Sumcheck 201

Ingonyama

Contents

Chapter 1

Algorithms for Sumcheck Parallelization

Suyash Bagad
Ingonyama
suyash@ingonyama.com

Karthik Inbasekar
Ingonyama
karthik@ingonyama.com

Abstract. In this chapter, we explore sum-check protocols from the point of view of parallelizable computation in devices such as GPUs. We explore algorithmic modifications to the sum-check protocol, for products of Multi-Linear Extensions (MLE) to improve parallel compute and address memory bottlenecks.

1.1 Technical Introduction

In this section we review computational aspects of the sum-check protocol for Multi-Linear Extensions. We restrict our attention to the sum-check protocol from the point of view of the prover, for a more comprehensive treatment we refer to the book Proofs, Arguments and Zero-Knowledge by Thaler [1]. Algorithms 1 and 2 discussed in this section are well known algorithms (see [2]) in the literature. We recommend going through this section to get used to the notation in this paper. In particular, We have re-written algorithms 1 and 2 in the bit-index notation which is very convenient from an implementation point of view. Our main contributions (see §1.4, §1.5) are the algorithms 3 and 4 which generalize the sum-check algorithms to arbitrary products. The latter algorithm 4 in particular makes use of parallelizable tensor structures for efficient computation. We also present a proof of concept of these algorithms which is open sourced. [1]

We use the following notation to denote the n dimensional boolean hypercube $\{0,1\}^n$. The Lagrange basis in a boolean hypercube is defined as

$$\chi_{\vec{s}}(\vec{X}) = \prod_{i=1}^{n} \left(\bar{X}_i \cdot \bar{s}_i + X_i \cdot s_i\right), \tag{1.1.1}$$

[1] https://github.com/ingonyama-zk/super-sumcheck

where $\vec{s}_n(k) = \{s_{\mathsf{MSB}(k)}, \ldots, s_{\mathsf{LSB}(k)}\}$ represents the n bit representation of an integer k, $\vec{X}_n = \{X_n, X_{n-1}, \ldots X_1\}$, and $\bar{X}_i = 1 - X_i$ and $\bar{s}_i = 1 - s_i$. When $\vec{X} = \vec{s}' \in \{0,1\}^n$ the Lagrange basis satisfy the following relation

$$\chi_{\vec{s}}(\vec{s}') = \delta_{s,s'}. \tag{1.1.2}$$

For our purpose, we assume that $n = \log_2 N$. A Multi-Linear Extension (MLE) is a unique representation of a tuple (from $\mathbb{F}^{2^n}$ in our case) as evaluations of a polynomial on the boolean hypercube $\{0,1\}^n$. It is conveniently expressed in the Lagrange basis as follows

$$F(\vec{X}) = \sum_{k=1}^{2^n} \chi_{\vec{s}_n(k-1)}(\vec{X}) \cdot F(\vec{s}_n(k-1)). \tag{1.1.3}$$

The uniqueness property follows directly from the completeness of the Lagrange basis (1.1.2).

Consider a simple example, the tuple $\{f_0, f_1, f_2, f_3\} \in \mathbb{F}^4$ is represented on $\{0,1\}^2$ as as shown in table 1.1. The evaluations f_i for all $i \in \{0,1,2,3\}$ occupy vertices on the boolean hypercube (a square in 2 dimensions) as shown in fig. 1.1.

Evaluation	$F(\vec{s}_2(0)) = f_0$	$F(\vec{s}_2(1)) = f_1$	$F(\vec{s}_2(2)) = f_2$	$F(\vec{s}_2(3)) = f_3$
Bit representation of i	$\vec{s}_2(0) = \{0,0\}$	$\vec{s}_2(1) = \{0,1\}$	$\vec{s}_2(2) = \{1,0\}$	$\vec{s}_2(3) = \{1,1\}$

Table 1.1: Lagrange interpolation in boolean hypercube

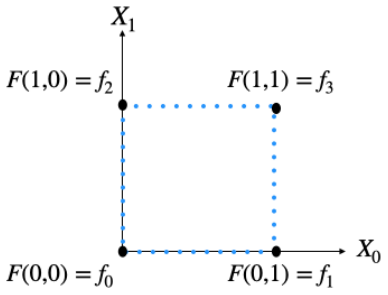

Figure 1.1: Evaluations of a Multilinear polynomial $F(X_2, X_1)$ on a boolean hypercube $\{0,1\}^2$.

$$\begin{aligned} F(X_2, X_1) &= \sum_{k=1}^{4} \chi_{\vec{s}_2(k-1)}(\vec{X}) \cdot F(\vec{s}_2(k-1)), \\ &= \bar{X}_2\bar{X}_1 \cdot f_0 + \bar{X}_2 X_1 \cdot f_1 + X_2\bar{X}_1 \cdot f_2 + X_2 X_1 \cdot f_3. \end{aligned} \tag{1.1.4}$$

The statement of the sum-check problem for an MLE on the boolean hypercube $\{0,1\}^n$ is defined as follows. The prover claims knowledge of a tuple $\mathbb{F}^{2^n}$, represented as evaluations

on $\{0,1\}^n$ such that the trace over all dimensions on the boolean hypercube is a unique invariant value $C \in \mathbb{F}$.

$$\sum_{X_i \in \{0,1\}} F(\vec{X}) = C \tag{1.1.5}$$

The most straightforward way to check the claim is to evaluate $F(\vec{X})$ on $\{0,1\}^n$. This costs the verifier to compute 2^n evaluations which is very expensive.

