T U M

I N S T I T U T F Ü R I N F O R M A T I K

Evaluating Non-Square Sparse Bilinear Forms on Multiple Vector Pairs in the I/O-Model

Gero Greiner Riko Jacob

TUM-I1015
Oktober 10

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-10-I1015-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Evaluating Non-Square Sparse Bilinear Forms on Multiple Vector Pairs in the I/O-Model

Gero Greiner and Riko Jacob

Technische Universität München

Abstract. We consider evaluating one bilinear form defined by a sparse $N_y \times N_x$ matrix $\mathbf{A}$ having h entries for w pairs of vectors. The model of computation is the semiring I/O-model with main memory size M and block size B. For a range of low densities (small h), we determine the I/O-complexity of this task for all meaningful choices of N_x, N_y, w, M and B, as long as $M \geq B^2$ (tall cache assumption). To this end, we present asymptotically optimal algorithms and matching lower bounds. Moreover, we show that this has essentially the same worst-case I/O-complexity as multiplying the matrix $\mathbf{A}$ with w vectors.

1 Introduction

We consider the problem of computing w scalars $z^{(i)} = {\mathbf{y}^{(i)}}^T \mathbf{A}\mathbf{x}^{(i)}$, $1 \leq i \leq w$, where $\mathbf{A}$ is a sparse $N_y \times N_x$ matrix with h non-zero entries, and all $\mathbf{x}^{(i)}$ and $\mathbf{y}^{(i)}$ are (dense) vectors. This is highly related to the matrix vector products $\mathbf{A}\mathbf{x}^{(i)}$, $1 \leq i \leq w$, and we show that the complexity of both tasks in our model only differs in reading the matrix up to w times. The evaluation of bilinear forms and matrix vector products are important tasks in computational science: fields of application include scientific computing, iterative linear system solvers, least-squares problems, eigenvalue problems, data mining and web search. While, from a traditional point of view, bilinear form and matrix vector product are easily obtained with a number of multiplications equal to the number of matrix entries, the sparseness sometimes induces irregular access patterns that lead to situations where memory access becomes the bottleneck of computation. Empirical studies show that for the naive algorithm, CPU-usage can be as low as 10% [6,9].

One way of dealing with this problem is the construction of algorithms where, instead of CPU-cycles, the movement of data between layers of the memory hierarchy is optimised. In this paper, we use a slight modification of the I/O-model [1], the semiring I/O-model with the parameters M and B, denoting the memory size, and the size of a block, see Section 2 for details. In this model, Bender, Brodal, Fagerberg, Jacob and Vicari [2,3] determined the I/O-complexity of computing the sparse matrix vector product for square matrices. These results are generalised here to the case of non-square matrices. Furthermore, we extend these results to the evaluation of multiple products, i.e., the matrix vector products of multiple vectors with the same matrix. Considering the evaluation of matrix vector products on multiple vectors is a step towards closing the gap

between sparse matrix vector multiplication and sparse matrix dense matrix multiplication since the set of w vectors $\mathbf{x}^{(i)}$ constitutes a dense $N_x \times w$ matrix $\mathbf{X}$.

The hardness of computing a bilinear product directly implies a similar hardness for computing a matrix vector product because an algorithm for the matrix vector product can be used to derive the bilinear form directly. The converse can also be made precise: one can extract from an I/O-efficient algorithm for the bilinear product a similarly I/O-efficient algorithm for the matrix vector product. This is used to simplify some expositions.

Related work Evaluating the matrix vector product $\mathbf{Ax}$ for an $N \times N$ matrix $\mathbf{A}$ with kN entries has been investigated in [3]. They show that the I/O-complexity of this task for matrices in the so called column major layout, as will be defined in Section 2, is $\Theta\left(\min\left\{\frac{kN}{B}\,\overline{\log}_{\frac{M}{B}}\,\frac{N}{\max\{M,k\}}, kN\right\}\right)$[1], and for a layout chosen by the program it is $\Theta\left(\min\left\{\frac{kN}{B}\,\overline{\log}_{\frac{M}{B}}\,\frac{N}{Mk}, kN\right\}\right)$.

The multiplication of two dense matrices has been examined by Hong and Kung in [5], showing a bound of $\Theta\left(\frac{N^3}{B\sqrt{M}}\right)$ on the number of I/Os. Very recently, in [4] the multiplication of a sparse matrix with a dense matrix was considered. For sparse $N \times N$ matrices with kN entries, it is shown that for certain ranges of k, the performance can be increased by finding denser than average submatrices of $\mathbf{A}$. More precisely, a submatrix does not have to consist of a consecutive part of the matrix, but is defined by the entries a_{ij} of $\mathbf{A}$ that belong to both, a set of row indices S_r and a set of column indices S_c. It was proven that such submatrices exist with a certain density greater or equal to a threshold δ, and that there are matrices without any submatrices of higher density than δ.

The evaluation of bilinear forms in the I/O-model has been considered as an optimisation problem in [7]. There it has been shown that an optimal program for the evaluation of matrix vector products is $\mathcal{NP}$-hard to find, even for $B = 1$. In [8], the I/O-complexity of evaluating the bilinear form for a (non-square) $N_y \times N_x$ matrix with h entries that form a diagonal band, i.e., entries are only placed near the diagonal, is determined to be $\Theta\left(\frac{h}{BM} + \frac{N_x+N_y}{B}\right)$.

Our results In this work, we consider the case where the number of entries for almost all submatrices in $\mathbf{A}$ is proportional to the number of rows and columns of the submatrix. This is possible if the average number of entries per column in $\mathbf{A}$ is some $k \leq \frac{N_y}{M^{1-\epsilon}N_x^{\epsilon}}$ with constant $\epsilon > 0$. For such k, a modification of the proof of [4] shows that the I/O-complexity of $\mathbf{AX}$ for any $w \geq B$ is $\Theta\left(\frac{wh}{B}\right)$. This will be stated in Section 6.3. As a lower bound, the proof directly extends to the case of multiplying a sparse matrix with w vectors, even if the program is allowed to choose the layout of the vectors.

The case of $w \leq B$ is examined here in detail, forming a bridge between the results of [3] and [4]. For matrices of the described density ($h/N_x \leq \frac{N_y}{M^{1-\epsilon}N_x^{\epsilon}}$), we cover all dimensions of $\mathbf{A}$, and parameters B, M and $w \leq B$. However, for all

[1] $\overline{\log}_b(x) := \max\{\log_b(x), 1\}$

other choices of h, we present upper bounds in form of algorithms. For certain cases where B/w is small the algorithms are indeed optimal for all ranges of h.

Furthermore, we show that evaluating the bilinear form has almost the same I/O-complexity as multiplying vectors with the same matrix.

Theorem 1. *Given a matrix* $\mathbf{A}$ *with fixed layout and fixed parameters* M *and* B*. Evaluating* w *bilinear forms with* $\mathbf{A}$ *has the same semiring I/O-complexity as evaluating the matrix vector product of* $\mathbf{A}$ *with* w *vectors if at least* $\ell = \Omega\left(\frac{hw}{B}\right)$ *I/Os are required.*

This result is explained in Section 4 and allows us to extend the results from [3] to matrices in row major layout, i.e., where the entries are given in external memory in a consecutive ordering by their row index, and their column index to break ties. Moreover, our main results hold for bilinear forms and matrix vector products, both, in column major and row major layout. Note that for Theorem 2, the dimensions N_x and N_y have to be swapped if $\mathbf{A}$ is given in row major layout.

For the complexity of the task, we are going to prove the following theorems depending on the layout of the matrix $\mathbf{A}$. Similar to [4], our bounds are only asymptotically tight if we assume a tall cache, i.e., $M \geq B^2$. However, this is only necessary to transpose $\mathbf{X}$ for some of the presented algorithms.

Our results for the case that the matrix $\mathbf{A}$ is stored in column major layout, are given in the following theorem.

Theorem 2. *Given an* $N_y \times N_x$ *matrix* $\mathbf{A}$ *with* h *entries in column major layout and parameters* M*,* B*. Assume* $M \geq 4B$*,* $h/N_x \leq N_y^\epsilon$*, and* $\log N_x \leq N_y^\epsilon$ *for some* $0 < \epsilon < 1$*. Then, evaluating* w *bilinear products with* $\mathbf{A}$ *has (worst-case) complexity in the semiring I/O-model*

$$\Theta\left(\min\left\{h\,,\,\frac{h\log N_y}{\log N_x}\,,\frac{h}{B}\log_{\frac{M}{B}}\min\left\{\frac{N_y}{M},\frac{N_xN_y}{h}\right\}+\frac{hw}{B}\log_{\frac{M}{B}}\frac{N_xN_y}{hM}\right\}\right)$$

unless this term is asymptotically smaller than $\frac{hw}{B}$*.*

The lower bounds are obtained by a modification of the proof in [3] for a single matrix vector multiplication, also keeping track of the different matrix dimensions. Additionally, the lower bounds of Theorem 3 apply which yields the second term of the sum.

On the algorithmic side, for very asymmetric matrices $\mathbf{A}$, where the number of columns is much higher than the number of rows, building a table of tuples of elements of the $\mathbf{y}$ vectors can be superior to the direct algorithm. The direct algorithm simply scans over $\mathbf{A}$ and loads for each a_{ij} the corresponding vector elements $x_j^{(1)},\ldots,x_j^{(w)}$ and $y_i^{(1)},\ldots,y_i^{(w)}$ to create products.

In [3], an algorithm is presented based on sorting the matrix entries to build row sums. This sorting approach can be used to initially change the layout of $\mathbf{A}$ to the best-case layout. Then, the sorting algorithm for best-case layout can be applied for one vector pair after another.

For the best-case layout, i.e., if the algorithm is allowed to choose the layout of the matrix, the following theorem holds.

