Faster Update Time for Turnstile Streaming Algorithms

Josh Alman* Huacheng Yu†

October 31, 2019

Abstract

In this paper, we present a new algorithm for maintaining linear sketches in turnstile streams with faster update time. As an application, we show that $\log n$ `Count` sketches or `CountMin` sketches with a constant number of columns (i.e., buckets) can be implicitly maintained in *worst-case* $O(\log^{0.582} n)$ update time using $O(\log n)$ words of space, on a standard word RAM with word-size $w = \Theta(\log n)$. The exponent $0.582 \approx 2\omega/3 - 1$, where ω is the current matrix multiplication exponent. Due to the numerous applications of linear sketches, our algorithm improves the update time for many streaming problems in turnstile streams, in the high success probability setting, without using more space, including ℓ_2 norm estimation, ℓ_2 heavy hitters, point query with ℓ_1 or ℓ_2 error, etc. Our algorithm generalizes, with the same update time and space, to maintaining $\log n$ linear sketches, where each sketch

1. partitions the coordinates into $k < \log^{o(1)} n$ buckets using a c-wise independent hash function for constant c,
2. maintains the sum of coordinates for each bucket.

Moreover, if arbitrary word operations are allowed, the update time can be further improved to $O(\log^{0.187} n)$, where $0.187 \approx \omega/2 - 1$. Our update algorithm is adaptive, and it circumvents the non-adaptive cell-probe lower bounds for turnstile streaming algorithms by Larsen, Nelson and Nguyên (STOC'15).

On the other hand, our result also shows that proving unconditional cell-probe lower bound for the update time seems very difficult, even if the space is restricted to be (nearly) the optimum. If $\omega = 2$, the cell-probe update time of our algorithm would be $\log^{o(1)} n$. Hence, proving any higher lower bound would imply $\omega > 2$.

*Harvard University. joshuaalman@gmail.com. Supported in part by NSF CCF-1651838 and NSF CCF-1741615.

†Princeton University. yuhch123@gmail.com. Supported in part by ONR grant N00014-17-1-2127, a Simons Investigator Award and NSF Award CCF 1715187.

1 Introduction

Linear sketching has numerous applications in streaming algorithms (to list a few [AMS99, CCF04, Ind06, Li08, KNW10]), especially for *turnstile* streams [Mut05]. In the turnstile streaming model, the data structure wants to maintain a vector $\nu \in \mathbb{Z}^n$, under updates of form (u, Δ) for $u \in [n]$ and $\Delta \in \mathbb{Z}$, which changes ν_u to $\nu_u + \Delta$. Each entry ν_i's is usually assumed to be bounded by $\operatorname{poly} n$ at any time. Occasionally, the data structure must answer queries about ν. Special cases of turnstile streams include the incremental streams ($\Delta = 1$) and singleton insertions and deletions ($\Delta = \pm 1$). For turnstile streams, a linear-sketching streaming algorithm maintains a vector $A\nu$ in memory, for some random (but carefully sampled) matrix $A \in \mathbb{Z}^{s \times n}$ and $s \ll n$, such that the (much shorter) vector $A\nu$ provides sufficient information to answer the queries with good probability. Due to the nature of linear transformations, the updates can be maintained easily. To handle an update (u, Δ), the algorithm simply computes $\Delta \cdot Le_u$, and adds it to $A\nu$, where e_u is the u-th unit vector.

However, one single instance of linear sketch is often not very reliable, e.g., one basic `Count` sketch [AMS99, CCF04] could only provide a constant approximation of x's second moment ($\|x\|_2^2$) with a (small) constant probability. The standard approach to boost the reliability, say to $1/n^2$ failure probability, is to maintain $O(\log n)$ independent sketches, i.e., independently sample $O(\log n)$ A (or equivalently, sample a taller A with $\log n$ times more rows). This approach would naturally cause a blowup in the space by a factor of $\log n$, as well as a slowdown in the update time. It was shown by Jayram and Woodruff [JW13] that the space blowup is necessary, however, it is unclear if one also needs to pay the $\log n$ factor in the update time. The update time could sometimes be even *more important* than the space [LNN15], since the updates can arrive from the data stream at an extremely high rate (e.g., see [TZ12]). A single instance of the linear sketch usually occupies logarithmic or less space, an extra $\log n$ factor is often acceptable. However, if the update processing speed does not at least match the rate at which the updates in the stream arrive, the whole system might fail.

In linear-sketch based algorithms, the space is proportional to the number of rows in A, and the update time is proportional to the *column sparsity*, which is the number of non-zero entries in each column (since we only need to update the non-zero entries in $\Delta \cdot Ae_u$). Hence, one conceivable and common approach to speed up the update time is to use a more sparse A. Unfortunately, this approach is known to have limitations. Larsen, Nelson and Nguyên [LNN15] proved that A cannot have few rows and be column sparse at the same time for several turnstile streaming problems. In fact, they showed tradeoffs between update time and space for the more general *non-adaptive* data structures. That is, the memory locations written or read during each update, can only depend on the updated index u and the random bits (but not the memory contents of the data structure). Note that all linear-sketch based algorithms are non-adaptive. Their lower bounds also apply to cell-probe data structures, i.e., the above lower bounds hold even if we only count the number of memory words read or written by the data structure. For the problems considered in [LNN15], all known solutions use non-adaptive update algorithms. Hence, in order to obtain faster update times, we would require a completely new strategy.

1.1 Our results

We propose a generic solution for efficiently maintaining linear sketches using fast matrix multiplication, and apply it to a large class of streaming problems, including the problems discussed in [LNN15]. Our new algorithm is adaptive, and hence, it circumvents the previous non-adaptive lower bound.

Our algorithm applies to any linear sketches of the following form:

1. partition all coordinates $[n]$ into k buckets using a c-wise independent hash function h for some constant c;[1]
2. maintain the sum $\sum_{u:h(u)=b} \nu_u$ for each bucket $b \in [k]$ (equivalently, so far A has k rows, and each column has exactly one 1 in a c-wise independently chosen row);
3. repeat T times independently.

Denote such a linear sketch by lin-skt(k, c, T). Maintaining a lin-skt(k, c, T) using the straightforward approach takes $O(kT)$ words of space and has $O(T)$ update time. When k is small, the corresponding matrix A is dense. It is worth noting that the celebrated `Count` sketch [CCF04] and the `CountMin` sketch [CM05] both have this form.[2] These two linear sketches have wide applications to many streaming problems, including ℓ_2 norm estimation, ℓ_1 heavy hitters, ℓ_2 heavy hitters, point query with ℓ_1 error, point query with ℓ_2 error, etc.

In this paper, we present a direct improvement on the update time of lin-skt(k, c, T) for small k, without using more space. In particular, when $k < \log^{o(1)} n$ and $T = \Theta(\log n)$, our algorithm has the following guarantees.

Theorem 1. *For any problem that admits a lin-skt(k, c, T) linear sketch solution, where $k < \log^{o(1)} n$ is a power of two, and $T = O(\log n)$, there is an algorithm that*

- *uses $O(k \log n)$ words of space,*
- *has* worst-case *update time $O(\log^{0.582} n)$, and*
- *additive* extra *query time $O(\log^{1.582} n)$*

on a standard word RAM with word-size $w = \Theta(\log n)$, where the exponent $0.582 \approx 2\omega/3 - 1$, and ω is the current matrix multiplication exponent. Moreover, if we allow arbitrary w-bit word operations, the update time and extra query time can be further reduced to $O(\log^{\omega/2-1+o(1)} n) = O(\log^{0.187} n)$ and $O(\log^{\omega/2+o(1)} n) = O(\log^{1.187} n)$ respectively.

