Speeding Up Sum-Check Proving[⋆]

Suyash Bagad[1], Quang Dao[2], Yuval Domb[1], and Justin Thaler[3]

[1] Ingonyama
[2] Carnegie Mellon University
[3] a16z crypto research and Georgetown University

Abstract. At the core of the fastest known SNARKs is the sum-check protocol. In this paper, we describe two complementary optimizations that significantly accelerate sum-check proving in key applications.

The first targets scenarios where polynomial evaluations involve small values, such as unsigned 32-bit integers or elements of small subfields within larger extension fields. This setting is common in applications such as Jolt, a state-of-the-art zero-knowledge virtual machine (zkVM) built on the sum-check protocol. Our core idea is to replace expensive multiplications over large fields with cheaper operations over smaller domains, yielding both asymptotic speedups and significant constant-factor improvements. The second optimization addresses a common pattern where sum-check is applied to polynomials of the form $g(x) = \mathsf{eq}(r, x) \cdot p(x)$, where eq is the multilinear extension of the equality function. We present a technique that substantially reduces the prover's cost associated with the equality polynomial component. We also describe how to combine both optimizations, which is essential for applications like Spartan within Jolt.

We have implemented and integrated our optimizations into the Jolt zkVM. Our benchmarks show consistent 2-3× speedups for proving the first sum-check of Spartan within Jolt, with performance gains reaching 20× or more when baseline methods approach their memory limits.

1 Introduction

The sum-check protocol [33] is a fundamental tool in the design of succinct interactive proofs. It provides an information-theoretically sound method for verifying claims of the form

$$\sum_{x \in \{0,1\}^\ell} g(x) = t,$$

where g is a multivariate polynomial over a finite field $\mathbb{F}$ and t is a specified value. From the verifier's perspective, the sum-check protocol acts as a *reduction* from the expensive task of summing 2^ℓ evaluations of g to the (hopefully) much cheaper task of evaluating g at a single random point in $\mathbb{F}^\ell$.

When used in standalone interactive settings, the sum-check protocol is information theoretically secure, meaning it requires no cryptographic commitments whatsoever. When used within SNARKs, the polynomial g is typically derived in some manner from a committed witness. In this context, the sum-check protocol generally *minimizes* (but does not eliminate) the amount of data the SNARK prover must commit to, and *also* minimizes the time required to prove that this committed data is well-formed.

These features have made sum-check a cornerstone of modern SNARK constructions. It underpins SNARKs for proving satisfiability of arithmetic circuits [26,49,53], R1CS [38], Plonkish and CCS [15,41], the fastest known lookup arguments [40,42], polynomial commitment schemes [3,21,22,37], folding schemes [11,30,31,12,36], and zero-knowledge virtual machines (zkVMs) like Jolt [4].

A key motivating application for our work is Jolt, a zkVM targeting the RISC-V instruction set. Sum-check lies at the core of every part of Jolt. The system consists of three main components:

[⋆] This manuscript combines and improves upon results from [6,20].

- A read-only memory checking protocol, also known as a *lookup argument* or *batch-function-evaluation protocol*. Currently Jolt uses Lasso [42] for read-only memory checking, but is imminently switching to Shout [40]),
- A read/write memory checking protocol (Jolt is currently using Spice [39], but is imminently switching to Twist [40]),
- And a highly structured R1CS instance, which is proven using a variant of Spartan [38] optimized for constraint systems with repeated structure.

Roughly, Jolt uses read-only memory checking to process reads into program bytecode and to execute primitive instructions, read/write memory checking to process reads and writes to the VM's registers and RAM, and Spartan to capture the "fetch-decode" logic of the VM's "fetch-decode-execute" loop (i.e., to make sure that at each cycle, the correct primitive instruction is applied to the correct inputs and written to the correct destination register or memory cell).

All three components use sum-check as their core protocol. As the memory-checking components have been optimized, the Spartan subprotocol within Jolt has become a major bottleneck. This reflects a broader trend in SNARK design: *the sum-check protocol is so effective at reducing commitment costs that its own proving time now dominates the prover's end-to-end cost.*

Virtually every application of the sum-check protocol in SNARK design, Jolt included, uses it to prove that a polynomial of the form

$$g(X) = \widetilde{\mathsf{eq}}(r, X) \cdot p(X)$$

sums to a specified value, where $\widetilde{\mathsf{eq}}$ is the multilinear extension[4] of the equality function and $p(X)$ is a product of multilinear polynomials. This includes every one (of several dozen) invocations of the sum-check protocol within Jolt.

Even more can be said about one of the key prover bottlenecks in Jolt, namely its application of the sum-check protocol within Spartan. In this application, $g(X)$ has the form $\widetilde{\mathsf{eq}}(r, X) \cdot (a(X) \cdot b(X) - c(X))$. Here, a, b, and c are multilinear extensions derived from a committed "execution trace" vector z via public matrices A, B, and C (i.e., $a = \widetilde{Az}$, $b = \widetilde{Bz}$, and $c = \widetilde{Cz}$.[5]). The equality polynomial is not committed because it can be evaluated directly by the verifier in time $O(\ell)$ (which is logarithmic in the instance size, 2^ℓ).

We make the crucial observation that the values being summed up by Spartan's sum-check (ignoring the $\widetilde{\mathsf{eq}}$ factor) are *small*. This is because execution traces contain values that are storable within the VM's registers. In Jolt today, these registers store 32-bit words. In fact, many values within an execution trace are much smaller than 32 bits (e.g., binary flags for instruction selection). Yet, the proving field used in Jolt is very large (e.g., 256-bit prime fields used for elliptic curve-based polynomial commitments, and 128-bit fields for other commitment schemes).

With the above context in mind, the focus of our work is the following question:

*Can we speed up the sum-check prover, both **asymptotically** and **in practice**, in common settings in modern SNARKs and zkVMs?*

1.1 Our Results

We present new algorithms for the sum-check prover that improve upon existing methods in the following settings:

- Small-value sum-check, where the values being summed are small relative to the field size,

[4] The multilinear extension $\widetilde{f}$ of a function $f\colon \{0,1\}^\ell \to \mathbb{F}$ denotes the unique multilinear polynomial satisfying $\widetilde{f}(x) = f(x)$ for all $x \in \{0,1\}^\ell$.

[5] A tilde over a vector such as Az of length 2^ℓ denotes the multilinear extension of the function over domain $\{0,1\}^\ell$ whose evaluation table is given by vector.

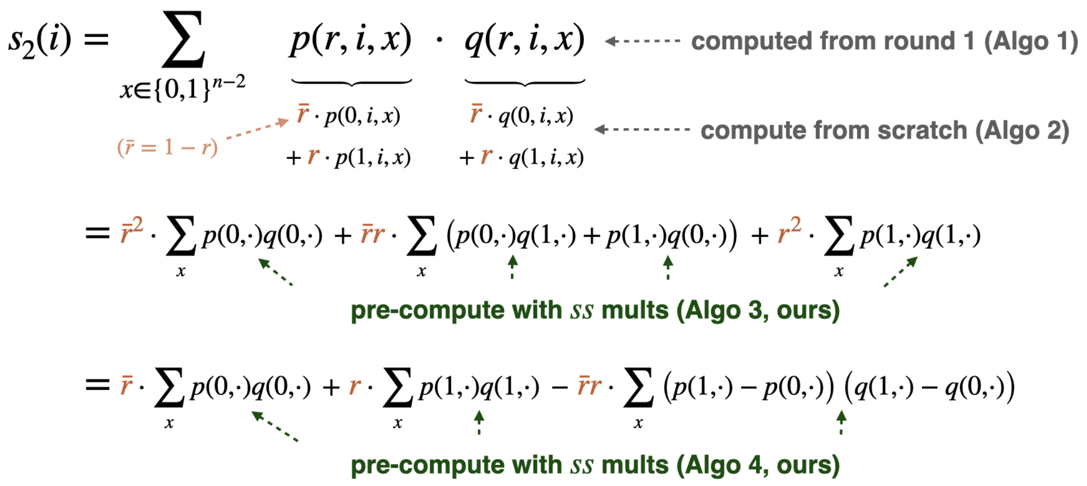

computed from round 1 (Algo 1)
compute from scratch (Algo 2)
pre-compute with ss mults (Algo 3, ours)
pre-compute with ss mults (Algo 4, ours)

Fig. 1: Overview of existing sum-check algorithms and our optimization in the small-value setting. Here $g(X) = p(X) \cdot q(X)$ for multilinear polynomials p, q, and we illustrate our small-value optimization on round 2 of the sum-check protocol.

- Eq-poly sum-check, where the polynomial being summed includes an $\widetilde{\mathsf{eq}}$ factor,
- The combination of the above two settings, as arises in Spartan-in-Jolt.

Our algorithms give an asymptotic speedup to the prover when applied in the above settings. Our speedups in practice range between 2.5-4× for the Spartan-in-Jolt sum-check invocation.

Beyond speed, our results substantially reduce memory usage. Our eq-polynomial optimization eliminates the 2^n-sized equality polynomial table common in prior work [47,44]. Furthermore, our small-value algorithm enables a streaming prover, using small space by processing values on-the-fly, as in Spartan-in-Jolt. Subsequent work [35] has leveraged this to achieve a small-space Jolt prover with minimal time overhead.

Our optimizations have been integrated into the Jolt zkVM.[6] We also provide a standalone implementation for testing a wider range of algorithms and field settings, though it is not yet fully optimized.[7]

Small-Value Sum-Check Optimization. Our first optimization targets scenarios where the values being summed are small relative to the field size. In such settings, field multiplications fall into three categories:

$$\mathfrak{ss} : \text{small} \times \text{small}, \quad \mathfrak{sl} : \text{small} \times \text{large}, \quad \mathfrak{ll} : \text{large} \times \text{large}.$$

Empirically (and asymptotically as well), $\mathfrak{ss} \ll \mathfrak{sl} \ll \mathfrak{ll}$, with large-large multiplications being an order of magnitude more expensive than the $\mathfrak{ss}$.[8] Our optimization reduces the number of $\mathfrak{ll}$ multiplications from $O(N)$ (where $N = 2^\ell$ is the size of the summation domain) based on prior work [18,38] to $O(N/\mathrm{poly}(\kappa))$, where κ is the factor difference between $\mathfrak{ll}$ and $\mathfrak{ss}$ multiplications.

The precise poly(κ) factor prover savings depends on: (1) the number d of multilinear factors in $g(x)$, and (2) the relative cost of $\mathfrak{ll}$ versus $\mathfrak{sl}$ versus $\mathfrak{ss}$ multiplications. For example, in the Spartan-in-Jolt setting mentioned above, where $d = 2$ and $\mathfrak{ll}$ multiplications are $\kappa := \lambda^2$ times more expensive than $\mathfrak{ss}$ (as with

[6] See `https://github.com/a16z/jolt/pull/690`
[7] See `https://github.com/ingonyama-zk/smallfield-super-sumcheck/tree/sb/eq-optimized`
[8] The difference between $\mathfrak{sl}$ and $\mathfrak{ll}$ depends on the specific field setting; see Section 2.1.

Montgomery multiplication for λ-bit prime fields [34,23]), our algorithm reduces the dominant cost of $\mathfrak{ll}$ multiplications by a factor of roughly $\lambda^{0.63}$.

To achieve this speedup, we need both our new sum-check proving algorithm *and* an optimized implementation for $\mathfrak{sl}$ multiplications in large prime fields, which we view as an independent contribution (see Section 7). Specifically, we show how to multiply a small value (fitting in a single machine word) by a large value (requiring $N > 1$ machine words) in time linear in N, improving upon standard Montgomery multiplication for two N-word values, which takes $O(N^2)$ time.[9]

Small-value algorithm overview. Figure 1 illustrates the different algorithms for the sum-check prover (existing ones as well as our new algorithms for the small-value setting, for the special case where g is the product of *two* multilinear polynomials). To explain the figure, we first recall what the sum-check prover needs to send the verifier in each round. For round i, the prover needs to compute:

$$s_i(u) = \sum_{x \in \{0,1\}^{n-i}} \prod_{k=1}^{d} p_k(r_1, \dots, r_{i-1}, u, x), \tag{1}$$

where $r_1, \dots, r_{i-1}$ are the verifier's challenges from previous rounds and u ranges over a large enough set of evaluation points (e.g., $u \in \{0, 1, \dots, d\}$ suffices).

While the prover's computation of its first-round message $s_1(u)$ trivially benefits from the fact that all $p_k(u, x)$ are small, starting from round $i \geq 2$, the evaluations $p_k(r_1, \dots, r_{i-1}, u, x)$ are no longer small, due to the presence of the random challenges $r_1, \dots, r_{i-1} \in \mathbb{F}$. This means that the standard linear-time prover (Algorithm 1 in Figure 1) exclusively performs $\mathfrak{ll}$ multiplications from round 2 onward. The (previously known) small-space proving algorithm (Algorithm 2 in Figure 1) recomputes the products "from scratch" each round (instead of using a linear-sized caching data structure), which involves many cheaper $\mathfrak{sl}$ multiplications but remains inefficient due to redundant recalculations.

As Algorithm 1's cost is concentrated in the first few rounds, our key idea is to replace these rounds with cheap $\mathfrak{ss}$ and $\mathfrak{sl}$ multiplications. By *expanding out* the products in Algorithm 2, we can re-express the computation as an inner product of a coefficient vector (depending on large field elements $r_1, \dots, r_{i-1}$) of length $(d+1)^{i-1}$, with a vector of small-value elements (obtained via summation over various products of p_i). This gives Algorithm 3 in Figure 1.

A downside of Algorithm 3 is that one replaces each $\mathfrak{ll}$ multiplication performed by Algorithm 1 with $O(2^{d \cdot i})$ $\mathfrak{ss}$ multiplications for each round i. As the round number increases, eventually we hit a point where Algorithm 1 (the linear-time prover) is more efficient than our Algorithm 3; at that point, we simply switch back to Algorithm 1. More precisely, to switch back to Algorithm 1, the prover requires arrays that store evaluations $\{p_k(r_{[<i]}, x)\}_{k \in [d], x \in \{0,1\}^{n-i}}$, where $r_{[<i]}$ is shorthand for $(r_1, \dots, r_{i-1})$. These can be quickly computed by running Algorithm 2 for one round (see Section 4.1 for details).

We can improve Algorithm 3 further, reducing the $O(2^{d \cdot i})$ overhead in $\mathfrak{ss}$ multiplications to $O((d+1)^i)$ for the same round i. The idea is to treat the product $\prod_k p_k(r_1, \dots, r_{i-1}, u, x)$ as the evaluation of a polynomial $F(X_1, \dots, X_{i-1})$ on $(r_1, \dots, r_{i-1})$, then use *polynomial interpolation* (in each variable X_i) to compute F instead. This approach, which we call Algorithm 4, draws from the Toom-Cook multiplication algorithm for polynomials [46] (see Section 4.2 for details).

Eq-Poly Sum-Check Optimization. Our second optimization targets the ubiquitous scenario where $g(x)$ includes an $\widetilde{\mathsf{eq}}(w, X)$ term. In this setting, we leverage the decomposition of $\widetilde{\mathsf{eq}}$ into a product: $\widetilde{\mathsf{eq}}(w, X) = \prod_{i=1}^{\ell} \widetilde{\mathsf{eq}}(w[i], X_i)$. Because of this, we may rewrite the computation of the i-th sum-check

[9] While our techniques are standard (e.g., optimized Barrett reduction), we know of no existing exposition or implementation for this specific setting.

message $s_i(X)$ as:

$$s_i(X) = \sum_{x \in \{0,1\}^{\ell-i}} \widetilde{\mathsf{eq}}(w_{[<i]}, r_{[<i]}) \cdot \widetilde{\mathsf{eq}}(w_i, X) \cdot t_i(X), \tag{2}$$

where $t_i(X)$ is given by the iterated sum:

$$\sum_{x_L \in \{0,1\}^{\ell/2-i}} \widetilde{\mathsf{eq}}(w_L, x_L) \sum_{x_R \in \{0,1\}^{\ell/2}} \widetilde{\mathsf{eq}}(w_R, x_R) \prod_{k=1}^{d} p_k(r_{[<i]}, X, x_L, x_R).$$

We perform two rewritings in the above equation: first, exploiting the fact that $\widetilde{\mathsf{eq}}$ can be written as a product of three terms, and second, rewriting the sum in Equation (2) as an iterated sum. While the first property was previously exploited by Gruen [28] to reduce the prover cost contributed by the $\widetilde{\mathsf{eq}}$ factor (in a way that is orthogonal to, and can be combined with, our own optimization)[10], the second rewriting is our novel contribution.

Our sum-check prover now pre-computes the smaller tables

$$\left\{\widetilde{\mathsf{eq}}(w_R, x_R) : x_R \in \{0,1\}^{\ell/2}\right\}, \quad \left\{\widetilde{\mathsf{eq}}(w_{[<i]}, x) : x \in \{0,1\}^{\ell/2-i}\right\}$$

for all $i = 0, \ldots, \ell/2 - 1$. The prover then uses these tables to compute the sum-check message in each of the first $\ell/2$ rounds of the protocol. The final $\ell/2$ rounds then follow the standard linear-time sum-check prover algorithm. This is in contrast to prior sum-check prover implementations [18,44], which require pre-computing the 2^ℓ-sized table $\{\widetilde{\mathsf{eq}}(w,x)\}_{x \in \{0,1\}^\ell}$. See Section 5 for details.

This optimization is especially impactful because almost all applications of the sum-check protocol in SNARKs involve the $\widetilde{\mathsf{eq}}$ polynomial. In particular, it benefits all three components of Jolt: the Spartan-based subprotocol for proving R1CS, the Twist/Spice protocols for read-write memory checking, and the Shout/Lasso protocols for read-only memory checking (i.e., lookups).

Combining Both: Optimizing Spartan-in-Jolt. We show how the two optimizations can be combined to achieve a speedup in the combined eq-poly and small-value setting. Our speedup here is not as large, since most of the $\mathfrak{ss}$ multiplications in Algorithms 3 and 4 are now $\mathfrak{sl}$ multiplications, due to the extra $\widetilde{\mathsf{eq}}$ factor lying in the large field. Asymptotically, this gives a speedup of about the square root of the speedup we get in the small-value-only setting.[11] See Section 6.1 for details.

In the case of Spartan's first sum-check, we describe two further optimizations. First, we observe that for the particular relation arising in Spartan, the honest prover's first-round message $s_1(X)$ is a univariate polynomial whose evaluations at 0 and 1 are always equal to 0. Hence, the prover does not have to invest any work to compute these two evaluations. Second, the fact that we may use evaluation "at infinity" [12] when specifying each prover message s_i means that we can omit the lower-degree term $\widetilde{Cz}$ in any calculation the prover performs involving evaluation at infinity. For more details, see Section 6.2.

Implementation and Concrete Speedups. In Section 8, we describe our implementation and the demonstrated speedups. To summarize, our optimization gives a 3× speedup for the first sum-check of Spartan-in-Jolt, growing to 20× or more when the linear-time algorithm gets memory-bound. Future extensions to our implementation will also handle a *streaming* prover without much overhead. As Jolt is currently a moving target (switching from Lasso-Spice to Twist-Shout), we leave benchmarking the end-to-end performance improvement of our optimization to future work.

[10] See also a small variation proposed by Setty and Thaler [40] that avoids altering the prover's message in each round of sum-check

[11] $\mathfrak{sl}$ costs are typically geometrically between $\mathfrak{ss}$ and $\mathfrak{ll}$ costs. For example, when applying Montgomery multiplication to N-word values, $\mathfrak{ss}$ multiplications incur one machine multiplication, $\mathfrak{sl}$ multiplications incur $O(N)$ (see Section 7), while $\mathfrak{ll}$ multiplications incur $O(N^2)$.

[12] The evaluation of s_i at infinity refers to the highest-degree coefficient of s_i in the standard monomial basis. See Lemma 2.2 for details.

1.2 Related Work

Origins of the Sum-Check Protocol. The sum-check protocol was introduced by Lund, Fortnow, Karloff, and Nisan [33] as a key component in an interactive proof for #**P** and later **PSPACE** [43]. The prover's in these protocols took superpolynomial time even when applied to very simple problems (e.g., those solvable in logarithmic space). The sum-check protocol was later harnessed by Goldwasser, Kalai, and Rothblum (GKR) [26], who used it to construct an interactive proof for any arithmetic circuit of polylogarithmic depth and polynomial size, with a prover that runs in polynomial time.

Efficient Sum-Check Provers. Cormode, Thaler, and Yi [18] introduced a linear-time implementation of the sum-check prover when applied to a product of multilinear polynomials, along with a space-efficient variation that runs in quasilinear time using only logarithmic space. These algorithms have since been refined and deployed in a variety of systems. For example, they were used in a long line of works to implement linear-time variants of the GKR protocol [17,47,44,48,49,55,53,54]. They also form the prover backbone in Spartan [38], which underlies one of the major components of Jolt. A line of works has also applied the sum-check protocol to obtain efficient memory-checking arguments, meaning a way of forcing an untrusted prover to correctly process reads and writes to a large memory (or just reads, in the case of real-only memories) [56,39,42]. Most recent in this line of work is Twist and Shout [40].

Gruen's Optimizations. A recent work of Gruen [28] also seeks to optimize the sum-check prover when working over fields of small characteristic. Two optimizations in that work are relevant. The first reduces prover overhead in the eq-poly setting. This optimization is orthogonal to our result in this setting. We discuss both Gruen's optimization and how ours can be combined with it in Section 5.

