Efficient Interactive Proofs for Linear Algebra

Graham Cormode
University of Warwick, UK
g.cormode@warwick.ac.uk

Chris Hickey
University of Warwick, UK
c.hickey@warwick.ac.uk

Abstract

Motivated by the growth in outsourced data analysis, we describe methods for verifying basic linear algebra operations performed by a cloud service without having to recalculate the entire result. We provide novel protocols in the streaming setting for inner product, matrix multiplication and vector-matrix-vector multiplication where the number of rounds of interaction can be adjusted to tradeoff space, communication, and duration of the protocol. Previous work suggests that the costs of these interactive protocols are optimized by choosing $O(\log n)$ rounds. However, we argue that we can reduce the number of rounds without incurring a significant time penalty by considering the total end-to-end time, so fewer rounds and larger messages are preferable. We confirm this claim with an experimental study that shows that a constant number of rounds gives the fastest protocol.

2012 ACM Subject Classification Theory of Computation → Models of Computation

Keywords and phrases Streaming Interactive Proofs, Linear Algebra

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2019.51

Funding *Graham Cormode*: Supported by European Research Council grant ERC-2014-CoG 647557
Chris Hickey: Supported by European Research Council grant ERC-2014-CoG 647557

1 Introduction

The pitch for cloud computing services is that they allow us to outsource the effort to store and compute over our data. The ability to gain cheap access to both powerful computing and storage resources makes this a compelling offer. However, it brings increased emphasis on questions of trust and reliability: to what extent can we rely on the results of computations performed by the cloud? In particular, the cloud provider has an economic incentive to take shortcuts or allow buggy code to provide fast results, if they are hardly noticed by the client.

Prior work has developed the idea of using interactive proofs to independently verify outsourced computations without duplicating the effort. Originally invented as tools in the realm of computational complexity, recent work has sought to argue that interactive proofs can indeed be practically used for verification. Modern research takes two main approaches, from highly general methods with currently far-from-practical costs, to tackling specific fundamental problems where the overhead of verification is negligible.

In this work, we focus on the 'negligible' end of the spectrum and study primitive computations within linear algebra — a core set of tools with applications across engineering, data analysis and machine learning. We make four main contributions:

- We consider protocols for inner product and matrix multiplication and present lightweight tunable verification protocols for these problems. We also produce an entirely new protocol for vector-matrix-vector multiplication.
- Our protocols allow us to trade off computational effort and communication size against the number of rounds of interaction. We show it is often desirable to have fewer rounds of interaction.

- We optimize the costs for the cloud, and show that the protocols impose a computational overhead that is typically much smaller than the cost of the computation itself.
- Our experimental study confirms our analysis, and demonstrates that the absolute cost is minimal, with the client's cost significantly less than performing the computation independently.

1.1 Streaming Interactive Proofs

Our work adopts the model of *streaming interactive proofs* (SIPs), formalized in [7, 8].

▶ **Definition 1.** *We have two communicating computational entities, a helper, H, and a verifier, V, observing a stream $\mathcal{S}$. V wishes to know $f(\mathcal{S})$, for some function f. After viewing the stream, H and V have a conversation, culminating in V producing an output,* $\mathrm{Out}(V, \mathcal{S}, V_R, H)$*, where V_R represents a private random string belonging to V, so that*

$$\mathrm{Out}(V, \mathcal{S}, V_R, H) = \begin{cases} X & \text{if } V \text{ is convinced by } H \text{ that } f(\mathcal{S}) = X \\ \perp & \text{Otherwise} \end{cases}$$

We say the protocol used by the two parties is ***complete*** *for f if there exists an honest helper H such that*

$$\mathbb{P}[\mathrm{Out}(V, \mathcal{S}, V_R, H) = f(\mathcal{S})] = 1$$

and ***sound*** *if for any helper, H', and any input, $\mathcal{S}'$*

$$\mathbb{P}[\mathrm{Out}(V, \mathcal{S}', V_R, H') \notin \{f(\mathcal{S}'), \perp\}] \leq \frac{1}{3}$$

Informally, complete protocols always accept an honest answer, and sound protocols reject an incorrect answer most of the time (the constant probability $\frac{1}{3}$ is arbitrary and can be reduced to be vanishingly small via standard amplification techniques). If a protocol for V is both complete and sound, we call it a *valid protocol* for f. A valid protocol is characterized by costs in terms of required space and communication.

▶ **Definition 2.** *For a function f we say that there is a d-round $(h, v)-$protocol if there is a valid protocol for f with*

- ***Verifier Memory*** *v — Verifier uses $O(v)$ working memory.*
- ***Communication*** *h — The total communication between the two parties is $O(h)$. Note that we do not include the cost of sending the claimed solution in this cost.*
- ***Interactivity*** *d at most $2d$ messages sent from H to V or vice versa.*

Furthermore, we quantify the computational costs by

- ***Verifier Streaming Cost*** *— The work during the initial stream.*
- ***Verifier Checking Computation*** *— The work for the interactive stage.*
- ***Helper Overhead*** *— The additional work outside of solving the problem.*

Problem Statement.

We seek optimal or near optimal verification protocols for core linear algebra operations. The canonical (and previously studied) example is the multiplication of two matrices $A \in \mathbb{F}_q^{k\times n}, B \in \mathbb{F}_q^{n\times k'}$, where $\mathbb{F}_q$ is the finite field of integers modulo q, for some prime $q > M^2 n$, where $M = \max_{i,j}(A_{ij}, B_{ij})$ or chosen sufficiently large to not incur overflows. Our protocols work on any prime size finite field, consistent with prior work. This allows computation over fixed precision rational numbers, with appropriate scaling. For ease of exposition, we assume in this paper that $n = k = k'$, although all our algorithms work with rectangular matrices. The resulting matrix AB is assumed to be too large for the verifier to conveniently store, and so our aim is for the helper to allow the verifier to compute a *fingerprint* of AB [14], defined formally in Section 3.1, that can be used to check the helper's claimed answer.

1.2 Prior Work

Interactive proofs were introduced in the 1980s, primarily as a tool for reasoning about computational complexity [12]. A key result showed that the class of problems admitting interactive proofs is equivalent to the complexity class PSPACE [17]. Subsequent work in this direction led to the development of probabilistically checkable proofs (PCPs), where (in our terminology) the verifier only inspects a small fraction of the proof written by the helper. One distinction between this prior work and our setting is that PCPs consider a verifier who can devote polynomial time to inspecting the proof and has access to the full input; by contrast, we consider weaker verifiers, and try to more tightly bound their space and computational resources. The notion that interactive proofs could be a practical tool for verifying outsourced computation was advocated by Goldwasser, Kalai and Rothblum [11]. This paper introduced the powerful GKR (or 'muggles') protocol for verifying arbitrary computations specified as arithmetic circuits. Several papers have aimed to optimize the costs of the GKR protocol [7, 19, 18], or to provide systems for verifying general purpose computation under a variety of computational or cryptographic models [13, 16, 15]. The latter of which tackle large classes of problems using *arguments*, which consider a computationally bounded prover. We consider only proofs as we can achieve highly efficient protocols without requiring restriction on the prover, or use of cryptographic assumptions. Furthermore, some costs associated with such verification still remain high, such as requiring a large amount of pre-processing on the part of the helper, which can only be amortized over a large number of invocations. For the common and highly symmetric algebraic computations we work with in this paper, it is beneficial to build a specialised protocol.

Other work has considered engineering protocols for specific problems that are more lightweight, and so trade generality for greater practicality. The motivation is that some primitives are sufficiently ubiquitous that having special purpose protocols will outweigh the effort to design them. An early example of this is given by Frievalds' algorithm for verifying matrix multiplication [10]. This and similar algorithms unfortunately don't directly work for verifiers that can't store the entire input. This line of work was initiated for problems arising in the context of data stream processing, such as frequency analysis of vectors derived from streams [5]. Follow-up work addressed problems on graph data [8], data mining [9] and machine learning [6].

