M \geq 4B > 2^4$. Assume further $N \geq 2^{20}$, otherwise $\frac{N}{kM} < 2^{15}$ and hence the logarithmic term (the base $\frac{M}{B}$ is at least 4) is weaker than the scanning bound. To apply Lemma A.4 with $\alpha = \frac{N}{k}$ and $\beta = 18/7$, we have to check the implication $kN \geq (4M/B)^B \Rightarrow \frac{N}{kM} \geq \sqrt[18/7]{N}$, which is the statement of Lemma A.3 (iii), using the bounds on $B$ and $N$. Hence, we conclude

$$\ell \geq \min\left\{\frac{3}{16}\cdot\frac{7}{18}kN, \frac{1}{8}\cdot\frac{kN}{B}\log_{\frac{M}{B}}\frac{N}{kM}\right\}$$

which implies the statement of the theorem by observing that $\frac{7}{16\cdot 6} > \frac{1}{14}$. □

## 8 Lower Bound for Best Case Layout of Vector

We may consider allowing the algorithm to choose also the layout of the input and output vectors. In this section, we show that this essentially does not help algorithms already allowed to choose the layout of the matrix.

**Theorem 8.1** *Assume that an algorithm computes the matrix-vector product for all $k$-regular $N\times N$ matrices in the semiring I/O-model with best case matrix and best case vector layout, using at most $\ell = \ell(k,N)$ I/Os. Then for $M \geq 4B$ and $8 \leq k \leq \sqrt[3]{N}$ it holds that:*

$$\ell(k,N) \geq \min\left\{\frac{1}{43}kN, \frac{1}{16}\cdot\frac{kN}{B}\,\overline{\log}_{\frac{M}{B}}\frac{N}{kM}\right\}$$

**Proof.** We proceed as in the proof of Lemma 7.2 and Theorem 7.3. If there is no layout and program that perform the task in $kN/4$ I/Os, the theorem is trivially true. Hence, we assume $\ell \leq kN/4$. If $B < 35$, then the scanning lower bound implies the statement of the theorem. Similarly, if $N < 2^{16}$ the scanning lower bound justifies the second term. The input vector can be stored in $N!$ different orders. To count how many different set-based states this corresponds to, note that one set-based state of the vector gives rise to $B!^{\frac{N}{B}}$ different orders (by ordering each of the $N/B$ blocks of the vector), which implies that there are $N!/B!^{\frac{N}{B}}$ different set-based states possible for the input vector. This arguments applies also for the output vector. Hence, the following inequality must hold:

$$\binom{N}{k}^N \le \left(\left(\frac{4M}{B}\right)^B kN\right)^{2\ell} \cdot \binom{2\ell BM}{kN} \cdot \left(\frac{N!}{B!^{\frac{N}{B}}}\right)^2 .$$

Rearranging terms and using $(x/y)^y \le \binom{x}{y} \le (xe/y)^y$ and $(x/e)^x \le x! \le x^x$, this yields

$$\left(\frac{N}{k}\right)^{kN} \cdot \left(\frac{B}{eN}\right)^{2N} = \left(\frac{N^{1-2/k}B^{2/k}}{e^{2/k}k}\right)^{kN} \le \left(\left(\frac{4M}{B}\right)^B kN\right)^{2\ell} \cdot \binom{2\ell BM}{kN} .$$

To apply Lemma A.4 with $\alpha = \frac{N^{1-2/k}B^{2/k}}{e^{2/k}k}$ and $\beta = 8$ we have to check the implication $kN \ge (4M/B)^B \Rightarrow \frac{\alpha}{M} \ge \sqrt[8]{N}$, which follows from Lemma A.3 (iv) using that $k \ge 8$ gives $N^{1-2/k} \ge N^{3/4}$. Hence we get

$$\ell \ge \min\left\{\frac{3}{16\cdot 8}kN, \frac{1}{8}\cdot\frac{kN}{B}\log_{\frac{M}{B}}\frac{N^{1-2/k}B^{2/k}}{e^{2/k}kM}\right\} ,$$

where the first term implies the first term of the theorem. It remains to show that under the assumptions made it holds that

$$\log_{\frac{M}{B}}\frac{N^{1-2/k}B^{2/k}}{e^{2/k}kM} \ge \frac{1}{2}\log_{\frac{M}{B}}\frac{N}{kM}$$