The second of Gruen's optimizations is a technique Gruen calls the "univariate skip". This technique targets the same "small value" setting that we do, but has several downsides relative to our algorithms. The univariate skip technique applies sum-check to a multivariate polynomial g that has high degree in its first variable–roughly $d \cdot 2^i$ for some $i \geq 1$ and some constant d, and degree d in each of its remaining variables. The sum-check protocol then needs to be modified so that the sum over the first variable is over a larger set of size $d \cdot 2^i$. In this setting, all of the prover's work in round 1 can occur over the base field.

Most importantly, it is applicable *only* in small-characteristic fields, i.e., when values reside in a small subfield of a larger extension field. In contrast, our small-value optimization applies whenever the values being summed are small (even when those small values reside within large prime fields). Without this generality, our optimization would not apply to the current Jolt implementation. In addition, our approach yields a lower proof size, stemming from the high degree of g's first variable in the univariate skip. Finally, many SNARKs derive g from committed polynomials; applying the univariate skip technique would require modifying the commitment scheme to support polynomials with a high-degree first variable, which may add costs. In contrast, our techniques apply "as is" to existing sum-check-based SNARKs, yielding functionally equivalent implementations.

Sparse-Dense and Related Protocols. The *sparse-dense sum-check* prover algorithm [42] achieves a general time-space tradeoff, running in $O(cn)$ time and $O(n^{1/c})$ space for any integer $c > 0$, but applies only to structured sum-check instances, involving the product of an arbitrary multilinear polynomial and a structured multilinear polynomial. Its refinement, the *prefix-suffix inner product* algorithm [35], further generalizes applicability and improves efficiency for these structured scenarios. Jolt employs these protocols wherever possible as they consistently provide efficiency gains; however, their applicability remains limited to specific structured sum-check instances (in particular, they do not apply to many of the sum-check instances in Jolt, including those arising in Spartan). The Blendy protocol [16] rediscovered a special case of sparse-dense sum-check that applies only in the degenerate scenario where $g(x)$ is a single multilinear polynomial.

The above protocols benefit from our eq-poly optimization, but not from our small-value optimization.

Subsequent work. As mentioned above, after the release of an initial version of our results [6,20], Nair, Thaler, and Zhu [35] described an efficient small-space implementation of the Jolt prover (without invoking SNARK composition or recursion), whose concrete efficiency benefits substantially from both of our optimizations.

An exciting upcoming manuscript of Baweja et al. [10] implements the sum-check prover (applied to a product of a constant number of multilinear polynomials) in sublinear space and slightly superlinear time (i.e., in $O(N \cdot \log\log N)$ field operations and using space $N^{1/k}$ for any constant integer $k > 1$, where $N = 2^\ell$ is the size of the sum being computed). Our techniques can be combined with the algorithm of Baweja et al. in nontrivial ways to obtain a more efficient streaming version of the Jolt prover, which we will detail in a future version of this manuscript. Assuming natural conjectures, Baweja et al. also prove a matching lower bound of $\Omega(N \log\log N)$ field multiplications for the sum-check prover applied to a product of arbitrary multilinear polynomials, when using sublinear space.

Two recent works also focus on the sum-check protocol applied in the small-characteristic setting (like the univariate skip technique [28], these algorithms do not apply when summing small values that reside within a large-characteristic field). Liu and Zhang [32] focus on the setting of binary tower fields, building on a technique of Dao and Thaler [19] to give a sum-check prover algorithm for products of d Boolean-valued multilinear polynomials. However, for the most common settings of d, namely $d \in \{2, 3\}$, their algorithm is asymptotically slower than ours, and no implementation is provided. Wei et al. [50] obtain a fast sum-check prover implementation by combining parallel repetition with NeutronNova's [31] "sumfold" scheme that reduces many independent instances of the sum-check protocol to a single one.

2 Preliminaries

Notation. We denote ranges as follows: $[a, b] := \{a, \ldots, b\}$, $[n] := \{1, \ldots, n\}$, $[< i] := \{1, \ldots, i-1\}$, and $[> i] := \{i+1, \ldots, n\}$ (where n is assumed to be clear from context). We write $\mathsf{GF}[q]$ for the (unique) finite field of order q, and $\mathsf{nat}_b(x) := \sum_{i=1}^{\ell} x_i \cdot b^{i-1}$ for the natural number in base b corresponding to the vector $x \in [0, b-1]^\ell$, abbreviating $\mathsf{nat}(x)$ when $b = 2$. We index vectors from 1.

We denote the multiplication of two small field elements by $\mathfrak{ss}$, of a small field element by a large field element by $\mathfrak{sl}$, and of two large field elements by $\mathfrak{ll}$, and use the same notation for their cost. We occasionally use the shorthand "mults" for field multiplications.

2.1 Field Arithmetic

We recall relevant facts about the performance of field operations in the small-value setting. The particular cases we consider are: (1) w-bit (signed or unsigned) integers, considered inside a large prime field $\mathbb{F} = \mathsf{GF}[p]$, and (2) A *base* field $\mathbb{B} = \mathsf{GF}[p]$, considered inside a degree-k extension field $\mathbb{F} = \mathsf{GF}[p^k]$ of $\mathbb{B}$.

Case (1) is relevant to the current implementation of Jolt [4] because Jolt currently uses elliptic curve commitment schemes that operate over a 256-bit prime-order field. Future versions of Jolt may instead use binary fields and hashing-based commitment schemes [21,22,13], for which Case (2) will be relevant. Alternative future versions of Jolt may be based on lattice commitments [36] and use an extension of the 64-bit Goldilocks field; in this case, *both* Cases (1) and (2) may be applicable.

Small values inside large prime fields. This setting is exemplified by when the values are (say) 64-bit, and the big field is roughly 256-bit (or more), represented as four (or more) limbs of 64-bits each. These fields are subfields of the scalar group of elliptic curves, such as BN254 [8], BLS12-381 [7], or others.

In this setting, $\mathfrak{ss}$ multiplications can be done using native ALU instructions, though unlike case (2), the magnitude of the values grow as we perform more $\mathfrak{ss}$ multiplications. Thus, our optimizations work best when there are few consecutive $\mathfrak{ss}$ multiplications (in Spartan, for instance, we only do one such multiplication).

On the other hand, $\mathfrak{ll}$ multiplications are typically done using *Montgomery multiplication* [34,23], which costs about $2N^2 + 1$ $\mathfrak{ss}$ multiplications, where N is the number of limbs representing a field element (e.g., $N = 4$ limbs are needed to represent a 256-bit field element on a machine with 64-bit words).[13]

Finally, $\mathfrak{sl}$ multiplications can be done using a more optimized algorithm that uses a single round of Barrett reduction [9] instead of full Montgomery reduction. We work out the details of this algorithm in Appendix 7, which only requires roughly $2N$ $\mathfrak{ss}$ multiplications, and is thus 2.5-3 times faster than $\mathfrak{ll}$ multiplications in settings of interest (i.e. multiplying a BN254 scalar by a u64 value).

Base field inside extension fields. An element of the extension field $\mathbb{F}$ can be represented as a vector of coefficients $(a_1, \dots, a_k) \in \mathbb{B}^k$ relative to some basis $\beta_1, \dots, \beta_k$ of the vector space $\mathbb{F}$ over $\mathbb{B}$. The choice of basis affects which algorithms can be used to compute field multiplication and can substantially impact performance.

A standard choice of basis is the *monomial basis*, in which the extension field $\mathbb{F}$ can be viewed as the set of all degree-k polynomials over the base field $\mathbb{B}$, modulo a degree-k irreducible polynomial over $\mathbb{B}$. In this basis, an extension field element's representation is its coefficients when viewed as such a polynomial.

An alternative that can lead to faster multiplication algorithms is the *tower basis*. A degree-2^ℓ extension field $\mathbb{F}$ of $\mathbb{B}$ is said to be a *tower* extension if it is constructed from $\mathbb{B}$ by first constructing a degree-2 extension $\mathbb{B}^{(1)}$, and then constructing a degree-2 extension $\mathbb{B}^{(2)}$ of $\mathbb{B}^{(1)}$, and so on for ℓ iterations. This leads to a basis for $\mathbb{F}$ in which, for any integer $j \geq 0$, the first 2^j basis elements are in the degree-2^j extension field obtained after j iterations of the tower construction. Particularly fast and elegant tower field constructions are known for fields of characteristic two [51,24].

There are (at least) two benefits to tower basis that are important to applications of the sum-check protocol in SNARK design [21]. First, *(intermediate) subfield elements are compressed:* for any degree-2^j extension $\mathbb{B}^{(j)}$ in the tower construction, one can represent any element $x \in \mathbb{B}^{(j)}$ via just 2^j elements of $\mathbb{B}$. Second, one obtains *fast subfield-by-subfield multiplication:* for any two subfields $\mathbb{B}^{(j)}$ and $\mathbb{B}^{(j')}$ in the tower construction with $j < j'$, one can multiply any element $x \in \mathbb{B}^{(j)}$ by any element $y \in \mathbb{B}^{(j')}$ using just $2^{j'-j}$ multiplications in $\mathbb{B}^{(j)}$. Both of these benefits are not present for the standard monomial basis (see [21,22] for details).

$\mathfrak{ll}$ *multiplications.* In both choices of basis (monomial and tower), $\mathfrak{ll}$ multiplications can be done using variants of Karatsuba's algorithm [29], which costs $O(k^{\log_2(3)}) = O(k^{1.58496\dots})$ base-times-base multiplications for each extension-times-extension multiplication. In particular, for the monomial basis, we apply Karatsuba's algorithm for multiplying two polynomials, while for the tower basis, we apply it for recursively multiplying smaller subfield elements.

Hardware Considerations. In FPGAs and ASICs, multiplication in binary fields is highly efficient when field elements are represented using a tower basis. Current CPUs, however, do not natively support efficient multiplication in the tower basis, but they do support fast multiplication in a different (monomial) basis for $\mathsf{GF}(2^{128})$, known as POLYVAL. For instance, ARMv8 provides the PMULL instruction, which efficiently multiplies two 64-bit inputs. To extend multiplication to 128-bit inputs, one can apply the techniques described in [27, Algorithms 2 and 3], which invoke PMULL multiple times. Indeed, BearSSL[14] contains optimized implementations of $\mathsf{GF}(2^{128})$ multiplication in the POLYVAL basis for both x86_64 and ARM64 architectures, utilizing the PMULL instruction for ARM64 and the PCLMULQDQ instruction for x86_64 [21]. Additionally, Intel's Galois Field instruction set (GFNI), originally targeting $\mathsf{GF}(2^8)$ multiplications for AES computations, can be leveraged—together with Karatsuba's algorithm—to achieve reasonably efficient $\mathsf{GF}(2^{128})$ multiplications by decomposing larger operations into sequences of smaller ones.

[13] The rough idea is that we store field elements in Montgomery form, so that $a \bmod p$ is stored as $aR \bmod p$, for a suitably chosen R that is a power of two. Multiplication then first compute $(aR) \cdot (bR) = abR^2$ as integers, then use *Montgomery reduction*, which roughly divides by R and takes the remainder mod p, to reduce to $abR \bmod p$.

[14] `https://bearssl.org/`

Recap of relevant fields and their performance. Most SNARKs need to work over fields of size at least 2^{128}. Working over smaller fields and applying repetition generally does *not* lead to adequate security without major performance overheads, and this is *always* the case for sum-check-based SNARKs.

Given this context, popular fields in SNARK design today fall into the following three categories: (1) 256-bit prime order fields, for use with elliptic curve commitments, (2) A 128-bit extension of a 64-bit field such as Goldilocks, or a 31-bit field such as M31 or Babybear, (3) A binary tower field like $\mathsf{GF}(2^{128})$.

2.2 Multilinear polynomials

We state some facts about multilinear polynomials needed in our algorithms. We refer to standard references, e.g. [45], for derivations.

Multilinear extension. An ℓ-variate polynomial p is *multilinear* if it has degree at most 1 in each variable. The *multilinear extension* of a function $f : \{0,1\}^\ell \to \mathbb{F}$ is the *unique* multilinear polynomial p such that $p(x) = f(x)$ for all $x \in \{0,1\}^\ell$.

In particular, we have that the *equality polynomial* $\widetilde{\mathsf{eq}}$ defined as

$$\widetilde{\mathsf{eq}}(X,Y) = \prod_{i=1}^{\ell} (X_i \cdot Y_i + (1 - X_i) \cdot (1 - Y_i)) \tag{3}$$

is the multilinear extension of the *equality function* $(\cdot \overset{?}{=} \cdot) : \{0,1\}^\ell \times \{0,1\}^\ell \to \{0,1\}$, as we can verify that

$$\widetilde{\mathsf{eq}}(x,y) = \begin{cases} 1 \text{ if } x = y \\ 0 \text{ otherwise} \end{cases}, \quad \text{for all } x, y \in \{0,1\}^\ell.$$

Lemma 2.1. *Let $p \colon \mathbb{F}^\ell \to \mathbb{F}$ be a multilinear polynomial. Then for any $0 \le i \le \ell$, any $(r_1, \dots, r_i) \in \mathbb{F}^i$, and any $x' \in \{0,1\}^{\ell - i}$,*

$$p(r_1, \dots, r_i, x') = \sum_{y \in \{0,1\}^i} \widetilde{\mathsf{eq}}((r_1, \dots, r_i), y) \cdot p(y, x'). \tag{4}$$

Setting $i = 1$, Equation (4) simplifies to

$$p(r_1, x') = (1 - r_1) \cdot p(0, x') + r_1 \cdot p(1, x'). \tag{5}$$

Setting $i = \ell$, we get the *multilinear extension* formula

$$p(r_1, \dots, r_\ell) = \sum_{y \in \{0,1\}^\ell} \widetilde{\mathsf{eq}}(r_1, \dots, r_\ell, y) \cdot p(y). \tag{6}$$

Lagrange Interpolation. A (univariate) polynomial $s(X)$ of degree d is uniquely determined by its values at $d+1$ distinct evaluation points $U = \{x_0, \dots, x_d\} \subset \mathbb{F}$ via the *Lagrange interpolation* formula:

$$s(X) = \sum_{i=0}^{d} s(x_i) \cdot \mathcal{L}_{U,i}(X), \quad \text{where} \quad \mathcal{L}_{U,i}(X) = \prod_{j \ne i} \frac{X - x_j}{x_i - x_j}. \tag{7}$$

In our algorithms, we use a variant of Lagrange interpolation that takes into account the "evaluation at infinity" of a polynomial, defined as $s(\infty) :=$ the highest-degree coefficient of $s(X)$. We obtain the following variant of Lagrange interpolation with ∞ as one of the points.

Parameters:

- ℓ = number of variables (and rounds of sum-check)
- ℓ_0 = number of rounds for small-value optimization
- d = number of multilinear polynomials
- $n = 2^\ell$ is the number of variables of g.

Input:

- Multilinear polynomials $p_1, \ldots, p_d$ specified via their evaluations over $\{0,1\}^\ell$.
- For Algorithm 5 and 6, we also know $w \in \mathbb{F}^\ell$ which defines $\widetilde{\mathsf{eq}}(w, X)$.
- Targets $C_i \in \mathbb{F}$ for the i-th round of sum-check, with C_0 being the initial sum-check target.

Evaluation points: $U_d := (\infty, 0, 1, 2, \ldots, d-1)$, $\widehat{U_d} := U_d \setminus \{1\}$.[a]

[a] For binary tower fields, we use the embedding $i \mapsto \mathsf{bits}(i)$, so that $2 \mapsto z_1$, $3 \mapsto z_1 + 1$, $4 \mapsto z_2$, and so on, where $\mathsf{bits}(i)$ is the binary representation of i, and $z_1, z_2, \ldots$ are "multiplication-friendly" extension field elements [52,19].

Fig. 2: Shared notation for our algorithms

Lemma 2.2. *Let $s(X) = a \cdot X^d + \cdots \in \mathbb{F}[X]$ be a polynomial of degree d. Then for any distinct evaluation points $x_1, \ldots, x_d \in \mathbb{F}$, setting $U = \{\infty\} \cup \{x_1, \ldots, x_d\}$, we have the identity:*

$$s(X) = a \cdot \prod_{k=1}^{d} (X - x_k) + \sum_{k=1}^{d} s(x_k) \cdot \mathcal{L}_{\{x_i\},k}(X). \tag{8}$$

Thus, we can define $\mathcal{L}_{U,0}(X) := \prod_{k=1}^{d}(X - x_k)$ and $\mathcal{L}_{U,k+1}(X) := \mathcal{L}_{\{x_i\},k}(X)$ for $k \in [0, d-1]$.

Proof. Consider $t(X) := s(X) - a \cdot \prod_{i=0}^{d-1}(X - x_i)$. Then $t(X)$ is of degree $\leq d-1$, and $t(x_i) = s(x_i)$ for all $i \in \{0, \ldots, d-1\}$. We thus conclude using Lagrange interpolation formula (Equation 7) applied to $t(X)$.

Note that if $s(X) = \prod_{i=1}^{d} p_i(X)$ is a product of linear polynomials, then we have $s(\infty) = \prod_{i=1}^{d} p_i(\infty) = \prod_{i=1}^{d}(p_i(1) - p_i(0))$. More generally, when $s(X) = G(p_1(X), \ldots, p_d(X))$ for some polynomial G and linear polynomials $p_1, \ldots, p_d$, we can calculate $s(\infty)$ by evaluating *only* the terms of G of highest total degree.

Streaming model. For algorithms that rely on streaming access to data, we assume that the evaluations $p_1(x), \ldots, p_d(x)$ for $x \in \{0,1\}^\ell$ can be enumerated in linear time and small space, in lexicographic order from $x = 0^\ell$ to $x = 1^\ell$. This assumption holds in most SNARKs using the sum-check protocol, since $p_1, \ldots, p_d$ are typically the multilinear extensions of functions $f_1, \ldots, f_d \colon \{0,1\}^\ell \to \mathbb{F}$, whose evaluations over the Boolean hypercube can themselves be enumerated in linear time and small space. For example, in systems like Jolt, these functions are often derived from the execution trace of a virtual machine and can be enumerated simply by emulating the computation cycle by cycle.

3 Existing Algorithms for Sum-Check

3.1 Background on the sum-check protocol

In Figure 3, we describe the sum-check protocol that reduces checking a claim of the form

$$\sum_{x \in \{0,1\}^\ell} g(x) = C,$$

for a polynomial g of degree at most d in each variable, to checking the evaluation of g at a random point: $g(r) = v$.

Setup: Assume that the verifier has oracle access to the polynomial g, and knows the value C_0 claimed to equal

$$\sum_{x \in \{0,1\}^\ell} g(x) = C_0.$$

For each round $i = 1, \dots, \ell$:

1. $\mathcal{P}$ sends the univariate polynomial $s_i(X)$ claimed to equal

$$\sum_{(x_{i+1},\dots,x_\ell) \in \{0,1\}^{\ell-i}} g(r_1, \dots, r_{i-1}, X, x_{i+1}, \dots, x_\ell).$$

$\mathcal{P}$ does so by sending the evaluations $\{s_i(u) : u \in \widehat{U}_d\}$ to $\mathcal{V}$.

2. $\mathcal{V}$ sends a random challenge $r_i \leftarrow \mathbb{F}$ to $\mathcal{P}$.
3. $\mathcal{V}$ derives $s_i(1) := C_{i-1} - s_i(0)$, then sets $C_i := s_i(r_i)$.

After ℓ rounds, we reduce to the claim that

$$g(r_1, \dots, r_\ell) = C_\ell.$$

$\mathcal{V}$ checks this claim by making a single oracle query to g.

Fig. 3: The sum-check protocol

In each round j, the honest prover sends a univariate polynomial s_j of degree d. As any degree-d univariate polynomial is specified by its evaluations on any set of $d+1$ points, computing $s_j(c)$ for all $c \in \{\infty\} \cup \{0, 1, \dots, d-1\}$ suffices to uniquely specify s_j.[15] In fact, the prover can omit the evaluation $s_j(1)$, and instead let the verifier derive it from the previous round using the equation $s_j(1) = C_{j-1} - s_j(0)$, which holds for an honest protocol execution. This variant is the one we present in Figure 3.

Theorem 3.1. *The sum-check protocol applied to a polynomial g of individual degree at most d is perfectly complete and has soundness error of $d\ell/|\mathbb{F}|$.*

See, e.g., [45] for a proof of this standard result, attributed to Lund, Fortnow, Karloff, and Nisan [33].[16]

We recall two existing algorithms for the sum-check prover [18,44], with a focus on their cost profile in the small value setting. That is, we specialize to the case where the polynomial $g(x)$ is given as

$$g(X) = p_1(X) \cdot p_2(X) \cdots p_d(X), \tag{9}$$

where $p_1, \dots, p_d$ are d multilinear polynomials in ℓ variables such that $p_i(x)$ has small values for all $i \in \{1, \dots, d\}$ and $x \in \{0,1\}^\ell$.

3.2 Algorithm 1: Linear Time and Linear Space

In Algorithm 1 (see Figure 4), the prover maintains d arrays $P_1, \dots, P_d$, which initially store all evaluations of $p_1, \dots, p_d$ over $\{0,1\}^\ell$, e.g., we have $P_k[x] = p_k(x)$ for all $k \in [d]$ and $x \in \{0,1\}^\ell$. The prover halves the size of the arrays after each round i, via "binding" the arrays $\{P_k\}_{k\in[d]}$ to the challenge r_i received from the verifier.