Theorem 3. *Given an $N_y \times N_x$ matrix $\mathbf{A}$ with h entries in best-case layout and parameters M, B. Assume $M \geq 4B$, $w \leq \frac{B}{20}$, and $h/N_x \leq \sqrt[6]{N_y}$ for $N_y \leq N_x$. Then, evaluating w bilinear products with $\mathbf{A}$ has (worst-case) complexity in the semiring I/O-model*

$$\Theta\left(\min\left\{h\,,\, h\frac{\log\left(\frac{N_x N_y \log N_x}{BhM}\right)}{\log N_x}\,,\, \frac{hw}{B}\log_{\frac{M}{B}}\frac{N_x N_y}{hM}\right\}\right)$$

unless this term is asymptotically smaller than $\frac{hw}{B}$.

The algorithms are similar to the ones for column major layout. The second term of the complexity is met by an algorithm that creates tuples of vector elements in the beginning, and then reads the matrix in tiles. The last term of the minimum is a simple extension of the sorting algorithm in [3].

The lower bound instead has to be derived in a slightly different manner. While one can simply adapt the lower bound of [3] to non-square matrices, considering multiple vectors makes it more difficult. To tackle this, only the vector $\mathbf{x}^{(i)}$ that contributes the least I/O volume is considered.

All our algorithms require at least hw/B I/Os. In contrast, we do not know a corresponding lower bound that would hold for all choices of the parameters. However, if the dimensions are polynomially bounded in each other (which is expressed using a condition on the density in the lemma), the results of [4] can be extended to obtain a lower bound of $\Omega(hw/B)$ by the following lemma.

Lemma 1. *Let $\mathbf{A}$ be a sparse $N_y \times N_x$ matrix with h non-zero entries for $N_x \geq N_y$. For an average number of entries per column $h/N_x \leq \frac{N_y}{M^{1-\epsilon}N_x^{\epsilon}}$ with constant $\epsilon > 0$, evaluating w bilinear products over a semiring requires $\Omega\left(\frac{hw}{B}\right)$ I/Os unless $h < \frac{32M}{\epsilon w} + 2M$.*

The proof of this lemma can be found in Section 6.3. Hence, for a $N_y \times N_x$ matrix $\mathbf{A}$ with $h = \omega(M)$ non-zero entries, for $N_x \geq N_y$ but $N_x^{2\epsilon} \leq N_y$, and average number of entries per column $h/N_x \leq \sqrt{N_y}/M$, a lower bound of $\Omega(hw/B)$ is given.

1.1 Outline

The outline of the paper is as follows: In Section 2, the model of computation is introduced along with the terminology used in this paper. This is followed by the necessary (technical) lemmas used in the proofs. In Section 4, it is proven that bilinear forms and matrix vector product have essentially the same complexity. The main results are then proven by providing (optimal) algorithms for upper bounding the complexity in Section 5 and up to constant factors matching lower bounds in Section 6.

2 Model of Computation

We use a combination of the I/O-model described in [1] and the model used in [5], the so called *semiring I/O-model* introduced by Bender, Brodal, Fagerberg, Jacob and Vicari in [3]. It models two layers of memory hierarchy, namely a slow memory of unbounded size and a fast limited memory where calculations can be performed. Programs are assumed to work for an arbitrary semiring.

Definition [3] The *semiring I/O-machine* consists of an *internal memory* which can hold up to M elements at a time, and an *external memory* (aka disk) of unbounded size which is organised in *blocks* (aka tracks) of B elements. The numbers constituting input, output and intermediate results are assumed to belong to a *commutative semiring* $\mathcal{S} = (\mathbb{S}, +, \cdot)$, i.e., a set $\mathbb{S}$ with operations addition $(+)$ and multiplication $(\cdot)$ that are associative, commutative, and multiplication is distributive over addition. Further, there is a neutral element 0 for addition, 1 for multiplication and 0 is annihilating with respect to multiplication. In contrast to rings and fields, inverse elements are neither guaranteed for addition nor for multiplication, i.e., there is no subtraction or division. The current *configuration* of a machine is described by the content $\mathcal{M} = (m_1, \ldots, m_M)$, $m_i \in \mathbb{S}$ of internal memory, and an infinite sequence of blocks $t_i \in \mathbb{S}^B$, $i \in \mathbb{N}$ representing external memory. The positions in internal and external memory will be denoted as *cells*. An *operation* is a transformation of one configuration into another, which can be one of the following types

- *Computation*: perform either an algebraic operation $m_i := m_j + m_k$, $m_i := m_j \cdot m_k$, set $m_i := 1$, set $m_i := 0$ (deletion) or assign $m_j := m_i$ (copying).
- *Input*: move the elements of block t_i from external memory to elements $m_{i_1}, \ldots, m_{i_B}$ in $\mathcal{M}$ for arbitrary $i \in \mathbb{N}$, $i_1, \ldots, i_B \in [M]$. Before the operation, the elements in internal memory have to be $m_{i_1} = \cdots = m_{i_B} = 0$, and after the operation, the block t_i is empty.
- *Output*: move elements $m_{i_1}, \ldots, m_{i_B}$ of $\mathcal{M}$ to block t_i for arbitrary $i \in \mathbb{N}$, $i_1, \ldots, i_B \in [M]$. Before the operation, the block t_i has to be empty, and after the operation, $m_{i_1} = \cdots = m_{i_B} = 0$ holds.

To the latter two, we also refer as *I/O operations* or simply *I/Os*.

Using this, we define a *program* P as a finite sequence of operations. The number of I/O operations describes the I/O costs of P. For the lower bounds we use the non-uniform notion that an *algorithm* is a family of programs where the program can be chosen according to the parameters underlying the problem. In particular, for matrix multiplications, the parameters are the dimensions, the sparseness, the *conformation* of the matrix $\mathbf{A}$, i.e., the position of the non-zero entries in $\mathbf{A}$, the memory size M and the block size B.

The input of a program is specified by the *input variables* $x_j^{(i)}$, a_{ij} and $y_j^{(i)}$, with $1 \leq i \leq w$, $1 \leq j \leq N_x$, $1 \leq k \leq N_y$. If an element p is operand of an algebraic operation with result p', we call p a *predecessor* of p' and p' a *successor* of p. We extend this notation to the transitive closure such that any element p used to derive p' after several operations is called predecessor and p' is its

successor. By the non-uniform nature of an algorithm, we can assume that all intermediate results are predecessors of an output value (also called final result). The requirement that a program must work for an arbitrary semiring allows us to describe any intermediate result as a polynomial over input variables. These polynomials can be normalised to be a sum of monomials, i.e., products of variables multiplied by a natural number. The absence of inverse elements induces that the set of variables of the polynomial describing p is a subset of the variables of all successors. Similarly, if some product $a \cdot b$ is part of one of the monomials of p, then this will be the case for all successors of p.

We use the notation $[N] = \{1, \ldots, N\}$ in order to classify all intermediate results in the following. Products of the form $a_{jk}x_k^{(i)}$, $y_j^{(i)}a_{jk}$, $x_k^{(i)}y_j^{(i)}$ and $y_j^{(i)}a_{jk}x_k^{(i)}$, for $1 \leq i \leq w$, $j \in [N_x]$, $k \in [N_y]$, are referred to as *elementary products*. Sums of the form $\sum_{k \in S_x} a_{jk}x_k^{(i)}$, $\sum_{j \in S_y} y_j^{(i)}a_{jk}$, and $\sum_{j \in S_y}\sum_{k \in S_x} y_j^{(i)}a_{jk}x_k^{(i)}$, with $1 \leq i \leq w$, $S_x \subseteq [N_x]$ and $S_y \subseteq [N_y]$, are called *partial sums*. Altogether, the term *canonical partial result* refers to any of these forms. In the case of a matrix vector multiplication only the first such form can arise as detailed in [3]. For the case of evaluating a bilinear form, the distributive law can be useful, and some additional arguments are necessary. First, observe that no intermediate result that uses input from different pairs of vectors can be useful. Further, monomials with coefficient > 1 or with more than one $x_k^{(i)}$, $y_j^{(i)}$, or a_{jk}, are useless. The same is true if there is a mismatch in row or column between matrix coefficient and vector element. Hence, all monomials must be elementary products. Further, any polynomial with at least two monomials that are elementary products of different type will continue to have this property and are disallowed. Also a non-trivial sum of at least monomials of type $x_k^{(i)}y_j^{(i)}$ is useless because it eventually has to be multiplied with some a_{jk} which would lead to some mismatching monomials.

Since we can assume that every program requires at least one I/O, when writing complexity using $\mathcal{O}$, Θ, or Ω at least 1 is meant.

3 Math

The observations and lemmas in this section can be found, or are extensions of lemmas in [3] and [4].

Observation 1 *For $0 \leq x \leq 1/2$, it holds that $\ln(1-x) \geq -2x$.*

Proof. Consider $f(x) = \ln(1-x) + 2x$. Observe $f(0) = 0$ and $f'(x) = \frac{-1}{1-x} + 2 = \frac{1}{x-1} + 2 \geq \frac{-1}{1/2} + 2 = 0$. Hence, $f(x) \geq 0$ for $0 \leq x \leq 1/2$. □

Observation 2 *For $0 < a \leq e$, for any $x > 0$ it holds $x \geq a \ln x$.*

Proof. The first derivation of $f(x) = x - a\ln x$ is $f'(x) = 1 - a/x$. Hence, $x = a$ is an extremal point with function value $f(a) = a - a\ln a \geq 0$ since $\ln a \leq 1$. The second derivation yields $f''(x) = 2a/x$ which is strictly positive for all values of $a, x \geq 0$. Thus, the extremal point is a minimum. □

Observation 3 *For $x \geq y \geq 1$ it holds*

$$\left(\frac{x}{y}\right)^y \overset{(a)}{\leq} \binom{x}{y} \overset{(b)}{\leq} \left(\frac{ex}{y}\right)^y.$$

Proof. Inequality (a) is obvious since $\frac{x-i}{y-i} \geq \frac{x}{y}$ holds for all $i \geq 0$. Inequality (b) is given by Stirling's Inequality weakened to $y! \geq \left(\frac{y}{e}\right)^y$ and $\frac{x!}{(x-y)!} \leq x^y$. □

Observation 4 *For $n \geq k \geq a \geq 1$ it holds*

$$\binom{n}{k} \geq \left(\frac{n-k}{k}\right)^a \binom{n}{k-a}.$$

Proof. By definition of binomial coefficients

$$\binom{n}{k} \cdot \binom{n}{k-a}^{-1} = \frac{n!(n-k+a)!(k-a)!}{(n-k)!k!n!} = \prod_{i=1}^{a} \frac{n-k+i}{k-a+i} \geq \left(\frac{n-k}{k}\right)^a.$$

□

Observation 5 *For $D, f, x \geq 0$, the inequality $D \ln fD > x$ is fulfilled if $D > \frac{2x}{\ln 2xf}$.*

Proof. Substituting D yields

$$D \ln fD > \frac{2x}{\ln 2xf} \ln\left(f \frac{2x}{\ln 2xf}\right) \geq \frac{2x}{\ln 2xf} \ln \sqrt{2xf} = x$$

where we use $\sqrt{2xf} \geq 2 \ln \sqrt{2xf}$ given by Observation 2. □

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

Proof. See Lemma B.2 in [3].