These papers tend to consider either the non-interactive case (minimizing the number of rounds), or have a poly-logarithmic number of rounds (minimizing the total communication). For example, [8] introduces an interactive inner product protocol which can accommodate a variable number of rounds. The development assumes that setting the number of rounds to be $\log(n)$ will be universally optimal, an assumption we reassess in this work. Similarly, in [18] the matrix multiplication protocol takes place over $O(\log(n))$ rounds. Our observation is that the pragmatic choice may fall between these extremes of non-interactive and highly interactive. Taking into account latency and round-trip time between participants, the preferred setting might be a constant number of rounds, which yields a communication cost which is a small polynomial in the input size, but which is not significantly higher in absolute terms from the minimal poly-logarithmic cost.

We summarize the current state of the art for the problems of computing inner product

Method	$O(h)$	$O(v)$	Rounds	H overhead	V overhead + checking
This Work	$O(ld)$	$O(l+d)$	$d-1$	$O(n\log(l))$	$O(nld)+O(ld)$
Binary SC [8]	$O(\log(n))$	$O(\log(n))$	$\log(n)$	$O(n)$	$O(n\log(n))+O(\log(n))$
FFT LDEs [7]	$O(n^{1-a})$	$O(n^a)$	1	$O(n\log(n))$	$O(n)+O(\log(n))$

Table 1 Different SIPs for Inner Product with $u, v \in \mathbb{F}_q^n$, with $n = l^d$ and $a \in [0, 1]$.

Method	$O(h)$	$O(v)$	Rounds	H overhead	V overhead + checking
This Work	$O(ld)$	$O(l+d)$	d	$O(n^2)$	$O(n^2ld)+O(ld)$
Binary SC [18]	$O(\log(n))$	$O(\log(n))$	$\log(n)+1$	$O(n^2)$	$O(n^2\log(n))+O(\log(n))$
Fingerprints [5]	$O(n^2)$	$O(1)$	1	$O(1)$	$O(n^2)+O(n^2)$

Table 2 Different SIPs for Matrix Multiplication with $A, B \in \mathbb{F}_q^{n\times n}$ and $n = l^d$.

(Table 1) and matrix multiplication (Table 2), and show the results we obtain here for comparison.

Lastly, we comment that our results are restricted to the information-theoretically secure model of Interactive Proofs, and are separate from recent results in the computational (cryptographic) security model [3, 4].

1.3 Contributions and outline

Our main contribution is an investigation into the time-optimal number of rounds for a variety of protocols. We adapt and improve protocols for inner product and matrix multiplication, as well as introducing an entirely new protocol for vector-matrix-vector multiplication. We then perform experiments in order to evaluate the time component of each stage of interaction.

We begin in Section 2 by re-evaluating how to measure the communication cost of a protocol, and propose to combine the competing factors of latency and bandwidth into a total time cost. This motivates generalized protocols that take a variable number of rounds, where we can pick a parameter setting to minimizes the total completion time.

In Section 3 and 4 we build on previous protocols [8, 7] to construct novel efficient *variable round* protocols for core linear algebra operations. We begin by revisiting variable round protocols for inner product. We leverage these to obtain new protocols for matrix multiplication and vector-matrix-vector multiplication (which does not appear to have been studied previously) with similar asymptotic costs.

In Section 5, we thoroughly analyse the practical computation costs of the resulting protocols, and compare to existing verification methods. We perform a series of experiments to back up our claims, and draw conclusions on what we should want from interactive proofs. We show that it can be preferable to use fewer rounds, despite some apparently higher costs.

2 How Much Interaction Do We Want?

Prior work has sought to find ‘optimal’ protocols which minimize the total communication cost. This is achieved by *increasing* the number of rounds of interaction, with the effect of driving down the amount of communication in each round. The minimum communication is typically attained when the number of rounds is polylogarithmic [7]. The non-interactive case represents another extreme in this regard, requiring a single message from the helper to verifier. This allows the parties to work asynchronously at the cost of larger total communication.

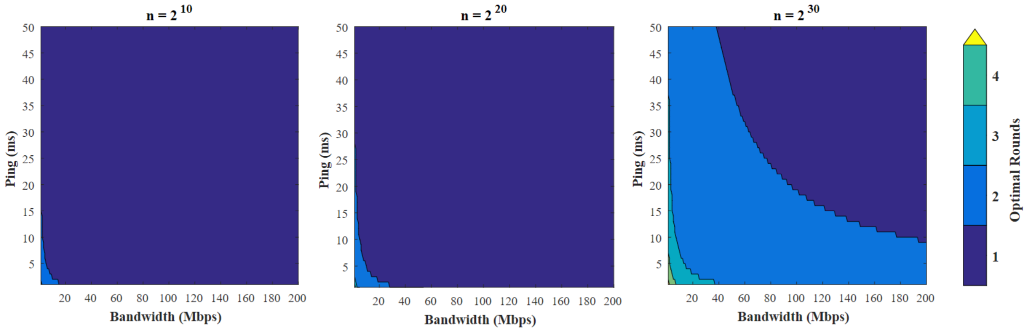

Figure 1 Optimal number of rounds for matrix multiplication of various sizes when considering only communication, with a field size $q = O(n^3)$.

In this section we argue that the right approach is neither the non-interactive case *nor* the highly-interactive case. Rather, we argue that a compromise of 'moderately interactive proofs' can yield better results. To do so we consider the overall time required to process the proof.

The key observation is that the time to process a proof depends not just on the amount of communication, but also the number of rounds. In the protocols from Table 1 and 2, each round cannot commence until the previous round completes, hence we incur a time penalty as a function of the *latency* between the two communicating parties. The duration of a round depends on the bandwidth between them. Thus, we aim to combine number of rounds and message size into a single intuitive quantity based on bandwidth and latency that captures the total wall-clock time cost of the protocol.

For matrix multiplication, the variable round protocols summarized in Table 2 spread the verification over d rounds, and have a total communication cost proportional to $dn^{1/d}$. Hence, we write the time to perform the communication of the protocol as $T = 2d\mathcal{L} + \frac{2dn^{1/d}\log(|\mathbb{F}|)}{\mathcal{B}}$, where latency ($\mathcal{L}$) is measured in seconds, and bandwidth ($\mathcal{B}$) in bits per second. This expression emerges due to the $2d$ changes in direction over the protocol, and considering a protocol that sends a total of $2dn^{1/d}$ field elements (from the analysis in Section 4.2).

We measured the cost using typical values of $\mathcal{L}$ and $\mathcal{B}$ observed on a university campus network, where the 'ping' time to common cloud service providers (Google, Amazon, Microsoft) is of the order of 20ms, and the bandwidth is around 100Mbps. From the above equation for T we see that, for a constant field size $|\mathbb{F}|$, the value of $2n^{1/d}d\log(|\mathbb{F}|)/\mathcal{B}$ is dominated by $2d\mathcal{L}$ for even small d under such parameter settings. Hence, we should prefer fewer rounds as latency increases. Figure 1 shows the number of rounds which minimizes the communication time as a function of the size of the input. We observe that the answer is a small constant, at most just two or three rounds, even for the largest input sizes, corresponding to exabytes of data.

3 Primitives

Before we introduce our protocols, we first describe the building blocks they rely on.

3.1 Fingerprints

Fingerprints can be thought of as hash functions for large vectors and matrices with additional useful algebraic properties. For $A \in \mathbb{F}_q^{n\times n}$ and $x \in \mathbb{F}_q$, define the matrix fingerprint as $F_x(A) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} A_{ij}x^{in+j}$. Similarly, for $u \in \mathbb{F}_q^n$ we have the vector fingerprint $F_x^{\text{vec}}(u) = \sum_{i=0}^{n-1} u_i x^i$. The probability of two different matrices having the same fingerprint (over the random choice of x) can be made arbitrarily small by increasing the field size.

▶ **Lemma 3** ([14]). *Given $A, B \in \mathbb{F}_q^{n\times n}$ and $x \in_R \mathbb{F}_q$, we have $\mathbb{P}[F_x(A) = F_x(B)|A \neq B] \leq \frac{n^2}{q}$.*

A similar result holds for F_x^{vec}. In our model, fingerprints can be constructed in constant space, and with computation linear to the input size.