As an application, our theorem implies that

- one can maintain a constant approximation of the ℓ_2 norm of ν with probability $1 - 1/n^{O(1)}$ using $O(\log n)$ words of space and worst-case update time $O(\log^{0.582} n)$;
- one can construct a data structure for nonnegative ν using $O(\log n)$ words of space and worst-case update time $O(\log^{0.582} n)$, supporting point queries with $n^{-O(1)}$ failure probability and ℓ_1-error guarantees, i.e., given an index $i \in [n]$, the data structure returns $\nu_i \pm 0.1\|\nu\|_1$ with probability $1 - 1/n^{O(1)}$.

Assuming one can do arbitrary word operations, the update times are further reduced to $O(\log^{0.187} n)$. It is worth noting that the lower bound by Larsen, Nelson and Nguyên asserts that any nonadaptive *cell-probe* data structure (even if we only count the number of memory accesses at the update time) for the above two problems must have update time at least $\tilde{\Omega}(\sqrt{\log n})$ if one uses $\operatorname{poly} \log n$ space. Hence, our new streaming algorithm "breaks" their lower bound by using adaptivity in the update algorithm.

[1] Most streaming algorithms need no more than constant-wise independence.

[2] The `Count` sketch assigns c-wise independent ± 1 random weights to each coordinate, and maintains the weighted sum. One may view all $+1$ coordinates forming one bucket and the -1 coordinates forming another bucket. Then the query algorithm may manually subtract the sums of the two buckets to obtain the weighted sum.

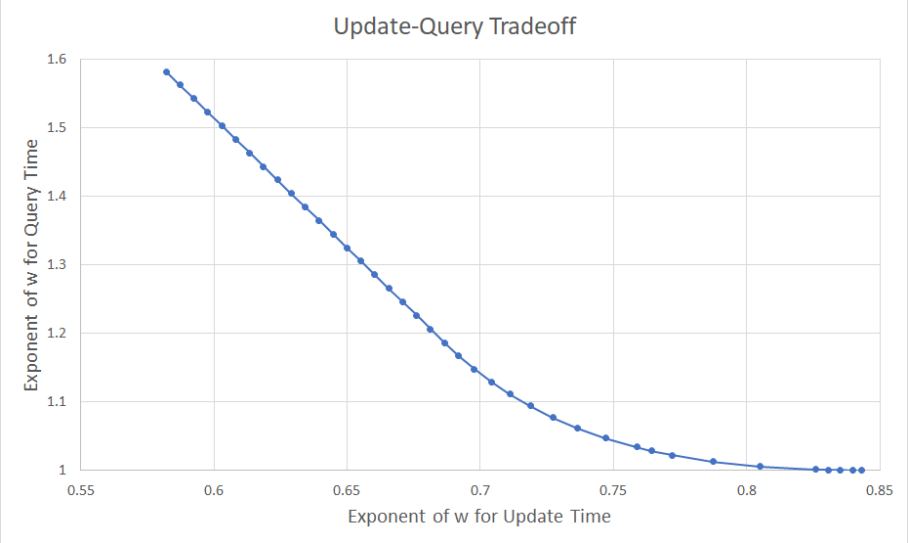

Figure 1: Trade-off between the exponent of the query time and update time in our algorithm in the word RAM model, from combining Theorems 1 and 4 below with the best known algorithm for rectangular matrix multiplication [GU18].

It turns out the only non-standard w-bit word operation needed to get $O(\log^{0.187} n)$ update time is to multiply two $w^{1/2}$ by $w^{1/2}$ 0-1 matrices. We refer to a word RAM equipped with this matrix multiplication operation as a word RAM$^{\text{MM}}$ (see Section 2.2.2).

Our new algorithm improves the update time by implicitly maintaining the linear sketch. Hence, in order to answer a query, one would need to first preprocess the data structure to recover the sketches. It may cause a slower query time by having a preprocessing stage with $O(\log^{1.582} n)$ time. Some applications may have a very slow query time to begin with (e.g., ℓ_1 heavy hitters [CM05]), in which case, the additive $O(\log^{1.582} n)$ time is insignificant. However, for applications with $O(\log n)$ query time (e.g., point query with ℓ_1 error [CM05]), the preprocessing time becomes the bottleneck.

To allow for a faster query time for those problems, we further provide a smooth tradeoff between the update time and the extra query time (see Figure 1). In particular, the other end of this tradeoff curve is an algorithm with $\log^{0.844} n$ update time and $\log^{1+o(1)} n$ extra query time. To the best of our knowledge, in every application, the query algorithm needs to spend at least a constant time on each of the $O(\log n)$ sketches, the query time was already at least $\Omega(\log n)$. This tradeoff point almost does not slow down the query time, while it still improves the update time non-trivially.

Theorem 2. *For any problem that admits a lin-skt(k, c, T) linear sketch solution, where $k < \log^{o(1)} n$ is a power of two, and $T = O(\log n)$, there is an algorithm that*

- *uses $O(k \log n)$ words of space,*
- *has* worst-case *update time $O(\log^{0.844} n)$, and*
- *additive* extra *query time $O(\log^{1+o(1)} n)$*

on a standard word RAM with word-size $w = \Theta(\log n)$, where the exponent $0.844 \approx 1 - \alpha/2$, and α is the current dual matrix multiplication exponent.

To obtain the faster update time, our new algorithms use the following simple observation that connects the *worst-case* update time to the running time of a computation problem.

Proposition 1 (informal)**.** *For any streaming algorithm $\mathcal{A}$ that uses $O(S)$ words of space, if one can perform $O(S)$ updates in $O(t_{bu})$ time and $O(S)$ words of (extra) space, then $\mathcal{A}$ can be simulated using $O(S)$ words and $O(t_{bu}/S)$* worst-case *update time.*

As a byproduct, we obtain a "hardness result for proving hardness." The above proposition implies that proving any super constant update time lower bound, even assuming that the data structure uses optimal space up to a constant factor, would result in a super linear time lower bound for linear space algorithms solving some computation problem in the RAM model, which seems beyond the scope of our current techniques. In particular, Theorem 1 implies that proving any $\log^{\epsilon} n$ lower bound for any problem that admits a lin-skt$(O(1), O(1), \log n)$ solution would imply a non-trivial lower bound for matrix multiplication. We note that although all computational problems admit linear time algorithm in the *cell-probe* model, Proposition 1 *does not* imply a streaming algorithm with faster update time in the *cell-probe* model. We also note that recently, Dvir, Golovnev and Weinstein [DGW19] proved a hardness result for proving space-*query time* tradeoff lower bounds, but with less "severe" consequences. See more discussions in the next subsection.

1.2 Our technique

The key steps that lead to the improved update time are

- observing that handling a batch of B updates in $o(B \cdot T)$ time implies an $o(T)$ *worst-case* update time algorithm;
- designing a faster batch update algorithm for linear sketches.

As a running example, let us focus on lin-skt$(2, 2, \log n)$, i.e., we partition $[n]$ into two buckets using pairwise hash families, maintain the sum of each bucket, and repeat $T = \log n$ times independently. Note that the straightforward implementation takes $O(\log n)$ space and update time.