Lemma 3. *Assume $2 \log kN_x \geq \frac{B}{w} \log \frac{6M}{B}$, $N_y \leq N_x \leq N_y^2$, $k \leq \sqrt[6]{N_y}$, $B \geq 2$, $1 \leq w \leq \frac{B}{20}$ and $M \geq B^2$. Then for $N_x \geq 2^{30}$, it holds $\frac{wN_y}{eBkM} > \sqrt[15]{N_y}$.*

Proof. By $M \geq B^2$ and $B \geq 2$, we have $\log \frac{6M}{B} \geq \log 12 > 3$. Thus $\frac{B}{w} \leq \frac{2 \log kN_x}{\log \frac{6M}{B}} \leq \frac{4}{3} \log N_x \leq \frac{4}{3} \sqrt[6]{N_x}$ for $N_x \geq 2^{30}$.

To see the claim, we rewrite the first assumption as $(kN_x)^{2w/B} \geq \frac{6M}{B}$. With $\frac{B}{w} \geq 20$, and $k \leq \sqrt[6]{N_y}$ we get: $\sqrt{M} \leq \frac{M}{B} \leq \frac{1}{6}(kN_x)^{2w/B} \leq \frac{1}{6} N_y^{13/60}$. Hence, together with $\frac{B}{w} \leq \frac{4}{3} \sqrt[6]{N_x} \leq \frac{4}{3} \sqrt[3]{N_y}$, we have $\frac{wN_y}{eBkM} \geq \frac{27 N_y^{2/3}}{e \sqrt[6]{N_y} N_y^{\frac{13}{30}}} \geq \sqrt[15]{N_y}$.

□

4 Transformations

In this section, we discuss how a program that evaluates bilinear forms can be transformed into one that computes matrix vector multiplications. In this, the operations that create elementary products play an important role. We say that an operation (it must be a multiplication) creates an elementary product, if the elementary product is one of the monomials of the result, but not part of any monomial in the direct predecessors. One multiplication can create many elementary products, but the total number of such operations is limited by the total number of elementary products. This relies upon the following Lemma, which is easy to prove.

Lemma 4. *In a normalised semiring I/O program evaluating a bilinear form on multiple vector pairs, no elementary product is created twice.*

Proof. In a normalized program, every elementary product that is ever created will become part of one of the w final results. If two intermediate results that both contain the same elementary product are added or multiplied, the result is useless. Hence, no correct normalized program can produce the same elementary product twice. □

For the proof of Theorem 1, we present transformations in both directions in the following lemmas. Note that the layout of the matrix $\mathbf{A}$ is not of concern, it just has to be the same for both tasks.

Lemma 5. *If the matrix vector products $\mathbf{A}\mathbf{x}^{(i)}$ for $1 \leq i \leq w$ can be computed for an arbitrary semiring with ℓ I/Os, then the bilinear forms ${\mathbf{y}^{(i)}}^T\mathbf{A}\mathbf{x}^{(i)}$ can be evaluated with at most $2\ell + 1$ I/Os.*

Proof. For each single vector pair $\mathbf{x}^{(i)}, \mathbf{y}^{(i)}$, the bilinear product is computed by multiplying $\mathbf{y}^{(i)}$ with the corresponding result vector $\mathbf{c}^{(i)} = \mathbf{A}\mathbf{x}^{(i)}$. Thus, one additional scan of the blocks of $\mathbf{y}^{(i)}$ that corresponds to the non-zero entries of $\mathbf{c}^{(i)}$ suffices. Because the algorithm at least wrote each $\mathbf{c}^{(i)}$, this scanning of $\mathbf{y}^{(i)}$ for each $1 \leq i \leq w$ takes certainly no more than an additional ℓ I/Os. Together with writing the results, the bilinear forms are evaluated with at most $2\ell + 1$ I/Os. □

Lemma 6. *If the bilinear forms ${\mathbf{y}^{(i)}}^T\mathbf{A}\mathbf{x}^{(i)}$, $1 \leq i \leq w$ can be evaluated in the semiring I/O-model with internal memory size M and block size B using ℓ I/Os, then the w products $\mathbf{c}^{(i)} = \mathbf{A}\mathbf{x}^{(i)}$ can be computed using $2\ell + \lceil 4wh/B \rceil$ I/Os with internal memory size $M + B$ and block size B.*

Proof. Let P be a program to evaluate the w bilinear forms using ℓ I/Os. By Lemma 4, there is a program $\hat{P}$ for the same task, which computes only canonical partial results and avoids producing an elementary product more than once, with at most ℓ I/Os.

We can then use $\hat{P}$ to construct a program for the matrix vector products. This construction is based on the following idea. During a simulation of $\hat{P}$,

canonical partial results can be extracted, and written to external memory. In a second phase, $\hat{P}$ is simulated time-reversed, as will be described later on, and the movement of $y_j^{(i)}$ variables in $\hat{P}$ can be used to lead the previously extracted results to the corresponding dimension in $\mathbf{y}^{(i)}$. In the end, the memory cells, where $\mathbf{y}^{(i)}$ is expected for P, constitute $\mathbf{c}^{(i)}$.

Construction For the first phase, we create a program $\overrightarrow{P}$ for the semiring I/O-machine with internal memory size $M + B$. We use the first M cells in memory for a simulation of $\hat{P}$, and reserve the last B cells $(m_{M+1}, \ldots, m_{M+B}) =: \mathcal{B}$ for further output operations. Let p be the maximal index of a block in external memory that is accessed by $\hat{P}$. During $\overrightarrow{P}$, the i-th output of elements in $\mathcal{B}$ will be written to block t_{p+i}.

During the simulation of $\hat{P}$, the following additional operation is performed. *Modification 1:* If a computation σ in $\hat{P}$ performs a multiplication of an element m_r consisting of some $y_j^{(i)}$, i.e., $m_r = y_j^{(i)}$, $m_r = y_j^{(i)} x_k^{(i)}$, or $m_r = \sum_{j \in S} y_j^{(i)} a_{jk}$ for $S \subseteq [N_y]$, with an element m_s then m_s is copied into an empty position of $\mathcal{B}$ immediately before σ is performed. For further analysis, we call a copy operation created by Modification 1 a *snapshots* and σ its associated operation. If $\mathcal{B}$ is full after a snapshot operation, i.e., no element in $\mathcal{B}$ is 0, an output of $\mathcal{B}$ is performed after the associated operation.

The program $\overrightarrow{P}$ is executed with input $\mathbf{A}$ and $\mathbf{x}^{(i)}$, $1 \leq i \leq w$ as given, but all vectors $\mathbf{y}^{(1)} = \cdots = \mathbf{y}^{(w)} = (1, \ldots, 1)$. For each elementary product $y_j^{(i)} a_{jk} x_k^{(i)}$ created in $\hat{P}$, there are at most two elements copied to $\mathcal{B}$ (the corresponding $x_k^{(i)}$ and a_{jk}). Recall that in $\hat{P}$ elementary products are only created once. Since there are at most wh complete elementary products of the form $y_j^{(i)} a_{jk} x_k^{(i)}$ necessary, $\overrightarrow{P}$ performs no more than $\ell + \lceil \frac{2wh}{B} \rceil$ I/Os.

For the second phase, we have to time-reverse $\overrightarrow{P}$. Additionally, we remove all elements in the simulation of $\hat{P}$ that do not consist of a polynomial containing some $y_j^{(i)}$. By definition, an input becomes an output when time is inverted, and vice versa. Hence, we only have to define how to handle computation operations. To this end, each copy operation of $\hat{P}$ that sets $m_r := m_s$ becomes a sum operation $m_s := m_s + m_r$ in the time-reversed program $\overleftarrow{P}$. Each sum operation $m_q := m_r + m_s$ in $\overrightarrow{P}$ becomes a copy operation $m_r := m_s := m_q$ in $\overleftarrow{P}$. Operations of $\overrightarrow{P}$ that set a cell to 0 or 1 can simply be ignored in $\overleftarrow{P}$, i.e., nothing has to be created.

The additional copy operations introduced in $\overrightarrow{P}$ are only made when some $y_j^{(i)}$ is involved in a computation operation σ. Considering the different cases of computation operations, the elements that where extracted in $\overrightarrow{P}$ can now be copied, multiplied, or added into one of the cells that are accessed by σ. For each operation σ that is associated to a snapshot, the corresponding snapshot element s is by construction in $\mathcal{B}$ before σ should be performed in $\overleftarrow{P}$. Say σ is of the form $m_q := m_r \cdot m_s$ and m_r is a polynomial containing $y_j^{(i)}$ in $\overrightarrow{P}$. *Modification 2:* Then, instead of σ, s is copied into m_r. Additionally, the operation $m_r := m_r \cdot m_q$

is performed. Note that this describes all remaining multiplication operations after removing partial results of $\hat{P}$ that do not consist of monomials including some $y_j^{(i)}$.