which is equivalent to

$$\log_{\frac{M}{B}}\frac{N}{kM} - \frac{2}{k}\log_{\frac{M}{B}}\frac{eN}{B} \ge \frac{1}{2}\log_{\frac{M}{B}}\frac{N}{kM}$$

which is equivalent to

$$\frac{1}{2}\log_{\frac{M}{B}}\frac{N}{kM} \ge \frac{2}{k}\log_{\frac{M}{B}}\frac{eN}{B}$$

which is equivalent to

$$4\log_{\frac{M}{B}}\frac{eN}{B} = 4\left(1 + \log_{\frac{M}{B}}\frac{eN}{M}\right) \le k\log_{\frac{M}{B}}\frac{N}{kM}$$

which is, using $\log_{\frac{M}{B}} e \le 1$, implied by

$$4\left(2 + \log_{\frac{M}{B}}\frac{N}{M}\right) \le k\log_{\frac{M}{B}}\frac{N}{kM}$$

which is because of $\log_{\frac{M}{B}}\frac{N}{M} \ge 8$ (otherwise the lemma holds trivially by the scanning bound) implied by

$$5\log_{\frac{M}{B}}\frac{N}{M} \le k\log_{\frac{M}{B}}\frac{N}{kM}$$

which is equivalent to

$$5 \ln \frac{N}{M} \le k \ln \frac{N}{kM} \,.$$

The derivative with respect to $k$ of the r.h.s. is $\ln \frac{N}{kM} - 1 \ge 0$ (otherwise we again refer to the scanning bound), hence it is sufficient to show that the inequality holds for the smallest claimed $k = 8$:

$$5 \log \frac{N}{M} \le 8 \log \frac{N}{M} - 8 \log 8 \qquad \Leftrightarrow \qquad 8 \log 8 \le 3 \log \frac{N}{M}$$

which holds because $8 \log 8 = 8 \cdot 3 \le 3 \cdot 10$, assuming $\log \frac{N}{M} \ge 10$ (otherwise the lemma holds trivially by the scanning bound). □

# 9 Extensions, Discussion, and Future Work

This section collects ideas and insights that extend the presentation of the earlier sections, but are somehow a distraction from the main flow of thought.

## 9.1 High probability

Having seen the argument of the lower bounds, we observe that the number of programs depends exponentially on the allowed running time. This immediately suggest that allowing somewhat less running time will reduce the number of achievable tasks dramatically, as exemplified in detail by the following theorem.

**Theorem 9.1** *Let $\ell = \ell(k, N, M, B)$ be an upper bound on the number of I/O-operations necessary to compute a uniformly chosen instance of the $N$-$kN$ copy problem (create $kN$ copies of $N$ variables) with probability at least $2^{-kN}$. Then, for $M \ge 4B$ there is the lower bound*

$$\ell \ge \min \left\{ \frac{kN}{4B} \overline{\log}_{M/B} \frac{N}{M} \,, \frac{kN}{6} \right\} .$$

**Proof.** We get the following slightly weaker version of Lemma 4.3

$$2^{-kN} \cdot N^{kN} = \left(\frac{N}{2}\right)^{kN} \le \left( \left(\frac{4M}{B}\right)^{B} 4\ell \right)^{\ell} \cdot B^{kN} .$$

From this the statement of the theorem follows by Lemma 4.4 with $H = kN$, where we again profit from the slightly weaker assumption of Lemma 4.4, when compared to the conclusion of Lemma 4.3. □

## 9.2 Smaller $M/B$

Throughout the sections, we used the assumption $M \ge 4B$ without further comment because it simplified the calculations and lead to reasonable constants in the lower bounds. It is not surprising that we can relax this assumption and only slightly weaken the results.

**Theorem 9.2** *Assume we have for task $T$ a lower bound on the number of I/Os $\ell(k, N, M, B)$ under the assumption $M \geq 4B$. Then under the weaker assumptions $M \geq 2B'$ we have the lower bound for $T$ of $\ell'(k, N, M, B') \geq \frac{1}{2}\ell(k, N, M, \frac{1}{2}B')$.*

**Proof.** A program that works for $B'$ can be simulated to work with track size $B = B'/2$, which doubles the number of I/Os. For this program, we get the lower bound of $\ell$, and we conclude $2\ell' \geq \ell$, hence the statement of the theorem. $\square$

As an example, we formulate the immediate consequence of this and Theorem 7.3, similar statements follow for the other considered tasks.