Handling B updates in $o(BT) = o(B \log n)$ time naturally implies an algorithm with $o(\log n)$ *amortized* update time: temporarily store each update in a buffer of size B, and handle the updates all at once for every B updates. By applying a trick similar to *global rebuilding* of Overmars [Ove83], we show that a faster batch update algorithm also leads to lower *worst-case* update time. To this end, we store two buffers each of size B, one of which is the *active* buffer. We assume that the first buffer is active in the beginning. The new updates are always appended to the active buffer. Then after B updates, the active buffer becomes full. Now we switch the second buffer to be active, and in the next B updates, we fill up the second buffer while *gradually flushing the first buffer*. That is, if there is an $O(Bt_u)$ time algorithm handling all B updates, we are going to simulate it for $O(t_u)$ steps in each of the next B updates. Therefore, when the second buffer becomes full again, we will have already emptied the first buffer, and now we can switch the first buffer back to active and start to flush the second buffer. Each update takes $O(t_u)$ time in worst case. We present the details in Section 3. Note that this reduction *does not* work for cell-probe algorithms, because it requires us to simulate the algorithm for a small number of steps in each update. Such simulation is only possible on a RAM for a RAM algorithm, but not possible in the cell-probe model for a cell-probe algorithm.

For lin-skt$(2, 2, \log n)$, the space usage is $O(\log n)$ words. Hence, we can afford to set $B = \log n$ without using more space (except for a constant factor). We then show that $\log n$ updates can be applied all together in $o(\log^2 n)$ time. Each update (u, Δ) adds the u-th column of A multiplied by Δ, to $A\nu$.

Thus, $\log n$ updates add the sum of $\log n$ columns multiplied by (possibly different) numbers. To compute this sum, it suffices to multiply a $\log n$ by $\log n$ 0-1 matrix, where each column is a column in A, by a $\log n$-dimensional vector with $\log n$-bit entries, where each entry is a Δ.

The task is boiled down to the following: a) given $\log n$ indices $u \in [n]$, compute for each u, the u-th column in A, which reduces to evaluating $\log n$ pairwise independent hash functions for each u; b) compute the above matrix-vector product. We show that for a carefully chosen pairwise hash family, both parts can be reduced to $\log n \times \log n \times \log n$ 0-1 matrix multiplications.

Finally, we show that this matrix multiplication can be computed in $\log^{2\omega/3} n$ time on a word RAM with word-size $w = \log n$, and $\log^{\omega/2} n$ time on a word RAM$^{\text{MM}}$, where $\omega < 2.373$ is the current matrix multiplication exponent, using $O(\log n)$ words of space.[3] We do this by combining the usual recursive approach to designing fast matrix multiplication algorithms, together with larger-than-usual base cases which can be multiplied in constant time using word operations. In the word RAM model, this involves packing many short vectors into two words so that, when the words are multiplied as integers, the pairwise inner products between those vectors can be read off from the result.

The extra query time of the above algorithm is $O(\log^{2\omega/3} n) = O(\log^{1.582} n)$ time, since we will have to complete the buffer-flushing algorithm, before the actual query algorithm can be launched. To reduce the extra query time, we reduce the buffer size, so that flushing the buffer takes only $\log^{1+o(1)} n$ time. The above arguments still apply, but now, the problem is reduced to rectangular matrix multiplication. The buffer size is reduced sufficiently so that the three dimensions in the matrix multiplication problem are very imbalanced. Finally, we present an algorithm that multiplies $w \times w^\theta \times w$ 0-1 matrices in $w^{1+o(1)}$ time for some constant $\theta > 0$, which is almost linear in the output size.

1.3 Organization

In Section 3, we present the connection between computing batch update efficiently and faster worst-case update time. In Section 4, we present how to do the batch update using matrix multiplication. In Section 5, we present the matrix multiplication algorithms for small matrices.

2 Preliminaries

2.1 Notation

Throughout this paper, we write $\tilde{O}$ to hide $\operatorname{poly}\log\log(n)$ multiplicative factors, so $\tilde{O}(T) = T \cdot \operatorname{poly}\log\log n$ and $\tilde{O}(1) = \operatorname{poly}\log\log n$. When q is a power of a prime, we write $\mathbb{F}_q$ for the finite field of order q. For $n \in \mathbb{N}$, we write $[n] := \{1, 2, \ldots, n\}$.

2.2 Model of Computation

We focus in this paper on the word RAM model of computation [FW90]. In the model for word size w (typically we pick $w = \Theta(\log n)$ where n is the input size), the algorithm has random access to words of memory, each of which stores w bits. An algorithm is allowed to perform any "standard" word operations, which only take as input a constant number of words, in constant time. Of course, the efficiency of an algorithm can vary depending on what word operations are considered standard; here we consider two options.

[3] Note that the input and output sizes are both $O(\log n)$ words.

2.2.1 Standard word RAM

In the first option, which we will call the word RAM model, we only allow for the following "simple" word operations: $+, -, \times, /$, bit-wise AND, OR, XOR, negation, and bit-shifts.[4] This is the most basic definition of the word RAM model used in the literature, and as these operations are so standard, that algorithms designed in this model should be implementable in any word RAM architecture. Nonetheless, many more complicated operations are known to require only $\tilde{O}(1)$ time as well, by combining the simple operations in clever ways. We will make use of the following operations from past work:

Proposition 2 ([BMM97, Proposition 1])**.** *For any fixed permutation π on m symbols, given a bit vector $x[1]x[2]\cdots x[m]$, we can compute the permutation $x[\pi(1)]x[\pi(2)]\cdots x[\pi(m)]$ in time $\tilde{O}(\lceil m/w \rceil)$.*

Proposition 3 ([AH74, Theorem 8.7])**.** *Given two polynomials $f, g \in \mathbb{F}_2[z]$ of degree at most w, we can compute $f(z) \cdot g(z)$ and $f(z) \pmod{g(z)}$ in time $\tilde{O}(1)$.*

Therefore, if we represent each field element in $\mathbb{F}_{2^w}$ as a degree-w polynomial $f(z)$, and encode it by writing down the coefficients, then any field operation can be computed in $\tilde{O}(1)$ time.

2.2.2 Word RAM$^{\text{MM}}$

In the second option, which we call the word RAM$^{\text{MM}}$ model, an algorithm is additionally allowed to multiply any two matrices over $\mathbb{Z}$ which each fit into a single word as a single word operation. For instance, the algorithm could multiply two $\sqrt{w} \times \sqrt{w}$ matrices whose entires are $O(1)$-bit integers in $\tilde{O}(1)$ time. This model is not necessarily unrealistic: in practice, engineers may design specific hardware to handle certain important word operations in order to speed up algorithms, and in particular, there has been substantial work on hardware for the fast multiplication of small matrices (see e.g. [FSH04, Fuj08, Mni09, LKC$^+$10]).

That said, this model is particularly interesting from a theoretical perspective, because of its relationship with the *cell-probe model*. Lower bounds against streaming algorithms are typically proved in the cell-probe model, in which an algorithm is only charged for accessing words of memory, and not for any computation on the contents; the word RAM$^{\text{MM}}$ model is still weaker than the cell-probe model. We will show that a lin-skt$(2, 2, \log n)$ linear sketch can be maintained with $O(w^{\omega/2-1})$ update time and $O(w^{\omega/2})$ query time in the word RAM$^{\text{MM}}$ model, where ω is the matrix multiplication constant[5]. If $\omega = 2$ then this would result in $\tilde{O}(1)$ update time and $\tilde{O}(w)$ query time. Hence, any cell-probe lower bounds for this problem could have substantial implications in arithmetic complexity theory.