After the run of $\overrightarrow{P}$ as described above, $\overleftarrow{P}$ is run where each of the input elements $z^{(i)}$ is considered as 1. This finishes the second phase and the result vector $\mathbf{c}^{(i)}$ can be found at the positions of $\mathbf{y}^{(i)}$ in external memory for $1 \leq i \leq w$.

Correctness Note that in a normalised program canonical partial results never contain input variables from different vector pairs. Hence, it suffices to consider the operations of each vector separated.

In the following, we denote a cell m_r that contains $y_j^{(i)}$ in $\overrightarrow{P}$ as $y_j^{(i)}$-*container* in $\overleftarrow{P}$. Observe, that if an elementary product is written into an $y_j^{(i)}$-container in $\overleftarrow{P}$, it will finally be contained as a summand in the initial position of $y_j^{(i)}$ in external memory. Since m_r is an input variable in $\overrightarrow{P}$, there is no other algebraic operation performed with it before. It can only result from a direct input of $y_j^{(i)}$ or be a copy of it. In case it was copied in $\overrightarrow{P}$, it is simply summed up with other results in the time-inverted $\overleftarrow{P}$. Eventually, with the initial input of $y_j^{(i)}$ in $\overrightarrow{P}$, i.e., its final output in $\overleftarrow{P}$, the sum of elementary products is written to the cell where $y_j^{(i)}$ is expected as an input for P.

During $\hat{P}$, elementary products can either be created explicitly during an operation, i.e., there is an element $m_q := y_j^{(i)} a_{jk} x_k^{(i)}$ in internal memory, or implicitly, caused by distributive law. If they are created explicitly, this can result from a multiplication operation $\sigma : m_q := m_r \cdot m_s$ with

a) $m_r = y_j^{(i)}$ and $m_s = a_{ij} x_k^{(i)}$,
b) $m_r = y_j^{(i)} a_{ij}$ and $m_s = x_k^{(i)}$, or
c) $m_r = y_j^{(i)} x_k^{(i)}$ and $m_s = a_{ij}$.

The first case is the simplest one. By Modification 1, a snapshot of the elementary product $a_{jk} x_k^{(i)}$ is made in $\overrightarrow{P}$. In $\overleftarrow{P}$, by Modification 2, this snapshot element is copied into m_r which is a $y_j^{(i)}$-container. Since in $\overrightarrow{P}$ the result m_q is eventually summed with other results (if existent) to compose $z^{(i)}$, in $\overleftarrow{P}$, m_q is a copy of $z^{(i)} = 1$. Hence, the additional multiplication $m_r := m_r \cdot m_q$ will not modify m_r.

For case b), there is a snapshot produced of $x_k^{(i)}$ in $\overrightarrow{P}$ by Modification 1. By Modification 2, in $\overleftarrow{P}$ the snapshot element $x_k^{(i)}$ is copied into m_r and not further modified since $m_q = 1$ holds. Since in $\overrightarrow{P}$, m_r contains the product $y_j^{(i)} a_{jk}$, there is an operation $\sigma' : m'_q := m'_r \cdot m'_s$ with $m'_r = y_j^{(i)}$ and $m'_s = a_{jk}$ before σ in $\overrightarrow{P}$. Thus, there is a snapshot made of a_{jk} during $\overrightarrow{P}$ by Modification 1. In $\overleftarrow{P}$, the snapshot element $x_k^{(i)}$ is traveling backwards, and before σ', it holds $m'_q = x_k^{(i)}$. By Modification 2, instead of σ', the snapshot a_{jk} is copied into m'_r. Then, $m'_q = x_k^{(i)}$ is multiplied to m'_r such that it now contains the elementary product $a_{jk} x_k^{(i)}$.

This elementary product belongs to $c_j^{(i)}$ and is now in a $y_j^{(i)}$-container. Case c) is analogous to case b), but with roles of $x_k^{(i)}$ and a_{jk} interchanged.

If the elementary products are not created explicitly, then there must be a multiplication of $m_r = y_j^{(i)}$ with a sum $m_s = \sum_{k \in S} a_{jk} x_k^{(i)}$, or a multiplication $m_r = x_k^{(i)}$ with $m_s = \sum_{j \in S'} y_j^{(i)} a_{jk}$. In the first case, there is a snapshot of $m_s = \sum_{k \in S} a_{jk} x_k^{(i)}$ made during the first phase. By Modification 2, m_s is then copied into the cell m_r in $\overleftarrow{P}$, which is a $y_j^{(i)}$-container.

The second case is slightly more complicated. In this case, by Modification 1, there is a snapshot of $x_k^{(i)}$ performed in $\overrightarrow{P}$. In $\overleftarrow{P}$, $x_k^{(i)}$ is then copied into m_s by Modification 2. Each operation that summed partial results together, building $\sum_{j \in S'} y_j^{(i)} a_{jk}$, is transformed into a copy operation in $\overleftarrow{P}$, copying $x_k^{(i)}$. Recall that due to the semiring, each of the monomials $y_j^{(i)} a_{jk}$, $j \in S'$ is derived from an independent multiplication operation in $\overrightarrow{P}$. For each $j \in S'$, when the operation $\sigma' : m'_q := m'_r \cdot m'_s$ is performed with $m'_r = y_j^{(i)}$ and $m'_s = a_{jk}$ before σ in $\overrightarrow{P}$, then we have $m'_q = x_k^{(i)}$ before σ' in $\overleftarrow{P}$. By Modification 2, instead of σ', the snapshot of m'_s is copied into the cell m'_r. Then, m'_q is multiplied to m'_r forming the elementary product $a_{jk} x_k^{(i)}$. Since m'_r further contained the input variable $y_j^{(i)}$ in $\overrightarrow{P}$, it is a $y_j^{(i)}$-container, and the elementary product will be summed into the corresponding field in external memory.

Since every canonical partial result that includes some $y_j^{(i)}$ has an input of the element $y_j^{(i)}$ as its predecessor in $\overrightarrow{P}$, in the time-reversed $\overleftarrow{P}$, all created partial results will be transferred to the initial position of $y_j^{(i)}$ in external memory. Furthermore, since all hw complete elementary products $y_j^{(i)} a_{jk} x_k^{(i)}$ have to be created for the bilinear product, and an $y_j^{(i)}$ is a predecessor for each, the created vectors $\mathbf{c}^{(i)}$, $1 \leq i \leq w$ are complete. □

5 Algorithms

Note that all the presented algorithms can be used for both, evaluating matrix vector products and bilinear forms, by Theorem 1. This yields asymptotically the same upper bounds for both problems. If an algorithm is described for bilinear forms and has to be transformed to evaluating matrix vector products, an internal memory size $M/2$ can be used for the algorithms to meet the conditions of Lemma 6 without changing the asymptotic behaviour.

5.1 Direct Algorithm

The computation of $w \leq B$ bilinear forms is possible with $\mathcal{O}(h)$ I/Os by considering the non-zero entries of $\mathbf{A}$ in an arbitrary order. For every entry a_{jk} the elementary products $x_k^{(1)} a_{jk} y_j^{(1)}, \ldots, x_k^{(w)} a_{jk} y_j^{(w)}$ are added to the respective

current partial sums $z^{(1)}, \ldots, z^{(w)}$. For this to inccur only a constant number of I/Os, the values $x_k^{(1)}, \ldots, x_k^{(w)}$ need to be stored in one block (or at least consecutively on disk), similarly to $y_j^{(1)}, \ldots, y_j^{(w)}$. This can be achieved by transposing the matrices $\mathbf{X} = \left[\mathbf{x}^{(1)} \ldots \mathbf{x}^{(w)}\right]$ and $\mathbf{Y} = \left[\mathbf{y}^{(1)} \ldots \mathbf{y}^{(w)}\right]$, which takes $\mathcal{O}\left((N_x + N_y)/B\right) = \mathcal{O}\left(h\right)$ I/Os [1], given the tall cache assumption $M \geq B^2$.

5.2 Sorting Based Algorithm

In [3] a sorting based approach for evaluating the product $\mathbf{Ax}$ for square matrices is presented. These algorithms can be extended straightforwardly to the matrix vector product of a non-square matrix $\mathbf{A}$ with one vector $\mathbf{x}$.

Column Major Layout If $\mathbf{A}$ is given in column major layout the algorithm works as follows: In a first step scan over $\mathbf{x}$ and $\mathbf{A}$ simultaneously, and multiply the elements of $\mathbf{x}$ into $\mathbf{A}$ to create elementary products. If $M > h/N_x$, i.e., internal memory is bigger than the average number of entries per column, then loading $\mathbf{A}$ (that consists now of elementary products) in runs of M elements, and sorting the elements by row index in internal memory yields h/M pre-sorted runs. Otherwise, the columns of $\mathbf{A}$ constitute N_x sorted runs. Using the M/B-way Merge sort from [1], the $r = \min\{N_x, h/M\}$ runs are sorted according to their row indices until there are h/N_y runs remaining. This merging process takes $\mathcal{O}\left(\frac{h}{B} \log_{M/B} rN_y/h\right)$ I/Os. By summing elementary products that belong to the same row immediately throughout the merging, each of the h/N_y remaining runs contains at most N_y elements. Finally, all runs are summed into the first run, which is possible with $\mathcal{O}\left(h/B\right)$ I/Os. Thus, the result vector $\mathbf{c} := \mathbf{Ax}$ is created with $\mathcal{O}\left(\frac{h}{B} \log_{M/B} \min\left\{\frac{N_x N_y}{h}, \frac{N_y}{M}\right\}\right)$ I/Os.