The sum-check protocol achieves verifier efficiency of $\mathcal{O}(n + E(\vec{\alpha}))$, where $E(\vec{\alpha}_n)$ is the cost of evaluating $F(\vec{X})$ on $\vec{\alpha}_n \in \mathbb{F}^n$. Assuming that the verifier has oracle access to $F(\vec{X})$, the protocol consists of n rounds. In round p the prover reduces the claim over a 2^{n-p+1} sized instance by folding it to a claim over a 2^{n-p} sized instance. We can use Lemma 1 in [2], to sequentially derive the computational steps in the sum-check protocol

$$F(X,Y) = \bar{Y} \cdot F(X,0) + Y \cdot F(X,1). \tag{1.1.6}$$

In the first round the prover folds the hypercube into the X_1 direction by performing 2^{n-1} boolean evaluations. The resultant univariate round polynomial $r_1(X)$ takes the form

$$\begin{aligned} r_1(X) &= \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-1}, X) \\ &= \bar{X} \cdot \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-1}, 0) + X \cdot \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-1}, 1) \\ &= \sum_{r=1}^{2} \left(\chi_{\vec{s}_1(r-1)}(X) \circ \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-1}, \vec{s}_1(r-1)) \right). \end{aligned} \tag{1.1.7}$$

where $\circ$ represents the Hadamard product, the first term is just the sum of all even indexed coefficients in $F(\vec{X})$ and the second term is the sum of all odd indexed coefficients in $F(\vec{X})$. The notation

$$\vec{X}_{n-t} = (X_n, X_{n-1}, \ldots X_{t+1}) \tag{1.1.8}$$

will be used throughout the article with $t \leq n-1$. The prover sends $r_1(X)$ to the verifier, who then checks

$$\sum_{X \in \{0,1\}} r_1(X) \stackrel{?}{=} C \tag{1.1.9}$$

The verifier then binds the variable X_1 with a random challenge α_1 and the protocol pro-

ceeds to the next round. The prover computes the second round polynomial $r_2(X)$ as

$$\begin{aligned} r_2(X) &= \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-2}, X, \alpha_1) \\ &= \bar{X} \cdot \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-2}, 0, \alpha_1) + X \cdot \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-2}, 1, \alpha_1) \\ &= \bar{X} \left(\bar{\alpha_1} \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-2}, 0, 0) + \alpha_1 \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-2}, 0, 1) \right) + \\ &\quad X \left(\bar{\alpha_1} \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-2}, 1, 0) + \alpha_1 \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-2}, 1, 1) \right) \\ &= \sum_{r=1}^{2^2} \left(\chi_{\vec{s}_2(r-1)}(X, \vec{\alpha}_1) \circ \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-2}, \vec{s}_2(r-1)) \right) \end{aligned} \tag{1.1.10}$$

We have written the computation explicitly, so that the computational structure stands out. In particular note that in the last line, the parts factorize into terms which depend on the challenges, and terms which depend only on the witness. The coefficients that do not depend on the challenges may be pre-computed and stored at the cost of increased memory.

In general, the folding in the pth round consists of folding the hybercube into the X_p direction by performing boolean evaluations over all directions $k \in [p+1, n]$. The resulting univariate polynomial has the structure

$$\begin{aligned} r_p(X) &= \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-p}, X, \vec{\alpha}_{p-1}) \\ &= \sum_{r=1}^{2^p} \left(\chi_{\vec{s}_p(r-1)}(X, \vec{\alpha}_{p-1}) \circ \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-p}, \vec{s}_p(r-1)) \right) \end{aligned} \tag{1.1.11}$$

where $\vec{\alpha}_{p-1} = \{\alpha_{p-1}, \ldots, \alpha_1\}$ is the challenge vector for the bounded variables in the previous rounds. The verifier then performs a boolean evaluation on the polynomial sent by the prover and checks

$$\sum_{X \in \{0,1\}} r_p(X) \stackrel{?}{=} r_{p-1}(\alpha_{p-1}) \tag{1.1.12}$$

The verifier then binds the X_p coordinate (X in the above equation) to a new challenge $\alpha_p \leftarrow \mathbb{F}$. This cycle continues till the nth round, where the prover sets

$$r_n(X) = F(X, \vec{\alpha}_{n-1}) \tag{1.1.13}$$

and sends it to the verifier. The verifier then checks the folding in the X_n direction with

$$\sum_{X \in \{0,1\}} r_n(X) \stackrel{?}{=} r_{n-1}(\alpha_{n-1}), \tag{1.1.14}$$

and checks the following relation with a single oracle query to $F(\vec{X})$

$$r_n(\alpha_n) \stackrel{?}{=} F(\vec{\alpha}_n). \tag{1.1.15}$$

The protocol is complete, the soundness bound is $\frac{n}{|\mathbb{F}|}$, since the prover sends a univariate polynomial of degree 1, in each of the n rounds.