3 Reduction to Batch Problem

Consider a dynamic problem $\mathcal{P}$ with updates $\mathcal{U}$ (and queries $\mathcal{Q}$) on word RAM with word-size w. For now, let us assume each update can fit in one word, i.e., $|\mathcal{U}| \leq 2^w$. Let S be the minimum number of *words* required to solve $\mathcal{P}$. In the following, we show that an algorithm that handles a batch of updates can be transformed into a data structure with *worst-case* update time.

More specifically, fix a data structure $\mathcal{D}$ for $\mathcal{P}$ that uses S words of space and has query time t_q, and consider the following computational problem.

[4] In fact, multiplication can be replaced with just bit-shifts, and multiplication can still be performed in $\tilde{O}(1)$ time [BMM97].

[5] Here we are using the 'usual' definition of ω, in terms of the size of an arithmetic circuit for performing matrix multiplication. Matrix multiplication can be performed faster than this in the cell probe model, but as discussed in Section 1.2, it is not evident how to use this in conjunction with our approach to design a faster cell probe algorithm.

Batch-Update$_B^{\mathcal{D}}$: Given a memory state M of $\mathcal{D}$ and B updates $u_1, u_2, \ldots, u_B \in \mathcal{U}$, compute the new memory state after all B updates are applied in the order. If $\mathcal{D}$ is randomized, sample a new memory state according to the distribution defined by $\mathcal{D}$.

Note that the input length of this problem is $O(S + B)$ words. The following theorem asserts that if the B updates can be handled in a batch efficiently, then $\mathcal{P}$ has a solution with *worst-case* update time.

Proposition 1 (restated). *For any $B \leq O(S)$, if there is a RAM algorithm $\mathcal{A}$ that solves Batch-Update$_B^{\mathcal{D}}$ in time t_{bu} and space $O(S)$ words, then there is a data structure for $\mathcal{P}$ that*

- *uses $O(S)$ words of space,*
- *has worst-case update time $O(t_{bu}/B)$, and*
- *worst-case query time $O(t_{bu} + t_q)$.*

In particular, the contrapositive implies if the update time must be super constant, then Batch-Update$_B^{\mathcal{D}}$ has no linear time algorithm on RAM.

Proof. To construct a data structure with fast update time, the natural idea is to buffer the updates and handle the updates in a batch using $\mathcal{A}$. The update time would then be $O(t_{\text{bu}}/B)$ *amortized*. However, it turns out that a simple trick can deamortizes it.

We will use two buffers *buf*$_0$ and *buf*$_1$, both of size B. Each time we receive an update, it is put into *buf*$_0$. Once *buf*$_0$ becomes full, we are going to put the subsequent updates into *buf*$_1$, while at the same time we gradually flush the first buffer *buf*$_0$. That is, each time we receive a new update, it is put into *buf*$_1$, then we simulate $\mathcal{A}$, which handles all B updates in *buf*$_0$ in t_{bu} time, for $O(t_{\text{bu}}/B)$ steps.[6] Since *buf*$_1$ can hold another B updates, we will be able to finish simulating $\mathcal{A}$ before it gets full. Once *buf*$_1$ becomes full, we will switch the roles of the two buffers: put the subsequent updates into *buf*$_0$ and gradually flush *buf*$_1$ using $\mathcal{A}$. To answer a query, it suffices to finish flushing the buffer, and then run the query algorithm of $\mathcal{D}$. See Figure 2 for details.

Since $\mathcal{A}$ uses $O(B)$ words of space, the total space usage is $O(B)$. The update time is $O(t_{\text{bu}}/B)$ in worst-case, and the query time is $O(t_{\text{bu}} + t_q)$.

□

4 Update Efficient Streaming Algorithm

In this section, we present our update-efficient streaming algorithm for lin-skt(k, c, T), and prove Theorem 1.

Theorem 1 (restated). *For any problem that admits a lin-skt(k, c, T) linear sketch solution, where $k < \log^{o(1)} n$ is a power of two, and $T = O(\log n)$, there is an algorithm that*

- *uses $O(k \log n)$ words of space,*
- *has* worst-case *update time $O(\log^{0.582} n)$, and*

[6] Note that a RAM algorithm uses only $O(1)$ registers, including a pointer to the line of code it is currently executing. Hence, with an extra counter, one can run a RAM algorithm for a certain number of steps and pause. Next time, we may continue from there.

memory:
1. M: memory state of $\mathcal{D}$, initialized according to $\mathcal{D}$ // S words
2. buf_0, buf_1: two initially empty buffers that each can store up to B updates // B words each
3. b: indicate the active buffer, initially set to 0 // 1 bit
4. *temp*: the working memory of $\mathcal{A}$ // $O(S)$ words

update(u): // handle an update $u \in \mathcal{U}$
1. append u to buf_b
2. **if** buf_b is full **then**
3. $\quad b \leftarrow 1 - b$
4. $\quad$ start a new instance of background_update()
5. **if** background_update() has not terminated **then**
6. $\quad$ simulate background_update() for $O(t_{\text{bu}}/S)$ steps

background_update():
1. reset *temp*
2. run $\mathcal{A}$ on (M, buf_{1-b}) and obtain new memory state M'
3. copy M' to M
4. empty buf_{1-b}

query(q): // handle a query $q \in \mathcal{Q}$
1. **if** background_update() has not terminated **then**
2. $\quad$ finish background_update()
3. $b \leftarrow 1 - b$
4. background_update()
5. run the query algorithm of $\mathcal{D}$

Figure 2: Data structure for $\mathcal{P}$ with $O(t_{\text{bu}}/B)$ worst-case update time and $O(S)$ space.

- *additive* extra *query time* $O(\log^{1.582} n)$

on a standard word RAM with word-size $w = \Theta(\log n)$*, where the exponent* $0.582 \approx 2\omega/3 - 1$*, and* ω *is the current matrix multiplication exponent. Moreover, the algorithm can be implemented on a word RAM*^{MM} *with the update time and extra query time* $O(\log^{\omega/2-1+o(1)} n) = O(\log^{0.187} n)$ *and* $O(\log^{\omega/2+o(1)} n) = O(\log^{1.187} n)$ *respectively.*

To prove the theorem, we first apply Proposition 1 and set the buffer size $B = \log n$, and reduce the problem to applying $\log n$ updates in batch. It turns out that to apply $\log n$ updates, it suffices to

1. evaluate $\log n$ c-wise independent hash families on all $\log n$ updated indices, and
2. compute a matrix-vector product, where the matrix is a $\log n \times \log n$ binary matrix, and the vector has $\log n$ dimensions with $\log n$-bit numbers in the entries.

In the subsections below, we show that both tasks can be done efficiently using fast matrix multiplication for small matrices.

Lemma 1. *For any constant* $c \geq 2$*, there is a* c*-wise independent hash family* $h_s : \{0,1\}^w \to \{0,1\}$ *for* $s \in \{0,1\}^{cw}$*, such that given* w *seeds* $s_1, \ldots, s_w$ *and* w *inputs* $u_1, \ldots, u_w$*, one can compute* w w*-bit strings* $v_1, \ldots, v_w$ *in* $O(w^{1.582})$ *time and* $O(w)$ *words of space, such that* v_i *stores the* w *hash values of* u_i*, i.e., the* j*-th bit of* v_i *is equal to* $h_{s_j}(u_i)$ *for all* $i, j \in [w]$*. Moreover, the running time can be reduced to* $O(w^{1.187})$ *on a word RAM*^{MM}*.*

Lemma 2. *Given as input a matrix $A \in \{0,1\}^{w \times w}$ and a vector $v \in \mathbb{Z}^w$ of w-bit integers, one can compute the product Av in $O(w)$ words of space and time $O(w^{1.582})$ on a word RAM, and in time $O(w^{1.187})$ on a word RAM^{MM}.*

Now, we proof Theorem 1 using the above lemmas.