**Theorem 9.3** *Let $\ell = \ell(k, N)$ be an upper bound on the number of I/O-operations necessary to solve the SpMV problem with worst case layout. Then, for $M \geq 2B$ and $k \leq N/2$ there is the lower bound*

$$\ell(k, N) \geq \min\left\{\frac{1}{28} \cdot kN, \frac{1}{16} \cdot \frac{kN}{B} \log_{\frac{M}{B}} \frac{N}{kM}\right\}$$

**Proof.** Remember the bound given in Theorem 7.3 for $M \geq 4B$:

$$\ell(k, N) \geq \min\left\{\frac{1}{14} \cdot kN, \frac{1}{8} \cdot \frac{kN}{B} \log_{\frac{M}{B}} \frac{N}{kM}\right\}$$

Changing to $B/2$ increases the base of the logarithmic term, such that this term reduces its value in the worst case to half its original value. This is counterbalanced by the $\frac{kN}{B}$ term doubling. Hence, the overall effect on the lower bound is that it is weakened by a factor $1/2$ as claimed. $\square$

## 9.3 Discussion of the model of Computation

Another point worth of discussion is the model of computation used in the lower bound proofs. This model might seem somewhat arbitrary, and hence we want to justify some of our modeling choices in this section.

Ideally, we would like to make as few algebraic assumptions as possible. In our context, some assumptions are necessary: If we would, for example, consider a model that has integer numbers and arbitrary operations, then it would be possible to encode a pair of numbers into a single number, and the notion of limited space would be meaningless, and all computational tasks would require precisely scanning the input and scanning the output.

The model we used 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 [14] that relies upon subtraction and with this achieves a better running than is possible in the semiring model, as argued by Hong and Kung [9].

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-model have been considered before, mainly in the context of matrix-matrix multiplication, in [16], implicitly in [15], and also in [9]. 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 [11].

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 the non-uniformity of 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, and standard address arithmetic.

A possible apparent strengthening of the model of computation is to allow comparisons. More precisely, we could assume that the elements of the semiring cannot only be added and multiplied, but there is additionally a total order on the semiring, i.e., that two elements can be compared ($\leq$). For the real numbers $\mathbb{R}$, this comparison could coincide with the usual "not smaller than" ordering. A program for the machine then consists of a tree $T_p$, with 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 (read, write or compute in internal memory) the node represents. A comparison amounts to checking whether a non-trivial polynomial of the data present in the memory is greater than zero, equal to zero, or smaller than zero. 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. We measure the performance of $T_p$ on input $x$ by counting the number of read and write nodes encountered when following $T_p$ on input $x$.

It would perhaps be reasonable to assume that $T_p$ is finite, but this might exclude interesting programs that perform complicated computations in internal memory and still use only few I/Os. Hence, we allow $T_p$ to be infinite as long as every allowed input leads to a finite computation, i.e., a path in $T_p$ ending at a leaf.

The following lemma shows that all this apparent additional computational power is in fact not usable.

**Lemma 9.4** *Assume that a semiring comparison I/O-tree $T_p$ computes a polynomial $p : \mathbb{R}^d \to \mathbb{R}^e$ with worst-case $\ell$ I/Os using comparisons ($\leq$), then there is a semiring comparison I/O-tree $T'_p$ without any comparisons computing $p$ with at most $\ell$ I/Os.*

**Proof.** Every node $v$ of $T_p$ is reached by a certain set $S_v \subseteq \mathbb{R}^d$ of inputs. It is possible that $T_p$ is infinite, but every $x \in \mathbb{R}^d$ must lead to a leaf of $T_p$ (on a finite path).