[15] We assume a mapping from $\{0, 1, \dots, d-1\}$ to distinct field elements. In large prime fields, these integers are naturally identified with their corresponding field elements. In binary extension fields, any d distinct field elements can be chosen, but a natural choice is to use those whose binary representation (in the chosen basis) matches that of each integer in $0, 1, \dots, d-1$.

[16] Not only does the sum-check protocol have soundness error at most $d\ell/|\mathbb{F}|$, it satisfies a stronger property: it has *round-by-round soundness* at most $d/|\mathbb{F}|$ in each round. This ensures that the protocol remains secure (in the random oracle model) even after it is rendered non-interactive via the Fiat-Shamir transformation. See [14] for details.

```
Algorithm 1: LinearTime_SC
Initialize: Length-$2^\ell$ arrays $P_1^{(0)}, \dots, P_d^{(0)}$ such that
    $P_k^{(0)}[x] = p_k(x)$   for all $k = 1, \dots, d$ and $x \in \{0,1\}^\ell$.
For each round $i = 1, \dots, \ell$:
    1. For $u \in \widehat{U}_d$, compute $s_i(u)$ using the formula
        $s_i(u) := \sum_{x' \in \{0,1\}^{\ell-i}} \prod_{k=1}^{d} \left( \left( P_k^{(i)}[1,x'] - P_k^{(i)}[0,x'] \right) \cdot u + P_k^{(i)}[0,x'] \right)$
    2. Send $\left\{ s_i(u) : u \in \widehat{U}_d \right\}$ to the verifier.
    3. Receive challenge $r_i \in \mathbb{F}$ from the verifier.
    4. For $k = 1, \dots, d$ and $x' \in \{0,1\}^{\ell-i}$, update arrays:
        $P_k^{(i+1)}[x'] := \left( P_k^{(i)}[1,x'] - P_k^{(i)}[0,x'] \right) \cdot r_i + P_k^{(i)}[0,x']$.
// Invariant: $P_k^{(i)}[x'] = p_k(r_{[1:i]}, x')$   for all $k \in [d], i \in [0,\ell], x' \in \{0,1\}^{\ell-i}$.
```

Fig. 4: Linear time, linear space prover

Cost Analysis of Algorithm 1 In round 1, since the initial evaluations $\{p_k(x) : k \in [d], x \in \{0,1\}^\ell\}$ are small, all multiplications are $\mathfrak{ss}$. Since we need to compute d evaluations $\{s_1(u) : u \in \widehat{U}_d\}$, and each evaluation is a sum over $2^{\ell-1}$ products of d small values, the total cost is $d \cdot (d-1) \cdot 2^{\ell-1}$ $\mathfrak{ss}$ mults. We then take $d \cdot 2^{\ell-1}$ $\mathfrak{sl}$ multiplications for updating the arrays. From round $i \geq 2$, all array values are large, so we incur $d \cdot (d-1) \cdot 2^{\ell-i}$ $\mathfrak{ll}$ mults for computing all evaluations, and $d \cdot 2^{\ell-i}$ $\mathfrak{ll}$ mults for updating the arrays. Thus, we can summarize the cost of Algorithm 1 as follows.

Lemma 3.2. *When the evaluations of $p_1, \dots, p_d$ are small, Algorithm 1's dominant cost is $d^2 \cdot 2^{\ell-1}$ $\mathfrak{ll}$ multiplications, followed by $d \cdot 2^{\ell-1}$ $\mathfrak{sl}$ mults, $d(d-1) \cdot 2^{\ell-1}$ $\mathfrak{ss}$ mults, and $O(n)$ field additions.*

3.3 Algorithm 2: Quasilinear Time and Square-Root Space

Next, we describe Algorithm 2 (see Figure 5), a sum-check prover implementation with $O(\ell \cdot 2^\ell)$ time and $O(2^{\ell/2})$ space. While [17] presented an algorithm with better, logarithmic $O(\ell)$ space complexity, we opt to present this simpler, square-root space variant for two reasons. First, its structure facilitates the explanation of our optimizations later in the paper. Second, it leads to a faster prover in practice, with negligible increase in memory usage compared to the log-space approach, even for large instance sizes ($\ell \approx 30$).[17]

The main idea is to avoid updating the arrays $P_1, \dots, P_d$ in each round $i \leq \ell/2$, and instead compute $\prod_{k=1}^{d} p_k(r_1, \dots, r_{i-1}, u, x')$ directly from Equation (4). The cost of these rounds is dominated by two sources: (1) $d \cdot 2^\ell$ $\mathfrak{sl}$ mults per round to compute the product between the $\widetilde{\mathsf{eq}}$ terms and the evaluations $p_k(b, u, x')$ (counting over u), and (2) $(d-1) \cdot 2^{\ell-i}$ $\mathfrak{ll}$ multiplications for the product $\prod_{k=1}^{d}(\cdot)$. Besides these, we have lower-order costs of $d(d-1) \cdot 2^{\ell-1}$ $\mathfrak{ss}$ mults to compute the evaluations $\{p_k(b, u, x')\}_{k,b,u,x'}$, and 2^{i-1} $\mathfrak{ll}$ mults to compute the evaluations $\{\widetilde{\mathsf{eq}}(r_{[<i]}, b)\}_{b \in \{0,1\}^{i-1}}$ and store them in an array.

After $\ell/2$ rounds, as both the time and space needed for the $\widetilde{\mathsf{eq}}$ evaluations goes past $2^{\ell/2}$, we will switch to Algorithm 1, which incurs a negligible cost of $d^2 \cdot 2^{\ell/2}$ $\mathfrak{ll}$ mults. Note that in practice, one could switch as soon as the prover is no longer space-constrained, which may happen well before round $\ell/2$.

[17] For instance, when $\ell = 30$, square root space becomes $\approx 2^{15} = 32768$ field elements, and if the field size is ≈ 256 bits, this is only about 1MB.

Algorithm 2: $\mathsf{SqrtSpace}_{\mathsf{SC}}$

For each round $i = 1, \dots, \ell/2$**:**

1. **For** $u \in \widehat{U}_d$, compute $s_i(u)$ using the formula

$$s_i(u) = \sum_{x' \in \{0,1\}^{\ell-i}} \prod_{k=1}^{d} \left(\sum_{b \in \{0,1\}^{i-1}} \widetilde{\mathsf{eq}}(r_{[<i]}, b) \cdot p_k(b, u, x') \right), \tag{10}$$

where $\left\{\widetilde{\mathsf{eq}}(r_{[<i]}, b)\right\}_{b \in \{0,1\}^{i-1}}$ is computed using Procedure 2, and $\{p_k(b, u, x')\}_{k,b,u,x'}$ is computed using Procedure 5.

2. Send $\{s_i(u) : u \in \widehat{U}_d\}$ to the verifier.
3. Receive challenge $r_i \in \mathbb{F}$ from the verifier.

Store: the evaluations $p_k(r_{[<\ell/2]}, b, x')$ for all $k \in [d]$, $b \in \{0,1\}$, and $x' \in \{0,1\}^{\ell/2}$, obtained while computing round $\ell/2$.

For each round $i = \ell/2 + 1, \dots, \ell$**:** Follow Algorithm 1.

Fig. 5: Quasilinear time, square-root space prover

Lemma 3.3. *When the evaluations of $p_1, \dots, p_d$ are small, Algorithm 2's cost is dominated by $d \cdot \ell/2 \cdot 2^\ell$ $\mathfrak{sl}$ mults (spread over the first $\ell/2$ rounds), and $(d-1) \cdot 2^\ell$ $\mathfrak{ll}$ mults. The lower-order costs include $d(d-1) \cdot \ell/2 \cdot 2^{\ell-1}$ $\mathfrak{ss}$ mults and $O(n)$ field additions.*

4 First Optimization: Sum-Check for Polynomials with Small Values

In the linear-time prover implementation (Algorithm 1), the prover performs exclusively $\mathfrak{ll}$ multiplications starting from round 2, due to the need to bind the multilinear polynomials after round 1. If we want to reduce the number of $\mathfrak{ll}$ multiplications, we must delay this binding process. A natural attempt at this leads to Algorithm 3, which is typically faster than Algorithm 1 in the first few rounds, and uses *significantly* less memory. We then optimize this further via polynomial interpolation (i.e., techniques similar to the Toom-Cook algorithm), leading to Algorithm 4.

4.1 Algorithm 3: A Warm-up Attempt

We obtain Algorithm 3 (see Figure 6) by expanding the product in Equation (10), and re-arranging the terms to group together those with the same factors. In other words, we rewrite Equation (10) as

$$\begin{aligned}
&\sum_{x' \in \{0,1\}^{\ell-i}} \sum_{\boldsymbol{b}_1, \dots, \boldsymbol{b}_d \in \{0,1\}^{i-1}} \prod_{k=1}^{d} \left(\widetilde{\mathsf{eq}}(r_{[<i]}, \boldsymbol{b}_k) \cdot p_k(\boldsymbol{b}_k, u, x') \right) \\
&= \sum_{x'} \sum_{\boldsymbol{b}_1, \dots, \boldsymbol{b}_d} \prod_{k=1}^{d} \left(\prod_{j=1}^{i-1} \left((1-r_j)^{\boldsymbol{b}_k[j]} \cdot r_j^{1-\boldsymbol{b}_k[j]} \right) \cdot p_k(\boldsymbol{b}_k, u, x') \right) \\
&= \sum_{x'} \sum_{\boldsymbol{b}_1, \dots, \boldsymbol{b}_d} \prod_{j=1}^{i-1} (1-r_j)^{\sum_{k=1}^{d} \boldsymbol{b}_k[j]} r_j^{d - \sum_{k=1}^{d} \boldsymbol{b}_k[j]} \cdot \prod_{k=1}^{d} p_k(\boldsymbol{b}_k, u, x').
\end{aligned}$$

In the above, the first sum is obtained from Equation (10) by exchanging the product with the inner sum; the second sum is obtained by unfolding the definition of $\widetilde{\mathsf{eq}}$; the third sum is obtained by exchanging the two products; and the fourth sum is obtained by "pulling in" the sum over x', which do not affect the sum over $\{\boldsymbol{b}_k\}_{k \in [d]}$ and the product over $j \in [i-1]$.

Algorithm 3: $\mathsf{SmallValue\text{-}Unoptimized}_{\mathsf{SC}}$

Pre-computation: Use Procedure 4 to compute the accumulators

$$\mathsf{A}_i(\boldsymbol{v}, u) := \sum_{x' \in \{0,1\}^{\ell-i}} \sum_{\substack{\boldsymbol{b}_1,\ldots,\boldsymbol{b}_d \in \{0,1\}^{i-1} \\ \boldsymbol{v} = \sum_{k=1}^d \boldsymbol{b}}} \sum_{b_1^*,\ldots,b_d^* \in \{0,1\}} u^{\sum_{k=1}^d b_k^*} \cdot (1-u)^{d-\sum_{k=1}^d b_k^*} \cdot \prod_{k=1}^d p_k(\boldsymbol{b_k}, b_k^*, x'). \tag{11}$$

for all $i \in [\ell_0]$, $\boldsymbol{v} \in [0,d]^{i-1}$, and $u \in \widehat{U}_d$.

Initialize: $\mathsf{R}_1 = (1)$.

For each round $i = 1, \ldots, \ell_0$**:**

1. **For** $u \in \widehat{U}_d$, compute $s_i(u)$ via
$$s_i(u) := \sum_{\boldsymbol{v} \in [0,d]^{i-1}} \mathsf{R}_i[\mathsf{nat}_{d+1}(\boldsymbol{v})] \cdot \mathsf{A}_i(\boldsymbol{v}, u).$$
2. Send $\left\{ s_i(u) : u \in \widehat{U}_d \right\}$ to the verifier.
3. Receive challenge $r_i \in \mathbb{F}$ from the verifier.
4. Compute $\mathsf{R}_{i+1} := \mathsf{R}_i \otimes \left((1-r_i)^k \cdot r_i^{d-k}\right)_{k=0}^d$.

For round $\ell_0 + 1$**:** Follow Algorithm 2, storing the evaluations $p_k(r_{[1:\ell_0-1]}, b, x')$ for all $k \in [d]$, $b \in \{0,1\}$, and $x' \in \{0,1\}^{\ell-\ell_0}$.

Receive challenge $r_{\ell_0+1} \in \mathbb{F}$ from the verifier, and perform updates to get $p_k(r_{[1:\ell_0]}, x')$ for all k and x', similar to Algorithm 1.

For each round $i = \ell_0 + 2, \ldots, \ell$**:** Follow Algorithm 1.

Fig. 6: First attempt to reduce $\mathfrak{ll}$ multiplications

At this point, we have rewritten the computation of $s_i(u)$ as an inner product between two vectors of length $2^{d(i-1)}$ (indexed by $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_d$): one dependent on the challenges $r_1, \ldots, r_{i-1}$, and the other dependent on the multilinear polynomials $p_1, \ldots, p_d$. We can do slightly better by noticing that the challenge-dependent terms only depend on $\boldsymbol{v} := \left(\sum_{k=1}^d \boldsymbol{b}_k[j]\right)_{j \in [i-1]}$ and not on the individual $\{\boldsymbol{b}_k\}_{k \in [d]}$—there are only $(d+1)^{i-1}$ distinct such terms. By re-indexing based on $\boldsymbol{v} \in [0,d]^{i-1}$, we can further rewrite

$$s_i(u) = \sum_{\boldsymbol{v} \in [0,d]^{i-1}} \prod_{j=1}^{i-1} (1-r_j)^{\boldsymbol{v}[j]} r_j^{d-\boldsymbol{v}[j]} \cdot \sum_{x' \in \{0,1\}^{\ell-i}} \sum_{\substack{\boldsymbol{b}_1,\ldots,\boldsymbol{b}_d \in \{0,1\}^{i-1} \\ \boldsymbol{v} = \sum_{k=1}^d \boldsymbol{b}_k}} \prod_{k=1}^d p_k(\boldsymbol{b}_k, u, x'),$$

which is equal to the inner product

$$\left\langle \bigotimes_{j=1}^{i-1} \left((1-r_j)^{v_j} \cdot r_j^{d-v_j}\right)_{v_j=0}^d \, , \, (\mathsf{A}_i(\boldsymbol{v}, u))_{\boldsymbol{v} \in [0,d]^{i-1}} \right\rangle.$$

Since the accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ do *not* depend on the challenges $r_1, \ldots, r_{i-1}$, the prover can precompute them before the protocol begins. The final change needed to obtain the pre-computation step in Figure 6 is to notice that, when computing the accumulators $\mathsf{A}_1(\boldsymbol{v}, u), \ldots, \mathsf{A}_{\ell_0}(\boldsymbol{v}, u)$ over the first ℓ_0 rounds, we

```
Algorithm 4: SmallValue_SC
Pre-computation: Use Procedure 7 to compute the accumulators
    $\mathsf{A}_i(\boldsymbol{v}, u) := \sum_{x' \in \{0,1\}^{\ell-i}} \prod_{k=1}^{d} p_k(\boldsymbol{v}, u, x')$
for all $i \in [\ell_0]$, $\boldsymbol{v} \in {U_d}^{i-1}$, and $u \in \widehat{U}_d$.
Initialize: $\mathsf{R}_1 = (1)$.
For each round $i = 1, \ldots, \ell_0$:
    1. For $u \in \widehat{U}_d$, compute $s_i(u)$ via
           $s_i(u) := \sum_{\boldsymbol{v} \in [0,d]^{i-1}} \mathsf{R}_i[\mathsf{nat}_{d+1}(\boldsymbol{v})] \cdot \mathsf{A}_i(\boldsymbol{v}, u)$.
    2. Send $\left\{ s_i(u) : u \in \widehat{U}_d \right\}$ to the verifier.
    3. Receive challenge $r_i \in \mathbb{F}$ from the verifier.
    4. Compute $\mathsf{R}_{i+1} := \mathsf{R}_i \otimes (\mathcal{L}_{U_d,k}(r_i))_{k=0}^{d}$.
For round $\ell_0 + 1$: Follow Algorithm 2, storing the evaluations $p_k(r_{[1:\ell_0-1]}, b, x')$ for all $k \in [d]$, $b \in \{0,1\}$, and $x' \in \{0,1\}^{\ell-\ell_0}$.
Receive challenge $r_{\ell_0+1} \in \mathbb{F}$ from the verifier, and perform updates to get $p_k(r_{[1:\ell_0]}, x')$ for all $k$ and $x'$, similar to Algorithm 1.
For each round $i = \ell_0 + 2, \ldots, \ell$: Follow Algorithm 1.
```

Fig. 7: Small-value sum-check prover. This improves upon Algorithm 3 by using ideas from Toom-Cook multiplication.

actually repeat a lot of the same computations for the product of the evaluations $\{p_k(\boldsymbol{b}_1, u, x')\}_{k \in [d]}$. For instance, when $d = \ell_0 = 2$, we compute the product $p_1(0,0,x') \cdot p_2(0,0,x')$ in both $\mathsf{A}_1(0)$ and $\mathsf{A}_2(0)$. To save on unnecessary computations, we can instead compute each product $\prod_k p_k(\boldsymbol{b}_k, u, x')$ once, then insert the result into the appropriate accumulators over all ℓ_0 rounds. We work out the details of this process in Section A.2.

Cost Analysis. Algorithm 3 uses $O((d+1)^{\ell_0})$ space for rounds 1 to ℓ_0 (accumulators and challenge vectors) and $d \cdot 2^{\ell-\ell_0}$ space thereafter (cached results). Since $(d+1)^{\ell_0} \ll d \cdot 2^{\ell-\ell_0}$ in practice, the overall space complexity is dominated by the latter term, representing a space saving factor of roughly 2^{ℓ_0} compared to Algorithm 1.

Regarding runtime, Algorithm 3 is bottlenecked by the $(d-1) \cdot 2^{d \cdot \ell_0} \cdot 2^{\ell-\ell_0} = (d-1) \cdot 2^{\ell+(d-1)\ell_0}$ $\mathfrak{ss}$ mults needed to compute the products $\{\prod_{k=1}^{d} p_k(\boldsymbol{b}_k, x')\}_{\boldsymbol{b},x'}$ that go into the accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ in Equation (11). We also incur roughly $d \cdot 2^{\ell}$ $\mathfrak{sl}$ mults when using Algorithm 2 for the (ℓ_0+1)-th round, and roughly $d^2 \cdot 2^{\ell-\ell_0-1}$ $\mathfrak{ll}$ mults for the remaining rounds using Algorithm 1.

4.2 Algorithm 4: Optimizing with Toom-Cook Multiplication

While Algorithm 3 almost eliminates $\mathfrak{ll}$ multiplications for the first few rounds and slashes memory usage, we still end up performing a lot of $\mathfrak{ss}$ multiplications (namely, by a factor of $\approx 2^{(d-1)\cdot\ell_0}$ compared to the number of $\mathfrak{ll}$ mults in Algorithm 1). To reduce this cost, we need another way to rewrite the products $\prod_{k=1}^{d} p_k(r_{[<i]}, u, x')$.

The idea is to see the product as the evaluation of the polynomial

$$F(Y_1, \dots, Y_{i-1}) = \prod_{k=1}^{d} p_k(Y_1, \dots, Y_{i-1}, u, x') \tag{12}$$

at the challenge point $(r_1, \dots, r_{i-1})$. Since F has individual degree d, we can apply Lagrange interpolation to each variable to rewrite $F(Y_1, \dots, Y_{i-1})$ as

$$\sum_{v_1 \in U_d} \mathcal{L}_{U_d, v_1}(Y_1) \cdots \sum_{v_{i-1} \in U_d} \mathcal{L}_{U_d, v_{i-1}}(Y_{i-1}) \prod_{k=1}^{d} p_k(\boldsymbol{v}, u, x') \tag{13}$$

$$= \sum_{\boldsymbol{v} \in {U_d}^{i-1}} \prod_{j=1}^{i-1} \mathcal{L}_{U_d, \boldsymbol{v}_j}(Y_j) \cdot \prod_{k=1}^{d} p_k(\boldsymbol{v}, u, x'). \tag{14}$$

As a result, when we plug in the evaluation $(r_1, \dots, r_{i-1})$, we can rewrite the i-th sum-check polynomial $s_i(X)$ as

$$\begin{aligned} s_i(X) &= \sum_{x' \in \{0,1\}^{\ell-i}} \sum_{\boldsymbol{v} \in {U_d}^{i-1}} \prod_{j=1}^{i-1} \mathcal{L}_{U_d, \boldsymbol{v}_j}(r_j) \cdot \prod_{k=1}^{d} p_k(\boldsymbol{v}, u, x') \\ &= \sum_{\boldsymbol{v} \in {U_d}^{i-1}} \prod_{j=1}^{i-1} \mathcal{L}_{U_d, \boldsymbol{v}_j}(r_j) \cdot \sum_{x' \in \{0,1\}^{\ell-i}} \prod_{k=1}^{d} p_k(\boldsymbol{v}, u, x'), \end{aligned}$$

which is equal to the inner product

$$= \left\langle \bigotimes_{j=1}^{i-1} \left(\mathcal{L}_{U_d, v_j}(r_j)\right)_{v_j \in U_d} \,,\, \left(\mathsf{A}_i(\boldsymbol{v}, u)\right)_{\boldsymbol{v} \in {U_d}^{i-1}} \right\rangle.$$

Just like with Algorithm 3, we can precompute the accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ by computing each of the $\approx (d+1)^{\ell_0}$ product

$$\left\{ \prod_{k=1}^{d} p_k(\boldsymbol{v}, u, x') : \boldsymbol{v} \in {U_d}^{i-1}, u \in \widehat{U}_d, x' \in \{0,1\}^{\ell-i} \right\}$$

once, then inserting the result into the appropriate $\mathsf{A}_i(\boldsymbol{v}, u)$ terms over all $i \in [\ell_0]$ rounds. For details, see Section A.3.