Proof of Theorem 1. We first design a fast batch update algorithm, which handles $w = \log n$ updates to lin-skt$(k, c, \log n)$ in $O(w^{1.582})$ time on a word RAM. The algorithm is given as input, a memory state of lin-skt$(k, c, \log n)$ and w updates (u_1, Δ_1), ..., (u_w, Δ_w). By definition, lin-skt(k, c, w) uses w c-wise independent hash functions $h_1, \ldots, h_w : [n] \to [k]$, and stores the random seeds, as well as kw counters: $\sum_{u:h_j(u)=b} \nu_u$ for each $j \in [w]$ and $b \in [k]$. To perform the batch update, we first compute for all $i, j \in [w]$, which of the kw counters Δ_i needs to be added to.

To this end, we use the c-wise independent hash family and the batch evaluation algorithm in Lemma 1. Note that Lemma 1 only considers such hash functions with one bit output. To apply to our problem, we apply the lemma on each of the $\log k$ output bits, since k is a power of two. Therefore, in $O(w^{1.582})$ time, we compute for every u_i, $\log k$ bit-strings $v'_{i,1}, \ldots, v'_{i,\log k}$ such that for each $j \in [w]$, the j-th bits of the $\log k$ strings encode the binary representation of $h_j(u_i)$.

Next, we post-process the binary representations into indicator vectors. That is, we will compute a kw-bit string v_i, stored in k words, that encodes for each of the kw counters, whether Δ_i needs to be added to it. This can be done in $O(k \log k)$ time: For each $b \in [k]$, a w-bit string indicating if $h_j(u_i) = b$ for each $j \in [w]$, can be computed by doing $O(\log k)$ bit-wise ANDs and negations.

After such transformation, we compute the changes to the kw counters. More specifically, the l-th counter needs to increase by $\sum_{i:\text{ the } l\text{-th bit of } v_i = 1} \Delta_i$. The task is precisely computing the matrix-vector multiplication of

- a kw by w binary matrix, whose columns are $v_1, \ldots, v_w$, and
- a w dimensional vector, whose entries are w-bit integers Δ_i.

Next, we will use Lemma 2 to compute the product. We first use Proposition 2 to permute the bits in the matrix and the vector, so that it matches the input format of the lemma, which takes $\tilde{O}(kw)$ time. Then, we apply the lemma, since $k < \log^{o(1)} n$, the product can be computed in $O(w^{1.582})$ time. At last, we use Proposition 2 again to permute the bits in the output, so that the l-th word of the output contains the l-th entry in the product vector, i.e., the change to the l-th counter. We add the product to counters.

The batch update algorithm for w updates runs in $O(w^{1.582})$ time, and $O(kw)$ space. Finally, the theorem is proved by applying Proposition 1 for $B = w$. By applying the word RAM$^{\text{MM}}$ versions of Lemma 1 and Lemma 2, we obtain the claimed update and query times for word RAM$^{\text{MM}}$. □

Both of Lemma 1 and Lemma 2 use fast matrix multiplication for small matrices.

Theorem 3. *In the word RAM^{MM} model, one can perform $w \times w \times w$ matrix multiplication over $\mathbb{F}_q$ for $q \leq \text{poly}(w)$ in time $O(w^{\omega/2+\varepsilon})$ for any $\varepsilon > 0$.*

Theorem 4. *In the word RAM model, for any $p \in [0,1]$, and any $\varepsilon > 0$, one can perform $w \times w^p \times w$ matrix multiplication over $\mathbb{F}_q$ for $q \leq \text{poly}(w)$ in time*

- $O(w^{(1+p)\omega/3+\varepsilon}) \leq O(w^{0.791\cdot(1+p)})$ *if $p \geq 1/2$,*
- $O(w^{\omega(1,2p,1)/2+\varepsilon})$ *if $p \leq 1/2$.*

The proof of the two theorems are deferred to Section 5. In the next two subsections, we prove Lemma 1 and Lemma 2 respectively.

4.1 Evaluating c-wise Hash Functions

In the following, we prove Lemma 1.

Lemma 1 (restated). *For any constant $c \geq 2$, there is a c-wise independent hash family $h_s : \{0,1\}^w \to \{0,1\}$ for $s \in \{0,1\}^{cw}$, such that given w seeds $s_1, \ldots, s_w$ and w inputs $u_1, \ldots, u_w$, one can compute w w-bit strings $v_1, \ldots, v_w$ in $O(w^{1.582})$ time and $O(w)$ words of space, such that v_i stores the w hash values of u_i, i.e., the j-th bit of v_i is equal to $h_{s_j}(u_i)$ for all $i, j \in [w]$. Moreover, the running time can be reduced to $O(w^{1.187})$ on a word RAM^{MM}.*

Proof. We use the following c-wise independent family h_s: given input $u \in \{0,1\}^w$, first generate a vector $g(u) \in \mathbb{F}_2^{cw}$ such that for any c different $u_1, \ldots, u_c \in \{0,1\}^w$, the corresponding vectors $g(u_1), \ldots, g(u_c)$ are linearly independent; then we take the seed s also in $\mathbb{F}_2^{cw}$, and let

$$h_s(u) := \langle s, g(u) \rangle .$$

Note that h_s is c-wise independent, because for any $u_1, \ldots, u_c \in \{0,1\}^w$, and $y_1, \ldots, y_c \in \{0,1\}$, we have

$$\Pr_s[\forall i, \langle s, g(u_i) \rangle = y_i] = 2^{-c},$$

due to the linear independence of $\{g(u_i)\}$.

Suppose such $g(u)$ can be computed efficiently, then evaluating all h_{s_j} on all u_i becomes to compute a matrix multiplication over $\mathbb{F}_2$: Let $\mathbf{S}$ be a w by cw matrix, where the j-th row is s_j for all j, and let $\mathbf{X}$ be a cw by w matrix, where the i-th column is $g(u_i)$ for all i, then the (j,i)-th entry of the product matrix $\mathbf{SX}$ is exactly the inner product $\langle s_j, g(u_i) \rangle$, i.e., $h_{s_j}(u_i)$. By Theorem 3 and Theorem 4, $\mathbf{SX}$ can be computed in $O(w^{1.582})$ time on a word RAM, or $O(w^{1.187})$ time on a word RAM$^{\text{MM}}$.

Next, we give a construction of $g(u)$ with the above c-wise linear independence property, and show that it can be computed efficiently. We first view $u \in \{0,1\}^w$ as an element in $\mathbb{F}_{2^w}$, encoded in a canonical form. That is, we fix any irreducible degree-w polynomial $Q(z) \in \mathbb{F}_2[z]$, and store it explicitly in memory. Suppose $u = (u(0), u(1), \ldots, u(w-1))$, we view it as the polynomial $u(0) + u(1)z + \cdots + u(w-1)z^{w-1}$ in $\mathbb{F}_2[z]$, which is an element in $\mathbb{F}_2[z]/(Q) \cong \mathbb{F}_{2^w}$.