We start by picking an arbitrary point $x_0 \in \mathbb{R}^d$, set $\varepsilon_0 = 1$, and start the computation at the root $n_0$ of $T_p$. We define a sequence of $x_i$'s by the following rule: As long as the open ball $B_i$ of radius $\varepsilon_i$ around $x_i$ behaves in the same way by the comparison at node $n_j$ (or this node is not a comparison node), we set $n_{j+1}$ to the corresponding child of $n_j$. If it is a comparison, the $<$ and $>$ outcomes are defined by open sets (defined by some polynomial in the input variables being positive or negative), and one of them must have non-empty intersection $O$ with $B_i$. In this case, we choose $x_{i+1}$ and $\varepsilon_{i+1}$ such that the ball $B_{i+1}$ is contained in $O$, and $0 < \varepsilon_{i+1} \leq \varepsilon_i/2$. This construction must lead to a leaf $w$ of $T_p$: Otherwise, the $x_i$'s form a Cauchy-sequence and hence converge to an input vector $x$, and for this $x$ the computation path would be the infinite sequence of the $n_j$, contradicting the definition that every input must lead to a leaf.

The new tree $T'_p$ is induced by following the path in $T_p$ to $w$, and leaving out all the comparisons. The result of $T'_p$ is given by a polynomial $q$ over the input variables, and we have to show $p = q$. This is true because $p$ and $q$ evaluate to the same number in an open subset of $\mathbb{R}^d$ around $x = x_i$. Hence for any $y \in \mathbb{R}^d$ different from $x$, the single-variable polynomials $p'$ and $q'$ that map the line through $x$ and $y$ to the value given by $p$ and respectively $q$ have infinitely many equality points and are equal. By the fundamental theorem of algebra we can conclude $p(y) = q(y)$, which means that $T'_p$ is correct. $\square$

## 9.4 Context 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.

# Appendix

## A Technical Lemmas

**Lemma A.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. $\square$

**Lemma A.2** *Let $b \geq 2$ and $s, t > 0$. For all positive real numbers $x$, we have $x \geq \frac{\log_b(s/x)}{t}$ implies $x \geq \frac{1}{2} \frac{\log_b(s \cdot t)}{t}$.*

**Proof.** If $s \cdot t \leq 1$, the implied inequality holds trivially. Hence, in the following we assume $s \cdot t > 1$. 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 A.1. This contradiction to the assumed upper bound on $x$ establishes the lemma. $\square$

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

*(i)* $N \geq 2^{20}$ *implies* $B \leq \frac{4}{e}\sqrt[6]{N}$.
*(ii)* $0 < \varepsilon < 1$, $N \geq 2^{2/(1-\varepsilon)^2}$ *implies* $B \leq \frac{3}{2e} N^{1-\varepsilon}$.
*(iii)* $N \geq 2^{20}$, $B \geq 12$, *and* $k \leq \sqrt[3]{N}$ *implies* $\frac{N}{keM} \geq \sqrt[18/7]{N}$.

*(iv)* $N \geq 2^{16}$, $B \geq 35$, *and* $k \leq \sqrt[3]{N}$ *implies* $\frac{N^{3/4}}{ekM} \geq \sqrt[8]{N}$.

**Proof.** By $M \geq 2B$, we have $\log \frac{4M}{B} \geq \log 8 = 3$. Thus $B \leq \frac{\log kN}{\log \frac{4M}{B}} \leq \frac{2}{3} \log N \frac{6}{e} \sqrt[6]{N}$. for $N \geq 2^{20}$, which proves (i), by the additionally observation that $\log 2^{20} \leq \frac{6}{e}\sqrt[6]{2^{20}}$.

Similarly follows statement (ii): It is clear, that once the lemma is true for some $N > 1$, it holds for all $N$. Hence, it remains to show the lemma for $N = 2^{2/(1-\varepsilon)^2}$. The important inequality certainly holds for $\varepsilon = 0$, which is sufficient if the derivative with respect to $\varepsilon$ is positive. To see this, we check that the derivative evaluates positively at 0 and compute the second derivative, which can easily be seen to be positive in the considered range.

To see (iii), we rewrite the main assumption as $(kN)^{1/B} \geq 4M/B$. With $B \geq 12$, $k \leq \sqrt[3]{N}$, and (i) we get: $eM \leq \frac{e}{4} B (kN)^{1/B} \leq \frac{e}{4} \frac{4}{e} \sqrt[6]{N} (kN)^{1/12} \leq N^{\frac{1}{6} + \frac{4}{3} \cdot \frac{1}{12}} = N^{\frac{5}{18}}$. Hence $\frac{N}{keM} \geq \sqrt[18/7]{N}$.