Cost Analysis. Similar to Algorithm 3, Algorithm 4 also saves a factor of 2^{ℓ_0} in space. Unlike Algorithm 3, however, it incurs only $(d-1) \cdot (d+1)^{\ell_0} \cdot 2^{\ell-\ell_0}$ $\mathfrak{ss}$ mults, instead of $(d-1) \cdot 2^{d \cdot \ell_0} \cdot 2^{\ell-\ell_0}$. This factor can be roughly 5 times smaller for common parameters (e.g. $d = 2$ and $\ell_0 = 5$). The cost of $\mathfrak{sl}$ and $\mathfrak{ll}$ mults are the same as in Algorithm 3.

Theorem 4.1. *Algorithm 4's cost is dominated by* $(d-1)((d+1)/2)^{\ell_0} \cdot N$ $\mathfrak{ss}$ *mults, followed by* $d \cdot N$ $\mathfrak{sl}$ *mults and* $d^2/2^{\ell_0+1} \cdot N$ $\mathfrak{ll}$ *mults.*

Optimal Switchover Point. Determining the best switchover point ℓ_0 is dependent on the cost analysis we presented above. Essentially, we want to find the point ℓ_0 that makes the cost equal between the $\mathfrak{ss}$ and $\mathfrak{ll}$ mults. If we denote κ as the factor difference between $\mathfrak{ss}$ and $\mathfrak{ll}$ mults, then we want to find ℓ_0 that satisfies

$$(d-1)((d+1)/2)^{\ell_0} \cdot N = \kappa \cdot d^2/2^{\ell_0+1} \cdot N,$$

which simplifies to

$$(d-1)(d+1)^{\ell_0} = \kappa \cdot d^2/2 \iff \ell_0 = \frac{\log\left(\frac{\kappa \cdot d^2}{2(d-1)}\right)}{\log(d+1)}.$$

Given this optimal choice of ℓ_0, we get a speedup factor of $\approx 2^{\ell_0}$ compared to Algorithm 1. This matters both *asymptotically* and for practical parameters.

In terms of asymptotics, we can see that $2^{\ell_0} \approx \kappa^{1/\log(d+1)}$ for constant d; concretely, for $d = 2$ we get a $\kappa^{0.63}$ speedup, while for $d = 3$ we get a $\kappa^{0.5}$ speedup.

Concretely, if we work over elliptic curve scalar fields, then we have $\kappa \approx 2N^2 \approx 30$ (for u64 mults versus BN254 mults). If we further set $d = 2$, then we get $\ell_0 \approx 3.72$ and a speedup of about 4 times. We refer to Section 8 for concrete numbers which fall short of this estimate, due to our implementation not yet having incorporated all available optimizations.

5 Second Optimization: Sum-Check involving Equality Polynomials

Our second optimization applies to the setting

$$g(X) = \widetilde{\mathsf{eq}}(w, X) \cdot \prod_{k=1}^{d} p_k(X), \tag{15}$$

with $w \in \mathbb{F}^\ell$ a vector of verifier's challenges, and $p_1(X), \ldots, p_d(X)$ are multilinear polynomials. As our algorithm builds on top of another optimization by Gruen [28], we first present Gruen's optimization, followed by our contribution.

5.1 Recap: Existing Optimization from Gruen

The main idea of this optimization by Gruen [28] (and a variant thereof in [40]) is to reduce the computation of the degree-$(d+1)$ polynomial $s_i(X)$ sent by the prover in each round i, to the computation of a degree-d factor $t_i(X)$ and a linear factor $\mathsf{l}_i(X)$. This way, the linear term is "free" to compute, and the prover saves an evaluation when computing $t_i(X)$ instead of $s_i(X)$.

This is possible due to the special structure of the $\widetilde{\mathsf{eq}}$ polynomial, which for every $i \in [\ell]$ can be decomposed as

$$\widetilde{\mathsf{eq}}(w, (r_{[<i]}, X, x')) = \widetilde{\mathsf{eq}}(w_{[<i]}, r_{[<i]}) \cdot \widetilde{\mathsf{eq}}(w_i, X) \cdot \widetilde{\mathsf{eq}}(w_{[>i]}, x').$$

Thus, if we set $\mathsf{l}_i(X) := \widetilde{\mathsf{eq}}(w_{[<i]}, r_{[<i]}) \cdot \widetilde{\mathsf{eq}}(w_i, X)$ and

$$t_i(X) := \sum_{x' \in \{0,1\}^{\ell-i}} \widetilde{\mathsf{eq}}(w_{[>i]}, x') \cdot \prod_{k=1}^{d} p_k(r_{[<i]}, X, x'),$$

then we have $s_i(X) = \mathsf{l}_i(X) \cdot t_i(X)$.[18] The sum-check prover now proceeds to compute d evaluations of $t_i(u)$ at $u \in \widehat{U}_d$, then rely on the equality $s_i(0) + s_i(1) = C_i$ to compute

$$t_i(1) = \mathsf{l}_i(1)^{-1} \cdot (C_i - \mathsf{l}_i(0) \cdot t_i(0)). \tag{16}$$

The prover now has $d+1$ evaluations of t_i, and thus can interpolate $t_i(X)$ and compute $s_i(X)$. Note that computing $\{t_i(u) : u \in \widehat{U}_d\}$ can be done generically using either Algorithm 1 or Algorithm 2; since our focus is on optimizing time rather than space, we assume that Algorithm 1 is used.

[18] Gruen's writeup computes and sends t_i directly to the verifier, which modifies the sum-check protocol itself and increases the verifier's costs. Setty and Thaler [40] suggest instead computing t_i and then quickly deriving s_i from it, and only sending s_i to the verifier. This ensures that the sum-check verifier is unchanged, and we follow this approach.

Algorithm 5: $\mathsf{EqPoly}_{\mathsf{SC}}$

Initialize: Length-2^ℓ arrays $P_1^{(0)}, \dots, P_d^{(0)}$ such that

$$P_k^{(0)}[x] = p_k(x) \quad \text{for all } k = 1, \dots, d \text{ and } x \in \{0,1\}^\ell.$$

Pre-computation: Use Procedure 3 to compute the evaluations

$$\left\{\widetilde{\mathsf{eq}}(w_{[1:i]}, x_L), \quad \widetilde{\mathsf{eq}}(w_{[\ell/2:\ell/2+i]}, x_R)\right\}$$

for all $i \in [\ell/2]$, $x_L \in \{0,1\}^{\ell/2-i}$, and $x_R \in \{0,1\}^{\ell/2-i}$.

For each round $i = 1, \dots, \ell$:

1. **For** $u \in \widehat{U}_d$, if $i < \ell/2$, compute $t_i(u)$ using the formula

$$\sum_{x_R \in \{0,1\}^{\ell/2}} \widetilde{\mathsf{eq}}(w_R, x_R) \cdot \sum_{x_L \in \{0,1\}^{\ell/2-i}} \widetilde{\mathsf{eq}}(w_L, x_L) \cdot \prod_{k=1}^{d} \left(\left(P_k^{(i)}[1, x_L, x_R] - P_k^{(i)}[0, x_L, x_R] \right) \cdot u + P_k^{(i)}[0, x_L, x_R] \right).$$

If $i \geq \ell/2$, compute $t_i(u)$ using the formula

$$\sum_{x_R \in \{0,1\}^{\ell/2-i}} \widetilde{\mathsf{eq}}(w_R, x_R) \cdot \prod_{k=1}^{d} \left(\left(P_k^{(i)}[1, x_R] - P_k^{(i)}[0, x_R] \right) \cdot u + P_k^{(i)}[0, x_R] \right).$$

2. Use Procedure 8 to compute $s_i(X)$ from $\left(\{t_i(u)\}, w, r_{[<i]}\right)$.
3. Send $\left\{s_i(u) : u \in \widehat{U}_d\right\}$ to the verifier.
4. Receive challenge $r_i \in \mathbb{F}$ from the verifier.
5. **For** $k \in [d]$ and $x' \in \{0,1\}^{\ell-i}$, update arrays:

$$P_k^{(i+1)}[x'] := \left(P_k^{(i)}[1, x'] - P_k^{(i)}[0, x'] \right) \cdot r_i + P_k^{(i)}[0, x'].$$

Fig. 8: Eq-poly sum-check prover, including Gruen's optimization, and using Algorithm 1 for the non-eq terms

5.2 Algorithm 5: Additional Speedup via Splitting Eq-Poly

We now present our optimization (see Figure 8), which leads to a further reduction of roughly $2^{\ell-1}$ $\mathfrak{ll}$ multiplications associated with the eq polynomial. Our insight is that we can further split off the $\widetilde{\mathsf{eq}}$ terms in Equation (16) for the first $i < \ell/2$ rounds, so that

$$t_i(u) = \sum_{x_R \in \{0,1\}^{\ell/2}} \widetilde{\mathsf{eq}}(w_R, x_R) \cdot \sum_{x_L \in \{0,1\}^{\ell/2-i}} \widetilde{\mathsf{eq}}(w_L, x_L) \cdot \tag{17}$$

$$\prod_{k=1}^{d} p_k(r_{[<i]}, u, x_L, x_R). \tag{18}$$

for all $u \in \widehat{U}_d$, where $w_R = w_{[i+1:\ell/2]}$ and $w_L = w_{[\ell/2+1:n]}$. The sum-check prover now computes all inner sums over x_L, then all outer sums over x_R. For rounds $i \geq \ell/2$, we no longer have the inner sum, and proceed similarly to Algorithm 1. Note that we put the left part $\widetilde{\mathsf{eq}}(w_L, x_L)$ in the inner sum, since it leads to better memory locality in practice, assuming the evaluations of p_k are streamed in big-endian order (so that x_L are low-order bits compared to x_R, which means their chunks are contiguous in memory).

```
Algorithm 6: EqPoly-SmallValue_SC
Pre-computation: Use Procedure 9 to compute the accumulators
    $\mathsf{A}_i(\boldsymbol{v}, u) := \sum_{x_R \in \{0,1\}^{\ell/2-i}} \widetilde{\mathsf{eq}}(w_R, x_R) \sum_{x_L \in \{0,1\}^{\ell/2}} \widetilde{\mathsf{eq}}(w_L, x_L) \cdot \prod_{k=1}^{d} p_k(\boldsymbol{v}, u, x_L, x_R)$
for all $i \in [\ell_0]$, $\boldsymbol{v} \in {U_d}^{i-1}$, and $u \in \widehat{U}_d$.
Initialize: $\mathsf{R}_1 = (1)$.
For each round $i = 1, \ldots, \ell_0$:
    1. For $u \in \widehat{U}_d$, compute $t_i(u)$ via
        $t_i(u) := \sum_{\boldsymbol{v} \in [0,d]^{i-1}} \mathsf{R}_i[\mathsf{nat}_{d+1}(\boldsymbol{v})] \cdot \mathsf{A}_i(\boldsymbol{v}, u)$.
    2. Use Procedure 8 to compute $s_j(X)$ from $\big(\{t_j(u)\}, w, r_{[<i]}\big)$.
    3. Send $\Big\{s_j(u) : u \in \widehat{U}_d\Big\}$ to the verifier.
    4. Receive challenge $r_j \in \mathbb{F}$ from the verifier.
    5. Compute $\mathsf{R}_{i+1} := \mathsf{R}_i \otimes (\mathcal{L}_{U_d,k}(r_j))_{k=0}^{d}$.
For round $\ell_0 + 1$: Follow Algorithm 2, storing the evaluations $p_k(r_{[1:\ell_0-1]}, b, x')$ for all $k \in [d]$, $b \in \{0,1\}$, and $x' \in \{0,1\}^{\ell-\ell_0}$.
Receive challenge $r_{\ell_0+1} \in \mathbb{F}$ from the verifier, and perform updates to get $p_k(r_{[1:\ell_0]}, x')$ for all $k$ and $x'$, similar to Algorithm 1.
For rounds $\ell_0 + 2, \ldots, \ell$: Follow Algorithm 5.
```

Fig. 9: Combined small-value and equality-polynomial sum-check optimizations

Cost Analysis. Our optimization avoids computing the $2^{\ell-i}$-sized table of evaluations $\{\widetilde{\mathsf{eq}}(w_{[>i]}, x') : x' \in \{0,1\}^{\ell-i}\}$ for each round i, and instead only compute two square-root-sized tables, which costs $2^{\ell/2}$ $\mathfrak{ll}$ mults for $\widetilde{\mathsf{eq}}(w_R, x_R)$ and $2^{\ell/2-i}$ $\mathfrak{ll}$ mults for $\widetilde{\mathsf{eq}}(w_L, x_L)$. In Procedure 3, we show that with memoization, these evaluations can be computed for all rounds i in $2^{\ell/2}$ $\mathfrak{ll}$ mults *total*. The (negligible) downside is that we need to compute an extra $2^{\ell/2}$ $\mathfrak{ll}$ mults when computing the outer sum over x_R.

Using the same process for cost analysis as in Lemma 3.2, we have the following cost breakdown for Algorithm 5.

Lemma 5.1. *Algorithm 5's dominant cost is $d(d+1)/2 \cdot N\mathfrak{ll}$ mults, reducing from Algorithm 1's cost in the same setting by roughly N $\mathfrak{ll}$ mults.*

6 Combined Optimizations

In this section, we present Algorithm 6, our optimization in the combined eq-poly and small value settings. The presence of the extra eq-poly term means that Algorithm 6 is less efficient than Algorithm 4, which is in the simpler setting (without eq-poly). We then present details of how Algorithm 6 specializes to the case of Spartan-in-Jolt.

6.1 Algorithm 6: Combined Eq-Poly and Small Value Optimization

Our starting point is Equation (17) in the exposition of Algorithm 5. We then apply the multivariate Lagrange decomposition formula in Equation (14) to the evaluations $\prod_{k=1}^{d} p_k(r, u, x_L, x_R)$, which gives

$$\begin{aligned} t_i(u) &= \sum_{x_R \in \{0,1\}^{\ell/2-i}} \widetilde{\mathsf{eq}}(w_R, x_R) \cdot \sum_{x_L \in \{0,1\}^{\ell/2}} \widetilde{\mathsf{eq}}(w_L, x_L) \cdot \\ &\qquad \sum_{\boldsymbol{v} \in {U_d}^{i-1}} \prod_{j=1}^{i-1} \mathcal{L}_{U_d, \boldsymbol{v}_j}(r_j) \cdot \prod_{k=1}^{d} p_k(\boldsymbol{v}, u, x_L, x_R) \\ &= \left\langle \bigotimes_{j=1}^{i-1} \left(\mathcal{L}_{U_d, v}(r_j)\right)_{v \in U_d} \, , \, \mathsf{A}_i(u) \right\rangle, \end{aligned}$$

where the accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ are now of the form

$$\mathsf{A}_i(\boldsymbol{v}, u) = \sum_{x_R \in \{0,1\}^{\ell/2-i}} \widetilde{\mathsf{eq}}(w_R, x_R) \sum_{x_L \in \{0,1\}^{\ell/2}} \widetilde{\mathsf{eq}}(w_L, x_L) \prod_{k=1}^{d} p_k(\boldsymbol{v}, u, x_L, x_R).$$

Unlike the small-value only case, here because of the presence of the $\widetilde{\mathsf{eq}}(w, \cdot)$ terms, we need to use $\mathfrak{sl}$ (instead of $\mathfrak{ss}$) multiplications in computing the accumulators, and the additions are over the big field as well. We also note that we have swapped the lengths of the vectors x_R and x_L in the accumulator formula. This is so that we can compute the *same* inner sum across the accumulators over all rounds, and we can dispatch these inner sums to different accumulators in different rounds afterwards, with negligible cost. See Section A.4 for more details.

Cost Analysis. Besides the extra subtleties as noted above, the cost estimate of Algorithm 6 is fairly straightforward and follows directly from combining the estimates of Algorithm 4 and Algorithm 5. In particular, the small-value precomputation process is now dominated by $(d+1)^{\ell_0} \cdot 2^{\ell-\ell_0}$ $\mathfrak{sl}$ mults. Next, the streaming round takes roughly $d \cdot 2^{\ell}$ $\mathfrak{sl}$ mults, and the rest of the linear-time sumcheck rounds cost roughly $d^2 \cdot 2^{\ell-\ell_0-1}$ $\mathfrak{ll}$ mults. We thus summarize as follows.

Theorem 6.1. *Algorithm 6's cost is dominated by* $\left(\left(\frac{d+1}{2}\right)^{\ell_0} + d\right) N$ $\mathfrak{sl}$ *mults and* $d^2/2^{\ell_0+1} N$ $\mathfrak{ll}$ *mults.*

Optimal Switchover Point. Our analysis here is similar to that of Section 4.2, except for the fact that the relevant ratio κ is now between $\mathfrak{sl}$ and $\mathfrak{ll}$ mults. This ratio is about square-root worse than the ratio between $\mathfrak{ss}$ and $\mathfrak{ll}$ mults (refer to Section 2.1 for details), making our asymptotic speedups a square-root factor worse. Our concrete speedups are also more modest, as can be seen in Section 8.

6.2 Improving Spartan-in-Jolt

Recall that the first sum-check in Spartan involves proving $\sum_{x \in \{0,1\}^{\ell}} G(x) = 0$ for

$$G(X) = \widetilde{\mathsf{eq}}(w, X) \cdot \left(\widetilde{Az}(X) \cdot \widetilde{Bz}(X) - \widetilde{Cz}(X)\right), \tag{19}$$

where $\widetilde{Az}(X), \widetilde{Bz}(X), \widetilde{Cz}(X)$ are multilinear extensions of the R1CS vectors $A \cdot Z$, $B \cdot Z$, and $C \cdot Z$. Our combined Algorithm 6 can be specialized to this setting. In particular, we make two key observations that drastically simplify the precomputation phase (e.g., Procedure 9):

- **Binary points yield zero:** If an accumulator $\mathsf{A}_i(v, u)$ corresponds to an evaluation point (v, u) consisting entirely of binary values (0s and 1s), then $\mathsf{A}_i(v, u) = 0$. This follows directly from the R1CS relation, which implies that $Az(x) \cdot Bz(x) = Cz(x)$ for all $x \in \{0,1\}^n$, and the formula for $\mathsf{A}_i(v, u)$.

– **Linear term vanishes for ∞ points:** If the evaluation point (v, u) contains at least one ∞ coordinate, then the linear term $Cz(X)$ does not contribute to the accumulator $\mathsf{A}_i(v, u)$. This is because the evaluation at ∞ only considers the highest-degree term, which precludes the linear term $\widetilde{Cz}(X)$.

These observations allow for significant optimization. For instance, setting $\ell_0 = 3$, the precomputation only needs to explicitly calculate the 19 non-zero accumulator terms for round 3 (as detailed in Appendix A.4), which are derived solely from combinations of Az and Bz evaluations. The accumulators for rounds 1 and 2 can be reconstructed from these.

Cost Analysis. By specializing our cost analysis of Algorithm 6 to the case of Spartan-in-Jolt, taking into account the optimizations above, we get the following cost breakdown (with details of our derivation in Section B.2):

– The optimal number of small-value rounds for Spartan-in-Jolt is 3. With this configuration, we incur roughly 4.5 N $\mathfrak{sl}$ mults and $N/4$ $\mathfrak{ll}$ mults.[19]

– In contrast, for Algorithm 5 with the same Spartan-specific optimizations, we incur roughly 3.5 N $\mathfrak{ll}$ mults.

We stress that these numbers are rough estimates. In practice (as with our implementation), we need to take into account the relative *sparsity* of the R1CS vectors Az, Bz, and Cz, and their expected bit-lengths.[20] We also do not count the cost of building the Az, Bz, and Cz evaluations from the program trace, nor any potential overhead related to data handling and parallelization.

7 Optimized Field Multiplication with Small Integers

We present an optimized algorithm for multiplying an element in a large prime field, such as BN254, with a 64-bit unsigned integer. The element is assumed to be in Montgomery form, and the multiplication result is also in Montgomery form. Our algorithm incurs only $2N+1$ base (u64) multiplications, compared to $2N^2+1$ for the full Montgomery multiplication, with practical speedup of $\sim 2.4\times$ to $\sim 3\times$ in micro-benchmarks, depending on the $\sim$256-bit field modulus.

We first give a brief recap of Montgomery multiplication, and then present our optimized algorithm.