1.2 Single MLE - Algorithm 1: Collapsing arrays

In this computational approach (memoization), the idea is to parallelize each round computation efficiently, while collapsing the vector sizes by half each round. Each consecutive round only depends on the previous round polynomial. The computation proceeds in a recursive manner as we describe below. We can write the polynomial in round 1 ((1.1.7) as an indexed vector

$$r_1(\vec{X}_{n-1}, X) = \bar{X} \cdot F(\vec{X}_{n-1}, 0) + X \cdot F(\vec{X}_{n-1}, 1) \tag{1.2.1}$$

and store it in an array of size 2^n, where we can identify the odd: $F(\vec{X}_{n-1}, 1)$ and even: $F(\vec{X}_{n-1}, 0)$ elements as consecutive elements in the array. Then we sum over $\{0,1\}^{n-1}$ to compute the first round polynomial

$$r_1(X) := \sum_{X_i \in \{0,1\}} r_1(\vec{X}_{n-1}, X) = \bar{X} \cdot \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-1}, 0) + X \cdot \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-1}, 1) \tag{1.2.2}$$

and send $r_1(X)$ to the verifier to obtain $\alpha_1 \in \mathbb{F}$. The computation of this sum can entirely be parallelized. As a preparatory step for the next round polynomial, we first compute

$$r_1(X_n, \ldots, X_2, \alpha_1) = \bar{\alpha}_1 \cdot F(X_n, \ldots, X_2, 0) + \alpha_1 \cdot F(X_n, \ldots, X_2, 1) \tag{1.2.3}$$

and store it in a 2^{n-1} dimensional array which we call the intermediate representation. Another way to think of this is that the elements of $r_1(X)$, indexed by the coordinates $X_n, \ldots, X_2$, form a 2^{n-1} dimensional hypercube, which can be used to compute the next round polynomial. We note that

$$\begin{aligned} r_2(\vec{X}_{n-2}, X, \alpha_1) = \bar{X} &\left(\bar{\alpha_1} \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-2}, 0, 0) + \alpha_1 \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-2}, 0, 1) \right) \\ &+ X \left(\bar{\alpha_1} \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-2}, 1, 0) + \alpha_1 \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-2}, 1, 1) \right) \\ &= \bar{X} \cdot r_1(\vec{X}_{n-2}, 0, \alpha_1) + X \cdot r_1(\vec{X}_{n-2}, 1, \alpha_1) \end{aligned} \tag{1.2.4}$$

can be entirely written in terms of (1.2.3) and we can discard the original 2^n array. In general, the round polynomial array for the p'th round can be expressed in terms of the recursive formula

$$r_p(\vec{X}_{n-p}, X, \vec{\alpha}_{p-1}) = \bar{X} \cdot r_{p-1}(\vec{X}_{n-p}, 0, \vec{\alpha}_{p-1}) + X \cdot r_{p-1}(\vec{X}_{n-p}, 1, \vec{\alpha}_{p-1}) \tag{1.2.5}$$

where once again the coefficients of $\bar{X}$ and X are consecutive even and odd indexed elements in the 2^{n-p} dimensional array. This temporary array must be stored in memory, to be

used for the next round. The prover can compute the round polynomial $r_p(X)$ from this temporary array as

$$r_p(X) = \sum_{X_i \in \{0,1\}} r_p(\vec{X}_{n-p}, X, \vec{\alpha}_{p-1}). \tag{1.2.6}$$

Formula (1.2.5) implies that the new round polynomial only depends on the intermediate representation. The sum-check algorithm that utilizes (1.2.5) to compute the round polynomials is summarized in algorithm 1. The features of algorithm 1 are:

- The computation in each round is highly parallelizable in Phase 1 (Accumulation) and Phase 3 (Intermediate representation). The compute cost is

$$\begin{aligned} &\textbf{Phase 1:}\ 2(2^n - n - 1)A \\ &\textbf{Phase 3:}\ 4(2^{n-1} - 1)M, 2(2^{n-1} - 1)A \end{aligned} \tag{1.2.7}$$

  where M and A stand for field multiplications and field additions respectively.

- The memory bound in the first round is 2^n and decreases by half in each subsequent round. Hence the compute problem is bounded by instance size.

- At the end of Phase 3 in round p, we need only a 2^{n-p} dimensional intermediate representation to compute round polynomial of the $p+1$'th round, which can be done with in-place computation efficiently.

- A disadvantage of the method is that device utilization decreases as the number of rounds increases.

- Algorithm 1 and its generalizations for product have an additional challenge when small fields are used. Especially in the intermediate representation (line 17). In small fields < 64 bit the challenges are in a suitable extension field. Hence in Phase 3 the t_{inter} computation after round 1 is always in the extension fields domain with only extension field - extension field multiplications (1.2.7). This can cause algorithm 1 (and its generalization algorithm 3) to be less effective for small fields since memory is not the primary bottleneck.