We define $g(u) := (1, u, u^2, \ldots, u^{c-1})$, where each u^i is computed in $\mathbb{F}_{2^w}$ and uses the above encoding. Thus, $g(u)$ can either be viewed as a vector in $\mathbb{F}_{2^w}^c$ or a vector in $\mathbb{F}_2^{cw}$. Note that in both views, adding two vectors yield the same result (both are bit-wise XOR). By the fact that Vandermonde matrices have full rank, for any different $u_1, \ldots, u_c$, and $\alpha_1, \ldots, \alpha_c \in \mathbb{F}_{2^w}$ that are not all zero, we have

$$\alpha_1 \cdot g(u_1) + \cdots + \alpha_c \cdot g(u_c) \neq 0.$$

In particular, it holds for any $\alpha_1, \ldots, \alpha_c \in \{0,1\}$ that are not all zero, which implies that $g(u_1), \ldots, g(u_c)$ are linearly independent as vectors in $\mathbb{F}_2^{cw}$. Finally, by Proposition 3, each $g(u)$ can be computed in $\tilde{O}(c)$ time. This proves the lemma. □

4.2 Matrix-Vector Multiplication

In this subsection, we prove Lemma 2.

Lemma 2 (restated). *Given as input a matrix $A \in \{0,1\}^{w \times w}$ and a vector $v \in \mathbb{Z}^w$ of w-bit integers, one can compute the product Av in $O(w)$ words of space and time $O(w^{1.582})$ on a word RAM, and in time $O(w^{1.187})$ on a word RAM^{MM}.*

Proof. First, construct the matrix $B \in \{0,1\}^{w \times w}$, whose entry $B[i,j]$ is the jth bit of $v[i]$ when written out in binary; in particular, $v[i] = \sum_{j=1}^{w} 2^{j-1} B[i,j]$. Next, use the given algorithm to compute the product $C := AB$ over $\mathbb{F}_q$. Note that since $q > w$ and the entries of A and B are all in $\{0,1\}$, it follows that C is also the product of A and B over $\mathbb{Z}$. Finally, output the vector $v' \in \mathbb{Z}^w$ given by $v'[i] = \sum_{j=1}^{w} 2^{j-1} C[i,j]$. To see that it is correct, note that:

$$\begin{aligned} v'[i] &= \sum_{j=1}^{w} 2^{j-1} C[i,j] \\ &= \sum_{j=1}^{w} 2^{j-1} \left(\sum_{k=1}^{w} A[i,k] B[k,j] \right) \\ &= \sum_{k=1}^{w} A[i,k] \left(\sum_{j=1}^{w} 2^{j-1} B[k,j] \right) \\ &= \sum_{k=1}^{w} A[i,k] v[k], \end{aligned}$$

which is exactly the desired output. The bottleneck of the running time is to compute the matrix product AB, which by Theorem 3 and Theorem 4, has the claimed running time. □

4.3 Faster query time

In this subsection, we describe how to obtain $\log^{1+o(1)} n$ extra query time. Again, we use Proposition 1. The idea is to set the buffer size B to be smaller, so that B updates can be handled in $\log^{1+o(1)} n$ time. Hence, the worst-case update time is $(\log^{1+o(1)} n)/B$. To this end, let θ be a positive number such that $w \times w^\theta \times w$ matrix multiplication can be computed in $w^{1+o(1)}$ time. By Theorem 4, we can set $\theta > 0.156$, according to the current best rectangular matrix multiplication algorithm. We set $B = \log^\theta n$.

A similar argument to the proof of Theorem 1 shows that the problem reduces to evaluating $\log n$ c-wise hash functions on $\log^\theta n$ points, as well as computing a matrix-vector product, where the matrix is a 0-1 matrix of size $(k \log n) \times \log^\theta n$, and the vector has dimension $\log^\theta n$ and $(\log n)$-bit values. Finally, similar to the proofs of Lemma 1 and Lemma 2, both problems can be reduced to computing $\log n \times \log^\theta n \times \log n$ matrix multiplication, which takes $\log^{1+o(1)} n$ by the definition of θ. This gives us an algorithm with update time $O(\log^{0.844} n)$, and extra query time $\log^{1+o(1)} n$, proving Theorem 2. We omit the rest of the details.

Theorem 2 (restated). *For any problem that admits a lin-skt(k, c, T) linear sketch solution, where $k < \log^{o(1)} n$ is a power of two, and $T = O(\log n)$, there is an algorithm that*

- *uses $O(k \log n)$ words of space,*
- *has* worst-case *update time $O(\log^{0.844} n)$, and*
- *additive* extra *query time $O(\log^{1+o(1)} n)$*

on a standard word RAM with word-size $w = \Theta(\log n)$, where the exponent $0.844 \approx 1 - \alpha/2$, and α is the current dual matrix multiplication exponent.

5 Fast Matrix Multiplication

5.1 Tensor Rank and Matrix Multiplication

In this section, we show how to take advantage of the word RAM model to speed up matrix multiplication when the dimensions of the matrices are polynomials in the word size w. We begin by reviewing useful notation related to fast matrix multiplication algorithms.

Let $\mathbb{F}$ be any field, a, b, c be any nonnegative real numbers, n be any positive integer, and $X = \{x_{i,j}\}_{i\in[n^a],j\in[n^b]}$, $Y = \{y_{j,k}\}_{j\in[n^b],k\in[n^c]}$, and $Z = \{z_{i,k}\}_{i\in[n^a],k\in[n^c]}$ be three sets of formal variables. The *rank of $n^a \times n^b \times n^c$ matrix multiplication over $\mathbb{F}$*, denoted $R_{\mathbb{F}}(\langle n^a, n^b, n^c\rangle)$, is the smallest integer r such that there are values $\alpha_{i,j,\ell}, \beta_{j,k,\ell}, \gamma_{i,k,\ell} \in \mathbb{F}$ for all $i \in [n^a]$, $j \in [n^b]$, $k \in [n^c]$ and $\ell \in [r]$ such that

$$\sum_{\ell=1}^{r} \left(\sum_{i\in[n^a],j\in[n^b]} \alpha_{i,j,\ell} x_{i,j} \right) \left(\sum_{j\in[n^b],k\in[n^c]} \beta_{j,k,\ell} y_{j,k} \right) \left(\sum_{i\in[n^a],k\in[n^c]} \gamma_{i,k,\ell} z_{i,k} \right) = \sum_{i\in[n^a],j\in[n^b],k\in[n^c]} x_{i,j} y_{j,k} z_{i,k}. \tag{1}$$

Proposition 4. *[BCS13, Proposition 15.1] For any positive real a, b, c and field $\mathbb{F}$, suppose there is a $t > 0$ and an algorithm, in the arithmetic circuit model, which performs $n^a \times n^b \times n^c$ matrix multiplication over the field $\mathbb{F}$ using $n^{t+o(1)}$ field operations. Then, for every $\varepsilon > 0$, there is a positive integer q such that $R_{\mathbb{F}}(\langle q^a, q^b, q^c\rangle) \le q^{t+\varepsilon}$.*

We define $\omega_{\mathbb{F}}(a, b, c) := \liminf_{q\in\mathbb{N}} \log_q(R_{\mathbb{F}}(\langle q^a, q^b, q^c\rangle))$. It is known (and we will show below in Theorem 5) that $n^a \times n^b \times n^c$ matrix multiplication over $\mathbb{F}$ can be performed in $O(n^{\omega_{\mathbb{F}}(a,b,c)+\varepsilon})$ field operations. Although $\omega_{\mathbb{F}}(a, b, c)$ may differ depending on the field $\mathbb{F}$, all known constructions achieve the same value for all $\mathbb{F}$, so we will typically drop the $\mathbb{F}$ and simply write $\omega(a, b, c)$ as in past work. We also write $\omega = \omega(1, 1, 1)$. We note a couple of simple properties:

- for all $a, b, c, d \ge 0$ we have $\omega(d \cdot a, d \cdot b, d \cdot c) = d \cdot \omega(a, b, c)$.
- $\omega(a, b, c) = \omega(b, c, a)$ (or more generally any permutation of the three arguments) by the symmetry of the right-hand side of (1).