7.1 Recap: Montgomery Multiplication

Assume we work over a prime field $\mathbb{F}_p$ that is represented as N limbs of 64-bit unsigned integers (common values are $N = 4$ for $\sim$ 256-bit fields, or $N = 6$ for $\sim$ 384-bit fields). Assume further that elements in the prime field are represented in Montgomery form, namely that an element a is represented as $aR \mod p$ for $R = 2^{64N}$. In this setting, Montgomery [34] presented an algorithm that allows for faster multiplication of field elements. This multiplication happens in two stages:[21]

1. **Bignum Multiplication.** Given $a' = aR \mod p$ and $b' = bR \mod p$, consider them as big unsigned integers, and perform an integer multiplication, giving a $2N$-limb bignum $c = a' \cdot b'$. This costs around N^2 base (u64) multiplications.
2. **Montgomery Reduction.** Given a $2N$-limb bignum $c \mod p$, compute $cR^{-1} \mod p$, which would be the Montgomery form $abR \mod p$ of the product $a \cdot b$. This costs around $N^2 + 1$ base multiplications.[22]

[19] Here, $N = 2^\ell$ is the total number of constraints in the R1CS instance, which is itself the product of the (padded) number of constraints per cycle, and the (padded) number of cycles in a program.

[20] In more detail, the coefficients are only guaranteed to fit inside a signed 128-bit integer in the worst case. However, this does not apply to most of the coefficients, which are much smaller, with most in $\{0, 1\}$.

[21] Some implementations intersperse these two stages for improved efficiency (i.e. finely-integrated operand scanning, or FIOS). We present them separately for clarity.

[22] Traditionally, the cost is $N^2 + N$ multiplications, but a recent blog post [23] shows a reduction to $N^2 + 1$ multiplications.

Procedure 1 Optimized multiplication of a field element (in Montgomery form) by a 64-bit unsigned integer

```
Input:
  • $r = 2^{64}$ and $R = 2^{64N}$
  • Field element $a' = aR \mod p$ as $N$ limbs $a'_0, \dots, a'_{N-1} \in \mathbb{Z}_{2^{64}}$.
  • Integer $b \in \mathbb{Z}_{2^{64}}$.
  • Modulus $p$ as $N$ limbs $p_0, \dots, p_{N-1}$.
  • Precomputed Barrett reduction constant $\mu$.
Output: Product $c' = (a' \cdot b \mod p)$, equivalent to $abR \mod p$.
                                              ▷ Stage 0: Precomputation (once and for all)
1: Compute $R' = 2^{n_p}$, where $n_p = \lfloor \log_2 p \rfloor + 1$
2: Compute $\mu = \lfloor rR'/(2p) \rfloor \in \mathbb{Z}_{2^{64}}$
                                              ▷ Stage 1: Bignum Multiplication
3: Compute $c = a' \cdot b$ as a $(N+1)$-limb integer. Let $\tilde{c} = \lfloor c/R' \rfloor$.
                                              ▷ Stage 2: Barrett Reduction (optimized for $(N+1)$-limb input)
4: Compute $m := \lfloor (\tilde{c} \cdot \mu)/r \rfloor$.          ▷ $m \in \{q-1, q\}$, where $q = \lfloor c/(2p) \rfloor$
5: Compute $r := c - m \cdot 2p$.          ▷ Result $r < 4p$ is $c \pmod{2p}$ or $c \pmod{2p} + 2p$
6: Find $k \in \{0,1,2,3\}$ such that $kp \le r < (k+1)p$, and compute $c' = r - kp$.
                                              ▷ Result $c'$ is $c \pmod p$
7: return $c'$
```

Altogether, Montgomery multiplication costs around $2N^2 + 1$ base multiplications, and other operations (e.g. base addition & subtraction) of second-order complexity.

7.2 Optimizing Multiplication by u64

While the above algorithm works best when multiplying two field elements in Montgomery form, we need a different approach for multiplying a field element (in Montgomery form as $aR \mod p$) with a raw unsigned 64-bit number b. The first stage, bignum multiplication, is the same as above, and only incurs N base multiplications due to the single-limb input b. However, we do *not* want to apply Montgomery reduction on the result, as this would lead to the "wrong" answer $ab \mod p$ that is not in Montgomery form, and is also expensive as the reduction algorithm does not take advantage of the fact that b occupies a single limb.

Instead, we will apply an optimized variant of *Barrett reduction* [9], which also performs a reduction modulo p, but do *not* divide by R at the same time. Barrett reduction typically performs multiple passes over the $2N$ limbs of the bignum multiplication, each time reducing the largest limb modulo p; the whole process takes $N^2 + 1$ base multiplications, similar to Montgomery reduction. However, since in our setting, we only get $N + 1$ limbs for the multiplication result, it suffices to do a *single* pass to reduce the highest limb, costing $N + 1$ multiplications.

More details. Our algorithm is described in Procedure 1, and here we give some intuition for the Barrett reduction step.

In this step, as c has $N + 1$ limbs, we may write it as $c = c_N r^N + c_{N-1} r^{N-1} + \cdots + c_0$. We will compute $c \mod kp$ for some multiple k, which will then imply $c \mod p$ by a conditional subtraction (by the right multiple of $0, p, \dots, (k-1)p$). We get this quantity by finding the quotient m of c when divided by kp, and then computing $r = c - m \cdot kp$. To find m, we use the following approximation process:

$$\begin{aligned} m = \left\lfloor \frac{c}{kp} \right\rfloor &= \left\lfloor \frac{c_N \cdot r^N + c_{N-1} r^{N-1} + \cdots + c_0}{kp} \right\rfloor \\ &\approx \left\lfloor \frac{c_N \cdot R}{kp} \right\rfloor \\ &\approx \left\lfloor \frac{c_N \cdot \left\lfloor \frac{r \cdot R}{kp} \right\rfloor}{r} \right\rfloor . \end{aligned}$$

At this point, we may precompute the value $\mu = \left\lfloor \frac{r \cdot R}{kp} \right\rfloor$, then compute the product $c_N \cdot \mu$ and divide by r to get m. If we choose k to be the least multiple of p such that $kp > R$, then μ will be a 64-bit integer, and m is simply the high part of the product $c_N \cdot \mu$.

Of course, we will need to justify that the approximations above are actually equality, and we will do so in Lemma 7.1. However, before doing so, we give a further optimization in the case where k may be larger than 2 (i.e. p is less than half of R). This is the case for the scalar field of the BN254 curve, where p is a 254-bit prime and $k = 6$. We wish to prevent having to perform conditional subtraction on 6 different multiples $0, p, \ldots, 5p$.

We can do so by choosing $k = 2$ and modifying the algorithm to use a *modified* $R' = 2^{n_p} < R$, where n_p is the bit-length of p. This ensures that $k' = \lceil R'/p \rceil = 2$, and we just need to precompute the modified constant $\mu' = \lfloor rR'/(2p) \rfloor$, and derive $m' = \lfloor c/(2p) \rfloor$. Intuitively, this works because $(aR \mod p) \cdot b < p \cdot b < R' \cdot r$, so we can decompose c into chunks of size r, starting from the *most* significant digit:

$$c = c_N R' + c_{N-1} R'/r + \cdots + c_1 R'/r^{N-1} + c_0,$$

where $c_N, \ldots, c_1 < r$ and $c_0 < R'/r^{N-1}$ (assuming that $r^{N-1} < R' < R = r^N$, i.e. p fits in exactly N limbs). The rest of the approximation is the same as above, with R' replacing R and 2 replacing k.

Lemma 7.1. *The multiplication algorithm in Procedure 1 is correct.*

Proof. We need to show the Barrett reduction (Stage 2) correctly computes $c' = c \mod p$, where $c = a' \cdot b$. Let $q = \lfloor c/(2p) \rfloor$. The core of the reduction relies on approximating this quotient q.

Line 4 computes an estimate $m := \lfloor (\tilde{c} \cdot \mu)/r \rfloor$, where $\mu = \lfloor rR'/(2p) \rfloor$ and $\tilde{c} = \lfloor c/R' \rfloor$, so that $c = \tilde{c} \cdot R' + c'$ for some $c' < R'$. The key claim is that this estimate satisfies:

$$m \in \{q - 1, q\}. \tag{20}$$

By expanding the definition of m and q, we get:

$$\left\lfloor \frac{c}{2p} \right\rfloor - 1 \leq \left\lfloor \frac{\tilde{c} \cdot \left\lfloor \frac{r \cdot R'}{2p} \right\rfloor}{r} \right\rfloor \leq \left\lfloor \frac{c}{2p} \right\rfloor . \tag{21}$$

The second inequality is true since m is always an underestimate of the true quotient q:

$$\left\lfloor \frac{c}{2p} \right\rfloor = \left\lfloor \frac{\tilde{c} \cdot R' + c'}{2p} \right\rfloor = \left\lfloor \frac{\tilde{c} \cdot \frac{r \cdot R'}{2p}}{r} + \frac{c'}{2p} \right\rfloor \geq \left\lfloor \frac{\tilde{c} \cdot \left\lfloor \frac{r \cdot R'}{2p} \right\rfloor}{r} \right\rfloor . \tag{22}$$

The first inequality is true since m cannot underestimate q by more than 1. In other words, it suffices to show that

$$\left\lfloor \frac{c}{2p} \right\rfloor < \left\lfloor \frac{\tilde{c} \cdot \left\lfloor \frac{r \cdot R'}{2p} \right\rfloor}{r} \right\rfloor + 2. \tag{23}$$

This is true since:

$$\begin{aligned}\left\lfloor\frac{c}{2p}\right\rfloor &= \left\lfloor\frac{\tilde{c}\cdot\frac{r\cdot R'}{2p}}{r}+\frac{c'}{2p}\right\rfloor\\ &< \left\lfloor\frac{\tilde{c}\cdot\left(\left\lfloor\frac{r\cdot R'}{2p}\right\rfloor+1\right)}{r}+\frac{c'}{2p}\right\rfloor\\ &= \left\lfloor\frac{\tilde{c}\cdot\left\lfloor\frac{r\cdot R'}{2p}\right\rfloor}{r}+\frac{\tilde{c}}{r}+\frac{c'}{2p}\right\rfloor\\ &< \left\lfloor\frac{\tilde{c}\cdot\left\lfloor\frac{r\cdot R'}{2p}\right\rfloor}{r}\right\rfloor+2.\end{aligned}$$

The first inequality follows from $\lfloor a\rfloor < a+1$ for all a, and the second inequality follows from $\tilde{c} < r$ and $c' < R' < 2p$.

We have proven the key estimate in (21). The rest of the algorithm then follows, since we now have $r = c - m\cdot 2p$ is equal to either $c - \lfloor c/2p\rfloor\cdot 2p = c \mod 2p$, or to $c - (\lfloor c/2p\rfloor - 1)\cdot 2p = (c \mod 2p) + 2p$. This means that $r < 2p + 2p = 4p$, and that $r \mod p = (c \mod 2p) \mod p = c \mod p$. Hence, the final conditional subtraction in line 6 correctly computes $c' = r \mod p$. □

7.3 Extensions to Signed and 128-bit Integers

We briefly note that our algorithm can easily be extended to the following settings:

- Signed 64-bit integers: we simply check the sign of b. If it is negative, we multiply by $-b$ and negate the result. This costs an amortized 0.5 field subtractions compared to the unsigned case.
- Unsigned 128-bit integers: we can perform *two* steps of Barrett reduction, after computing bignum multiplication to get a $(N+2)$-limb number.
- Signed 128-bit integers: we combine the above two cases. Namely, perform 128-bit unsigned multiplication on the absolute value of b, and then negate the result if b was negative.

More details on multiplications by u128. To expand on our above point, assume that we have computed the bignum multiplication of $(aR \mod p)$ with b, where b is a 128-bit unsigned integer. This costs $2N$ base (u64) multiplications, and gives us a $(N+2)$-limb number

$$c = c_{N+1}r^{N+1} + c_N r^N + \cdots + c_1 r + c_0.$$

We will perform one pass of Barrett reduction on the $(N+1)$-high limbs of c, namely on $c_{\mathsf{high}} = c_{N+1}r^N + c_N r^{N-1} + \cdots + c_1$. The result will be a N-limb number $c' = c_{\mathsf{high}} \mod p$. We now perform a second pass of Barrett reduction on the $(N+1)$-limb number $c'\cdot r + c_0$, to get the final result $c'' = (c'\cdot r + c_0) \mod p$. We can easily verify that

$$c'' \equiv c'\cdot r + c_0 \equiv c_{\mathsf{high}}\cdot r + c_0 \equiv c \mod p,$$

and hence the algorithm is correct. This algorithm takes $2N + (2N+2) = 4N+2$ base (u64) multiplications.

We leave to future work the task of further optimizing the above algorithms for these settings. This could be in the form of a "native" signed Barrett reduction, or a more optimized algorithm for the 128-bit case that fuses two Barrett reductions into a single pass.

8 Implementation and Experiments

In this section, we report on the implementation of our optimizations and associated speedups. Our implementation has two parts: (1) a standalone library implementing all algorithms over binary tower fields, and (2) an integration of our optimization into the Jolt zkVM, speeding up the first sum-check in Spartan (described in Section 6.2). We did so to demonstrate our optimizations in two different field settings: binary tower fields and large scalar prime fields (Jolt currently works over the BN254 scalar field). However, this splitting of efforts also means that our standalone library is not yet fully optimized—we hope to address this in the near future.

8.1 Experimental Setup

For our standalone implementation, we implemented binary tower fields from scratch, achieving the aforementioned optimal speedup factor between $\mathfrak{ss}$, $\mathfrak{sl}$, and $\mathfrak{ll}$ multiplications (see Section 2.1).

For Jolt, we build on the library tooling. Our main modification is to the Spartan proving subroutine, which we replace with our optimized algorithm described in Section 6.2. We also make minor modifications to enable comparisons between the old and new Spartan prover.

We benchmark the prover runtimes on a M4 Pro Macbook with 24GB of RAM and 14 CPU cores.

Small-value settings. For our standalone implementation, we draw values from the 32-bit tower subfield of the 128-bit binary tower field. For Jolt, we inherit the existing R1CS constraints and the 32-bit RISC-V trace structure. This means that the values of Az, Bz, Cz in Spartan are within the range $\{-2^{64}, \dots, 2^{64}\}$ (necessitating their datatypes to be `i128`), though we stress that most of these values fit in `i64` and hence benefit from the more optimized version of our multiplication algorithm (Section 7).

List of Benchmarks. For our standalone implementation, we compare the following:

– Algorithm 1 versus Algorithm 4 (in the small-value-only setting). We experiment with various settings of the number of small value optimization rounds ℓ_0, and for degree $d = 2$.

– Algorithm 5 versus Algorithm 6 (in the combined eq-poly and small-value setting), again varying ℓ_0.

For Jolt, we benchmark proving repeated evaluations of SHA-2 (i.e. a SHA-2 chain) of increasing depth, from 8 to 2048 iterations, with factor of 2 increases. This corresponds to roughly 2^{17} to 2^{26} number of RISC-V cycles. We focus our benchmark on the first sum-check in Spartan (where our optimizations apply), and do not include a full end-to-end benchmark of the entire Jolt prover.

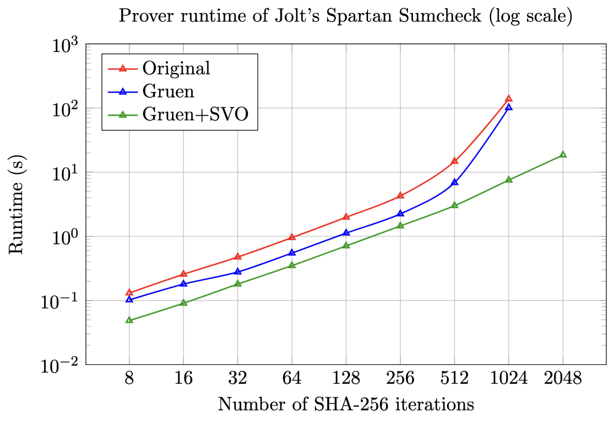
When we started, the Spartan implementation in Jolt had the following drawbacks: (1) it only implements the split-eq-poly optimization, but *not* Gruen's optimization (see Section 5 for the difference), (2) it does not incorporate our Spartan-specific optimizations described in Section 6.2, and (3) certain data movements are not optimized. For fair benchmarking, we have fixed these issues in an *intermediate* version of the Spartan prover, where all optimizations in this paper have been applied *except* for the small-value optimization. This provides a fair baseline for the impact of the small-value optimization alone.

8.2 Results

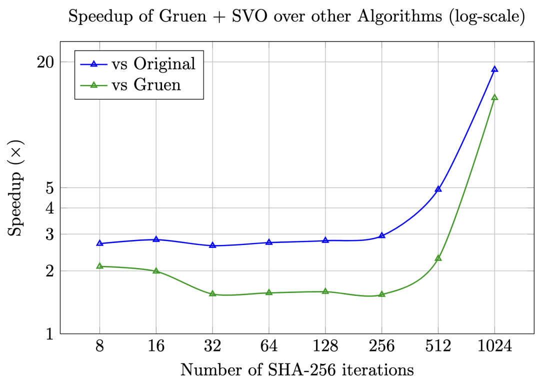
We give the results of our Jolt benchmarks in Figure 10, and of our standalone benchmarks in Figure 11.

The Jolt results reveal two distinct performance regimes. For smaller instances (8 to 256 iterations), our optimizations provide consistent speedups of around 3× over the original implementation and 1.5-2× over Gruen's optimization alone. From 512 to 2048 iterations, Gruen+SVO demonstrates its memory efficiency (as discussed in our cost estimates), giving almost 20× speedup over both other methods for 1024 iterations, and scaling smoothly to 2048 iterations while the other two algorithms run out of memory.

Our standalone implementation shows more modest but consistent improvements, due to the lack of optimizations. Algorithm 6 (our combined approach) provides 20-40% speedups over Algorithm 5 (eq-poly

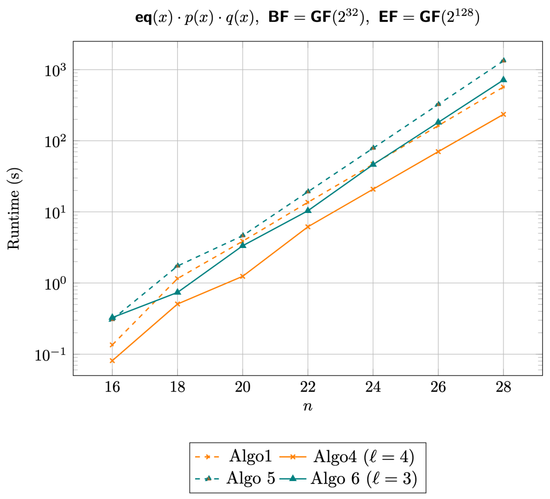


(a) Absolute Runtimes

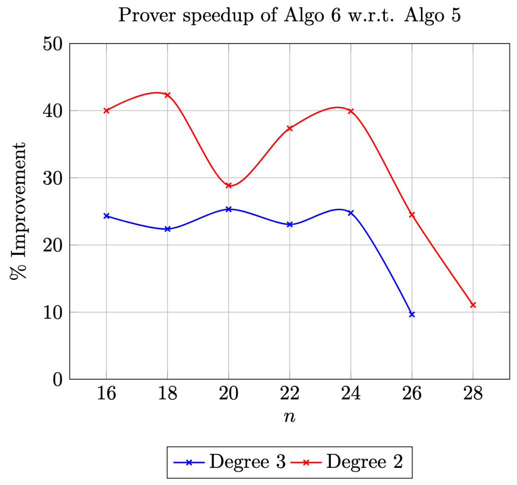


(b) Speedup

Fig. 10: Performance of the first Spartan sum-check in Jolt. (a) Runtimes for proving SHA-2 chain computation comparing Original Jolt, Jolt with Gruen's optimization, and Jolt with Gruen's & Small-Value Optimization (SVO). (b) Speedup factor of Gruen+SVO over Gruen-only and Original for SHA2 benchmarks.



(a) Absolute Runtimes



(b) Percentage Speedup

Fig. 11: Performance of standalone sum-check algorithms. (a) Absolute runtimes for $\sum \text{eq}(x)\cdot p(x)\cdot q(x)$ with base field BF=$\mathsf{GF}2^{32}$ and extension field EF=$\mathsf{GF}2^{128}$. "Algo1" is baseline; "Algo4 ($\ell_0 = 4$)" is small-value optimization; "Algo 5" is eq-polynomial optimization; "Algo 6 ($\ell_0 = 3$)" is combined. (b) Percentage improvement of Algorithm 6 over Algorithm 5 for degree 2 and degree 3 polynomials, varying n.

optimization alone) across $n = 16$ to $n = 28$ number of sum-check rounds, with degree-2 polynomials generally showing better improvements than degree-3. We also see the improvement in the non-eq-poly setting, with Algorithm 4 outperforming Algorithm 1 by roughly the same amount. While these gains are smaller than Jolt due to our implementation being less optimized, they help to partially validate our cost estimates, and demonstrate the potential for further improvements.

Overall Impact on Jolt. We now discuss the implications of our benchmarks for the end-to-end performance of Jolt. Thanks to other optimizations [2], the proving time of Spartan-in-Jolt is dominated by the first sum-check—hence our $3\times$ speedup is also for the whole Spartan proving process.

When Jolt moves to a *streaming* prover, our optimization will give a significant speedup compared to the naive streaming algorithm (Algorithm 2). This because, instead of needing to stream $O(\ell)$ rounds until falling below a certain memory threshold, our optimization (and the improvements to [10] described in a future version of this manuscript) will allow us to stream for only $O(\log \ell)$ rounds. Concretely, for a billion ($\approx 2^{30}$) RISC-V cycles, and keeping prover's memory to $2GB$ or less, we only need to stream the input 3 times instead of 10 or more times. We will give more detailed cost estimates in a future version of this manuscript.