3.2 Low Degree Extensions

Low degree extensions (LDEs) have been used extensively in interactive proofs. LDEs have been used in conjunction with sum-check (Section 3.3) in a variety of contexts [11, 7, 8]. Formally, for a set of data S an LDE is a low degree polynomial that goes through each data point. Typically, we think of S as being laid out as a vector or d-dimensional tensor indexed over integer coordinates. This polynomial can then be evaluated at a random point r with the property that, like fingerprinting, two different data sets are unlikely to evaluate to the same value at r (inversely proportional to the field size).

Given input as a vector $u \in \mathbb{F}_q^n$, we consider two new parameters, l and d with $n \leq l^d$, and re-index u over $[\ell]^d$. The d-dimensional LDE of u satisfies $\tilde{f}_u(k_0, ..., k_{d-1}) = u_k$ for $k \in [n]$ where $k_0...k_{d-1}$ is the base l representation of k. For a random point $r = (r_0, ..., r_{d-1}) \in \mathbb{F}_q^d$, we have

$$\tilde{f}_u(r_0, ..., r_{d-1}) = \sum_{k_0}^{l-1} \cdots \sum_{k_{d-1}}^{l-1} u_k \chi_k(r) \tag{1}$$

$$\chi_k(r) = \prod_{j=0}^{d-1} \prod_{\substack{i=0 \\ i\neq k_j}}^{l-1} \frac{r_j - i}{k_j - i}, \tag{2}$$

where χ is the Lagrange basis polynomial. Note that $\tilde{f}_u : \mathbb{F}_q^d \to \mathbb{F}_q$ and $q \geq l$. A similar definition can be used for a matrix $A \in \mathbb{F}_q^{n\times n}$, by reshaping into a vector in $\mathbb{F}_q^{n^2}$.

The polynomials can be evaluated over a stream of updates in space $O(d)$ and time per update $O(ld)$ [8]. The time cost of our verifier to evaluate an LDE at one location, r, is $O(nld)$ (for sparse data, n can be replaced with the number of non-zeros in the input).

3.3 Sum-Check Protocol

Our final primitive is the sum-check protocol [12]. Sum-check is a multi-round protocol for verifying the sum

$$G = \sum_{k_0=0}^{l-1} \sum_{k_1=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} g(k_0, k_1, ..., k_{d-1}) \text{ for } g : \mathbb{F}_q^d \to \mathbb{F}_q. \tag{3}$$

For our purposes, g will be a polynomial derived from the LDE of a dataset of size $n = l^d$ (i.e. the d-dimensional tensor representation of the data), and each polynomial used in the protocol will have degree λ, with $\lambda = O(l)$; however, we keep the parameter λ for completeness. Provided that all the checks are passed then the verifier is convinced that (except with small probability) the value G was as claimed in (3). The original descriptions of the sum-check protocol [12, 2] use $l = 2$, however we shift to using arbitrary l, similar to [1, 7, 8]. The protocol goes as follows:

Stream Processing: V randomly picks $r \in \mathbb{F}_q^d$ and computes $g(r_0, ..., r_{d-1})$.

Round 1: H computes and sends G and $g_0 : \mathbb{F}_q \to \mathbb{F}_q$, where

$$g_0(k_0) = \sum_{k_1=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} g(k_0, k_1, ..., k_{d-1}).$$

V checks that $G = \sum_{k_0=0}^{l-1} g_0(k_0)$, computes $g_0(r_0)$ and sends r_0 to H.

$\vdots$

Round $j+1$: H has received $r_0, \ldots, r_{j-1}$ from V, and sends $g_j : \mathbb{F}_q \to \mathbb{F}_q$, where

$$g_j(k_j) = \sum_{k_{j+1}=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} g(r_0, ..., r_{j-1}, k_j, ..., k_{d-1}).$$

V checks if $g_{j-1}(r_{j-1}) = \sum_{k_j=0}^{l-1} g_j(k_j)$, computes $g_j(r_j)$ and sends r_j to H.

$\vdots$

Round d: H sends $g_{d-1} : \mathbb{F}_q \to \mathbb{F}_q$, where $g_{d-1}(k_{d-1}) = g(r_0, ..., r_{d-3}, r_{d-2}, k_{d-1})$.
V checks that $g_{d-2}(r_{d-2}) = \sum_{k_{d-1}=0}^{l-1} g_{d-1}(k_{d-1})$, computes $g_{d-1}(r_{d-1})$, and finally checks this is $g(r_0, ..., r_{d-2}, r_{d-1})$.

H can express the polynomial g_j as a set $G_j = \{(g_j(x), x) : x \in [\lambda]\}$. In each round V sums the first l elements of this set, and checks it is $g_{j-1}(r_{j-1})$ for $j > 0$, then evaluates the LDE of G_j at r_j, giving a computation cost per round of $O(l + \lambda)$. The verifier also has to do some work in the streaming phase, evaluating the function g at r, with cost $O(n\lambda d)$. The helper's computation time comes from having to evaluate g at l^{d-j} points in the jth round, and so ultimately evaluating g at $\sum_{j=1}^{d-1} l^{d-j} = O(n)$ points, with a cost per point of $O(\lambda d)$ (we subsequently show how this can be reduced in our protocols for linear algebra). The costs of performing sum-check are summarized as follows:

Communication $O(\lambda d)$ words, spread over d rounds.

Helper costs $O(n\lambda d)$ time for computation.

Verifier costs $O(\lambda + d)$ memory cost, $O(n\lambda d)$ overhead to compute LDE and checking cost $O(d(l + \lambda))$.

In our implementations, we will optimize our methods to 'stop short' the sum-check protocol and terminate at round $d - 1$ (this idea is implicit in the work of Aaronson and Wigderson [1, Section 7.2]). In this setting, the verifier finds the set

$$\{g(r_0, ..., r_{d-3}, r_{d-2}, k_{d-1}) : k_{d-1} \in [l]\}.$$

in the stream processing stage, and then checks this against the claimed set of values provided by the helper in round $d - 1$. This increases the space used by the verifier to maintain these l LDE evaluations. However, this does not affect the asymptotic space usage of the verifier, since we assume that V already keeps space proportional to l to handle H's messages. It does not affect the streaming overhead time, since each update affects only the LDE point with which it shares the final coordinate. Equivalently, this can be viewed as running l instances of sum-check in parallel on the data divided into l partitions. Hence, this appears as an all-round improvement, at least in theory.

4 Protocols for Linear Algebra Primitives

Using the previously discussed primitives for SIPs, we show how they have been used in inner product [7]. We then use this to construct a new variable round method for matrix multiplication, and extend it to achieve a novel vector-matrix-vector multiplication protocol.

4.1 Inner Product

Given two vectors $a, b \in \mathbb{F}_q^n$, the verifier wishes to receive $a^T b \in \mathbb{F}_q$ from the helper. We give a straightforward generalization of the analysis of a protocol in [8], as an application of sum-check on the LDEs of a and b. This variable round protocol has costs detailed below.

▶ **Theorem 4.** *Given $a, b \in \mathbb{F}_q^n$, there is a $(d-1)$-round $(ld, l+d)$-protocol with $n = l^d$ for verifying $a^T b$ with helper computation time $O\left(\frac{n\log(n)}{d}\right)$, verifier overhead $O(nld)$, and checking cost $O(ld)$.*

The analysis from [8] sets $l = 2$ and $d = \log(n)$, and the computational cost for the verifier is $O(\log(n))$ while the cost for the helper is $O(n\log(n))$. For general l and d these costs become $O(ld)$ and $O(nld)$ for the verifer and helper respectively.

In [7] it is shown how the helper's cost can be reduced to $O(n\log(n))$ for $d = 2$ and $l = \sqrt{n}$ using the Discrete Fast Fourier Transform to make a fast non-interactive protocol. We extend this for arbitrary d and l, and show how by combining with sum-check we can keep the helper's computation low, proving Theorem 1.

▶ **Lemma 5.** *Given $a, b \in \mathbb{F}_q^n$ the sum*

$$a^T b = \sum_{k_0=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \tilde{f}_a(k_0, ..., k_{d-1})\tilde{f}_b(k_0, ..., k_{d-1}) \tag{4}$$

can be verified using a $(d-1)$-round $(ld, l+d)$-protocol with helper computation time $O(\frac{n\log(n)}{d})$, and verifier computation time $O(ld)$, overhead time $O(nld)$.