We illustrate algorithm 1 in a simple example as shown in fig. 1.2. Consider an MLE on a 3 dimensional boolean hypercube as defined below,

$$\begin{aligned} F(X_3, X_2, X_1) = f_0 \cdot \bar{X}_3\bar{X}_2\bar{X}_1 + f_1 \cdot \bar{X}_3\bar{X}_2X_1 + f_2 \cdot \bar{X}_3X_2\bar{X}_1 + f_3 \cdot \bar{X}_3X_2X_1 + \\ f_4 \cdot X_3\bar{X}_2\bar{X}_1 + f_5 \cdot X_3\bar{X}_2X_1 + f_6 \cdot X_3X_2\bar{X}_1 + f_7 \cdot X_3X_2X_1 \end{aligned} \tag{1.2.8}$$

The round polynomial in the first round is given by

$$\begin{aligned} r_1(X) &= \sum_{X_i \in \{0,1\}} F(X_3, X_2, X) = \sum_{i=0}^{3} r_1^{(i)}(X) \\ r_1^{(i)}(X) &= f_{2i} \cdot \bar{X} + f_{2i+1} \cdot X \end{aligned} \tag{1.2.9}$$

Algorithm 1 MLE recursive Sumcheck Algorithm 1

Input: MLE $F(X_n, \ldots, X_1)$ of 2^n evaluations $\{f_0, f_1, \ldots f_{2^n-1}\}$ and C claimed sum.
let $T = [\mathsf{public}]$
let $t \leftarrow [f_0, f_1, \ldots, f_{2^n-1}]$ ▷ Initial size 2^n
let $\alpha_0 \leftarrow \mathsf{hash}(T, C)$
for $p = 1, 2 \ldots, n$ **do** ▷ rounds
    Phase 1: Accumulation
    let $r_p := [0, 0]$ ▷ Two evals for a linear round poly
    for $i = 0, 1, \ldots, \frac{\mathsf{len}(t)}{2} - 1$ **do**
        $r_p[0]$ += $t[2i]$ ▷ Accumulate even coeff
        $r_p[1]$ += $t[2i+1]$ ▷ Accumulate odd coeff
    end for
    Phase 2: Challenge generation
    $T.\mathsf{append}(r_p)$
    $\alpha_p \leftarrow \mathsf{hash}(\alpha_{p-1}, r_p)$
    Phase 3 : Intermediate representation
    for $i = 0, 1, \ldots, \frac{\mathsf{len}(t)}{2} - 1$ **do**
        $t_{\mathsf{inter}}[i] = \bar{\alpha}_p \cdot t[2i] + \alpha_p \cdot t[2i+1]$
    end for
    $t \leftarrow t_{\mathsf{inter}}$ ▷ update
end for
return T

After substituting $X = \alpha_1$, we represent $r_1^{(i)}(\alpha_1)$ as entries on the vertex of a 2 dimensional boolean hypercube or equivalently a 4 dimensional array, following which we can compute $r_2(X)$ directly from the above data as

$$r_2(X) = \sum_{X_i \in \{0,1\}} F(X_3, X, \alpha_1) = \sum_{i=0}^{1} r_2^{(i)}(X)$$
$$r_2^{(i)}(X) = r_1^{(2i)}(\alpha_1) \cdot \bar{X} + r_1^{(2i+1)}(\alpha_1) \cdot X \tag{1.2.10}$$

similarly we represent $r_2^{(i)}(\alpha_2)$ as a two dimensional array and compute

$$\begin{aligned} r_3(X) &= F(X, \alpha_2, \alpha_1) \\ &= r_2^{(0)}(\alpha_2) \cdot \bar{X} + r_2^{(1)}(\alpha_2) \cdot X \end{aligned} \tag{1.2.11}$$

as explained in fig. 1.2.