9 Conclusion

In this work, we present a suite of optimizations for the sum-check prover in two common settings: small-value evaluations, and involvement of the equality polynomial. Our optimizations reduce both proving time and memory usage. We have implemented our algorithms and demonstrate significant speedups in a relevant application, that of improving Spartan-in-Jolt.

For future work, it would be valuable to investigate our algorithms in new contexts, such as in sum-check over rings (for lattice-based proof systems such as [11]), or in the sparse-dense setting. We also plan to revise our standalone implementation to incorporate more optimizations, which should fully realize our estimated speedups.

Disclosure of Interests. Justin Thaler is a Research Partner at a16z crypto and is an investor in various blockchain-based platforms, as well as in the crypto ecosystem more broadly (for general a16z disclosures, see `https://www.a16z.com/disclosures/`.)

References

1. Jolt codebase (2025), `https://github.com/a16z/jolt`
2. Arasu Arun: Simplified second-Spartan-sum-check (via simpler uniform R1CS product-vector MLE). `https://github.com/a16z/jolt/issues/347` (2024), accessed: 2025-05-03
3. Arnon, G., Chiesa, A., Fenzi, G., Yogev, E.: WHIR: Reed–solomon proximity testing with super-fast verification. Cryptology ePrint Archive, Paper 2024/1586 (2024), `https://eprint.iacr.org/2024/1586`
4. Arun, A., Setty, S., Thaler, J.: Jolt: Snarks for virtual machines via lookups. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 3–33. Springer (2024)
5. Attema, T., Fehr, S., Klooß, M.: Fiat-shamir transformation of multi-round interactive proofs. In: Theory of Cryptography Conference. pp. 113–142. Springer (2022)
6. Bagad, S., Domb, Y., Thaler, J.: The sum-check protocol over fields of small characteristic. Cryptology ePrint Archive, Paper 2024/1046 (2024), `https://eprint.iacr.org/2024/1046`
7. Barreto, P.S., Lynn, B., Scott, M.: Constructing elliptic curves with prescribed embedding degrees. In: International conference on security in communication networks. pp. 257–267. Springer (2002)
8. Barreto, P.S., Naehrig, M.: Pairing-friendly elliptic curves of prime order. In: International workshop on selected areas in cryptography. pp. 319–331. Springer (2005)
9. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Conference on the Theory and Application of Cryptographic Techniques. pp. 311–323. Springer (1986)

10. Baweja, A., Fenzi, G., Chiesa, A., Mishra, P., Zitek-Estrada, A., Fedele, E., Mopuri, T.: Time-space trade-offs for sumcheck (2025), manuscript
11. Boneh, D., Chen, B.: LatticeFold: A lattice-based folding scheme and its applications to succinct proof systems. Cryptology ePrint Archive, Paper 2024/257 (2024), https://eprint.iacr.org/2024/257
12. Boneh, D., Chen, B.: Latticefold+: Faster, simpler, shorter lattice-based folding for succinct proof systems. Cryptology ePrint Archive (2025)
13. Brehm, M., Chen, B., Fisch, B., Resch, N., Rothblum, R.D., Zeilberger, H.: Blaze: Fast snarks from interleaved raa codes. Cryptology ePrint Archive (2024)
14. Canetti, R., Chen, Y., Holmgren, J., Lombardi, A., Rothblum, G.N., Rothblum, R.D.: Fiat-shamir from simpler assumptions. Cryptology ePrint Archive (2018)
15. Chen, B., Bünz, B., Boneh, D., Zhang, Z.: HyperPlonk: Plonk with linear-time prover and high-degree custom gates (2023)
16. Chiesa, A., Fedele, E., Fenzi, G., Zitek-Estrada, A.: A time-space tradeoff for the sumcheck prover. Cryptology ePrint Archive, Paper 2024/524 (2024), https://eprint.iacr.org/2024/524
17. Cormode, G., Mitzenmacher, M., Thaler, J.: Practical verified computation with streaming interactive proofs. In: ITCS (2012)
18. Cormode, G., Thaler, J., Yi, K.: Verifying computations with streaming interactive proofs. Proc. VLDB Endow. **5**(1), 25–36 (2011). https://doi.org/10.14778/2047485.2047488, http://www.vldb.org/pvldb/vol5/p025_grahamcormode_vldb2012.pdf
19. Dao, Q., Thaler, J.: Constraint-packing and the sum-check protocol over binary tower fields. Cryptology ePrint Archive, Paper 2024/1038 (2024), https://eprint.iacr.org/2024/1038
20. Dao, Q., Thaler, J.: More optimizations to sum-check proving. Cryptology ePrint Archive, Paper 2024/1210 (2024), https://eprint.iacr.org/2024/1210
21. Diamond, B.E., Posen, J.: Succinct arguments over towers of binary fields. Cryptology ePrint Archive, Paper 2023/1784 (2023), https://eprint.iacr.org/2023/1784, https://eprint.iacr.org/2023/1784
22. Diamond, B.E., Posen, J.: Polylogarithmic proofs for multilinears over binary towers. Cryptology ePrint Archive, Paper 2024/504 (2024), https://eprint.iacr.org/2024/504, https://eprint.iacr.org/2024/504
23. Domb, Y.: The Barrett-Montgomery duality and a new multi-precision modular reduction scheme with only $n^2 - 1$ operations. https://medium.com/@ingonyama/the-barrett-montgomery-duality-and-a-new-multi-precision-modular-reduction-scheme-with-only-n2-1-b40ede4792da (2024), accessed: 2025-04-09
24. Fan, J.L., Paar, C.: On efficient inversion in tower fields of characteristic two. In: Proceedings of IEEE International Symposium on Information Theory. p. 20. IEEE (1997)
25. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. pp. 186–194 (1986)
26. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. Journal of the ACM (JACM) **62**(4), 1–64 (2015)
27. Gouvêa, C.P., López, J.: Implementing gcm on armv8. In: Topics in Cryptology—CT-RSA 2015: The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings. pp. 167–180. Springer (2015)
28. Gruen, A.: Some improvements for the piop for zerocheck. Cryptology ePrint Archive (2024)
29. Karatsuba, A.A.: The complexity of computations. Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation **211**, 169–183 (1995)
30. Kothapalli, A., Setty, S.: Hypernova: Recursive arguments for customizable constraint systems. In: Annual International Cryptology Conference. pp. 345–379. Springer (2024)
31. Kothapalli, A., Setty, S.: NeutronNova: Folding everything that reduces to zero-check. Cryptology ePrint Archive, Paper 2024/1606 (2024), https://eprint.iacr.org/2024/1606
32. Liu, T., Zhang, Y.: Efficient SNARKs for boolean circuits via sumcheck over tower fields. Cryptology ePrint Archive, Paper 2025/594 (2025), https://eprint.iacr.org/2025/594
33. Lund, C., Fortnow, L., Karloff, H., Nisan, N.: Algebraic methods for interactive proof systems. In: FOCS (Oct 1990)
34. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**(170), 519–521 (1985). https://doi.org/10.2307/2007970, https://doi.org/10.2307/2007970
35. Nair, V., Thaler, J., Zhu, M.: Proving CPU executions in small space. Cryptology ePrint Archive, Paper 2025/611 (2025), https://eprint.iacr.org/2025/611
36. Nguyen, W., Setty, S.: Neo: Lattice-based folding scheme for ccs over small fields and pay-per-bit commitments. Cryptology ePrint Archive (2025)
37. Novakovic, A., Angeris, G.: Ligerito: A small and concretely fast polynomial commitment scheme (2025)

38. Setty, S.: Spartan: Efficient and general-purpose zksnarks without trusted setup. In: Annual International Cryptology Conference. pp. 704–737. Springer (2020)
39. Setty, S., Angel, S., Gupta, T., Lee, J.: Proving the correct execution of concurrent services in zero-knowledge (Oct 2018)
40. Setty, S., Thaler, J.: Twist and shout: Faster memory checking arguments via one-hot addressing and increments. Cryptology ePrint Archive, Paper 2025/105 (2025), `https://eprint.iacr.org/2025/105`
41. Setty, S., Thaler, J., Wahby, R.: Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552 (2023), `https://eprint.iacr.org/2023/552`
42. Setty, S., Thaler, J., Wahby, R.: Unlocking the lookup singularity with lasso. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 180–209. Springer (2024)
43. Shamir, A.: Ip= pspace. Journal of the ACM (JACM) **39**(4), 869–877 (1992)
44. Thaler, J.: Time-optimal interactive proofs for circuit evaluation. In: CRYPTO (2013)
45. Thaler, J.: Proofs, arguments, and zero-knowledge. Foundations and Trends in Privacy and Security **4**(2–4), 117–660 (2022)
46. Toom, A.: The complexity of a scheme of functional elements realizing the multiplication of integers. In: Soviet Mathematics Doklady, No 3. pp. 714–716 (1963)
47. Vu, V., Setty, S., Blumberg, A.J., Walfish, M.: A hybrid architecture for verifiable computation (2013)
48. Wahby, R.S., Ji, Y., Blumberg, A.J., Shelat, A., Thaler, J., Walfish, M., Wies, T.: Full accounting for verifiable outsourcing (2017)
49. Wahby, R.S., Tzialla, I., Shelat, A., Thaler, J., Walfish, M.: Doubly-efficient zkSNARKs without trusted setup (2018)
50. Wei, Y., Wang, K., Xiang, B., Zhang, X., Wang, H., Deng, Y., Zhu, X., Lin, L.: Packed sumcheck over fields of small characteristic with application to verifiable FHE. Cryptology ePrint Archive, Paper 2025/719 (2025), `https://eprint.iacr.org/2025/719`
51. Wiedemann, D.: An iterated quadratic extension of gf (2). Fibonacci Quart **26**(4), 290–295 (1988)
52. Wiedemann, D.: An iterated quadratic extension of gf (2). The Fibonacci Quarterly **26**(4), 290–295 (1988)
53. Xie, T., Zhang, J., Zhang, Y., Papamanthou, C., Song, D.: Libra: Succinct zero-knowledge proofs with optimal prover computation (2019)
54. Zhang, J., Xie, T., Zhang, Y., Song, D.: Transparent polynomial delegation and its applications to zero knowledge proof. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 859–876. IEEE (2020)
55. Zhang, Y., Genkin, D., Katz, J., Papadopoulos, D., Papamanthou, C.: vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases (2017)
56. Zhang, Y., Genkin, D., Katz, J., Papadopoulos, D., Papamanthou, C.: vRAM: Faster verifiable RAM with program-independent preprocessing (2018)

A Auxiliary Procedures

In this section, we list the procedures that are called by the algorithms in the main body of the paper.

A.1 Equality polynomial evaluations

Procedure 2 computes the equality polynomial evaluations $\widetilde{\mathsf{eq}}(w, x)$ for $w \in \mathbb{F}^\ell$ and $x \in \{0,1\}^\ell$ using 2^ℓ $\mathfrak{ll}$ multiplications. Procedure 3 is a memoized version of Procedure 2 that additionally stores the intermediate result $\{e(w_{[1:i]}, x)\}_{i=1}^{\ell}$ at each step i.

Procedure 2 Compute $\widetilde{\mathsf{eq}}(w,x)$ for all $x \in \{0,1\}^\ell$

```
Input: $w \in \mathbb{F}^\ell$
Output: $\mathbf{v}$ such that $\mathbf{v}[\mathsf{nat}(x)] = \widetilde{\mathsf{eq}}(w,x)$ for all $x \in \{0,1\}^\ell$
 1: Initialize: an all-one vector $\mathbf{v} := (1,1,\dots,1) \in \{0,1\}^\ell$
 2: Initialize: $s := 1$
 3: for $i = 0,\dots,\ell-1$ do
 4:     for $j = s-1,\dots,0$ do
 5:         $\mathbf{v}[2\cdot j+1] := \mathbf{v}[j] \cdot w_i$
 6:         $\mathbf{v}[2\cdot j] := \mathbf{v}[j] - \mathbf{v}[2\cdot j+1]$
 7:     end for
 8:     $s := s \cdot 2$
 9: end for
10: return $\mathbf{v}$
```

Procedure 3 Compute $\{\widetilde{\mathsf{eq}}(w_{[1:i]},x) : x \in \{0,1\}^i\}$ for all $i \in [\ell]$

```
Input: $w \in \mathbb{F}^\ell$
Output: $\mathbf{v}_1,\dots,\mathbf{v}_\ell$ such that $\mathbf{v}_i[\mathsf{nat}(x)] = \widetilde{\mathsf{eq}}(w_{[1:i]},x)$ for all $i = 1,\dots,\ell$ and $x \in \{0,1\}^i$
 1: Initialize: empty vectors $\mathbf{v}_i$ of size $2^i$ for all $i = 1,\dots,\ell$.
 2: Initialize: $\mathbf{v}_0 := (1)$ and $s := 1$
 3: for $i = 0,\dots,\ell-1$ do
 4:     for $j = s-1,\dots,0$ do
 5:         $\mathbf{v}_{i+1}[2\cdot j+1] := \mathbf{v}_i[j] \cdot w_i$
 6:         $\mathbf{v}_{i+1}[2\cdot j] := \mathbf{v}_i[j] - \mathbf{v}_{i+1}[2\cdot j+1]$
 7:     end for
 8:     $s := s \cdot 2$
 9: end for
10: return $\mathbf{v}_1,\dots,\mathbf{v}_\ell$
```

A.2 Procedures for Algorithm 3

To allow for an optimized implementation where each product $\prod_{k=1}^d p_k(\boldsymbol{b}_k, x'')$ (for $\boldsymbol{b}_k \in \{0,1\}^{i-1}, x'' \in \{0,1\}^{\ell-\ell_0}$) is computed only once across all accumulators and all rounds $i \in [\ell_0]$, we can rewrite the definition of the accumulators $\mathsf{A}_i(\boldsymbol{v},u)$ by further splitting up the sum over $x' \in \{0,1\}^{\ell-i}$ to be over $x'' \in \{0,1\}^{\ell-\ell_0}$ and $y \in \{0,1\}^{\ell_0-i}$:

$$\mathsf{A}_i(\boldsymbol{v},u) := \sum_{x''\in\{0,1\}^{\ell-\ell_0}} \sum_{y\in\{0,1\}^{\ell_0-i}} \sum_{\substack{\boldsymbol{b}_1,\dots,\boldsymbol{b}_d\in\{0,1\}^{i-1} \\ \boldsymbol{v}=\sum_{k=1}^d \boldsymbol{b}}} \sum_{b_1^*,\dots,b_d^*\in\{0,1\}} u^{\sum_{k=1}^d b_k^*} \cdot (1-u)^{d-\sum_{k=1}^d b_k^*} \cdot \prod_{k=1}^d p_k(\boldsymbol{b_k}, b_k^*, y, x'').$$

What we gain from this is that we can now iterate over the *combined* vectors $(\boldsymbol{b}_k, b_k^*, y)$, and flipping the order of evaluation, we get a procedure where we first iterate over $\boldsymbol{B} := (\boldsymbol{b}_1,\dots,\boldsymbol{b}_d) \in \left(\{0,1\}^{\ell_0}\right)^d$ (note the ℓ_0 index instead of $i-1$), compute the product term corresponding to $\boldsymbol{B}$ and y, and then based on the values of $\boldsymbol{B}$, decide which accumulators to add the product term to.

Definition A.1 (Algorithm 3: Mapping from product terms to accumulators). *Let $\boldsymbol{B} = (\boldsymbol{b}_1,\dots,\boldsymbol{b}_d) \in \left(\{0,1\}^{\ell_0}\right)^d$ be a tuple of d bitstrings of length ℓ_0. We define $\mathsf{idx3}(\boldsymbol{B})$ to be the set of tuples $(i \in [\ell_0], \boldsymbol{v} \in [d]^{i-1})$ such that $\boldsymbol{b}_1[>i] = \cdots = \boldsymbol{b}_d[>i]$ and $\boldsymbol{v} = \sum_{k=1}^d \boldsymbol{b}_k[<i]$.*

Intuitively, the set of indices $\mathsf{idx3}(\boldsymbol{b}_1, \dots, \boldsymbol{b}_d)$ tracks where the product term $\prod_{k=1}^{d} p_k(\boldsymbol{b}_k, x')$ needs to go to in the various accumulators. It looks at the first deviation (from right to left) in the bitstrings $\boldsymbol{b}_1, \dots, \boldsymbol{b}_d$; if they deviate after index $i \in [\ell_0]$, then this product *cannot* be present in any accumulator at round $i-1$ or before.

Example A.2. Consider $\ell_0 = 3$ and $d = 2$. We have the following:

$$\mathsf{idx3}((0,0,0),(1,1,0)) = \{(2,(1,1)),(3,(1,1,0))\},$$

meaning that for every $x'' \in \{0,1\}^{\ell-3}$, the product $p_1(0,1,0,x'') \cdot p_2(1,1,0,x'')$ goes into

- the accumulator $\mathsf{A}_2(0,1)$ at round 2,
- the accumulator $\mathsf{A}_3(1,1,0)$ at round 3.

The product cannot go into any accumulator in the first round, since they differ in the first two bits. In other words, since the products needed for the first round has the form $p_1(b_1, x') \cdot p_2(b_2, x')$ for $b_1, b_2 \in \{0,1\}$ and $x' \in \{0,1\}^{\ell-1}$, they possibly differ only in the first bit.

Procedure 4 Compute accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ for Algorithm 3

```
Input: A stream of evaluations $\{p_k(\beta, x'')\}_{k\in[d],\beta\in\{0,1\}^{\ell_0}}$ for each $x'' \in \{0,1\}^{\ell-\ell_0}$
Output: The accumulators $\{\mathsf{A}_i(\boldsymbol{v},u) : i \in [\ell_0], \boldsymbol{v} \in [0,d]^{i-1}, u \in \widehat{U}_d\}$
1: Initialize: $\mathsf{A}_i(\boldsymbol{v},u) := 0$ for all $i, \boldsymbol{v}, u$.
2: for $x'' \in \{0,1\}^{\ell-\ell_0}$ do    ▷ Iterate over the suffix outside the $\ell_0$ prefix
3:     for $\boldsymbol{B} = (\boldsymbol{b}_1,\dots,\boldsymbol{b}_d) \in \left(\{0,1\}^{\ell_0}\right)^d$ do    ▷ Iterate over all length-$\ell_0$ prefix combinations
4:         Compute product $\mathsf{P} := \prod_{k=1}^{d} p_k(\boldsymbol{b}_k, x'')$.    ▷ Compute the product term once
5:         Compute index set $\mathcal{I} = \mathsf{idx3}(\boldsymbol{B})$.    ▷ Find which accumulators this product contributes to
6:         for $(i, \boldsymbol{j}) \in \mathcal{I}$ do    ▷ For each relevant round $i$ and its prefix sum $\boldsymbol{j}$ (length $i$)
7:             Let $\boldsymbol{v} = (\boldsymbol{j}_1,\dots,\boldsymbol{j}_{i-1})$.    ▷ Derive accumulator index $\boldsymbol{v}$ (length $i-1$) from $\boldsymbol{j}$
8:             for $u \in \widehat{U}_d$ do
9:                 Get the $i$-th bits: $b_k^* := \boldsymbol{b}_k[i]$ for $k = 1,\dots,d$.
10:                Compute coefficient $C_u := u^{\sum_{k=1}^{d} b_k^*} \cdot (1-u)^{d-\sum_{k=1}^{d} b_k^*}$.    ▷ Coefficient depends on round $i$ via $b_k^*$
11:                $\mathsf{A}_i(\boldsymbol{v},u) := \mathsf{A}_i(\boldsymbol{v},u) + C_u \cdot \mathsf{P}$.    ▷ Add scaled product to correct accumulator
12:            end for
13:        end for
14:    end for
15: end for
16: return $\{\mathsf{A}_i(\boldsymbol{v},u) : i \in [\ell_0], \boldsymbol{v} \in [0,d]^{i-1}, u \in \widehat{U}_d\}$
```

We can formally show that our optimization of reversing the order of evaluation is correct.