To see (iv), we rewrite the main assumption as $(kN)^{1/B} > 4M/B$. With $B \geq 35$, $k \leq \sqrt[3]{N}$, and (i) we get: $M \leq \frac{1}{4} B (kN)^{1/B} \leq \frac{2}{9} \sqrt[4]{N} (kN)^{1/35} \leq \frac{2}{9} N^{\frac{1}{4} + \frac{4}{3} \cdot \frac{1}{35}} \leq \frac{1}{3} N^{51/140}$. Hence $\frac{N^{3/4}}{keM} \geq \frac{3N^{3/4}}{\sqrt[4]{N} e N^{51/140}} \geq N^{\frac{3}{4} - \frac{1}{4} - \frac{51}{140}} \geq N^{\frac{19}{140}} \geq \sqrt[8]{N}$.

$\square$

**Lemma A.4** *Assume $M \geq 4B$, $kN/B \leq \ell \leq kN$, $\alpha > 0$, $\beta > 0$, $k \leq \sqrt[3]{N}$, and that $kN \geq (4M/B)^B$ implies $\frac{\alpha}{M} \geq \sqrt[\beta]{N}$. Then*

$$\alpha^{kN} \leq \left( \left( \frac{4M}{B} \right)^B kN \right)^{2\ell} \cdot \binom{2\ell BM}{kN}$$

*implies*

$$\ell \geq \min \left\{ \frac{3}{16\beta} kN, \frac{1}{8} \cdot \frac{kN}{B} \log_{\frac{M}{B}} \frac{\alpha}{M} \right\} .$$

**Proof.** Using $(x/y)^y \leq \binom{x}{y} \leq (xe/y)^y$ we arrive at

$$kN \log \alpha \leq 2\ell \left( \log(kN) + B \log \frac{4M}{B} \right) + kN \log \frac{2e\ell BM}{kN}$$

by taking logarithms. Rearranging terms yields

$$\frac{2\ell}{kN} \geq \frac{\log \frac{kN}{2\ell} \frac{\alpha}{eBM}}{\log(kN) + B \log \frac{4M}{B}} .$$

We take the statement of Lemma A.2 with $s := \frac{\alpha}{eBM}$, $t := \log(kN) + B \log \frac{4M}{B}$, and $x := 2\ell/kN$, and conclude

$$\frac{2\ell}{kN} \geq \frac{\log(st)}{2t} \geq \frac{\log \frac{\alpha}{M}}{2t} .$$

Now, we distinguish according to the leading term of $t$. If $\log(kN) < B \log \frac{4M}{B}$, then we get

$$\frac{2\ell}{kN} \geq \frac{\log \frac{\alpha}{M}}{2B \log \frac{4M}{B}} = \frac{\log \frac{\alpha}{M}}{2B(2 + \log \frac{M}{B})} \geq \frac{\log \frac{\alpha}{M}}{2B(2 \cdot \log \frac{M}{B})} ,$$

where the last inequality follows from $\log \frac{M}{B} \geq 2$. Hence, in this case, the statement of the lemma follows.

Otherwise, if $\log(kN) \geq B \log \frac{4M}{B}$, from the assumptions $\alpha/M \geq \sqrt[\beta]{N}$ follows, and hence

$$\frac{4\ell}{kN} \geq \frac{\frac{1}{\beta} \log N}{\log kN} \geq \frac{\frac{1}{\beta} \log N}{\frac{4}{3} \log N} = \frac{3}{4\beta}$$

from which the lemma follows. $\square$

## 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 and Visualization*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 139–159. American Mathematical Society, Providence, RI, 1999.

[3] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. 35th Annual ACM Symposium on Theory of Computing (STOC)*, pages 307–315, San Diego, CA, 2003. ACM Press.

[4] 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.

[5] 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.

[6] 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), 2005.

[7] 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.

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

[9] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. 13th Annual ACM Symposium on Theory of Computing (STOC)*, pages 326–333, New York, NY, USA, 1981. ACM Press.

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

[11] 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, 2004. ACM Press.

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

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

[14] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

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

[16] J. S. Vitter. External memory algorithms and data structures. In J. M. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–38. American Mathematical Society Press, Providence, RI, 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, 15 March 2006.

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