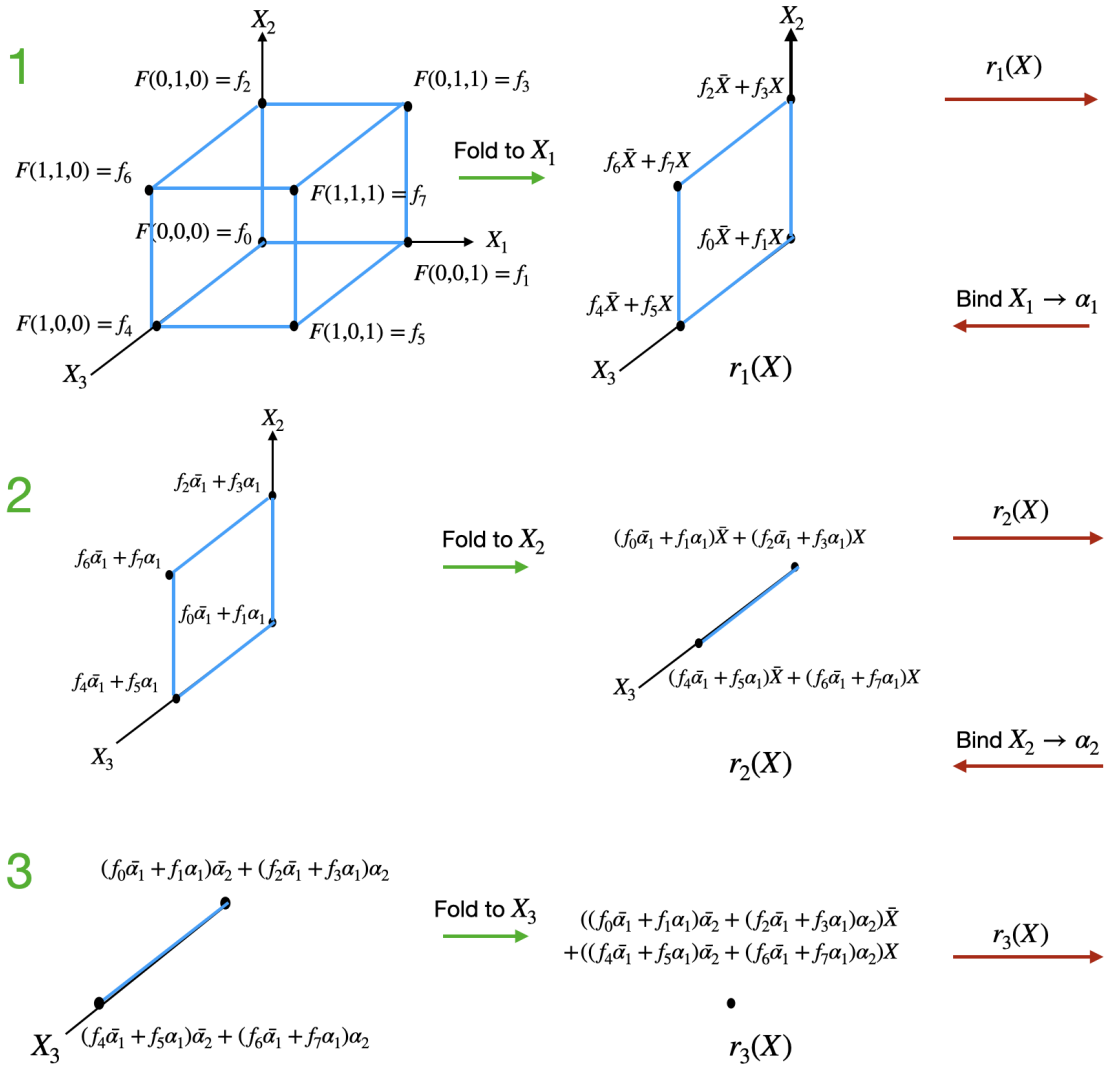

Figure 1.2: A schematic representation of algorithm 1 by recursion: Round polynomial computation for MLE $F(X_3, X_2, X_1)$. In each round, we trace over one dimension X_i in the boolean hypercube $\{0,1\}^3$, the round polynomial is the sum of all vertices of the folded cube in each round. The reduced MLE after the folding represent the intermediate arrays that can be used in the next round computation.

1.3 Single MLE - Algorithm 2: Precomputations

We notice that the round polynomial in the p'th round (1.3.1)

$$\begin{aligned} r_p(X) &= \sum_{X_i\in\{0,1\}} F(\vec{X}_{n-p}, X, \vec{\alpha}_{p-1}) \\ &= \sum_{r=1}^{2^p} \left(\chi_{\vec{s}_p(r-1)}(X, \vec{\alpha}_{p-1}) \circ \sum_{X_i\in\{0,1\}} F(\vec{X}_{n-p}, \vec{s}_p(r-1)) \right) \end{aligned} \tag{1.3.1}$$

is a sum of elements of the hadamard product of two 2^p dimensional vectors, one of which can be precomputed entirely before the sum-check protocol starts. We write the above equation as follows

$$r_p(X) = \sum_{r=1}^{2^p} \mathcal{F}_p[r] \circ \Xi_p[r] \tag{1.3.2}$$

The precomputation is performed as a memoized sum of coefficients necessary for the last round (input data) and recursively computing the rest of the coefficient arrays all the way to the first round. Following which, we store the n arrays $\mathcal{F}_p[r]$ indexed by $p = 1, \dots n$ in memory. Note that for each p, $\mathcal{F}_p[r]$ is of dimension 2^p.

$$\begin{aligned}
\mathcal{F}_{p-1}[r] = \sum_{X_i} F(\vec{X}_{n-p+1}, \vec{s}_{p-1}(r-1)) &= \sum_{X_{n-p+2}} \Bigg(\bar{X}_{n-p+2} \sum_{X_i} F(\vec{X}_{n-p-2}, 0, \vec{s}_{p-1}(r-1)) \\
&\qquad + X_{n-p+2} \sum_{X_i} F(\vec{X}_{n-p-2}, 1, \vec{s}_{p-1}(r-1)) \Bigg) \\
&= \sum_{X_i} F(\vec{X}_{n-p-2}, \vec{s}_p(r-1)) + \sum_{X_i} F(\vec{X}_{n-p-2}, \vec{s}_p(r-1+2^{p-1})) \\
&= \mathcal{F}_p(r) + \mathcal{F}_p(r+2^{p-1})
\end{aligned} \tag{1.3.3}$$

For the challenge vectors, we write a similar recursive relation for the arrays as shown below

$$\begin{aligned}
\Xi_p[r]\big|_{r=1}^{2^p}(X, \vec{\alpha}_{p-1}) &= \left\{\chi_{\vec{s}_p(r-1)}(X, \vec{\alpha}_{p-1})\right\}_{r=1}^{2^p} \\
&= \left(\bar{X} \cdot \left\{\chi_{\vec{s}_{p-1}(r-1)}(\vec{\alpha}_{p-1})\right\}_{r=1}^{2^{p-1}}, X \cdot \left\{\chi_{\vec{s}_{p-1}(r-1)}(\vec{\alpha}_{p-1})\right\}_{r=1}^{2^{p-1}}\right) \\
&= \left(\bar{X} \cdot \Xi_{p-1}(\vec{\alpha}_{p-1}), X \cdot \Xi_{p-1}(\vec{\alpha}_{p-1})\right)
\end{aligned} \tag{1.3.4}$$