5.2 New algorithms for small matrices

We now show how to design faster algorithms for multiplying small matrices, whose dimensions are polynomials in the word size w of the word RAM model. Our algorithm only slightly modifies the usual recursive algorithm for fast matrix multiplication by making use of a more efficient base case.

We state our result over the field $\mathbb{F}_p$ for $p \le \mathrm{poly}(w)$, but it generalizes to any field where operations can be performed efficiently in the word RAM model.

Theorem 5. *Let $p \le \mathrm{poly}(w)$ be any prime number. Suppose, for some nonnegative real numbers a', b', c', that there is an algorithm which performs $w^{a'} \times w^{b'} \times w^{c'}$ matrix multiplication over the field $\mathbb{F}_p$ in time $M(w)$. Then, for any nonnegative real numbers a, b, c, and any $\varepsilon > 0$, there is an algorithm which performs $w^{a+a'} \times w^{b+b'} \times w^{c+c'}$ matrix multiplication over $\mathbb{F}_q$ in time $O(M(w) \cdot w^{\omega(a,b,c)+\varepsilon})$.*

Proof. We design a recursive algorithm which, for all positive integers n, performs $(n^a w^{a'}) \times (n^b w^{b'}) \times (n^c w^{c'})$ matrix multiplication over $\mathbb{F}_p$ in time $O(M(w) \cdot n^{\omega(a,b,c)+\varepsilon})$. As the base case, when $n = 1$, such an algorithm is assumed to exist.

For the recursive step, let q be the positive integer (constant) which is guaranteed to exist by Proposition 4 such that $R(\langle q^a, q^b, q^c\rangle) \le q^{\omega(a,b,c)+\varepsilon} =: r$, and using the notation of subsection 5.1, let $\alpha_{i,j,\ell}, \beta_{j,k,\ell}, \gamma_{i,k,\ell} \in \mathbb{F}$ for $i \in [q^a]$, $j \in [q^b]$, $k \in [q^c]$, $\ell \in [r]$ be the corresponding coefficients in the rank expression.

Let A be the input matrix of dimensions $w^{a'}n^a \times w^{b'}n^b$ over $\mathbb{F}$, and B be the input matrix of dimensions $w^{b'}n^b \times w^{c'}n^c$ over $\mathbb{F}$. First, we partition A into a $q^a \times q^b$ block matrix, where each block is a $w^{a'}(n/q)^a \times w^{b'}(n/q)^b$ matrix; call the blocks $A_{i,j}$ for $i \in [q^a]$, $j \in [q^b]$. Similarly we partition B into a $q^b \times q^c$ block matrix, where each block is a $w^{b'}(n/q)^b \times w^{c'}(n/q)^c$ matrix; call the blocks $B_{j,k}$ for $j \in [q^b]$, $k \in [q^c]$. The algorithm first computes, for each $\ell \in [r]$, the linear combination

$$A'_\ell = \sum_{i\in[q^a], j\in[q^b]} \alpha_{i,j,\ell} A_{i,j},$$

and the linear combination

$$B'_\ell = \sum_{j\in[q^b], k\in[q^c]} \beta_{j,k,\ell} B_{j,k}.$$

Since q is a constant, this takes $\tilde{O}(n^{a+b}/w^{1-a-b} + n^{b+c}/w^{1-b-c})$ field operations. (More details here??)

Next, for each $\ell \in [r]$, the algorithm computes the $w^{a'}(n/k)^a \times w^{c'}(n/k)^c$ matrix $C'_\ell := A'_\ell \times B'_\ell$, by *recursively* performing $w^{a'}(n/k)^a \times w^{b'}(n/k)^b \times w^{c'}(n/k)^c$ matrix multiplication. By the inductive hypothesis, this requires $O(r \cdot M(w) \cdot (n/q)^{\omega(a,b,c)+\varepsilon}) = O(M(w) \cdot n^{\omega(a,b,c)+\varepsilon})$ time.

Finally, for each $i \in [q^a]$ and $k \in [q^c]$, the algorithm computes the linear combination

$$C_{i,k} = \sum_{\ell=1}^{r} \gamma_{j,\ell} C'_j,$$

in total time $O(n^{a+c}/w^{1-a-c})$. These are the blocks of the $w^{a'}n^a \times w^{c'}n^c$ matrix C which we output. We can see these are correct from the definition of the rank expression (equation (1) in subsection 5.1): if we substitute in $A_{i,j}$ for $x_{i,j}$ and $B_{j,k}$ for $y_{j,k}$ in (1), then from the left hand side of (1) we see that $C_{i,k}$ is the resulting coefficient of $z_{i,k}$, and from the right hand size of (1) we see that that coefficient is indeed $\sum_j A_{i,j}B_{j,k}$, which is the correct i,k block of the output matrix C.

To see that the $O(M(w) \cdot n^{\omega(a,b,c)+\varepsilon})$ running time for the recursive step dominates the other terms $O(n^{a+b}/w^{1-a-b})$, $O(n^{b+c}/w^{1-b-c})$ and $O(n^{a+c}/w^{1-a-c})$, simply note that, because of the time to read the input, we have $n^{\omega(a,b,c)} \ge \Omega(n^{a+b} + n^{b+c} + n^{a+c})$, and $M(w) \ge \Omega(w^{a+b-1} + w^{b+c-1} + w^{a+c-1})$. □

5.2.1 Word RAM$^{\text{MM}}$ model

We begin with the model of computation where matrices which fit into words can be multiplied in constant time. In particular:

Proposition 5. *In the word RAM^{MM} model, one can perform $w^{1/2} \times w^{1/2} \times w^{1/2}$ matrix multiplication over $\mathbb{F}_q$ for $q \le \mathrm{poly}(w)$ in time $\tilde{O}(1)$.*

Proof. A $w^{1/2} \times w^{1/2}$ matrix fits into $\tilde{O}(1)$ words. □

Theorem 3 (restated). *In the word RAM^{MM} model, one can perform $w \times w \times w$ matrix multiplication over $\mathbb{F}_q$ for $q \le \mathrm{poly}(w)$ in time $O(w^{\omega/2+\epsilon})$.*

Proof. Set $a = b = c = a' = b' = c' = 1/2$ in Theorem 5, combined with the base case from Proposition 5. □

5.2.2 Word RAM model

Lemma 3. *In the word RAM model with word size w, for any positive integers d_a, d_b, d_c, q such that $d_a \cdot d_b \cdot d_c \cdot \log(q) \leq \tilde{O}(w)$, one can compute $d_a \times d_b \times d_c$ matrix multiplication over $\mathbb{F}_q$ in time $\tilde{O}(1)$.*

Proof. For a vector $v \in \mathbb{F}_q^{d_b}$, let $\ell = \lceil \log_2(q) \rceil$, and for $i \in [d_b]$, let $v_i \in \{0,1\}^\ell$ be the binary representation of $v[i]$ (the ith entry of v, with leading zeroes added as necessary). Then, letting $g = \lceil \log_2(2qd_b) \rceil$, define $s(v) \in \{0,1\}^{gd_b}$ to be the string given by $s(v) = 0^{g-\ell} v_{d_b} 0^{g-\ell} v_{d_b-1} \cdots 0^{g-\ell} v_1$. Hence, $s(v)$ is a space-separated concatenation of the entries of v, and moreover, as an integer it is equal to $\sum_{i=1}^{d_b} 2^{g(i-1)} \cdot v[i]$. Similarly define $s^r(v) = 0^{g-\ell} v_1 0^{g-\ell} v_2 \cdots 0^{g-\ell} v_{d_b}$.