Proof. First, set

$$g(k_0, ..., k_{d-1}) = \tilde{f}_a(k_0, ..., k_{d-1})\tilde{f}_b(k_0, ..., k_{d-1}).$$

$g : \mathbb{F}_q^d \to \mathbb{F}_q$ is a degree $2l$ polynomial in each variable. Now, consider round $j+1$ of the sum-check protocol, where the helper is required to send

$$g_j(x) = \sum_{k_{j+1}=0}^{l-1} \cdots \sum_{k_d=0}^{l-1} g(r_1, ..., r_{j-1}, x, k_{j+1}, ..., k_d).$$

Here, g is degree $2l$ polynomial, sent to V as a set $G_j^{\Sigma} = \left\{(g_j(x), x) : x \in [2l]\right\}$. To compute this set we have H find the individual summands as

$$G_j = \left\{\left(g(r_1, ..., r_{j-1}, x, k_{j+1}, ..., k_{d-1}), x\right) : x \in [2l], k_{j+1}, ..., k_{d-1} \in [l]\right\}.$$

Naive computation of all the values in G_j takes time $O(nd)$ each, for a total cost of $O(nl^{d-j}d)$. However, instead of computing the LDE at l^{d-j} points with cost $O(ld)$ we can sum l^{d-j} convolutions of length $2l$ vectors to obtain the same result. We present the full proof of this claim in the Appendix. The total cost of each convolution is $O(l\log(l))$. Summing these l^{d-j} convolutions gives the cost of the jth round for the helper as $O\left(\frac{l^{d-j}\log(n)}{d}\right)$. Summing $\sum_{j=0}^{d-1} l^{d-j}$ over the d rounds gives us our cost of $O\left(\frac{n\log(n)}{d}\right)$. The remaining costs are as in our version of the sum-check protocol (Section 3.3). ◀

4.2 Matrix Multiplication

By combining the power of LDEs with the matrix multiplication methods from [6], we can create a protocol with only marginally larger costs than inner product.

▶ **Theorem 6.** *Given two matrices $A, B \in \mathbb{F}_q^{n\times n}$, we can verify the product $AB \in \mathbb{F}_q^{n\times n}$ using a d-round $(ld, l+d)$-protocol with verifier overhead time $O(n^2 ld)$, checking time $O(ld)$ and helper computation time $O(n^2)$.*

Proof. We make use of the matrix fingerprints from [6], and generate the fingerprint of AB for some $x \in \mathbb{F}_q$ by expressing matrix multiplication as a sum of outer products.

$$F_x(AB) = \sum_{i=0}^{n-1} F_{x^n}^{\text{vec}}(A_i^{\downarrow}) F_x^{\text{vec}}(B_i^{\rightarrow}) \tag{5}$$

where $A_i^{\downarrow}$ denotes the ith column of A and $B_j^{\rightarrow}$ is the jth row of B. We also define:

$$A_{\text{col}} = (F_{x^n}^{\text{vec}}(A_1^{\downarrow}), ..., F_{x^n}^{\text{vec}}(A_n^{\downarrow})) \text{ and } B_{\text{row}} = (F_x^{\text{vec}}(B_1^{\rightarrow}), ..., F_x^{\text{vec}}(B_n^{\rightarrow})).$$

Our fingerprint $F_x(AB)$ is then given by the inner product of A_{col} and B_{row}. We apply the inner product protocol of Theorem 4, hence we need to show the verifier can evaluate the LDE of the product of these two vectors at a random point,

$$\sum_{k_{d-1}=0}^{l-1} \tilde{f}_{A_{\text{col}}}(r_0, ..., r_{d-2}, k_{d-1}) \tilde{f}_{B_{\text{row}}}(r_0, ..., r_{d-2}, k_{d-1}),$$

which we denote as $\Sigma \tilde{f}_{A_{\text{col}}}(r)\tilde{f}_{B_{\text{row}}}(r)$. We can construct this value in the initial stream by storing, for each value of k_{d-1}, $\tilde{f}_{A_{\text{col}}}(r_0, ..., r_{d-1}, k_{d-1})$ and $\tilde{f}_{B_{\text{row}}}(r_0, ..., r_{d-1}, k_{d-1})$, which is done in space $O(ld)$ for the verifier. Each of these requires an initial verifier overhead of $O(ld)$ for each of the n^2 elements, then checking requires $O(ld)$ as in Theorem 4. The helper has to fingerprint the matrices to form A_{col} and B_{row}, at a cost of $O(n^2)$. The result follows by using the generated fingerprint to compare to the fingerprint of the claimed result AB (which is provided by the helper in some suitable form, and excluded from the calculation of the protocol costs). ◀

Note that the helper is not required to follow any particular algorithm to compute the matrix product AB. Rather, the purpose of the protocol is for the helper to assist the verifier in computing a fingerprint of AB from its component matrices. The time cost of this is much faster: linear in the size of the input.

Fingerprinting versus LDEs. Our protocol in Theorem 6 is stated in terms of fingerprints. In [18], a d-round protocol is presented which uses

$$\tilde{f}_{AB}(R_1, R_2) = \sum_{k_0=0}^{1} \cdots \sum_{k_{\log(n)-1}=0}^{1} \tilde{f}_A(R_1, k)\tilde{f}_B(k, R_2).$$

This uses the inner product definition of matrix multiplication, whilst we use the outer product property of fingerprints. Finding $\tilde{f}_{AB}(R_1, R_2)$ during the initial streaming has cost per update $O(\log(n))$. For our method, we find $\Sigma \tilde{f}_{A_{\text{col}}}(r)\tilde{f}_{B_{\text{row}}}(r)$, which has cost $O(ld)$. In the case $l = 2, d = \log(n)$, we see these two methods are very similar. The methods differ in how we respond to receiving the result, AB. In [18], the verifier computes the LDE of

AB at a cost of $O(n^2ld)$, while our method takes time $\tilde{O}(n^2)$ to process the claimed AB, as we simply fingerprint the result. Thaler's method posesses some other advantages, for example it can chain matrix powers (finding A^m) without the Helper having to materialize the intermediate matrices. Nevertheless, in data analysis applications, it is often the case that only a single multiplication is required.

4.3 Vector-Matrix-Vector Multiplication

Vector-matrix-vector multiplication appears in a number of scenarios. A simple example arises in the context of graph algorithms: suppose that helper wishes to demonstrate that a graph, specified by an adjacency matrix A, is bipartite. Let v be an indicator vector for one part of the graph, then $v^T Av = (\mathbf{1} - v)^T A(\mathbf{1} - v) = 0$ iff v is as claimed. More generally, the helper can show a k colouring of a graph using k vector-matrix-vector multiplications between the adjacency matrix and the k disjoint indicator vectors for the claimed colour classes.

We reduce the problem of vector-matrix-vector multiplication (which yields a single scalar) to inner product computation, after reshaping the data as vectors. Formally, given $u, v \in \mathbb{F}^n$ and $A \in \mathbb{F}^{n\times n}$, we can compute $u^T Av$ as

$$u^T Av = \sum_{i=1}^{n}\sum_{j=1}^{n} u_i A_{ij} v_j = (uv^T)_{\text{vec}} \cdot A_{\text{vec}}$$

$u^T Av$ is equal to computing the inner product of A and uv^T written as length n^2 vectors. Protocols using this form will need to make use of an LDE evaluation of uv^T. We show that this can be built from independent LDE evaluations of each vector.