Best-Case Layout For the best-case layout, we assume the non-zero entries of $\mathbf{A}$ to be separated into meta-columns consisting of $M - B$ consecutive columns where each meta-column is written in row major layout in external memory. This allows us to load the $M - B$ vector elements $\mathbf{x}_{(j-1)(M-B)+1}, \ldots, \mathbf{x}_{j(M-B)}$ corresponding to the j-th meta-column of $\mathbf{A}$ into internal memory. Then, in the remaining block in internal memory, the corresponding meta-column of $\mathbf{A}$ can be scanned, and elementary products are created and written into $\mathbf{A}$. Afterwards, the algorithm works similarly to the description above: Meta-columns are merged together until there are h/N_y runs left, where throughout the merging process elementary products of the same row in $\mathbf{A}$ are summed immediately. The h/N_y runs are finally summed into the first run that will constitute the result vector $\mathbf{c}$ in the end. Since there are $\lceil N_x/(M-B) \rceil$ pre-sorted runs in the beginning, the task is possible with $\mathcal{O}\left(\frac{h}{B} \log_{M/B} \frac{N_x N_y}{Mh}\right)$ I/Os.

With slight modifications, this algorithm can also be described for a broader class of layouts. For this class of best-case layouts, the matrix $\mathbf{A}$ is given as a split-up of its columns into meta-columns where each meta-column is written in row major layout. Columns of a meta-column have to be continuous, each column is

assigned to only one meta-column, and each meta-columns consist of an arbitrary number of columns, but at most $M - B$. Additionally, the number of meta-columns is at most $\lceil N_x/B \rceil + 2\lceil h/N_y \rceil$. Since each meta-column consists of no more than $M - B$ continuous columns, for each meta-column, the corresponding elements of $\mathbf{x}$ can all be loaded into internal memory, and the meta-column is scanned to create elementary products which are then written back to $\mathbf{A}$. Afterwards, if $N_x/B > h/N_y$, meta-columns are merged together using Merge sort until there are at most h/N_y runs. Since in this case there are no more than $3\lceil N_x/B \rceil$ meta-columns, $\mathcal{O}\left(\frac{h}{B}\,\overline{\log}_{M/B}\frac{N_xN_y}{Bh}\right) = \mathcal{O}\left(\frac{h}{B}\,\overline{\log}_{M/B}\frac{N_xN_y}{Mh}\right)$ I/Os are sufficient. Otherwise, if $h/N_y \geq N_x/B$, there are at most $3\lceil h/N_y \rceil$ meta-columns. Since meta-columns and runs are in row major layout, with one scan of each meta-column / run, elements from the same row can be summed together, and meta-columns / runs become a single column. All created columns can then be summed into the first column with $\mathcal{O}\left(\frac{h}{N_y}\cdot\frac{N_y}{B}\right)$ I/Os. Hence, in all cases, the matrix vector product for a matrix $\mathbf{A}$ given in a layout meeting the conditions described can be determined with $\mathcal{O}\left(\frac{h}{B}\,\overline{\log}_{M/B}\frac{N_xN_y}{Mh}\right)$ I/Os.

Multiple Vectors For the evaluation of w matrix vector products, the algorithms can simply be run for each single vector, which increases the running time by a factor w. However, for column major layout, it can be faster to transform the layout of $\mathbf{A}$ into one belonging to the class of generalized best-case layouts described above, and then use that algorithm for each single vector.

The transformation of the layout has two cases, depending on the parameters. The first case handles situations with $N_x \leq h/(M - B)$, where the average column consists of more than $M - B$ already sorted entries. Then the N_x columns are bottom up merged using the M/B-way Merge sort, each time reducing the number of meta-columns by a factor of M/B. This is continued as long as the resulting meta columns have width at most $M - B$, and the number of meta columns is still greater than h/N_y. Hence, the running time of this merging is $\mathcal{O}\left(\frac{h}{B}\log_{M/B}\min\left\{M, \frac{N_xN_y}{h}\right\}\right)$ I/Os, and there are at most $\max\left\{\lceil N_x/(M-B)\rceil, \lceil h/N_y\rceil\right\}$ meta-columns that can contain less than $N_y/2$ entries, i.e., they form a generalised best case layout.

The second case assumes $h/(M - B) \leq N_x$, and mimics the creation of initial runs of length $M - B$. The possibility of columns having vastly different number of entries makes this slightly more involved. First, the columns of $\mathbf{A}$ are split into $2h/N_y$ continuous groups such that each group contains at most N_y entries. Then, each group that spans at most $M - B$ columns is transformed into row major layout using the classical M/B-way Merge sort in $\mathcal{O}\left(\frac{h}{B}\,\overline{\log}_{M/B}\frac{N_y}{M}\right)$ I/Os. Groups that span more than $M - B$ columns are divided into subgroups that span at most $M - B$. This can be achieved by greedily assigning the blocks of a group to subgroups such that each subgroup spans no more than $M - B$ columns. Splitting blocks that belong to two columns is possible with $\mathcal{O}(N_x/M)$ I/Os. The subgroups are then transformed into row major layout using Merge sort.

Since there are at most N_y/B blocks per group, this can be done with another $\mathcal{O}\left(\frac{h}{B}\,\overline{\log}_{M/B}\,\frac{N_y}{M}\right)$ I/Os.

Because one block spans at most B columns, each of the subgroups, except at most one per group, spans at least $M-2B$ columns. Since we assume $M \geq 4B$ in this paper, for each group, there can be at most one meta-column with span less than $2B$. Hence, in the created layout, we have at most $\lceil N_x/(2B)\rceil + \lceil 2h/N_y\rceil$ meta-columns, each spanning at most $M-B$ columns, and the algorithm for the class of generalised best-case layouts can be applied.

5.3 Table Based Algorithm

For very asymmetric cases of $\mathbf{A}$ where $N_x \gg N_y$, the construction of tuples of rows of $\mathbf{Y}$, such that arbitrary dimensions of each vector can be loaded within one I/O, can be superior. We present the following algorithms in the setting of evaluating bilinear products.

Column Major Layout The bilinear forms ${\mathbf{y}^{(i)}}^T\mathbf{A}\mathbf{x}^{(i)}$ for w vector pairs can be evaluated with $\mathcal{O}\left(\max\left\{\frac{wh}{B}, h\frac{\log N_y}{\log N_x}\right\}\right)$ I/Os as follows. The algorithm starts by creating a table of all c-tuples of rows of $\mathbf{Y}$ in lexicographical order of the row indices. To this end, define $c := \min\left\{\left\lfloor\frac{B}{w}\right\rfloor - 1, \left\lfloor\frac{\log N_x}{2\log N_y}\right\rfloor\right\}$ such that a c-tuple of rows of $\mathbf{Y}$ does not exceed one block. Since we assume that $\mathbf{Y}$ is in column major layout, it first has to be transposed which is possible with $\mathcal{O}(wN_y/B)$ I/Os (cf. Section 5.1). Further, since we assume a tall cache, i.e. $M \geq B^2$, internal memory can hold c blocks at a time and one for the output of a created tuple.

A table of all c-tuples can be created by scanning over the transposed $\mathbf{Y}$ for each of the c tuple dimensions: For every B-th tuple, a new block of $\mathbf{Y}$ is loaded for the least significant position of the tuple. The other positions $i = 1, \ldots, c-1$ change only every N_y^i tuple, yielding in total another $O(1/B)$ I/Os per tuple. Hence, the I/Os are dominated by writing the generated tuples to external memory. The size of the output table is $wcN_y^c \leq wc\sqrt{N_x} \leq wN_x$, where the last inequality relies upon $c \leq \frac{1}{2}\log N_x \leq \sqrt{N_x}$, which is true for all N_x. The table can easily be created in $\mathcal{O}(wN_x/B)$ I/Os, a term dominated by the I/Os needed to read $\mathbf{X}$.

After creating a table of all c-tuples of rows of $\mathbf{Y}$, the algorithm simultaneously scans the entries of $\mathbf{A}$ and the corresponding elements of $\mathbf{X}$. To this end, $\mathbf{X}$ is transposed first. Since we have $M \geq 4B$, we use one block for the scanning of $\mathbf{A}$, one for elements of $\mathbf{X}$, one for a c-tuple of $\mathbf{Y}$, and the last block to sum elementary results together for each of the $w \leq B$ vectors. Throughout the scanning of $\mathbf{A}$, for each $c \leq B$ entries $a_{i_1,j_1}, \ldots, a_{i_c,j_c}$, the c-tuple containing the corresponding rows $i_1, \ldots i_c$ of $\mathbf{Y}$ is loaded, and elementary products for each pair of vectors are created. These can be summed immediately into the block reserved for the results. Hence, $\mathcal{O}(h/c)$ I/Os are sufficient to evaluate the w bilinear forms.

Best-case Layout If $w < B$ and $N_x \geq N_y^2$, the table-based approach for column major layout can be improved using a different layout of $\mathbf{A}$. Similar to the column major case, the matrices $\mathbf{X}$ and $\mathbf{Y}$ are transposed in the beginning.

Again, we create a table of c-tuples of rows of $\mathbf{Y}$, but for different c. Furthermore, we assume again that c is at most B/w. But in contrast to the column major case, the tables are created only for ranges of elements instead. To this end, the matrix $\mathbf{A}$ is organised in tiles with dimensions $s \times t$ such that each tile contains on average $(M-B)/w$ non-zero entries. Using this, a tile can be loaded and all elementary products can be created while still one free block is available in internal memory to read and write vector elements. However, for the ease of calculations, we use $M/(2w)$ as the number of entries per tile. The dimensions of the tiles will be determined throughout this section. Note that the width of a tile becomes larger than MB/w such that it is not possible to keep all required vector elements of one dimension in internal memory for more than one tile, i.e. both $\mathbf{X}$ and $\mathbf{Y}$ have to be accessed for each tile again. For the width s of a tile, on average there are $s\frac{h}{N_xN_y}$ non-zero entries per row in each tile. Thus, on average, $d = \frac{cN_xN_y}{sh}$ rows of a tile contain c non-zero entries, and it is sufficient to create $\lceil N_y/d \rceil$ tables of c-tuples of rows $di+1, \ldots, \min\{d(i+1), N_y\}$ of $\mathbf{Y}$, for $0 \leq i \leq \lceil N_y/d \rceil - 1$.