Lemma A.3. *Procedure 4 computes the desired accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ (as defined in Figure 6) for all $i \in [\ell_0]$, $\boldsymbol{v} \in [0,d]^{i-1}$, and $u \in \widehat{U}_d$.*

Proof. This simply follows by rearranging the order of summation:

$$\begin{aligned}
\mathsf{A}_i(\boldsymbol{v}, u) &= \sum_{x'' \in \{0,1\}^{\ell-\ell_0}} \sum_{y \in \{0,1\}^{\ell_0 - i}} \sum_{\substack{\boldsymbol{b}_1,\ldots,\boldsymbol{b}_d \in \{0,1\}^{i-1} \\ \boldsymbol{v} = \sum_{k=1}^d \boldsymbol{b}}} \sum_{b_1^*,\ldots,b_d^* \in \{0,1\}} \\
&\qquad u^{\sum_{k=1}^d b_k^*} \cdot (1-u)^{d - \sum_{k=1}^d b_k^*} \cdot \prod_{k=1}^d p_k(\boldsymbol{b_k}, b_k^*, y, x'') \\
&= \sum_{x'' \in \{0,1\}^{\ell-\ell_0}} \sum_{\substack{\boldsymbol{b}_1,\ldots,\boldsymbol{b}_d \in \{0,1\}^{i-1} \\ \boldsymbol{v} = \sum_{k=1}^d \boldsymbol{b}}} \sum_{b_1^*,\ldots,b_d^* \in \{0,1\}} \sum_{y \in \{0,1\}^{\ell_0 - i}} \\
&\qquad u^{\sum_{k=1}^d b_k^*} \cdot (1-u)^{d - \sum_{k=1}^d b_k^*} \cdot \prod_{k=1}^d p_k(\boldsymbol{b_k}, b_k^*, y, x'') \\
&= \sum_{x'' \in \{0,1\}^{\ell-\ell_0}} \sum_{\substack{\boldsymbol{b}_1,\ldots,\boldsymbol{b}_d \in \{0,1\}^{\ell_0} \\ \boldsymbol{v} = \sum_{k=1}^d \boldsymbol{b}[<i], \quad y = \boldsymbol{b}_1[>i] = \cdots = \boldsymbol{b}_d[>i]}} \\
&\qquad u^{\sum_{k=1}^d \boldsymbol{b}_k[i]} \cdot (1-u)^{d - \sum_{k=1}^d \boldsymbol{b}_k[i]} \cdot \prod_{k=1}^d p_k(\boldsymbol{b}_k, x'') \\
&= \sum_{x'' \in \{0,1\}^{\ell-\ell_0}} \sum_{\substack{\boldsymbol{b}_1,\ldots,\boldsymbol{b}_d \in \{0,1\}^{\ell_0} \\ (i, \boldsymbol{v}) \in \mathsf{idx3}(\boldsymbol{b}_1,\ldots,\boldsymbol{b}_d)}} \\
&\qquad u^{\sum_{k=1}^d \boldsymbol{b}_k[i]} \cdot (1-u)^{d - \sum_{k=1}^d \boldsymbol{b}_k[i]} \cdot \prod_{k=1}^d p_k(\boldsymbol{b}_k, x'').
\end{aligned}$$

The first equality follows from our discussion at the beginning of this section. The second equality follows from moving the sum over y to the inside. The third equality follows from the renaming $(\boldsymbol{b}_k, b_k^*, y) \mapsto \boldsymbol{b}_k$ for all $k \in [d]$, and modifying the other terms appropriately. The final equality follows from the definition of $\mathsf{idx3}$. At this point, we now see that this order of summation is exactly followed by Procedure 4, thus finishing the proof.

Lemma A.4. *Algorithm 3 is correct. In particular, the output of the sum-check prover in Algorithm 3 is always the same as the output of the sum-check prover in the standard linear-time algorithm (Algorithm 1).*

Proof. This follows from our discussion in Section 4.1 and the correctness of the pre-computation routine in Procedure 4.

A.3 Procedures for Algorithm 4

Similar to the optimization for Algorithm 3 in Section A.2, we can improve the efficiency of computing the accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ for Algorithm 4 (Figure 7) by reversing the order of summation. The original definition sums over suffixes x' and then computes products based on the prefix $(\boldsymbol{v}, u)$. Reversing this involves iterating over all possible full evaluation prefixes $\beta \in {U_d}^{\ell_0}$ and suffixes $x'' \in \{0,1\}^{\ell-\ell_0}$, computing the full product term $\mathsf{P} = \prod_{k=1}^d p_k(\beta, x'')$ once, and then determining which accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ this term contributes to.

A challenge arises because the input polynomials p_k are typically provided via evaluations over the Boolean hypercube $\{0,1\}^\ell$. To compute $p_k(\beta, x'')$ where β contains elements from U_d (not just $\{0,1\}$), we need to extend the evaluations from the binary domain. Since p_k is multilinear in its first ℓ_0 variables (holding the last $\ell - \ell_0$ fixed as x''), we can compute these extended evaluations iteratively using the linearity in each variable. Procedure 6 describes this process, taking binary evaluations $\{p_k(b, x'')\}_{b\in\{0,1\}^{\ell_0}}$ and producing the required evaluations $\{p_k(\beta, x'')\}_{\beta\in {U_d}^{\ell_0}}$.

To map each computed product term $\mathsf{P} = \prod_{k=1}^{d} p_k(\beta, x'')$ to the correct accumulators, we use the index set $\mathsf{idx4}(\beta)$ defined in Definition A.5. This set identifies exactly the tuples $(i, \boldsymbol{v}, u)$ such that the prefix β is compatible with the structure required for accumulator $\mathsf{A}_i(\boldsymbol{v}, u)$ (i.e., $\beta = (\boldsymbol{v}, u, y)$ for some binary y).

Definition A.5 (Algorithm 4: Mapping from evaluation prefixes to accumulators). *Let* $\beta = (\beta_1, \ldots, \beta_{\ell_0}) \in {U_d}^{\ell_0-1}$. *We define* $\mathsf{idx4}(\beta)$ *to be the set of tuples* $(i, \boldsymbol{v}, u, y)$ *such that:*

- $i \in [\ell_0]$,
- $\boldsymbol{v} = (\beta_1, \ldots, \beta_{i-1}) \in {U_d}^{i-1}$,
- $u = \beta_i \in \widehat{U}_d$,
- $y = (\beta_{i+1}, \ldots, \beta_{\ell_0}) \in \{0,1\}^{\ell_0-i}$.

This set identifies all accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ *to which the product term computed using the prefix* β *contributes. The* y *component will be summed over when computing the accumulators.*

Procedure 5 Compute $\{p(u)\}_{u\in U_d}$ for a linear polynomial $p(X)$

Input: Evaluations $p(0), p(1)$
Output: Evaluations $p(\infty), p(2), \ldots, p(d-1)$, available in a streaming fashion.

```
1: Initialize: vector v of size $d-1$.
2: Set $\mathbf{v}[1] := p(1) - p(0)$ and $\mathbf{v}[2] := \mathbf{v}[1] + p(1)$.
3: for $i = 3, \ldots, d-1$ do
4:     $\mathbf{v}[i] := \mathbf{v}[i-1] + \mathbf{v}[1]$
5: end for
6: return v
```

Procedure 6 Compute $\{\mathsf{p}(\beta) : \beta \in {U_d}^{\ell_0}\}$ for a multilinear polynomial p

Input: Evaluations $\{\mathsf{p}(y)\}_{y\in\{0,1\}^{\ell_0}}$, stored in a map $\mathsf{CurrentEvals}$ with keys in $\{0,1\}^{\ell_0}$. p is multilinear.
Output: Evaluations $\{\mathsf{p}(\beta)\}_{\beta\in{U_d}^{\ell_0}}$, returned as a stream over the lexicographically ordered domain ${U_d}^{\ell_0}$.

```
 1: for $j = 1$ to $\ell_0$ do                                    ▷ Incrementally extend the dimension $j$
 2:     Initialize: An empty map $\mathsf{NextEvals}$.            ▷ Maps points in ${U_d}^j \times \{0,1\}^{\ell_0-j}$ to field elements
 3:     $\mathsf{PrefixDomain} := {U_d}^{j-1}$.                    ▷ Domain for $\beta_1, \dots, \beta_{j-1}$
 4:     $\mathsf{SuffixDomain} := \{0,1\}^{\ell_0-j}$.             ▷ Domain for $y_{j+1}, \dots, y_{\ell_0}$
 5:     $\mathsf{CurrentEvalDomain} := U_d$                        ▷ Default domain for current variable
 6:     for $\beta_{\text{prefix}} \in \mathsf{PrefixDomain}$ do
 7:         for $y_{\text{suffix}} \in \mathsf{SuffixDomain}$ do
 8:             Construct key $\mathsf{k}_0 = (\beta_{\text{prefix}}, 0, y_{\text{suffix}})$.   ▷ Tuple representing the point
 9:             Construct key $\mathsf{k}_1 = (\beta_{\text{prefix}}, 1, y_{\text{suffix}})$.   ▷ Tuple representing the point
10:             $\mathsf{p}_0 := \mathsf{CurrentEvals}[\mathsf{k}_0]$.            ▷ Get value at $X_j = 0$
11:             $\mathsf{p}_1 := \mathsf{CurrentEvals}[\mathsf{k}_1]$.            ▷ Get value at $X_j = 1$
12:             Compute extended slice values $\{\mathsf{v}_\gamma\}_{\gamma\in U_d}$ using Procedure 5($\mathsf{p}_0, \mathsf{p}_1$).
13:             for $\gamma \in \mathsf{CurrentEvalDomain}$ do
14:                 Construct key $\mathsf{k}_\gamma = (\beta_{\text{prefix}}, \gamma, y_{\text{suffix}})$.
15:                 $\mathsf{NextEvals}[\mathsf{k}_\gamma] := \mathsf{v}_\gamma$.     ▷ Store evaluation for $X_j = \gamma$
16:             end for
17:         end for
18:     end for
19:     $\mathsf{CurrentEvals} := \mathsf{NextEvals}$.   ▷ Update map; keys are now in ${U_d}^j \times \{0,1\}^{\ell_0-j}$ (or subset for $j = \ell_0$)
20: end for
21: return $\mathsf{CurrentEvals}$                                 ▷ Final map has keys in ${U_d}^{\ell_0}$
```

Procedure 7 Compute accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ for Algorithm 4

Input: A stream providing blocks of 2^{ℓ_0} evaluations $\{p_k(b, x'') : k \in [d], b \in \{0,1\}^{\ell_0}\}$, for each $x'' \in \{0,1\}^{\ell-\ell_0}$
Output: The accumulators $\{\mathsf{A}_i(\boldsymbol{v}, u) : i \in [\ell_0], \boldsymbol{v} \in {U_d}^{i-1}, u \in \widehat{U}_d\}$

```
 1: Initialize: $\mathsf{A}_i(\boldsymbol{v}, u) := 0$ for all $i \in [\ell_0]$, $\boldsymbol{v} \in {U_d}^{i-1}$, $u \in \widehat{U}_d$.
 2: for $x'' \in \{0,1\}^{\ell-\ell_0}$ do                         ▷ Iterate over the suffix
 3:     for $\beta \in {U_d}^{\ell_0}$ do                           ▷ Iterate over all possible full prefixes
 4:         Compute $p_k(\beta, x'')$ for all $k \in [d]$ using Procedure 6.
                                                                 ▷ Derive necessary evaluations
 5:         Compute product $\mathsf{P} := \prod_{k=1}^{d} p_k(\beta, x'')$.   ▷ Compute product term
 6:         Compute index set $\mathcal{I} = \mathsf{idx4}(\beta)$.           ▷ Find target accumulators
 7:         for $(i, \boldsymbol{v}, u, y) \in \mathcal{I}$ do                 ▷ For each target accumulator
 8:             $\mathsf{A}_i(\boldsymbol{v}, u) := \mathsf{A}_i(\boldsymbol{v}, u) + \mathsf{P}$.   ▷ Add product to accumulator
 9:         end for
10:     end for
11: end for
12: return $\{\mathsf{A}_i(\boldsymbol{v}, u) : i \in [\ell_0], \boldsymbol{v} \in {U_d}^{i-1}, u \in \widehat{U}_d\}$
```

Lemma A.6. *Procedure 7 computes the desired accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ (as defined in Figure 7) for all $i \in [\ell_0]$, $\boldsymbol{v} \in [0, d]^{i-1}$, and $u \in \widehat{U}_d$.*

Proof. We start from the definition of $\mathsf{A}_i(\boldsymbol{v}, u)$ in Figure 7:

$$\mathsf{A}_i(\boldsymbol{v}, u) := \sum_{x'\in\{0,1\}^{\ell-i}} \prod_{k=1}^{d} p_k(\boldsymbol{v}, u, x').$$

Let $x' = (y, x'')$ where $y \in \{0,1\}^{\ell_0 - i}$ and $x'' \in \{0,1\}^{\ell - \ell_0}$. The evaluation point for p_k requires the first ℓ_0 coordinates to be $(\boldsymbol{v}, u, y)$, where $\boldsymbol{v} \in {U_d}^{i-1}$, $u \in \widehat{U}_d$, and y is binary. We can rewrite the sum as:

$$\mathsf{A}_i(\boldsymbol{v}, u) = \sum_{x'' \in \{0,1\}^{\ell - \ell_0}} \sum_{y \in \{0,1\}^{\ell_0 - i}} \prod_{k=1}^{d} p_k((\boldsymbol{v}, u, y), x''). \tag{24}$$

Now, consider the computation performed by Procedure 7. For a fixed x'', it computes product terms $\mathsf{P}(\beta, x'') = \prod_{k=1}^{d} p_k(\beta, x'')$ for all relevant prefixes $\beta \in {U_d}^{\ell_0}$. It then adds $\mathsf{P}(\beta, x'')$ to the accumulator $\mathsf{A}_i(\boldsymbol{v}, u)$ if and only if $(i, \boldsymbol{v}, u, y) \in \mathsf{idx4}(\beta)$. The total value accumulated in $\mathsf{A}_i(\boldsymbol{v}, u)$ is therefore:

$$\sum_{x'' \in \{0,1\}^{\ell - \ell_0}} \sum_{\substack{\beta \in {U_d}^{\ell_0} \\ \text{s.t. } (i, \boldsymbol{v}, u, y) \in \mathsf{idx4}(\beta)}} \prod_{k=1}^{d} p_k(\beta, x''). \tag{25}$$

We argue that Equation (24) and Equation (25) are equivalent. By definition (Definition A.5), the condition $(i, \boldsymbol{v}, u, y) \in \mathsf{idx4}(\beta)$ holds if and only if β has the form $(\boldsymbol{v}, u, y)$ for some $y \in \{0,1\}^{\ell_0 - i}$. Therefore, the inner sum in Equation (25) iterates over precisely the same prefixes $\beta = (\boldsymbol{v}, u, y)$ as the inner sum in Equation (24) iterates over using y. The product terms are identical for corresponding prefixes. Thus, the sums are equal. This shows that Procedure 7 correctly computes $\mathsf{A}_i(\boldsymbol{v}, u)$ as defined in Figure 7.

Lemma A.7. *Algorithm 4 is correct. In particular, the output of the sum-check prover in Algorithm 4 is always the same as the output of the sum-check prover in the standard linear-time algorithm (Algorithm 1).*

Proof. This follows from our discussion in Section 4.2 and the correctness of the pre-computation routine in Lemma A.6.

A.4 Procedures for Algorithm 5 & Algorithm 6

We describe the procedures needed in Algorithm 5 and Algorithm 6 in the main body. The first, and simpler, procedure is the algorithm to compute the round polynomial $s_i(X) = \widetilde{\mathsf{eq}}(w_{[:i]}, (r_{[<i]}, X)) \cdot t_i(X)$ given evaluations $\{t_i(c)\}_{c \in \widehat{U}_d}$ and the claimed equality $s_i(0) + s_i(1) = T_i$, where T_i is the claimed value of the round. This is given in Procedure 8, and can easily be seen to be correct.

Procedure 8 Compute $s(X) = \ell(X) \cdot t(X)$ given evaluations $\{t(c)\}_{c \in \widehat{U}_d}$ and the condition $s(0) + s(1) = T$

```
Input: Evaluations $\{t(c)\}_{c \in \widehat{U}_d}$ for a polynomial $t(X)$ of degree $d$, a linear polynomial $\ell(X)$, and a target sum $T \in \mathbb{F}$
Output: Polynomial $s(X)$ of degree $d+1$ such that $s(0) + s(1) = T$ and $s(X) = \ell(X) \cdot t(X)$
1: compute $t(1) = \ell(1)^{-1} \cdot (T - \ell(0) \cdot t(0))$.
2: interpolate $t(X)$ from $\{t(c)\}_{c \in U_d}$ via Equation (8).
3: return $s(X) = \ell(X) \cdot t(X)$.
```

The more complex procedure is the pre-computation algorithm for Algorithm 6. Because of the extra equality polynomial factor $\widetilde{\mathsf{eq}}(w, X)$ split across different parts of the input variables (as introduced in Section 5.2), this computation is more involved than the analogous computation for Algorithm 4.

Recall the definition of the accumulators for Algorithm 6 (derived from combining the ideas in Section 5.2 and Section 4.2):

$$\mathsf{A}_i(\boldsymbol{v}, u) = \sum_{y \in \{0,1\}^{\ell_0 - i}} \sum_{x_{\mathsf{out}} \in \{0,1\}^{\ell/2 - \ell_0}} \widetilde{\mathsf{eq}}\left((w_{[(i+1):\ell_0]}, w_{[(\ell/2+\ell_0+1):]}), (y, x_{\mathsf{out}})\right) \tag{26}$$
$$\sum_{x_{\mathsf{in}} \in \{0,1\}^{\ell/2}} \widetilde{\mathsf{eq}}(w_{[(\ell_0+1):(\ell_0+\ell/2)]}, x_{\mathsf{in}}) \cdot \prod_{k=1}^{d} p_k(\boldsymbol{v}, u, y, x_{\mathsf{in}}, x_{\mathsf{out}})$$

Here, $(\boldsymbol{v}, u, y)$ forms the prefix of length ℓ_0, x_{in} forms the next $\ell/2$ variables, and x_{out} forms the final $\ell/2 - \ell_0$ variables. The $\widetilde{\mathsf{eq}}$ polynomial is split accordingly.

To compute these accumulators efficiently, we rearrange the summation order, similar to the approach used for Algorithm 4. The goal is to first compute the innermost sum involving x_{in} and the product term $\prod p_k$, weighted by the relevant part of the $\widetilde{\mathsf{eq}}$ polynomial. This inner sum depends on the prefix $\beta \in {U_d}^{\ell_0}$ (which generalizes $(\boldsymbol{v}, u, y)$ to allow non-binary values) and the suffix x_{out}. We can then distribute this computed inner sum to the appropriate final accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ using the $\mathsf{idx4}$ mapping, weighted by the remaining part of the $\widetilde{\mathsf{eq}}$ polynomial.

Specifically, we can rewrite Equation (26) by changing the order of summation. We bring the sums over x_{out} and the full prefix $\beta \in {U_d}^{\ell_0}$ to the outside, using $\mathsf{idx4}$ to relate β back to the required $(i, \boldsymbol{v}, u, y)$ for a specific accumulator:

$$\mathsf{A}_i(\boldsymbol{v}, u) = \sum_{x_{\mathsf{out}} \in \{0,1\}^{\ell/2 - \ell_0}} \sum_{\beta \in {U_d}^{\ell_0}} \sum_{\substack{(i', \boldsymbol{v}', u', y) \in \mathsf{idx4}(\beta) \\ \text{s.t. } i'=i, \boldsymbol{v}'=\boldsymbol{v}, u'=u}} \tag{27}$$
$$\widetilde{\mathsf{eq}}\left((w_{[(i'+1):\ell_0]}, w_{[(\ell/2+\ell_0+1):]}), (y, x_{\mathsf{out}})\right) \tag{28}$$
$$\cdot \left(\sum_{x_{\mathsf{in}} \in \{0,1\}^{\ell/2}} \widetilde{\mathsf{eq}}(w_{[(\ell_0+1):(\ell_0+\ell/2)]}, x_{\mathsf{in}}) \cdot \prod_{k=1}^{d} p_k(\beta, x_{\mathsf{in}}, x_{\mathsf{out}}) \right)$$

This rearranged form motivates Procedure 9. In this algorithm, we first compute the inner sum over x_{in} (stored in temporary accumulators tA) for each x_{out} and β, and then distribute this result, weighted by the outer $\widetilde{\mathsf{eq}}$ factor (corresponding to $\mathsf{E}_{\mathsf{out},i'}$ evaluated at the correct index i' which equals i), to the final accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ selected by $\mathsf{idx4}$. This avoids redundant computations of the inner sum.