► **Lemma 7.** *Given $u, v \in \mathbb{F}^n$ and $r \in_R \mathbb{F}^d$, with $n = l^d$*

$$f_{uv^T}(r_0, \dots r_{2d-1}) = f_u(r_0, \dots, r_{d-1}) f_v(r_d, \dots, r_{2d-1})$$

Proof. We abuse notation a little to treat uv^T as a vector of length n^2, and we assume that $n = l^d$ (if not, we can pad the vectors with zeros without affecting the asymptotic behaviour). We write $R_1 = (r_0, \dots, r_{d-1})$ and $R_2 = (r_d, \dots, r_{2d-1})$. The proof follows by expanding out expression (2) to observe that $\chi_k(r_0 \dots r_{2d-1}) = \chi_{k_0,\dots k_{d-1}}(R_1)\chi_{k_d,\dots,k_{2d-1}}(R_2)$ and so

$$\begin{aligned}
f_{uv^T}(r_0, \dots r_{2d-1}) &= \sum_{k_0=0}^{l-1} \dots \sum_{k_{2d-1}=0}^{l-1} \left[\left(uv^T\right)_k \chi_k(r)\right] \\
&= \sum_{i_0=0}^{l-1} \dots \sum_{i_{d-1}=0}^{l-1} \sum_{j_0=0}^{l-1} \dots \sum_{j_{d-1}=0}^{l-1} (u_i v_j)\chi_i(R_1)\chi_j(R_2) \\
&= f_u(R_1) f_v(R_2)
\end{aligned}$$

◄

The essence of the proof is that we can obtain all the needed cross-terms corresponding to entries of uv^T from the product involving all terms in f_u and all terms in f_v.

We can employ the protocol for inner product using f_A and f_{uv^T}, which we can compute in the streaming phase, as $f_{uv^T} = f_u f_v$ to give us Theorem 3.

► **Theorem 8.** *Given $u, v \in \mathbb{F}^n$ and $A \in \mathbb{F}^{n\times n}$, we can verify $u^T Av$ using a $(d-1)$ round $(ld, l+d)$-protocol for $n^2 = l^d$, with helper computation $O\left(\frac{n^2 \log(n)}{d}\right)$, verifier overhead $O(nld)$ and checking cost $O(l)$.*

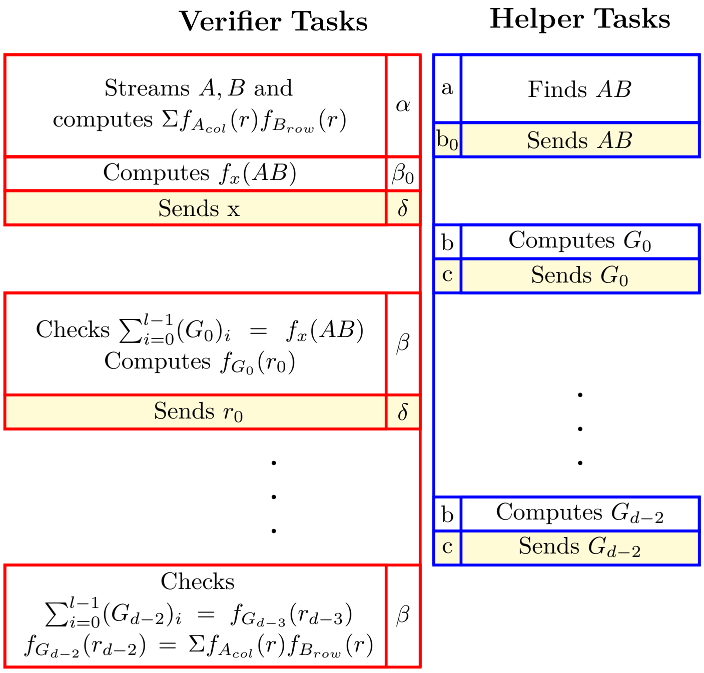

Figure 2 Detailed Matrix Multiplication Protocol

5 Practical Analysis

To evaluate these protocols in practice, we focus on the core task of matrix multiplication. In order to discuss the time costs associated with execution of our protocols in more detail, we break down the various steps into components as illustrated in Figure 2. Here, we use Greek characters to describe the costs for the verifier: the initial streaming overhead (t[α]), the checks performed in total in each round (t[β]), as well as the time to send responses (t[δ]). For the helper, we identify four groups of tasks, denoted by Latin characters: the computation of the matrix product itself (t[a]), the communication of this result to the verifier (t[b_0]), and the time per round to compute and send the required message (t[b] and t[c] respectively).

Recall our discussion in Section 2 on the effects of communication bandwidth and latency on the optimal number of rounds. In our simple model we focused on the tasks most directly involved with communication (the verifier round cost t[δ] and helper round cost t[c]). We implicitly treated the corresponding round computation costs (t[β] and t[b]) as nil. As the construction and sending of the solution (t[a] and t[b_0]) will dominate the first stage of the protocol, we focus our experimental study on measuring values of t[b], t[β_0] and t[β] to quantify a reasonable estimate for the length of time the interactive phase of the protocol takes with bandwidth $\mathcal{B}$ and latency $\mathcal{L}$.

We account for the cost required for computation and communication separately to find the total time, T, as follows:

$$T = \mathrm{t}[\mathrm{work}] + \mathrm{t}[\mathrm{comm}] = (\mathrm{t}[\beta_0] + \mathrm{t}[\beta] + \mathrm{t}[b]) + \left(2d\mathcal{L} + \frac{2dl \log(|\mathbb{F}|)}{\mathcal{B}}\right).$$

T is the total time for the protocol from receiving the answer to producing a conclusion of the veracity of the result. We can omit the verifier's streaming computation time t[α] from

l	d	t[b] (ms)	t[β] (μs)
2	12	0.230±0.02	9±2
4	6	0.120±0.01	14±1
8	4	0.099±0.01	35±7
16	3	0.097±0.01	35±7
64	2	0.110±0.01	43±5

(a) $n = 2^{12}$, $\mathrm{t}\,[\beta_0] = 149 \pm 15\mathrm{ms}$

l	d	t[b] (ms)	t[β] (μs)
2	16	3.5 ± 0.2	6 ± 1
4	8	2.0 ± 0.1	9 ± 1
16	4	1.6 ± 0.1	46 ± 3
256	2	1.8 ± 0.1	1700 ± 200

(b) $n = 2^{16}$, $\mathrm{t}\,[\beta_0] = 38.0 \pm 6.5\mathrm{s}$

l	d	t[b] (ms)	t[β] (μs)
2	18	14.1 ± 0.9	6 ± 1
4	9	8.0 ± 0.5	11 ± 3
8	6	6.3 ± 0.5	30 ± 3
64	3	7.1 ± 0.6	270 ± 30
512	2	7.8 ± 0.7	6400 ± 650

(c) $n = 2^{18}$, $\mathrm{t}\,[\beta_0] = 603 \pm 63\mathrm{s}$

Table 3 Interaction phase costs

n	t[a] (s)
2^{10}	0.61 ± 0.06
2^{11}	5.61 ± 0.7
2^{12}	47.9 ± 4.3
2^{13}	403 ± 34

Table 4 Matrix Multiplication Timings

the total protocol run time, as this can be overlapped with the helper's computation of the true answer, which should always dominate.

In what follows, we instantiate this framework and determine the costs of implementing protocols. These demonstrate that while computation cost for matrix multiplication ($t[a]$) grows superquadratically, the streaming cost ($t[\alpha]$) is linear in the input size n. The dominant cost during the protocol is $t[\beta_0]$, to fingerprint the claimed answer; other computational costs in the protocol are minimal. Factoring in the communication based on real-world latency and bandwith costs, we conclude that latency dominates, and indeed we prefer to have fewer rounds. In all our experiments, the optimal number of rounds is just 2. Extrapolating to truly enormous values of n suggest that still three rounds would suffice.

5.1 Setup

The experiments were performed on a workstation with an Intel Core i7-6700 CPU @ 3.40GHz processor, and 16GB RAM. Our implementations were written in single-threaded C using the GNU Scientific Library with BLAS for the linear algebra, and FFTW3 library for the Fourier Transform. The programs were compiled with GCC 5.4.0 using the -O3 optimization flag, under Linux (64-bit Ubuntu 16.04), with kernel 4.15.0. Timing was done using the `clock()` function for all readings except t[β], which used `getrusage()` as the timings were so small.

For the various tests performed, the matrices and vectors were generated using the C `rand()` function. Note that the work of the protocols is not affected by the data values, so we are not much concerned with how the inputs are chosen. The arithmetic field used was $\mathbb{F}_q$ with $q = 2^{31} - 1$ (larger fields, such as $q = 2^{61} - 1$ or $q = 2^{127} - 1$ could easily be substituted to obtain much lower probability of error, at a small increase in time cost). The work of the verifier and work of the helper were both simulated on the same machine.