The algorithm works in rounds where in each round a tile of $\mathbf{A}$ is loaded into internal memory, and the corresponding rows of $\mathbf{X}$ are scanned and multiplied to the entries of $\mathbf{A}$. Then, the elements of $\mathbf{Y}$ are read in c-tuples for each c entries of $\mathbf{A}$ to create elementary products. Since the tuples are only created for ranges of rows of $\mathbf{Y}$, it can be necessary to load multiple tuples for a range of d rows, and it might be necessary to load a tuple from which only few elements are used. However, on average at most 2 tuples are loaded for each d rows of a tile.

Loading the tuples hence incurs on average $\frac{M}{wc}$ I/Os per round. We also allow $\frac{M}{wc}$ I/Os per round for loading rows of $\mathbf{X}$, each consisting of w vector entries. This yields a width $s = \frac{MB}{w^2c}$ for each tile in $\mathbf{A}$. Because a tile of $\mathbf{A}$ shall contain $M/(2w)$ entries on average, for the height t of a tile, it has to hold $t\frac{MB}{w^2c}\frac{h}{N_xN_y} = \frac{M}{2w}$, that is $t = \frac{wcN_xN_y}{2hB}$. As described before, it is sufficient to create tuples only for each d rows of $\mathbf{Y}$ where d is the number of rows within a tile having c entries on average. Using the dimensions of a tile, we find the value of d to be $d = \frac{cN_xN_y}{sh} = \frac{w^2c^2N_xN_y}{hBM}$.

Now, we choose c as big as possible, such that the size of all tables together does not exceed the size of $\mathbf{X}$, i.e., $c\frac{N_y}{d}wd^c \leq wN_x$. By construction, d rows of a tile in $\mathbf{A}$ contain on average c entries. Observe that if $d < c$, i.e., the average number of non-zero entries per row in a tile is more than 1, there can only be one (ordered) tuple which can be obtained by simply reading $\mathbf{Y}$. Moreover, setting $d = c = B/w$ in this case yields an algorithm requiring $\mathcal{O}\left(\frac{hw}{B}\right)$ I/Os.

Thus, we consider in the following the case that $c \leq d$. It then suffices to maximise c for $N_ywd^c \leq wN_x$. Substituting d yields $\left(\frac{w^2c^2N_xN_y}{hBM}\right)^c \leq \frac{N_x}{N_y}$, and by taking logarithms we get $c\log\left(c^2\frac{w^2N_xN_y}{hBM}\right) \leq \log\frac{N_x}{N_y}$. From this we can already restrict ourselves to $c \leq \log\frac{N_x}{N_y}$, such that we get

$$c\left(2\log\log\frac{N_x}{N_y} + \log\frac{w^2N_xN_y}{hBM}\right) \leq \log\frac{N_x}{N_y},$$

and set

$$c = \left\lfloor \frac{\log \frac{N_x}{N_y}}{2 \log\log \frac{N_x}{N_y} + \log \frac{w^2 N_x N_y}{BhM}} \right\rfloor .$$

Since $wc \leq B$ the number of I/Os is dominated by the table accesses. If $c \leq 1$, the direct algorithm can be applied, otherwise we get a performance of

$$\mathcal{O}\left(h \frac{2 \log\log \frac{N_x}{N_y} + \log \frac{w^2 N_x N_y}{hBM}}{\log \frac{N_x}{N_y}}\right) = \mathcal{O}\left(h \frac{2 \log\log N_x + \log \frac{w^2 N_x N_y}{hBM}}{\log N_x}\right)$$

where we use $N_y^2 \leq N_x$.

In the following, we consider the complexity of this algorithm for the cases where it is asymptotically superior over the other presented algorithms. First, note that the term $\frac{w^2 N_x N_y}{hBM}$ never becomes smaller than $w^2 > 1$ unless the sorting based algorithm would be optimal. If the sorting based algorithm is not asymptotically optimal, $\frac{N_x N_y}{hM} > \frac{M}{B}$ has to hold (otw. a simple scanning bound of $\Omega\left(\frac{hw}{B}\right)$ matches its complexity). Hence, for the cases where the table based algorithm is the only asymptotically optimal algorithm of the presented ones, we can assume $\frac{w^2 N_x N_y}{hBM} > \frac{w^2 M}{B^2} \geq w^2$ where the last inequality is obtained by the tall cache assumption. This simplifies the complexity such that it is now

$$\mathcal{O}\left(h \frac{2 \log\log N_x + \log \frac{w^2 N_x N_y}{hBM}}{\log N_x}\right) = \mathcal{O}\left(h \frac{\log\left(\frac{w N_x N_y}{hBM} \log N_x\right)}{\log N_x}\right) .$$

Furthermore, it can be shown that the leading term of the logarithm will never be w, and can hence be ignored in Theorem 3. Observe that the assumption $\frac{N_x N_y}{hBM} \log^2 N_x < w$ implies that the table based algorithm is not superior over the sorting based algorithm: In this case, we have

$$c > \left\lfloor \frac{\log \frac{N_x}{N_y}}{3 \log w} \right\rfloor > \frac{\log N_x}{6 \log w} - 1 > \frac{\log N_x}{6 \log M/B} - 1$$

where we use the tall cache $M/B \geq B > w$, and $N_x \geq N_y^2$. Since we can assume $c \geq 2$ for the case of being superior and having $c \leq B/w$, we get $\frac{B}{w} \log M/B \geq \frac{1}{12} \log N_x$. Thus, it holds

$$\frac{\log\left(\frac{w N_x N_y}{hBM} \frac{B}{w} \log M/B\right)}{\frac{B}{w} \log M/B} \leq \frac{12 \log\left(\frac{w N_x N_y}{hBM} \log N_x\right)}{\log N_x}$$

such that the number of I/Os for the sorting based algorithm is asymptotically not higher than the number of I/Os of the table based algorithm. Hence, if the table based algorithm is superior, the asymptotical behaviour can be written as

$$\mathcal{O}\left(h \cdot \max\left\{\frac{\log\left(\frac{N_x N_y}{hBM} \log N_x\right)}{\log N_x}, \frac{w}{B}\right\}\right) .$$

6 Lower bounds

For the lower bounds, we only consider matrix vector products. By Theorem 1 this also implies lower bounds for bilinear products.

6.1 Column Major Layout

The following lower bound is only for single matrix vector products. However, on the algorithmic side, it makes sense to change the layout of A for multiple evaluations into best-case layout. Hence, the lower bounds of Theorem 2 are a combination of Lemma 7, and the lower bound for best-case layout in Section 6.2.

Note that the lower bounds do not require the tall cache assumption $M \geq B^2$.

Lemma 7. *Computing over an arbitrary semiring the matrix vector product with an $N_y \times N_x$ matrix A with h entries, stored in column major layout has (worst-case) I/O-complexity for M, B with $M \geq 4B$*

$$\Omega\left(\min\left\{\frac{h}{B}\log_{\frac{M}{B}}\frac{N_y}{M} \quad , \quad \frac{h}{B}\log_{\frac{M}{B}}\frac{N_xN_y}{h} \quad , \quad h \quad , \quad h\max\left\{\frac{1}{B}, \frac{\log N_y}{\log N_x}\right\}\right\}\right).$$

Proof. To proof this lemma, the dimensions of $\mathbf{A}$ have to be simply replaced in the proof in [3] which yields

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

for

$$\tau = \begin{cases} 3^h & \text{for } k > B, \\ 1 & \text{for } k = B, \\ \binom{2B}{k}^{N_x} & \text{for } k < B \end{cases}$$

where we have $k = h/N_x$.

For $k < B$, by taking logarithms we get

$$h\log\frac{N_y}{k} \leq \ell\left(\log h + B\log\frac{e(M+B)}{B}\right) + h\log(2eB/k)\,.$$

Because of $M + B < 4M/3$ since $M \geq 4B$, and $e4M/3 < 4M$ we have

$$\ell \geq h\frac{\log\frac{N_y}{2eB}}{\log h + B\log\frac{4M}{B}}\,. \tag{1}$$

For $k \geq B$, we have

$$h\log\frac{N_y}{k} \leq \ell\left(\log h + B\log\frac{e(M+B)}{B}\right) + h\log 3$$

which yields

$$\ell \geq h \frac{\log \frac{N_y}{3k}}{\log h + B \log \frac{4M}{B}} \,. \tag{2}$$

Combining (1) and (2), we obtain

$$\ell \geq h \frac{\log \frac{N_y}{\max\{3k, 2eB\}}}{\log h + B \log \frac{4M}{B}} \,, \tag{3}$$

and it remains to distinguish the leading terms of the denominator.

For $\log h \leq B \log \frac{4M}{B}$ this asymptotically matches the sorting based algorithm. Since $M \geq 4B$, we get $l \geq \frac{h}{4B} \log_{M/B} \frac{N_y}{\max\{3k,2eB\}} \geq \frac{h}{8B} \log_{M/B} \frac{N_y}{\max\{k,B\}}$ for $\log_4 \frac{N_y}{\max\{k,B\}} > 8$, otherwise, a scanning bound of $\frac{h}{B}$ holds for reading $\mathbf{A}$. Finally, if $B > k$, the matching bound is obtained since $\log_{M/B} \frac{N_y}{B} = 1 + \log_{M/B} \frac{N_y}{M}$.