For the challenge vectors the relation is of size 2^p in round p. Note that the challenges are generated only after the commitment of each round polynomial, and the recursion can be used only after obtaining the challenge. Algorithm 2 summarizes the computation of the round polynomials using (1.3.3) and (1.3.4). The main features of algorithm (2) are

- The precompute, Hadamard accumulator and Lagrange updater phase are all parallelizable. The compute cost is

$$\begin{aligned}
&\textbf{Phase 1:} \;\; 4(2^n - 1)M, 2(2^n - n - 1)A \\
&\textbf{Phase 3:} \;\; 2(2^{n-1} - 1)M
\end{aligned} \tag{1.3.5}$$

  where M and A stand for field multiplications and field additions respectively.

- The coefficient vector $\mathcal{F}_p(r)$ requires a storage memory of 2^p in round p, and in total requires a memory of $2 \cdot (2^n - 1)$ for the entire protocol. The challenge vector grows as 2^p in round p, however only requires 2^{p-1} storage due to repetitions. As a result, the intermediate storage grows exponentially with the round. This makes the method highly unsuitable for large fields.

- The hadamard multiplications are 2^p per round, between the challenge vector and the coefficients vector. In small fields (< 64) bits, the challenges are always in extension fields. We can see that the total number of extension field-extension field multiplications that occur in phase 3 (1.3.5) are half that of algorithm 1 (1.2.7) and the trade off is with the base field extension field multiplications, which are less expensive. Algorithm (2) and its generalization algorithm (4) are more suited for small fields case.

Algorithm 2 MLE: Algorithm 2: Precompute

Input: MLE $F(X_n, \ldots, X_1)$ of 2^n evaluations $\{f_0, f_1, \ldots f_{2^n-1}\}$ and C claimed sum.
Phase 0: Precompute
$\mathcal{F}_n \leftarrow \{f_0, f_1, \ldots f_{2^n-1}\}$
for $p = n, \ldots, 2$ **do**
    for $r = 1, \ldots, 2^{p-1}$ **do**
        $\mathcal{F}_{p-1}[r] \leftarrow \mathcal{F}_p[r] + \mathcal{F}_p[r + 2^{p-1}]$
    end for
end for
let $\alpha_0 \leftarrow \mathsf{hash}(T[0], C)$
let $\Xi_1(c)[\,] = [\bar{X}, X]$ ▷ Function definition, X are place holders
for $p = 1, 2 \ldots, n$ **do** ▷ rounds
    Phase 1 : Hadamard product accumulator
    for $r = 1, \ldots, 2^p$ **do**
        $r_p(X)[r]$ += $\Xi_p(X)[r] \cdot \mathcal{F}_p[r]$
    end for
    Phase 2: Challenge generation
    $T.\mathsf{append}(r_p)$
    $\alpha_p \leftarrow \mathsf{hash}(\alpha_{p-1}, r_p)$
    Phase 3 : Lagrange updater
    $\Xi_{p+1}(X)[\,] \leftarrow [\bar{X} \cdot \Xi_p(\alpha_p) \;||\; X \cdot \Xi_p(\alpha_p)]$ ▷ Memoize Lagrange base update
end for
return T

We illustrate relations (1.3.3) , (1.3.4) and algorithm 2 using a simple example which is shown in figure 1.3 for the polynomial $F(X_3, X_2, X_1)$ Below we write coefficients array