Table 5 Time taken for interactions (ping 20ms, bandwidth 100Mbps, $|\mathbb{F}| = 2^{31} - 1$)

n	l	d	Latency cost (ms)	Bandwidth cost (ms)
2^{12}	2	12	440	0.014
	4	6	200	0.012
	8	4	120	0.015
	16	3	80	0.020
	64	2	40	0.041
2^{16}	2	16	600	0.019
	4	8	280	0.018
	16	4	120	0.031
	256	2	40	0.163
2^{18}	2	18	680	0.022
	4	9	320	0.020
	8	6	200	0.026
	64	3	80	0.082
	512	2	40	0.328

Table 6 Verifier matrix multiplication time (ping 20ms, bandwidth 100Mbps, $|\mathbb{F}| = 2^{31} - 1$).

n	l	d	t[comm] (s)	t[work] (s)	T (s)
2^{12}	2	12	0.44	0.149	0.589
	4	6	0.20		0.349
	8	4	0.12		0.269
	16	3	0.08		0.229
	64	2	0.04		0.189
2^{16}	2	16	0.60	38	38.6
	4	8	0.28		38.3
	16	4	0.12		38.1
	256	2	0.04		38.0
2^{18}	2	18	0.68	603	604
	4	9	0.32		603
	8	6	0.20		603
	64	3	0.08		603
	512	2	0.04		603

5.2 Matrix Multiplication Results

Table 3 shows the experimental results for the matrix multiplication protocol for matrix sizes ranging from $n = 2^{12}$ to 2^{18}. Note, this means we are tackling matrices with tens of billions of entries. For completeness, we timed BLAS matrix multiplication on our machine for $n = 2^{10}$ to 2^{13} to give an idea of the comparative magnitude of a (Table 4), although further results were restricted by machine memory. Due to memory limitations, we tested our algorithms using freshly drawn random values in place of stored values of the required vectors or matrices. This does not affect our ability to compare the data, and allows us to increase the data size beyond that of the machine memory.

The computation cost t$[a]$ grows with the cost of matrix multiplication, which is super-quadratic in n, while t$[\alpha]$ grows linearly with the size of the input, which is strictly quadratic in n. Further, the verifier does not need to retain whole matrices in memory, and can compute the needed quantities with a single linear pass over the input.

We next study the helper's cost across all d rounds to compute the responses in each step of the protocol. Our analysis bounds this total cost as $O(\frac{n\log(n)}{d})$. However, we observe that in our experiments, this quantity tends to decrease as d decreases. We conjecture that while the cost does decrease each round, the amount of data needed to be handled quickly decreases to a point where it is cache resident, and the computation takes a negligible amount of time compared to the data access. Thus, this component of the helper's time cost is driven by the number of rounds during which the relevant data is still 'large', which is greater for larger d.

When we look at the contributory factors to t[work], we observe that the dominant term is by far t$[\beta_0]$, where the verifier reads through the claimed answer and computes the fingerprint. Thus, arguably, the *computational* cost of any such protocol once the prover finds the answer is dominated by the time the verifier takes to actually inspect the answer: all subsequent checks are minimal in comparison. This justifies our earlier modelling assumption to omit computational costs in our balancing of latency and bandwidth factors.

We now turn to the time due to communication, summarized in Table 5. Here, we can clearly see the huge difference of several orders of magnitude between the latency cost, $2d\mathcal{L}$, versus the bandwidth cost, $\frac{2dl\log(|\mathbb{F}|)}{\mathcal{B}}$. Note that these timing figures are simulated, based on the average values of latency and the corresponding average bandwidth found when pinging several cloud servers such as Google, Amazon and Microsoft from a university network. The dependencies on both latency and bandwidth are linear. Consequently, if the latency were reduced to 10ms, this would halve the times in the Latency cost column; similarly, if bandwidth were doubled, this would halve the times in the Bandwidth cost column. We observe then that for all but very low bandwidth scenarios, the latency cost will dominate.

Finally, we put these pieces together, and consider the total protocol time from both computation and communication components. We obtain the total time by summing t[work] and t[comm], in Table 6. These results confirm our earlier models, and the fastest time is achieved with a very small number of rounds. For all values of n tested in these experiments, we see the optimal value of d is 2, the minimally interactive scenario. The trend is such that, because of the sheer domination of latency and t[β_0], it is unlikely that more than two or three rounds will ever be needed for even the largest data sets. As n increases, the size of t[work] grows faster than t[comm], predominantly due to t[β_0]. Therefore to minimize the cost of verification one should prefer a small constant number of rounds.

6 Concluding Remarks

Our experimental study supports the claim that fewer rounds of interaction are preferable to allow efficient interactive proofs for linear algebra primitives. For large instances in our experiments, the optimal number of rounds is just two. These primitives allow simple implementation of more complex tools such as regression and linear predictors [6]. Other primitive operations, such as scalar multiplication and addition, are trivial within this model (since LDE evaluations and fingerprints are linear functions), so these primitives collectively allow a variety of computations to be efficiently verified. Further operators, such as matrix (pseudo)inversion and factorization are rather more involved, not least since they bring questions of numerical precision and representation [6]. Nevertheless, it remains open to show more efficient protocols for other functions, such as matrix exponentiation, and to allow sequences of operations to be easily 'chained together' to verify more complex expressions.

References

1. Scott Aaronson and Avi Wigderson. Algebrization: A new barrier in complexity theory. *ACM Trans. Comput. Theory*, 1(1):2:1–2:54, February 2009. URL: `http://doi.acm.org/10.1145/1490270.1490272`, `doi:10.1145/1490270.1490272`.
2. Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.
3. Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. Cryptology ePrint Archive, Report 2016/116, 2016. `https://eprint.iacr.org/2016/116`.
4. Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, and Ron D. Rothblum. Fiat-shamir from simpler assumptions. Cryptology ePrint Archive, Report 2018/1004, 2018. `https://eprint.iacr.org/2018/1004`.
5. Amit Chakrabarti, Graham Cormode, and Andrew Mcgregor. Annotations in data streams. *Automata, Languages and Programming*, pages 222–234, 2009.
6. Graham Cormode and Chris Hickey. Cheap checking for cloud computing: Statistical analysis via annotated data streams. In *International Conference on Artificial Intelligence and Statistics*, pages 1318–1326, 2018.
7. Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 90–112. ACM, 2012.
8. Graham Cormode, Justin Thaler, and Ke Yi. Verifying computations with streaming interactive proofs. *Proceedings of the VLDB Endowment*, 5(1):25–36, 2011.
9. Samira Daruki, Justin Thaler, and Suresh Venkatasubramanian. Streaming verification in data analysis. In *International Symposium on Algorithms and Computation*, pages 715–726. Springer, 2015.
10. Rūsiņš Freivalds. Fast probabilistic algorithms. *Mathematical Foundations of Computer Science 1979*, pages 57–69, 1979.
11. Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 113–122. ACM, 2008.
12. Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)*, 39(4):859–868, 1992.
13. Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *Commun. ACM*, 59(2):103–112, January 2016.
14. Michael O Rabin. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.
15. Srinath TV Setty, Richard McPherson, Andrew J Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, volume 1, page 17, 2012.
16. Srinath TV Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security Symposium*, pages 253–268, 2012.
17. Adi Shamir. IP=PSPACE. *Journal of the ACM (JACM)*, 39(4):869–877, 1992.
18. Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Advances in Cryptology–CRYPTO 2013*, pages 71–89. Springer, 2013.
19. Victor Vu, Srinath Setty, Andrew J Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 223–237. IEEE, 2013.

A Details of Proof of Lemma 5

▶ **Lemma 9** (Restatement of Lemma 5). *Given $a, b \in \mathbb{F}_p^n$ the sum*

$$a^T b = \sum_{k_0=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} f_a(k_0, ..., k_{d-1}) f_b(k_0, ..., k_{d-1})$$

can be verified using a $(d-1)$-round $(ld, l+d)$-protocol with helper overhead time $O(\frac{n\log(n)}{d})$, and verifier overhead time of $O(nld)$ and checking computation time $O(ld)$.