Now, consider the case of $\log h > B \log \frac{4M}{B}$. This is equivalent to $B < \frac{\log kN_x}{\log(4M/B)} \leq \frac{1}{4}(k + \log N_x)$ since we have $M \geq 4B$ and $k \leq N_y$. Assume $k < N_y^\varepsilon$ and $\log N_x < N_y^\varepsilon$ for some $0 < \varepsilon < 1$. Using this and $h \leq N_x N_y$, we have

$$\ell \geq h \frac{\log \frac{N_y}{\max\{3k,2eB\}}}{2 \log h} \geq h \frac{\log N_y^{1-\varepsilon}}{2 \log(N_x N_y)} = h \frac{(1-\varepsilon) \log N_y}{2 \log N_x + 2 \log N_y} \,.$$

Distinguishing the leading terms of the denominator, we get

$$\ell \geq h \frac{1-\varepsilon}{4} \min \left\{ 1, \frac{\log N_y}{\log N_x} \right\} \,.$$

□

6.2 Best-case Layout

As described in Section 2, for the best-case layout it is up to the program to choose the layout of $\mathbf{A}$. The proof of Lemma 7 is based on the task of computing row sums. To obtain a lower bound for the best-case layout, we have to use a different approach because producing row sums is trivial when using a row major layout. Therefore, we consider the sequence of configurations of a program and follow the movement of input variables of $\mathbf{X}$ and partial results of $\mathbf{Y}$. Furthermore, we allow accessing $\mathbf{A}$ for free. This can only weaken the lower bound.

We count the number of different matrix conformations that can be handled by programs for matrix vector multiplication with ℓ I/Os. For a given program, the conformation of a matrix can be identified by considering multiplication operations including input variables, and their results: When there is an input variable $x_j^{(i)}$ loaded, and it is used to form an elementary product that is a predecessor of $c_k^{(i)}$, this describes the existence of a non-zero entry a_{ik} in $\mathbf{A}$. Hence, by tracking all copies of input variables $x_j^{(i)}$ and all elements that are

predecessors of a unique result $c_k^{(i)}$ (this can be elementary products or partial sums), and by choosing the positions in a program where multiplications involving such elements are performed, the conformation of a matrix is uniquely determined. To do this, it suffices to consider the tracking of elements only for one of the w matrix vectors multiplication. All this information will be called *trace* in the following.

In order to describe the trace, we normalise programs which changes the number of I/Os only by constant factors. The following normalisation is a variation of [5, Theorem 3.1].

Lemma 8. *Assume there is an I/O program $\mathcal{A}$ performing ℓ I/Os for parameters M and B. Then there is an I/O program $\mathcal{B}$ computing the same function performing at most $3\ell + M/B$ I/Os for parameters $2M$ and B, that works in rounds: Each round consists of $2M/B$ input operations, an arbitrary number of computation operations followed by $2M/B$ output operations such that after each round internal memory is empty.*

Proof. A program $\mathcal{B}$ can be created by splitting the computation of $\mathcal{A}$ into rounds of M/B consecutive I/Os. With the additional memory, input and output can be serialised as claimed. The final memory content of each round can be transferred to the next round with $2M/B$ I/Os. □

In the following, we consider programs in rounds according to the above lemma. For the ease of notation, we ignore the doubling of internal memory since it will not change the asymptotic behaviour. To determine the traces of input variables and result predecessors in a round-based program, we consider the transfer of blocks between rounds, i.e. a block that is output by one round and input by another.

The movements of input variables can be described as follows. For a vector $\mathbf{x}^{(k)}$, we consider the subset $\mathcal{T}_{\mathcal{V},i}^{(k)} \subseteq [N_x]$ of indices of elements $x_i^{(k)}$ in a block i and trace the copying and deletion of variables in each round. For the trace of a predecessor of a unique result $c_j^{(k)}$, we abstract from the element itself, and consider only the index of the result j. Hence, we have the subset $\mathcal{T}_{\mathcal{R},i}^{(k)} \subseteq [N_y]$ of indices of unique result predecessors transferred by block i.

As written before, it suffices to consider the traces for one pair of input and result vector only. Every block of an I/O can be separated into values belonging to the w different tasks implied by the different pairs of vectors. Hence, for the l-th I/O, we have the number $u_l^{(k)}$ of elements belonging to vector pair k. By averaging we have $\sum_{0 \leq l \leq \ell} u_l^{(k)} \leq B\ell/w$ for some k, and we determine the traces for this pair of vectors in the following.

Describing the traces Because we will describe the traces of programs by blocks transferred between rounds, we view the input variables as output of rounds with no cost. Further, since we are aware that the task is possible in $\mathcal{O}(h)$, we are only interested in lower bounds at most h such that we assume $\ell \leq h$ in the following. Let R be the total number of rounds. For each block that is an output of a round, and input of another round there are $R^2 \leq h^2$ possibilities

to choose the origin and destination of the block. Because there are $\ell/2$ blocks transferred, h^ℓ is an upper bound on the total number of possible macroscopic structures of how blocks travel between rounds.

Further, the values of $u_l^{(k)}$ can be chosen which yields at most B^ℓ possibilities. Every traced element that is transferred by a block can terminate at the destination, i.e., it is not copied further. Hence, there are $2^{\ell B/w}$ choices of terminating elements. Each of the s_i non-terminal incoming elements of round i can appear up to M/B times in the outgoing blocks, namely once per outgoing block. Hence, there are $\binom{s_i M/B}{t_i}$ possibilities to choose the t_i outgoing elements of round i, for some $t_i \geq s_i$. Since we have $\sum_{1\leq i\leq R} t_i \leq B\ell/w$, the total number of possibilities for this is bounded by $\binom{M\ell/w}{B\ell/w}$.

Finally, we have to specify the subset of possible multiplications that are actually performed. To this end, let W_i be the number of partial results output by round i. Together with the number of vector variables U_i loaded in round i, there are $\sum_{i\leq \ell\frac{B}{M}} U_i W_i$ possible multiplications with matrix entries during the program. Additionally, we have the conditions $U_i \leq M$, $W_i \leq M$, and $\sum_{i\leq \ell\frac{B}{M}} (U_i + W_i) \leq \frac{\ell B}{w}$. The term $\sum_{i\leq \ell\frac{B}{M}} U_i W_i$ is hence maximised for $U_i = W_i = M$, for some indices $i \in \mathcal{I}$, $\mathcal{I} \subseteq [\ell\frac{B}{M}]$ with $|\mathcal{I}| \leq \frac{\ell B}{2Mw}$, and the size of the set of possible multiplications is at most $\frac{\ell B}{2Mw} \cdot M^2 < \frac{\ell MB}{w}$. From this, we select a subset of size h, yielding no more than $\binom{\ell MB/(2w)}{h}$ possibilities.

Calculations With the above discussion, we get

$$\binom{N_x N_y}{h} \leq h^\ell \cdot B^\ell \cdot 2^{\ell B/w} \cdot \binom{\ell M/w}{\ell B/w} \cdot \binom{\frac{\ell MB}{w}}{h}.$$

W.l.o.g. we assume $N_x \geq N_y$ in the following. Furthermore, define $k = h/N_x$, i.e., the average number of entries per column. Taking logarithms, estimating binomials according to Observation 3, and rearranging terms yields

$$\ell \geq h \frac{\log \frac{N_y}{k} - \log \frac{e\ell MB}{wh}}{\log h + \log B + \frac{B}{w}(1 + \log \frac{eM}{B})}.$$

In the following, we can assume $h \geq M \geq B$, and thus

$$\frac{\ell}{h} \geq \frac{\log\left(\frac{N_y}{k} \cdot \frac{wh}{e\ell MB}\right)}{2\log h + \frac{B}{w}(\log \frac{6M}{B})}.$$

Otherwise, if $h \leq M$, the task is trivial and a scanning bound of $\Omega\left(\frac{h}{B}\right)$ for reading $\mathbf{A}$ suffices. Applying Lemma 2 ($x = \ell/h$, $t = 2\log h + \frac{B}{w}(\log \frac{6M}{B})$, $s = \frac{wN_y}{ekMB}$), and estimating $t \geq \log N_x + 3\frac{B}{w}$, we get

$$\frac{2\ell}{h} \geq \frac{\log\left(\frac{wN_y}{ekMB} \cdot \left(\frac{3B}{w} + \log N_x\right)\right)}{2\log h + \frac{B}{w}(\log \frac{6M}{B})}.$$

Now, it remains to distinguish according to the leading term in the denominator.

Case 1 ($2\log h \leq \frac{B}{w}(\log \frac{6M}{B})$) yields

$$\frac{\ell}{h} \geq \frac{\log \frac{N_y}{kM}}{4\frac{B}{w}(\log \frac{6M}{B})}.$$

Using $M \geq 4B$, we get

$$\ell \geq \frac{hw}{B} \frac{\log \frac{N_y}{kM}}{4 \cdot \frac{5}{2} \log \frac{M}{B}} = \frac{hw}{10B} \log_{\frac{M}{B}} \frac{N_y}{kM}$$

which matches the sorting based bound.

Case 2 ($2\log h > \frac{B}{w}(\log \frac{6M}{B})$) yields

$$\frac{2\ell}{h} \geq \frac{\log \left(\frac{wN_y}{BekM} \log N_x \right)}{4 \log h} > \frac{\log \left(\frac{N_y}{BekM} \log N_x \right)}{4 \log h}$$

which matches the bound obtained by the table based algorithm.

Recall that the table based upper bound requires $N_x \geq N_y^2$. However, for $N_x \leq N_y^2$, a linear lower bound is obtained as follows. By Lemma 3, assuming $k \leq \sqrt[6]{N_y}$ and $B \geq 20w$, we get for the case $N_x \geq 2^{30}$

$$\frac{2\ell}{h} \geq \frac{\log \frac{wN_y}{BekM}}{2 \log h} \geq \frac{\log \sqrt[15]{N_y}}{2(2 + 1/6) \log N_y} \geq \frac{1}{65}$$

Otherwise, if $N_y \leq N_x < 2^{30}$, also h is bounded from above, and thus, the task is trivially possible in $\mathcal{O}(1)$.