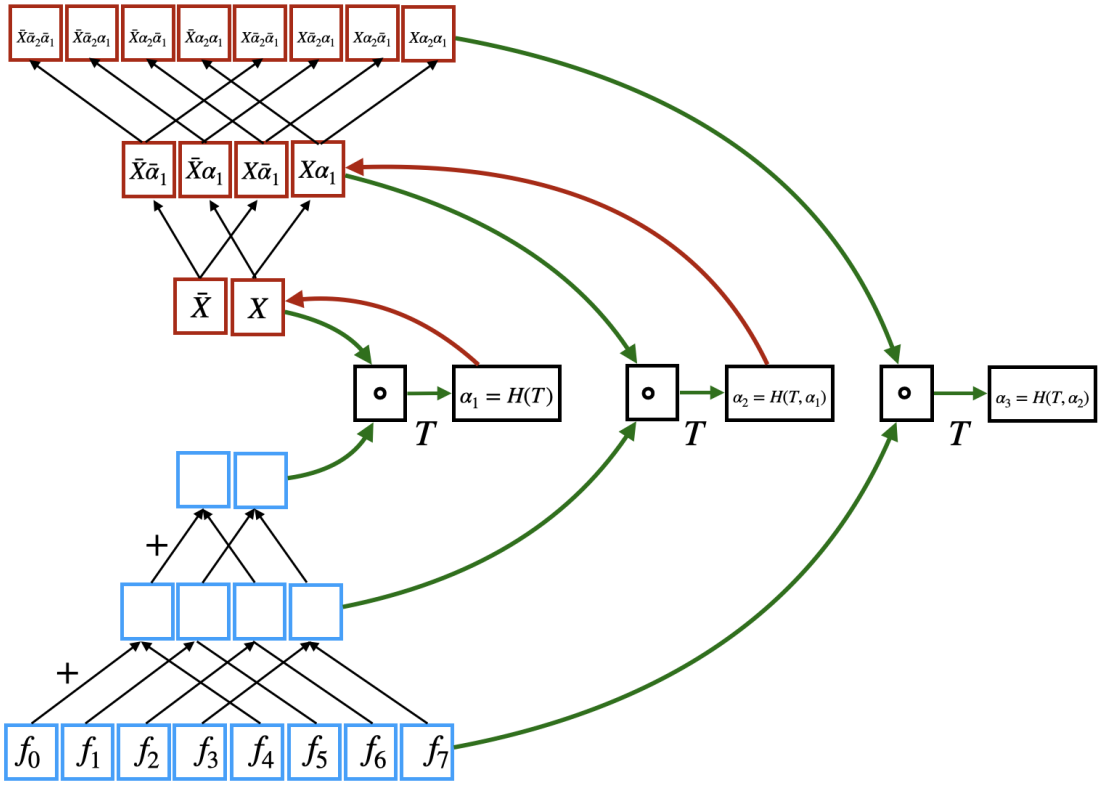

Figure 1.3: A schematic representation of algorithm 2 for the sum-check prover computation for $F(X_3, X_2, X_1)$. The blue boxes represent the precompute data, the arrows connecting other blue boxes represent the memoization which can parallelize the process. The red boxes, represent the challenge vectors, which are computed after the challenges are generated in a parallelized manner. The $\circ$ is the hadamard computation which is also done in parallel compute.

for $p = 1$ in terms of coefficients of $p = 2$

$$\begin{aligned}
\mathcal{F}_1[r]\big|_{r=1}^{2} = \sum_{X_i} F(\vec{X}_2, \vec{s}_1(r-1)) &= \left\{ \sum_{X_i} F(\vec{X}_2, 0), \sum_{X_i} F(\vec{X}_2, 1) \right\} \\
&= \Big\{ \sum_{X_2} \bar{X}_2 \cdot \sum_{X_i} F(\vec{X}_1, 0, 0) + \sum_{X_2} X_2 \cdot \sum_{X_i} F(\vec{X}_1, 1, 0) \\
&\quad , \sum_{X_2} \cdot \bar{X}_2 \sum_{X_i} F(\vec{X}_1, 0, 1) + \sum_{X_2} X_2 \cdot \sum_{X_i} F(\vec{X}_1, 1, 1) \Big\} \\
&= \{\mathcal{F}_2(1) + \mathcal{F}_2(3), \mathcal{F}_2(2) + \mathcal{F}_2(4)\} \qquad (1.3.6)
\end{aligned}$$

where the last line is written in terms of coefficients for $p = 2$. We can continue this

recursion further and write each of these in terms of coefficients for $p = 3$

$$\begin{aligned}\mathcal{F}_2[r]\big|_{r=1}^{2^2} = \sum_{X_i} F(\vec{X}_1, \vec{s}_2(r-1)) = \Big\{ &\sum_{X_i} F(\vec{X}_1, 0, 0), \sum_{X_i} F(\vec{X}_1, 0, 1) \\ &, \sum_{X_i} F(\vec{X}_1, 1, 0), \sum_{X_i} F(\vec{X}_1, 1, 1) \Big\} \\ &= \{\mathcal{F}_3(1) + \mathcal{F}_3(5), \mathcal{F}_3(2) + \mathcal{F}_3(6), \mathcal{F}_3(3) + \mathcal{F}_3(7), \mathcal{F}_3(4) + \mathcal{F}_3(8)\}\end{aligned} \tag{1.3.7}$$

The last round coefficients are just the input data

$$\mathcal{F}_3[r]\big|_{r=1}^{2^3} = (f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7) \tag{1.3.8}$$

Thus in general we can write, the coefficient vector for the $p-1$ round in terms of memoized sum of a vector of coefficients for the pth round. Similarly for the challenge vectors we begin with round 1, for $p = 1$ and since there are no challenges

$$\Xi_1(X)[r]\big|_{r=1}^{2} = \left\{\chi_{\vec{s}_1(0)}(X), \chi_{\vec{s}_1(1)}(X)\right\} = (\bar{X}, X) \tag{1.3.9}$$

where we set $\chi_{\vec{s}_0(0)}(\vec{\alpha}_0) = 1$. After committing the round polynomial $r_1(X)$ the prover recieves α_1 and computes $\Xi_1(X)[\alpha_1]$ which is used to construct

$$\begin{aligned}\Xi_2(X, \alpha_1)[r]\big|_{r=1}^{4} &= \left\{\chi_{\vec{s}_2(0)}(X, \alpha_1), \chi_{\vec{s}_2(1)}(X, \alpha_1), \chi_{\vec{s}_2(2)}(X, \alpha_1), \chi_{\vec{s}_2(3)}(X, \alpha_1)\right\} \\ &= \left(\bar{X} \cdot \bar{\alpha}_1, \bar{X} \cdot \alpha_1, X \cdot \bar{\alpha}_1, X \cdot \alpha_1\right) \\ &= \left(\bar{X} \cdot \Xi_1(\alpha_1), X \cdot \Xi_1(\alpha_1)\right)\end{aligned} \tag{1.3.10}$$