Proof. First, set

$$g(k_0, ..., k_{d-1}) = f_a(k_0, ..., k_{d-1}) f_b(k_0, ..., k_{d-1})$$

$g : \mathbb{F}_q \times ... \times \mathbb{F}_q \to \mathbb{F}_q$ is a degree $2l$ polynomial in each variable. Now, consider round $j+1$ of the sum-check protocol, where the helper is required to send

$$g_j(x) = \sum_{k_{j+1}=1}^{l} \cdots \sum_{k_d=1}^{l} g(r_1, ..., r_{j-1}, x, k_{j+1}, ..., k_d)$$

Here, g is degree $2l$ polynomial, sent to V as a set $G_j^{\Sigma} = \{(g_j(x), x) : x \in [2l]\}$. To compute this set we have H find the individual summands as

$$G_j = \Big\{ \big(g(r_1, ..., r_{j-1}, x, k_{j+1}, ..., k_{d-1}), x\big) : x \in [2l], k_{j+1}, ..., k_{d-1} \in [l] \Big\}$$

Naive computation of all the values in G_j takes time $O(nd)$ each, for a total cost of $O(nl^{d-j}d)$. However, instead of computing the LDE at l^{d-j} points with cost $O(ld)$ we can sum l^{d-j} convolutions of length $2l$ vectors to obtain the same result (See below). The total cost of the convolution is $O(l\log(l)) = O(\frac{l\log(n)}{d})$, using $n = l^d$. Summing these l^{d-j} convolutions gives the cost of the jth round for the helper as $O(\frac{l^{d-j}\log(n)}{d})$. Summing over the d rounds gives us our cost of $O(\frac{n\log(n)}{d})$. ◀

A.1 Finding G_j with Convolution

To simplify the argument, we consider the computation of $a^T a$ (also referred to as F_2). The general case of $a^T b$ follows the same steps but the notation quickly becomes cumbersome. So, given a vector $a \in \mathbb{F}_q^n$, we want to find $\sum_{i=0}^{n-1} a_i^2$. This is equivalent to finding the inner product of a with itself.

Consider a $d-1$ round protocol for the F_2 problem on $a \in \mathbb{F}_q^n$. We have $n = l^d$, and so for each round of interaction the helper sends

$$g_j(x) = \sum_{k_{j+1}=1}^{l} \cdots \sum_{k_{d-1}=1}^{l} f_A(r_0, ..., r_{j-1}, x, k_{j+1}, ..., k_{d-1})^2,$$

where the input is reshaped as the d-dimensional $A \in \mathbb{F}^{l \times l \times ... \times l}$. There are $d-1$ such polynomials to send over the course of the protocol, and each one has degree $2l-1$.

Round 1.

Consider first the opening round

$$g_0(x) = \sum_{k_1=1}^{l} \cdots \sum_{k_{d-1}=1}^{l} f_A(x, k_1, ..., k_{d-1})^2$$

This can be found by materializing the set of values $G_0 = \Big\{ \big(f_A(x, k_1, ..., k_d), x\big) : x \in [2l], k_1, ..., k_{d-1} \in [l] \Big\}$, and then summing over $k_1, \ldots, k_d$ to obtain G_0^Σ.

For the first half of the G_0^Σ, the computation is closely linked to the original input, and so we can simply compute the partial sums

$$\sum_{k_1=1}^{l} \cdots \sum_{k_{d-1}=1}^{l} f_A(x, k_1, ..., k_{d-1})^2.$$

These sums partition the input, so the total time is $O(n)$ to obtain the values for all $x \in [l]$.

However, for x values in the range $l+1 \ldots 2l$, we need to evaluate the LDE at locations not present in the original input. To avoid the higher cost associated with naive computation of all terms, we expand the definition of LDEs:

$$f_A(k_0, ..., k_{d-1}) = \sum_{p_0=0}^{l-1} \cdots \sum_{p_{d-1}=0}^{l-1} A_{p_0 p_1 \ldots p_{d-1}} \chi_{p_0 p_1 \ldots p_{d-1}}(k_0, ..., k_{d-1})$$

$$\chi_{p_0 p_1 \ldots p_{d-1}}(k_0, ..., k_{d-1}) = \prod_{j=0}^{d-1} \prod_{i=0, i \neq p_j}^{l-1} \frac{k_j - i}{p_j - i}$$

In what follows, we can make use of the fact that not all input values contribute to every LDE evaluation needed. We expand as follows:

$$\begin{aligned}
g_0(x) &= \sum_{k_1=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} f_A(x, k_1, ..., k_{d-1}) \\
&= \sum_{k_1=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \Bigg(\sum_{p_0=0}^{l-1} \sum_{p_1=0}^{l-1} \cdots \sum_{p_{d-1}=0}^{l-1} \\
&\qquad \left(A_{p_0 p_1 \ldots p_{d-1}} \left[\prod_{i=0, i \neq p_0}^{l-1} \frac{x - i}{p_0 - i} \right] \left[\prod_{i=0, i \neq p_1}^{l-1} \frac{k_1 - i}{p_1 - i} \right] \cdots \left[\prod_{i=0, i \neq p_{d-1}}^{l-1} \frac{k_{d-1} - i}{p_{d-1} - i} \right] \right) \Bigg)^2 \\
&= \sum_{k_1=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \left(\sum_{p_0=0}^{l-1} \left[A_{p_0 k_1 \ldots k_{d-1}} \prod_{i=0, i \neq p_0}^{l-1} \frac{x - i}{p_0 - i} \right] \right)^2 \\
&= \sum_{k_1=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \left(\sum_{p_0=0}^{l-1} \left[\left(A_{p_0 k_1 \ldots k_{d-1}} \prod_{i=0, i \neq p_0}^{l-1} \frac{1}{p_0 - i} \right) \left(\prod_{i=0}^{l-1} (x - i) \right) \left(\frac{1}{x - p_0} \right) \right] \right)^2 \\
&= \sum_{k_1=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \left(\left(\prod_{i=0}^{l-1} (x - i) \right) \sum_{p_0=0}^{l-1} \left[\left(A_{p_0 k_1 \ldots k_{d-1}} \prod_{i=0, i \neq p_0}^{l-1} \frac{1}{p_0 - i} \right) \left(\frac{1}{x - p_0} \right) \right] \right)^2
\end{aligned}$$

Note in the second step we use that

$$\sum_{p_j=0}^{l-1} \prod_{i=0, i \neq p_j}^{l-1} \frac{k_j - i}{p_j - i} = \begin{cases} 0 & p_j \neq k_j \\ 1 & p_j = k_j \end{cases}$$

We now introduce the helper functions

$$g(p) = \frac{1}{p} \quad ; \quad h(x) = \prod_{i=1}^{l} (x - i) \quad \text{and} \quad q(p) = \prod_{i=0, i \neq p}^{l-1} \frac{1}{p - i} \tag{6}$$

to simplify the notation. We define the vectors

$$b_{k_1 \dots k_{d-1}}(p) = \begin{cases} A_{p,k_1 \dots k_{d-1}} q(p) & \text{for } p \in [0, l-1], k_1, ..., k_{d-1} \in [0, l-1] \\ 0 & \text{for } p \in [l, 2l-1], k_1, ..., k_{d-1} \in [0, l-1] \end{cases}$$

and use these to rewrite in terms of convolutions

$$\begin{aligned} g_0(x) &:= \sum_{k_1=1}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \left(h(x) \sum_{p_0=0}^{l-1} \left[b_{k_1 \dots k_{d-1}}(p_0) g(x - p_0) \right] \right)^2 \\ &= h(x)^2 \sum_{k_1=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} (\mathrm{conv}(b_{k_1 \dots k_{d-1}}, g)[x])^2 \\ &= h(x)^2 \left(\sum_{k_2=1}^{l} \cdots \sum_{k_d=1}^{l} \mathrm{DFT}^{-1}(\mathrm{DFT}(b_{k_1 \dots k_{d-1}}) \cdot \mathrm{DFT}(g)) \right) [x]^2 \end{aligned}$$

Thus, by precomputing some arrays of values, we reduce the computation to several convolutions that can be evaluated quickly via fast Fourier transform. Observe that this FFT does not need to be computed over the same field as the matrix multiplication: we can choose any suitably large field for which there is an FFT (say, real vectors of size 2^j for some j), and then map the result back into $\mathbb{F}_q$. Forming $b_{k_1 \dots k_d}(p)$ takes time $O(l^d)$. We have to do $O(l^{d-1})$ convolutions on vectors of length $O(l)$, so each convolution takes time $O(l \log(l))$. Since $\log(l) = \log(n^{\frac{1}{d}})$, we can write the helper's time cost for the first round as $O(\frac{n}{d} \log(n))$.