For a large class of matrices a bound of $\Omega\left(\frac{hw}{B}\right)$ can be used to derive better constant factors for $N_x < 2^{30}$. This bound is obtained by an extension of the lower bound in [4] to non-square matrices, and is is given in the following section.

6.3 Extension of results from [4]

In this section, we present a lower bound of $\Omega\left(\frac{hw}{B}\right)$ on the number of I/Os for certain parameter ranges. To this end, we show that by loading M elements from $\mathbf{A}$ only few elementary products can be obtained. For the sake of illustration, we consider the matrix $\mathbf{A}$ as an adjacency matrix of a bipartite graph $G = (U \cup V, E)$, $|U| = N_x$, $|V| = N_y$, where $a_{ij} \neq 0$ constitutes a connection between the j-th node of U and the i-th node of V. By bounding the degree of subgraphs with at most M edges, a lower bound on the number of I/Os for the matrix vector product can be stated.

Lemma 9 ([4], Modification Lemma 4). *Let $\mathcal{G}$ be the family of bipartite graphs $G = (U \cup V, E)$ with $|U| = N_x$, $|V| = N_y$ and $|E| = h$ for $h \leq N_x N_y/2$.*

For any $M \leq h$ there is a graph $G \in \mathcal{G}$ such that G contains no subgraph $G_S = (U_S \cup V_S, E_S)$ with $|E_S| = M$ and average degree

$$D'_M > \max\left\{ \frac{8 \ln \frac{N_x+N_y}{2M}}{\operatorname{ln} \frac{16 N_x N_y \ln^2 \frac{N_x+N_y}{2M}}{hM}}, e^4 \cdot \sqrt{\frac{hM}{N_x N_y}} \right\}. \tag{4}$$

Proof. We will show this by upper bounding the number of graphs containing at least one such dense subgraph and compare this to the cardinality of $\mathcal{G}$. The upper bound is given by the number of possibilities to choose $2M/D'_M$ vertices from $U \cup V$ and the number of possibilities to insert M edges between the selected vertices. Furthermore, the remaining $h - M$ edges are chosen arbitrarily within the graph. The former presumes $2M/D'_M \leq N_x + N_y$. However, since $M \leq h$ and $D'_M > \sqrt{\frac{hM}{N_x N_y}}$ this is implied. Further, we can assume $D'_M \leq \sqrt{M}$ since this is the maximum average degree of a subgraph consisting of M edges. Hence, if the inequality

$$\binom{N_x + N_y}{2M/D'_M} \binom{(M/D'_M)^2}{M} \binom{N_x N_y}{h - M} < \binom{N_x N_y}{h}$$

holds for the parameters given, Lemma 9 is proven. Observation 4 yields

$$\binom{N_x + N_y}{2M/D'_M} \binom{(M/D'_M)^2}{M} < \left(\frac{N_x N_y - h}{h} \right)^M.$$

Estimating binomial coefficients according to Observation 3, taking logarithms and multiplying by D'_M/M, we obtain

$$2 \ln \frac{eD'_M(N_x + N_y)}{2M} + D'_M \ln \frac{eM}{{D'_M}^2} < D'_M \ln \frac{N_x N_y}{h} + D'_M \ln \left(1 - \frac{h}{N_x N_y}\right).$$

The last term can be estimated for $h \leq N_x N_y / 2$ by Observation 1 resulting in

$$2 \ln \frac{eD'_M(N_x + N_y)}{2M} + D'_M \ln \frac{eM}{{D'_M}^2} < D'_M \ln \frac{N_x N_y}{h} - D'_M \frac{2h}{N_x N_y}.$$

And by simple equivalence transformations, we obtain

$$D'_M \ln \frac{{D'_M}^2 N_x N_y}{hM} > \underbrace{2 \ln \frac{N_x + N_y}{2M}}_{\text{Term 1}} + \underbrace{2 \ln eD'_M + D'_M \left(1 + 2 \frac{h}{N_x N_y}\right)}_{\text{Term 2}}. \tag{5}$$

Equation 5 is implied if Terms 1 and 2 are both bounded by $\frac{1}{2} D'_M \ln \frac{{D'_M}^2 N_x N_y}{hM}$. We first check this for Term 2 only. By Observation 2, it holds $\ln(eD'_M) \leq D'_M$. Thus,

$$\frac{1}{2} D'_M \ln \frac{{D'_M}^2 N_x N_y}{hM} > 2 \ln(eD'_M) + 2D'_M$$

is implied by $D'_M > e^4 \cdot \sqrt{\frac{hM}{N_x N_y}}$ yielding the second term of the maximum in the final inequality (7). For any such D'_M Inequality 5 holds if

$$D'_M \ln \frac{D'_M \sqrt{N_x N_y}}{\sqrt{hM}} > 2 \ln \frac{N_x + N_y}{2M}. \tag{6}$$

By substitution of D'_M by $e^4 \cdot \sqrt{\frac{hM}{N_x N_y}}$, inequality (6) already holds if $\sqrt{h} > \frac{1}{2e^4} \sqrt{\frac{N_x N_y}{M}} \ln \frac{N_x+N_y}{2M}$. For $\sqrt{h} \leq \frac{1}{2e^4} \sqrt{\frac{N_x N_y}{M}} \ln \frac{N_x+N_y}{2M}$, we use Observation 5 with $f = \sqrt{\frac{N_x N_y}{hM}}$ and $x = 2 \ln \frac{N_x+N_x}{2M}$ yielding the first term of the maximum in (7). Altogether, for

$$D'_M > \max \left\{ \frac{8 \ln \frac{N_x+N_y}{2M}}{\overline{\ln} \frac{16 N_x N_y \ln^2 \frac{N_x+N_y}{2M}}{hM}}, e^4 \cdot \sqrt{\frac{hM}{N_x N_y}} \right\} \tag{7}$$

not all possible graphs in $\mathcal{G}$ are covered and therefore, Lemma 9 holds. Since the second term is a sufficient bound for any $\sqrt{h} > \frac{1}{2e^4} \sqrt{\frac{N_x N_y}{M}} \ln \frac{N_x+N_y}{2M}$, we use $\overline{\ln}$ instead of ln to derive a closed formula by bounding the first term. Finally, note that $D'_M > 4$ holds for $h \geq \max\{N_x, N_y\}$. □

Using this, Lemma 5 from [4] applies, and we can finally proof Lemma 1.

Lemma 10 ([4], Lemma 5).
For any $M \leq h$, there is a graph $G \in \mathcal{G}$ such that G contains at most $M - 1$ edges in subgraphs $G_S = (U_S \cup V_S, E_S)$ with $|E_S| \leq M$ and average degree $D' \geq 2D'_M$.

Proof (Proof of Lemma 1). To use Lemma 10, we normalise programs into rounds according to Lemma 8 of Section 6.2. Since for the evaluation of w bilinear forms in the semiring I/O-model wh elementary products have to be created, by giving an upper bound on the number of elementary products that can be created during one round, a lower bound on the number of required rounds and I/Os is obtained.

By Lemma 10, there are at most $w(M-1)$ elementary products which might be calculated faster than the rest. For the remaining $hw - wM + w$ elementary products, the following holds. Consider a round-based program to evaluate w bilinear forms with $\mathbf{A}$. Within each round, there are at most $2M$ elements of $\mathbf{A}$ loaded. Thus, for each vector pair, the number of newly created elementary products that can be achieved during this round is some $m_i \leq 2M$. By Lemma 10, for the i-th pair of vectors, at least $2m_i/D'_{2M}$ elements have to be loaded to create m_i elementary products. Since from $\mathbf{X}$ and $\mathbf{Y}$ there can be as well no more than $2M$ elements in internal memory during the round, $\sum_i 2m_i/D'_{2M} \leq 2M$ vector elements are loaded to obtain $\sum_i m_i$ elementary products. Hence, the number of elementary products created throughout one round is bounded from above by $2MD'_{2M}$.

For the average number of entries in $\mathbf{A}$ per column being $h/N_x \leq \frac{N_y}{M^{1-\epsilon}N_x^{\epsilon}}$ with constant $\epsilon > 0$, we have D'_{2M} upper bounded by $8/\epsilon$. This yields a lower bound on the number of rounds of

$$\frac{\epsilon(hw - wM + w)}{16M}.$$

Thus, we get a lower bound of

$$\frac{M}{B}\left(\frac{\epsilon(hw - wM + w)}{16M} - 1\right) = \Omega\left(\frac{hw}{B}\right)$$

I/Os for $h \geq \frac{32M}{\epsilon w} + 2M$. Note that the task becomes trivial for $h = \mathcal{O}(M)$. □

7 Acknowledgement

Thanks to Dan Roche and Clement Pernet for asking the question about multiplying many vector pairs. Many thanks to an anonymous referee for suggestions on an earlier draft of this article.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. In *Proceedings of SPAA '07*, pages 61–70, New York, NY, USA, 2007. ACM.
3. M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. *Theory of Computing Systems*, 47:934–962, 2010.
4. G. Greiner and R. Jacob. The I/O complexity of sparse matrix dense matrix multiplication. In *Proceedings of LATIN'10*, volume 6034 of *Lecture Notes in Computer Science*, pages 143–156. Springer, 2010.
5. Hong, Jia-Wei and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of STOC '81*, pages 326–333, New York, NY, USA, 1981. ACM.
6. R. Jacob and M. Schnupp. Experimental performance of I/O-optimal sparse matrix dense vector multiplication algorithms within main memory. Technical Report TUM-I1017, Technische Universität München, 2010.
7. T. Lieber. Combinatorial approaches to optimizing sparse matrix dense vector multiplication in the I/O-model. Master's thesis, Informatik Technische Universität München, 2009.
8. F. F. Roos, R. Jacob, J. Grossmann, B. Fischer, J. M. Buhmann, W. Gruissem, S. Baginsky, and P. Widmayer. Pepsplice: cache-efficient search algorithms for comprehensive identification of tandem mass spectra. *Bioinformatics*, 23(22):3016–3023, 2007.
9. R. W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, Fall 2003.