Notice that for vectors $v, w \in \mathbb{F}_q^{d_b}$, if we compute $a_{v,w} := s(v) \cdot s^r(w)$ (as a product over the integers) then

$$a_{v,w} = \left(\sum_{i_0=1}^{d_b} 2^{g(i_0-1)} \cdot v[i_0] \right) \cdot \left(\sum_{i_1=1}^{d_b} 2^{g(d_b-i_1)} \cdot w[i_1] \right) = \sum_{j=0}^{2d_b-2} 2^{gj} \sum_{i_0-i_1=j+1-d_b} v[i_0] \cdot w[i_1] = \sum_{j=0}^{2d_b-2} 2^{gj} P_{v,w}(j),$$

where we define $P_{v,w}(j) := \sum_{i_0-i_1=j+1-d_b} v[i_0] \cdot w[i_1]$. In particular, we know that $P_{v,w}(j)$, which is a sum of at most d products of two integers between 0 and $q-1$, fits in g bits. Hence, $a_{v,w}$ is a space-separated list of the $P_{v,w}(j)$ for all j. Notice in particular that $P_{v,w}(d_b - 1)$, when taken mod q, is exactly the inner product $\langle v, w \rangle$. Since $s(v), s^r(w)$ fit in gd_b bits, it follows that $a_{v,w}$ fits in $2gd_b$ bits.

Next, for any vectors $v_1, \ldots, v_{d_a} \in \mathbb{F}_q^{d_b}$ and $w_1, \ldots, w_{d_c} \in \mathbb{F}_q^{d_b}$, consider the two strings

$$s(v_1, v_2, \ldots, v_{d_a}) := 0^{gd_b} s(v_1) 0^{gd_b} s(v_2) \cdots 0^{gd_b} s(v_{d_a}),$$

which has length $m_1 := 2gd_b d_a$, and

$$s^r(w_1, w_2, \ldots, w_{d_c}) := 0^{m_1} s^r(w_1) 0^{m_1} s^r(w_2) \cdots 0^{m_1} s_r(w_{d_c}),$$

which has length $m_2 := d_c(m_1 + gd_b) = O(gd_a d_b d_c)$. These can be computed using Proposition 2 in $\tilde{O}(1)$ time. Similar to before, the string

$$a_{(v_1,\ldots,v_{d_a}),(w_1,\ldots,w_{d_c})} := s(v_1, v_2, \ldots, v_{d_a}) \cdot s^r(w_1, w_2, \ldots, w_{d_c})$$

is a string of length $O(gd_a d_b d_c)$ which consists of a space-separated list of a_{v_i,w_j} for all $i \in [d_a]$ and $j \in [d_c]$. As above, from this we can extract the inner product $\langle v_i, w_j \rangle$ for all $i \in [d_a]$ and $j \in [d_c]$, which is exactly the desired matrix product. □

Theorem 4 (restated). *In the word RAM model, for any $p \in [0,1]$, and any $\varepsilon > 0$, one can perform $w \times w^p \times w$ matrix multiplication over $\mathbb{F}_q$ for $q \leq \mathrm{poly}(w)$ in time*

- *$O(w^{(1+p)\cdot\omega/3+\varepsilon}) \leq O(w^{0.791\cdot(1+p)})$ if $p \geq 1/2$,*
- *$O(w^{\omega(1,2p,1)/2+\varepsilon})$ if $p \leq 1/2$.*

Proof. When $p \geq 1/2$, then applying Lemma 3 with $d_a = d_c = w^{(2-p)/3}$ and $d_b = w^{(2p-1)/3}$, we know there is an algorithm for $w^{(2-p)/3} \times w^{(2p-1)/3} \times w^{(2-p)/3}$ matrix multiplication over $\mathbb{F}_q$ running in time $\tilde{O}(1)$. Combining this with Theorem 5 with $a = b = c = (1+p)/3$, $a' = c' = (2-p)/3$, and $b' = (2p-1)/3$ yields that $w \times w^p \times w$ matrix multiplication over $\mathbb{F}_q$ can be done in the desired running time.

When $p \leq 1/2$, we instead apply Lemma 3 with $d_a = d_c = w^{1/2}$ and $d_b = 1$, then Theorem 5 with $a = c = a' = c' = 1/2$, $b = p$, and $b' = 0$. □

Remark 1. *When applying Theorem 4 with the best known bounds on ω [Wil12, LG14, GU18], the '$+\varepsilon$' terms in the exponents may be replaced by '$+o(1)$'.*

Acknowledgments. The authors would like to thank Jelani Nelson for proposing the problem to us, and we would like to thank Jelani Nelson, Virginia Vassilevska Williams and Ryan Williams for helpful discussions.

References

[AH74] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[BCS13] Peter Bürgisser, Michael Clausen, and Mohammad A Shokrollahi. *Algebraic complexity theory*, volume 315. Springer Science & Business Media, 2013.

[BMM97] Andrej Brodnik, Peter Bro Miltersen, and J Ian Munro. Trans-dichotomous algorithms without multiplicationsome upper and lower bounds. In *Workshop on Algorithms and Data Structures*, pages 426–439. Springer, 1997.

[CCF04] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.

[CM05] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[DGW19] Zeev Dvir, Alexander Golovnev, and Omri Weinstein. Static data structure lower bounds imply rigidity. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019.*, pages 967–978, 2019.

[FSH04] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137. ACM, 2004.

[Fuj08] Noriyuki Fujimoto. Faster matrix-vector multiplication on geforce 8800gtx. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[FW90] Michael L Fredman and Dan E Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, pages 1–7. ACM, 1990.

[GU18] François Le Gall and Florent Urrutia. Improved rectangular matrix multiplication using powers of the coppersmith-winograd tensor. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1029–1046. SIAM, 2018.

[Ind06] Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM*, 53(3):307–323, 2006.

[JW13] T. S. Jayram and David P. Woodruff. Optimal bounds for johnson-lindenstrauss transforms and streaming problems with subconstant error. *ACM Trans. Algorithms*, 9(3):26:1–26:17, 2013.

[KNW10] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. On the exact space complexity of sketching and streaming small norms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 1161–1178, 2010.

[LG14] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation*, pages 296–303. ACM, 2014.

[Li08] Ping Li. Estimators and tail bounds for dimension reduction in l_α $(0 < \alpha \leq 2)$ using stable random projections. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 10–19, 2008.

[LKC+10] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH computer architecture news*, 38(3):451–460, 2010.

[LNN15] Kasper Green Larsen, Jelani Nelson, and Huy L. Nguyên. Time lower bounds for nonadaptive turnstile streaming algorithms. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 803–812, 2015.

[Mni09] Volodymyr Mnih. Cudamat: a cuda-based matrix class for python. *Department of Computer Science, University of Toronto, Tech. Rep. UTML TR*, 4, 2009.

[Mut05] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*. Lecture Notes in Economic and Mathematical Systems. Springer-Verlag, 1983.

[TZ12] Mikkel Thorup and Yin Zhang. Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. *SIAM J. Comput.*, 41(2):293–331, 2012.

[Wil12] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC*, volume 12, pages 887–898. Citeseer, 2012.