Round j.

Similar rewritings are possible in subsequent rounds. Initially, it may seem that things are more complex for G_j, as each $f_A(r_0, ..., r_{j-1}, x, k_{j+1}, ..., k_{d-1})$ appears to require full inspection of the input to evaluate at $(r_0, ..., r_{j-1})$. However, we can again define an ancillary array $b_{k_1 \dots k_{d-1}}$ to more easily compute this. In the sum-check protocol after the helper sends G_0, it receives r_0, with which we define the array over $[l]^{d-1}$:

$$A^{(1)}_{r_0 k_1 \dots k_{d-1}} = \sum_{p=0}^{l-1} b_{k_1 \dots k_{d-1}}(p) \prod_{i=0, i \neq p}^{l-1} (r_0 - i)$$

This allows the Helper to form G_1 using the same idea as above, but with $A^{(1)}$ instead of A. Working in terms of $A^{(1)}$ reduces the Helper's cost from $O(l^{d-1} l d)$ for computing the $f_A(r_0, k_1, ..., k_{d-1})$ for each $k_i \in [l]$ to just $O(l^2)$ when combined with using $b_{k_1 \dots k_{d-1}}$.

In more detail, and with more generality, let us consider the jth round, where we are forming G_j and G_j^{Σ}. We define

$$A^{(j)}_{r_0, \dots, r_{j-1}, k_j \dots k_{d-1}} = \sum_{p=0}^{l-1} b_{k_j \dots k_{d-1}}(p) \prod_{i=0, i \neq p}^{l-1} (r_{j-1} - i)$$

Then we have the following computation for $x \in [l, 2l-1]$:

$$
\begin{aligned}
g_j(x) &= \sum_{k_{j+1}=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} f_A(r_0, ..., r_{j-1}, x, k_{j+1}, ..., k_{d-1})^2 \\
&= \sum_{k_{j+1}=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \left(\sum_{p_0=0}^{l-1} \cdots \sum_{p_{d-1}=0}^{l-1} \left(A_{p_0 \dots p_{d-1}} \left[\prod_{i=0, i \neq p_0}^{l-1} \frac{r_0 - i}{p_0 - i} \right] \cdots \left[\prod_{i=0, i \neq p_{j-1}}^{l-1} \frac{r_{j-1} - i}{p_{j-1} - i} \right] \right.\right. \\
&\qquad \left.\left. \left[\prod_{i=0, i \neq p_j}^{l-1} \frac{x - i}{p_j - i} \right] \left[\prod_{i=0, i \neq p_{j+1}}^{l-1} \frac{k_{j+1} - i}{p_{j+1} - i} \right] \cdots \left[\prod_{i=0, i \neq p_{d-1}}^{l-1} \frac{k_{d-1} - 1}{p_{d-1} - 1} \right] \right) \right)^2 \\
&= \sum_{k_{j+1}=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \left(\sum_{p_j=0}^{l-1} \left[A^{(j)}_{r_0 \dots r_{j-1} p_j k_{j+1} \dots k_{d-1}} \prod_{i=0, i \neq p_j}^{l-1} \frac{x - i}{p_j - i} \right] \right)^2 \\
&= \sum_{k_{j+1}=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \left(\sum_{p_j=0}^{l-1} \left[\left(A^{(j)}_{r_0 \dots r_{j-1} p_j k_{j+1} \dots k_{d-1}} \prod_{i=0, i \neq p_j}^{l-1} \frac{1}{p_j - i} \right) \left(\prod_{i=0}^{l-1} (x - i) \right) \left(\frac{1}{x - p_j} \right) \right] \right)^2 \\
&= \sum_{k_{j+1}=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \left(\left(\prod_{i=0}^{l-1} (x - i) \right) \sum_{p_j=0}^{l-1} \left[\left(A^{(j)}_{r_0 \dots r_{j-1} p_j k_{j+1} \dots k_{d-1}} \prod_{i=0, i \neq p_j}^{l-1} \frac{1}{p_j - i} \right) \left(\frac{1}{x - p_j} \right) \right] \right)^2
\end{aligned}
$$

We make use of the same set of helper functions specified in equation (6), and define the vectors

$$
b_{k_{j+1} \dots k_d}(p) = \begin{cases} A^{(j)}_{r_0 \dots r_{j-1} p k_{j+1} \dots k_{d-1}} q(p) & \text{for } p \in [0, l-1], k_{j+1}, ..., k_d \in [0, l-1] \\ 0 & \text{for } p \in [l, 2l-1], k_{j+1}, ..., k_{d-1} \in [0, l-1] \end{cases}
$$

We can now continue to express the computation in terms of convolutions

$$
\begin{aligned}
g_j(x) &:= \sum_{k_{j+1}=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \left(h(x) \sum_{p_j=0}^{l-1} \left[b_{k_{j+1} \dots k_{d-1}}(p_j) g(x - p_j) \right] \right)^2 \\
&= \sum_{k_{j+1}=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \left(h(x) \, \mathrm{conv}(b_{k_{j+1} \dots k_{d-1}}, g)[x] \right)^2 \\
&= h(x)^2 \left(\sum_{k_{j+1}=0}^{l-1} \cdots \sum_{k_{d-1}=0}^{l-1} \mathrm{DFT}^{-1}(\mathrm{DFT}(b_{k_j \dots k_d}) \cdot \mathrm{DFT}(g)) \right) [x]^2
\end{aligned}
$$

We can think of $A^{(j)}$ as a shrinking input array, where $A^{(j)} \in \mathbb{F}^{l \times l \times \dots \times l}$ is $d-j$ dimensional, and

$$
b_{k_{j+1} \dots k_d}(p_j) = A^{(j)}_{r_1 \dots r_{j-1} p_j k_{j+1} \dots k_d} \prod_{i=1, i \neq p_j}^{l} \frac{1}{p_j - i}
$$

$$
A^{(j)}_{r_0, \dots, r_{j-1}, k_j \dots k_{d-1}} = \sum_{p_{j-1}=0}^{l-1} A^{(j-1)}_{r_1 \dots r_{j-2} p_{j-1} k_j \dots k_d} \prod_{i=0, i \neq p_{j-1}}^{l-1} \frac{r_{j-1} - i}{p_{j-1} - i}
$$

Using this formulation, the dominant computation cost in round j will be from the FFT, which involves l^{d-j-1} convolutions of cost $O(\frac{l}{d} \log(n))$ each. Thus the final cost for the round is $O(\frac{l^{d-j}}{d}\log(n))$. The cost of running the entire protocol requires $d-1$ rounds, making the computational cost for the helper

$$O\left(\sum_{j=0}^{d-2}\frac{l^{d-j}}{d}\log(n)\right) = O\left(n\log(n)\frac{\sum_{j=0}^{d-2} l^{-j}}{d}\right) = O\left(\frac{n\log(n)}{d}\right)$$

since $l \geq 2$. Note that when $d = \log(n)$ and $l = 2$, we achieve $O(n)$ time for the helper. The cost increases with fewer rounds, up to a maximum of $O(n\log n)$ for a constant round protocol.

Cost summary.

For the verifier, the checking computation cost is $O(ld)$, which emerges from the d rounds, where in each round the verifier sums the first l elements of G_j^{Σ}, before evaluating the LDE of G_j^{Σ} at r_j, making for a total cost of $O(l)$. The streaming overhead for the verifier involves evaluating the LDE of the input A, for a cost of $O(nld)$ The verifier requires $O(l+d)$ memory to find the LDE of a at $r \in \mathbb{F}^d$. The communication will be $O(ld)$ as we have the helper sending d sets G_j of size $O(l)$. Hence, we summarize the various costs as

Rounds $d-1$
Communication $O(ld)$
Verifier Memory $O(l+d)$
Helper Computation Time $O(\frac{n\log(n)}{d})$
Verifier Overhead Time $O(nld)$
Verifier Checking Computation Time $O(ld)$