Procedure 9 Compute accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ for Algorithm 6

Input:

- A stream providing evaluations $\{p_k(b, x') : k \in [d], b \in \{0,1\}^{\ell_0}\}$ for each $x' \in \{0,1\}^{\ell-\ell_0}$
- Pre-computed eq-poly evaluations $\mathsf{E}_{\mathsf{in}} := \left(\widetilde{\mathsf{eq}}(w_{[(\ell_0+1):(\ell_0+\ell/2)]}, x_{in})\right)_{x_{in} \in \{0,1\}^{\ell/2}}$
- Pre-computed eq-poly evaluations for all $i \in [\ell_0]$:

$$\mathsf{E}_{\mathsf{out},i} := \left(\widetilde{\mathsf{eq}}((w_{[(i+1):\ell_0]}, w_{[(\ell/2+\ell_0+1):]}), (y, x_{out}))\right)_{(y,x_{out}) \in \{0,1\}^{\ell_0} \times \{0,1\}^{\ell/2-\ell_0}}.$$

Output: The accumulators $\{\mathsf{A}_i(\boldsymbol{v}, u) : i \in [\ell_0], \boldsymbol{v} \in {U_d}^{i-1}, u \in \widehat{U}_d\}$

```
 1: Initialize: $\mathsf{A}_i(\boldsymbol{v}, u) := 0$ for all $i, \boldsymbol{v}, u$.
 2: for $x_{out} \in \{0,1\}^{\ell/2-\ell_0}$ do
 3:     Initialize: temporary accumulators $\mathsf{tA}[\beta] := 0$ for all $\beta \in {U_d}^{\ell_0}$.
 4:     for $x_{in} \in \{0,1\}^{\ell/2}$ do                ▷ Inner loop over suffix inner bits (low-order)
 5:         for $\beta \in {U_d}^{\ell_0}$ do                ▷ Iterate all full prefixes
 6:             Compute $p_k := p_k(\beta, x_{in}, x_{out})$ for all $k \in [d]$ via Procedure 6.
 7:             Update $\mathsf{tA}[\beta] := \mathsf{tA}[\beta] + \mathsf{E}_{\mathsf{in}}[x_{in}] \cdot \prod_{k=1}^{d} p_k(\beta, x_{in}, x_{out})$.
 8:         end for
 9:     end for
10:     for $\beta \in {U_d}^{\ell_0}$ do                ▷ Distribute to final accumulators
11:         for $(i, \boldsymbol{v}, u, y) \in \mathsf{idx4}(\beta)$ do                ▷ Find target indices via idx4
12:             Update $\mathsf{A}_i(\boldsymbol{v}, u) := \mathsf{A}_i(\boldsymbol{v}, u) + \mathsf{E}_{\mathsf{out},i}[y, x_{out}] \cdot \mathsf{tA}[\beta]$.
13:         end for
14:     end for
15: end for
16: return $\{\mathsf{A}_i(\boldsymbol{v}, u)\}$
```

Lemma A.8. *Procedure 9 computes the desired accumulators $\mathsf{A}_i(\boldsymbol{v}, u)$ (as defined in Equation (26)) for all $i \in [\ell_0]$, $\boldsymbol{v} \in {U_d}^{i-1}$, and $u \in \widehat{U}_d$.*

Proof. First, we justify the equivalence between the original definition in Equation (26) and the rearranged form in Equation (27). The original definition is:

$$\mathsf{A}_i(\boldsymbol{v}, u) = \sum_{y \in \{0,1\}^{\ell_0 - i}} \sum_{x_{\mathsf{out}} \in \{0,1\}^{\ell/2-\ell_0}} \widetilde{\mathsf{eq}}\left((w_{[(i+1):\ell_0]}, w_{[(\ell/2+\ell_0+1):]}), (y, x_{\mathsf{out}})\right)$$
$$\sum_{x_{\mathsf{in}} \in \{0,1\}^{\ell/2}} \widetilde{\mathsf{eq}}(w_{[(\ell_0+1):(\ell_0+\ell/2)]}, x_{\mathsf{in}}) \cdot \prod_{k=1}^{d} p_k((\boldsymbol{v}, u, y), x_{\mathsf{in}}, x_{\mathsf{out}})$$

We change the order of summation, moving the sum over x_{out} to the outermost position. Then, instead of summing over the specific prefix $(\boldsymbol{v}, u, y)$, we sum over all possible evaluation prefixes $\beta \in {U_d}^{\ell_0}$. For each such prefix, we use the $\mathsf{idx4}$ mapping (Definition A.5) to select only those prefixes that contribute to the target accumulator $\mathsf{A}_i(\boldsymbol{v}, u)$. Recall that $(i', \boldsymbol{v}', u', y) \in \mathsf{idx4}(\beta)$ if and only if $\beta = (\boldsymbol{v}', u', \beta_{i'+1}, \ldots, \beta_{\ell_0-1})$ and $y = (\beta_{i'+1}, \ldots, \beta_{\ell_0-1})$ is binary. Thus, summing over β and then filtering using $\mathsf{idx4}$ with the condition $i' = i, \boldsymbol{v}' = \boldsymbol{v}, u' = u$ is equivalent to summing over the original y for the fixed $(\boldsymbol{v}, u)$. The product term $\prod p_k$ needs to be evaluated at the full prefix β corresponding to $(\boldsymbol{v}, u, y)$, and the outer $\widetilde{\mathsf{eq}}$ term uses the index $i' = i$ from the $\mathsf{idx4}$ condition. This gives the rearranged form:

$$\mathsf{A}_i(\boldsymbol{v}, u) = \sum_{x_{\mathsf{out}} \in \{0,1\}^{\ell/2-\ell_0}} \sum_{\beta \in {U_d}^{\ell_0}} \sum_{\substack{(i', \boldsymbol{v}', u', y) \in \mathsf{idx4}(\beta) \\ \text{s.t. } i'=i, \boldsymbol{v}'=\boldsymbol{v}, u'=u}}$$
$$\widetilde{\mathsf{eq}}\left((w_{[(i'+1):\ell_0]}, w_{[(\ell/2+\ell_0+1):]}), (y, x_{\mathsf{out}})\right)$$
$$\cdot \left(\sum_{x_{\mathsf{in}} \in \{0,1\}^{\ell/2}} \widetilde{\mathsf{eq}}(w_{[(\ell_0+1):(\ell_0+\ell/2)]}, x_{\mathsf{in}}) \cdot \prod_{k=1}^{d} p_k(\beta, x_{\mathsf{in}}, x_{\mathsf{out}}) \right)$$

This matches exactly Equation (27). We use this rearranged form to analyze the algorithm.

Now, let's analyze the computation performed by Procedure 9. The algorithm iterates through each x_{out}. For a fixed x_{out}, the inner loops compute temporary values $\mathsf{tA}[\beta]$ for every prefix β. Specifically, after the loop over x_{in} completes, the value stored is:

$$\mathsf{tA}[\beta] = \sum_{x_{\mathsf{in}} \in \{0,1\}^{\ell/2}} \mathsf{E}_{\mathsf{in}}[x_{\mathsf{in}}] \cdot \prod_{k=1}^{d} p_k(\beta, x_{\mathsf{in}}, x_{\mathsf{out}})$$

where $\mathsf{E}_{\mathsf{in}}[x_{\mathsf{in}}] = \widetilde{\mathsf{eq}}(w_{[(\ell_0+1):(\ell_0+\ell/2)]}, x_{\mathsf{in}})$ and the evaluations $p_k(\beta, x_{\mathsf{in}}, x_{\mathsf{out}})$ are obtained using Procedure 6.

The next loops iterate over all β and distribute the computed $\mathsf{tA}[\beta]$ values to the final accumulators. For a given β, the algorithm finds all $(i', \boldsymbol{v}', u', y) \in \mathsf{idx4}(\beta)$. For each such tuple, it updates the accumulator $\mathsf{A}_{i'}(\boldsymbol{v}', u')$ by adding $\mathsf{E}_{\mathsf{out},i'}[y, x_{\mathsf{out}}] \cdot \mathsf{tA}[\beta]$.

Consider a specific final accumulator $\mathsf{A}_i(\boldsymbol{v}, u)$. Its value is the sum of contributions from all β such that $(i, \boldsymbol{v}, u, y) \in \mathsf{idx4}(\beta)$ for some y. Summing these contributions across all iterations of the outermost loop (over x_{out}), the final value accumulated in $\mathsf{A}_i(\boldsymbol{v}, u)$ is:

$$\sum_{x_{\mathsf{out}} \in \{0,1\}^{\ell/2-\ell_0}} \sum_{\beta \in {U_d}^{\ell_0}} \sum_{\substack{(i',\boldsymbol{v}',u',y) \in \mathsf{idx4}(\beta) \\ \text{s.t. } i'=i, \boldsymbol{v}'=\boldsymbol{v}, u'=u}} \mathsf{E}_{\mathsf{out},i'}[y, x_{\mathsf{out}}] \cdot \mathsf{tA}[\beta]$$

Substituting the expression for $\mathsf{tA}[\beta]$ into this sum, we obtain exactly Equation (27). Since we showed Equation (27) is equivalent to the original definition Equation (26), this demonstrates that Procedure 9 computes the correct values.

Lemma A.9. *Algorithm 6 is correct. In particular, the output of the sum-check prover in Algorithm 6 is always the same as the output of the sum-check prover in the standard linear-time algorithm (Algorithm 1).*

Proof. This follows from our discussion above and the correctness of the pre-computation routine in Lemma A.8.

B Details of Integration with Jolt

In this section, we discuss the integration of our optimizations into the Jolt zkVM [1].[23] Our optimizations primarily apply to the first sum-check invocation of Spartan [38], which is sped up with our combined optimizations described in Section 5.2.

While Spartan also has a second invocation of sum-check, it is not of the form in Equation (15), and so our optimizations do not apply. However, the second invocation admits a different optimization owing to the highly uniform nature of the constraint system arising in Jolt. We do not discuss this optimization further in this paper, and refer the interested reader to [2].

B.1 Spartan and Jolt Preliminaries

Recap: The Spartan protocol. Spartan [38] is a proof system for the R1CS relation, which takes the form

$$(A \cdot Z) \circ (B \cdot Z) = C \cdot Z,$$

[23] Our implementation builds upon the 'main' branch of Jolt as of May 15, 2025. This does not yet include the Twist and Shout integration.

where $A, B, C \in \mathbb{F}^{n\times m}$ are public matrices, $x \in \mathbb{F}^k$ is the public input, $v \in \mathbb{F}^{m-k-1}$ is the private witness, $Z = (1, x, v)$, and $\circ$ denotes Hadamard (element-wise) product. To prove this relation, Spartan invokes the sum-check protocol twice, with the first invocation applied to the polynomial:

$$G(X) := \widetilde{\mathsf{eq}}(w, X) \cdot \left(\overline{A}(X) \cdot \overline{B}(X) - \overline{C}(X)\right),$$

to prove that $\sum_{x\in\{0,1\}^n} G(x) = 0$. Here $w \in \mathbb{F}^\ell$ is a random challenge vector, and $\overline{A}(X), \overline{B}(X), \overline{C}(X)$ are the multilinear extensions of the vectors $A \cdot Z$, $B \cdot Z$, and $C \cdot Z$, respectively. For Jolt, the vector Z is an encoding of the execution trace of a 32-bit RISC-V program, and the matrices A, B, C encode the constraints of the VM.[24] The Jolt prover can thus compute the vectors $A \cdot Z$, $B \cdot Z$, and $C \cdot Z$ as the execution trace is built; these vectors have small entries that fit in a signed 64-bit integer, but as we will see, many of these entries will be zero or one.

Our algorithm specializes to the case of $G(X)$ as follows. Recall that in round $i \in [\ell]$, the prover's message is the polynomial

$$s_i(X) = \sum_{x'\in\{0,1\}^{\ell-i}} G(r_{[<i]}, X, x') = \mathsf{eq}(w_{[\le i]}, (r_{[<i]}, X)) \cdot t_i(X),$$

where

$$t_i(X) = \sum_{x'\in\{0,1\}^{\ell-i}} \widetilde{\mathsf{eq}}(w_{[>i]}, x') \cdot \\ \left(\overline{A}(r_{[<i]}, X, x') \cdot \overline{B}(r_{[<i]}, X, x') - \overline{C}(r_{[<i]}, X, x')\right).$$

In particular, since $t_i(X)$ is of degree 2, and further satisfies an equation based on $s_i(0) + s_i(1) = T_i$, we only need to compute two evaluations of $t_i(X)$ at 0 and ∞ (denoting $r_{[<i]}$ as r'):

$$\begin{aligned} t_i(0) = &\sum_{x'\in\{0,1\}^{\ell-i}} \widetilde{\mathsf{eq}}(w_{[>i]}, x') \cdot \\ &\quad \left(\overline{A}(r', 0, x') \cdot \overline{B}(r', 0, x') - \overline{C}(r', 0, x')\right), \\ t_i(\infty) = &\sum_{x'\in\{0,1\}^{\ell-i}} \widetilde{\mathsf{eq}}(w_{[>i]}, x') \cdot \\ &\quad \left(\overline{A}(r', 1, x') - \overline{A}(r', 0, x')\right) \cdot \left(\overline{B}(r', 1, x') - \overline{B}(r', 0, x')\right). \end{aligned}$$

Note that we do *not* need to compute the $\overline{C}$ term in $t_i(\infty)$, since it is of lower degree (see Section 2.2).

Overview of Spartan constraints.

For the Lasso-and-Spice version of Jolt, there are 68 constraints and 87 variables per cycle. These constraints are of the following form:

- Binary constraints: $X \cdot (1 - X) = 0$
- Conditional equality: if X then $Y = Z$
- If-else: if X then Y else Z
- Product: $X \cdot Y = Z$,

where X, Y, Z are linear combinations of the public inputs and private witness (i.e., the program trace). This is padded to 128 constraints per cycle (and the number of cycles is itself padded to a power of

[24] These constraints together with a lookup argument and a read-write memory-checking argument are used to ensure that the execution trace is correct.

two). For most of these constraints, the Az term is binary or have very small values (i.e. fits in an i8). In particular, the polynomials $p \in \{Az, Bz, Cz\}$ have the following form:

$$p\,(\underbrace{x_0, \ldots, x_6}_{\text{constraint index}},\ \underbrace{x_7, \ldots, x_\ell}_{\text{cycle index}}). \tag{29}$$

In the upcoming Twist and Shout version of Jolt, there will only be less than 32 constraints per cycle, due to a simpler arithmetization and the removal of all binary constraints (to be checked by a specialized sum-check).

B.2 Comparative Cost Analysis for Spartan Sum-Checks

We now provide a detailed comparative cost analysis for the prover in Spartan's first sum-check. We contrast Algorithm 5 (eq-polynomial optimization only, reflecting current Jolt estimates) with Algorithm 6 (combined eq-polynomial and small-value optimization, using $\ell_0 = 3$ small-value rounds). This analysis is based on the derivations in Section 6.2. We denote $N = 2^\ell$ as the size of the domain. Costs are measured in "small-large" ($\mathfrak{sl}$) and "large-large" ($\mathfrak{ll}$) field multiplications.

Cost of Algorithm 5 (current Jolt). The estimated cost for Algorithm 5, based on current Jolt implementation characteristics as detailed in Section 6.2, is composed of:

- **Round 1:** Approximately $3N\mathfrak{sl}$ multiplications. This is the estimated cost for the operations in the first round within the current Jolt setup.
- **Rounds 2 to ℓ:** Approximately $4.5N\mathfrak{ll}$ multiplications. This is the cumulative cost for the subsequent standard sum-check rounds, applied to the degree-2 polynomial $t_i(X)$ that arises in each round, as per current Jolt estimates.

Thus, the total estimated cost for Algorithm 5 is $3N\ \mathfrak{sl} + 4.5N\ \mathfrak{ll}$.

Cost of Algorithm 6 (with $\ell_0 = 3$). For Algorithm 6, with $\ell_0 = 3$ rounds of small-value precomputation, the costs derived in Section 6.2 are broken down as follows:

- **Precomputation (First 3 rounds):** Approximately $2.375N\ \mathfrak{sl}$ multiplications. This dominant cost arises from computing product terms $(\widetilde{Az}(\ldots) \cdot \widetilde{Bz}(\ldots))$ for the 3^{ℓ_0} generalized evaluation points of the first ℓ_0 variables, scaled by the sub-problem size $N/2^{\ell_0}$. Since, for the 2^{ℓ_0} points on the binary hypercube $\{0,1\}^{\ell_0}$, the R1CS relation implies the relevant sum (for $t_i(0)$ terms often related to $\overline{AB} - \overline{C}$) is zero, the primary computational effort is for the remaining $3^{\ell_0} - 2^{\ell_0}$ evaluation patterns (those involving at least one ∞). For $\ell_0 = 3$, this results in $(3^3 - 2^3) \cdot N/2^3 = (27-8)N/8 = 19N/8 = 2.375N\ \mathfrak{sl}$. These operations are $\mathfrak{sl}$ as they primarily involve products of MLEs of small-valued vectors derived from the R1CS instance Z.
- **Streaming Round (Round 4, i.e., round $\ell_0 + 1$):** The costs for this round are:
  - $2.5N\ \mathfrak{sl}$: This cost is for evaluating the necessary polynomial components (related to $\widetilde{Az}$, $\widetilde{Bz}$, $\widetilde{Cz}$, and their differences when an ∞ is involved in the current round's variable) after fixing the first ℓ_0 variables to the challenges $r_{[\ell_0]}$. Your derivations indicate this is for approximately 5 key evaluations summed over the remaining variables, needed to construct $s_{\ell_0+1}(X)$.
  - $N/8\ \mathfrak{ll}$: This accounts for large-large multiplications when computing terms like $(\overline{A}(r'_{[\ell_0]}, 1, x') - \overline{A}(r'_{[\ell_0]}, 0, x')) \cdot (\overline{B}(r'_{[\ell_0]}, 1, x') - \overline{B}(r'_{[\ell_0]}, 0, x'))$ for $t_{\ell_0+1}(\infty)$, summed over a domain of size $N/2^{\ell_0}$ (since one variable is fixed to 0/1 within this round, the sum is over $N/2^{\ell_0+1}$ but repeated twice for 0 and 1, effectively $N/2^{\ell_0}$ work for the differences). For $\ell_0 = 3$, this is $N/8\mathfrak{ll}$.
  - $3N/16\ \mathfrak{ll}$: This is the cost of binding the verifier's challenge r_{ℓ_0+1}. The polynomial $s_{\ell_0+1}(X)$ is degree-2 in X_{ℓ_0+1}. Binding its 3 coefficients (or equivalent operations) with the challenge involves

multiplications by Lagrange coefficients over the sub-domain of size $N/2^{\ell_0+1}$. For $\ell_0 = 3$, this is $3N/16\mathfrak{ll}$.

- **Rounds 5 to ℓ (i.e., rounds $\ell_0 + 2$ to ℓ):** Approximately $7N/8$ $\mathfrak{ll}$ multiplications. These are $\ell - (\ell_0 + 1)$ standard sum-check rounds for a degree-2 polynomial $t_k(X)$. For $\ell_0 = 3$, this results in $7N/8\mathfrak{ll}$. This sum arises from evaluating $t_k(X)$ at two points (0 and ∞) and performing challenge binding in each of these remaining rounds.

Summing these components, the total estimated cost for Algorithm 6 (with $\ell_0 = 3$) is $(2.375N + 2.5N)$ $\mathfrak{sl}$ + $(N/8 + 3N/16 + 7N/8)$ $\mathfrak{ll} = 4.875N$ $\mathfrak{sl} + (2N/16 + 3N/16 + 14N/16)$ $\mathfrak{ll} = 4.875N$ $\mathfrak{sl} + 19N/16$ $\mathfrak{ll}$, which simplifies to $4.875N$ $\mathfrak{sl} + 1.1875N$ $\mathfrak{ll}$.

Comparison. Comparing the two approaches:

- Algorithm 5 (current Jolt) costs: $3N$ $\mathfrak{sl} + 4.5N$ $\mathfrak{ll}$.
- Algorithm 6 (with $\ell_0 = 3$) costs: $4.875N$ $\mathfrak{sl} + 1.1875N$ $\mathfrak{ll}$.

Algorithm 6 achieves a significant reduction in $\mathfrak{ll}$ operations (from $4.5N$ down to $1.1875N$) at the expense of a moderate increase in $\mathfrak{sl}$ operations (from $3N$ up to $4.875N$). Given that $\mathfrak{ll}$ operations are typically substantially more expensive than $\mathfrak{sl}$ operations, this trade-off generally results in a notable overall improvement in prover efficiency.

B.3 Other Improvements to Jolt

We also list a number of small improvements to Jolt that come from our optimizations in this paper.

The first is the integration of optimized multiplication for small values (u64, i64, u128, and i128). See Section 7 for details of these optimizations. Our changes have been merged into Jolt and led to an end-to-end speedup of about $5 - 10\%$ on the standard Jolt benchmarks that comes with CI.

The second is the integration of Gruen's optimizations [28] for the sum-checks involving *interleaved* polynomials, which appear in *grand product* arguments in Lasso, and various sum-checks in the upcoming Twist and Shout. We will implement these optimizations in a future version of Jolt.

C Why Extension Fields are Necessary for Sum-Check

One might ask whether our small-value optimization is truly necessary, as one could perhaps work fully over the base field, and apply sequential or parallel repetition to boost soundness. Unfortunately, although such techniques can boost soundness in the *interactive* setting, for sum-check-based SNARKs they fail to do so when combined with the Fiat-Shamir transformation [25] to render the protocol non-interactive (at least, not without major performance overheads).

The case of sequential repetition is widely known. In particular, given a protocol with one or more rounds and $1/\mathsf{poly}(\lambda)$ soundness, repeating the protocol for any $k = \mathsf{poly}(\lambda)$ times and applying Fiat-Shamir results in a *completely insecure* non-interactive argument. This is because an adversary against the Fiat-Shamired version can simply guess the "bad" challenge for the first repetition via varying the first message, then do the same for the second repetition via varying the first message of the second repetition, and so on. This is referred to as a "grinding attack" on the Fiat-Shamired protocol.

Grinding attacks are also effectiving when the Fiat-Shamir transformation is applied to the parallel (rather than sequential) repetition of a multi-round protocol, unless the number of repetitions is very large. Specifically, when applying parallel repetition followed by Fiat-Shamir to an ℓ-round interactive protocol, $\ell \cdot k$ repetitions are necessary to amplify λ/k bits of security to λ bits of security (i.e., there is an attack demonstrating that this security bound is tight [5].[25]).

[25] See `https://a16zcrypto.com/posts/article/17-misconceptions-about-snarks/#section--13` for an exposition of this attack.

In the context of the sum-check protocol, this results in $O(nk \log n)$ base field operations for the prover, which is typically worse than the linear-time algorithm (our Algorithm 1) that achieves $O(nk^2)$ base field operations. Indeed, in practice, we have $k \leq 4$ (i.e. 64-bit inside BN254, or 32-bit subfield inside 128-bit extension fields) while $\log n \geq 20$.

Subsequent to a preliminary version of our work [6], Wei et al. [50] introduced a variation of parallel repetition that is secure when combined with the Fiat-Shamir repetition. In this variation, each round of the repeated protocol is followed by a "many-to-one reduction" that itself invokes the sum-check protocol run over an extension field. Hence, even in this variation, efficient implementations of the sum-check protocol run over extension fields are required. We refer to our related work section (Section 1.2) for additional details.