Similarly the prover computes $\Xi_2(\alpha_2, \alpha_1)$ and constructs the challenge vector for $p = 3$

$$\begin{aligned}\Xi_3(X, \vec{\alpha}_2)[r]\big|_{r=1}^{8} &= \left\{\chi_{\vec{s}_3(0)}(X, \vec{\alpha}_2), \chi_{\vec{s}_3(1)}(X, \vec{\alpha}_2), \chi_{\vec{s}_3(2)}(X, \vec{\alpha}_2), \chi_{\vec{s}_3(3)}(X, \vec{\alpha}_2)\right\} \\ &= \left(\bar{X} \cdot \bar{\alpha}_2\bar{\alpha}_1, \bar{X} \cdot \bar{\alpha}_2\alpha_1, \bar{X} \cdot \alpha_2\bar{\alpha}_1, \bar{X} \cdot \alpha_2\alpha_1, X \cdot \bar{\alpha}_2\bar{\alpha}_1, X \cdot \bar{\alpha}_2\alpha_1, X \cdot \alpha_2\bar{\alpha}_1, X \cdot \alpha_2\alpha_1\right) \\ &= \left(\bar{X} \cdot \Xi_2(\vec{\alpha}_2), X \cdot \Xi_2(\vec{\alpha}_2)\right)\end{aligned} \tag{1.3.11}$$

1.4 Product MLE -Algorithm 3: Recursive product sum-check

The discussion in the previous sections §1.2, §1.3 can be easily generalized to arbitrary products of multilinear polynomials. While this certainly increases the memory, it also introduces new computational structures that improve parallelism.

We assume that the MLE's are defined on the same boolean hypercube of dimension $\{0, 1\}^n$. In general, we can use any combine rule to define an arbitrary expansion of products

$$\begin{aligned}\mathsf{combine}(F_1(\vec{X}), F_2(\vec{X}), F_3(\vec{X}), F_4(\vec{X})) =& F_1(\vec{X}) \cdot F_2(\vec{X}) \cdot F_3(\vec{X}) \cdot F_4(\vec{X}) + F_1(\vec{X}) \cdot F_2(\vec{X}) \cdot F_3(\vec{X}) \\ &+ F_1(\vec{X}) \cdot F_2(\vec{X}) + F_1(\vec{X})\end{aligned} \tag{1.4.1}$$

another example which is used in Spartan [3] is

$$\mathsf{combine}(F_1(\vec{X}), F_2(\vec{X}), F_3(\vec{X}), F_4(\vec{X})) = F_1(\vec{X}) \cdot F_2(\vec{X}) \cdot F_4(\vec{X}) - F_3(\vec{X}) \cdot F_4(\vec{X}) \tag{1.4.2}$$

One can imagine that this defines a "gate" in the MLE language. The generalization of algorithm 1 to the product case is straightforward and less interesting, one provides the combine function, expand and evaluate term by term and write to transcript the $m+1$ evaluations for m MLE products using intermediate representation. We refer to Algorithm 3 below for the full details

Algorithm 3 Recursive Sumcheck for product MLE Algorithm 3

Define: $C = \sum_{0,1}^{n} \mathsf{combine}(F_1, F_2, \ldots, F_m)$
Input: m MLE's $n = 2^k, F_1, F_2, \ldots F_m, C, \mathsf{combine}(F_1, F_2, \ldots, F_m)$
let $T = [\mathsf{public}]$
let $\alpha_0 \leftarrow \mathsf{hash}(T[0], C)$
for $p = 1, 2 \ldots, n$ **do** ▷ rounds
    Phase 1: Accumulation
    let $r_p := [0 : m+1]$ ▷ $m+1$ evals for a deg m round poly
    for $k = 0, 1, \ldots, m$ **do**
        for $i = 0, 1, \ldots, 2^{n-p-1} - 1$ **do**
            $r_p[k] = \mathsf{combine}\left(\{F_l[i] \cdot (1-k) + F_l[i + 2^{n-p-1}] \cdot k\}_{l \in [m]}\right)$
        end for
    end for
    Phase 2: Challenge generation
    $T.\mathsf{append}(r_p)$
    $\alpha_p \leftarrow \mathsf{hash}(\alpha_{p-1}, r_p)$
    Phase 3: Intermediate representation
    for $i = 0, 1, \ldots, 2^{n-p-1} - 1$ **do**
        for $l = 0, 1, \ldots m - 1$ **do**
            $F_l[i] = \bar{\alpha}_p \cdot F_l[i] + \alpha_p \cdot F_l[i + 2^{n-p-1}]$ ▷ Update each polynomial array
        end for
    end for
end for
return T

1.5 Algorithm 4: Precomputations for product sum-check

The generalization of algorithm (2) for arbitrary products is more interesting because new computational tensor structures emerge, which assists in improving parallelism. For the purpose of this section we define a general product of m MLE's, generalizing (1.1.3). Note that this will generate a multivariate polynomial of degree m in each $X_i \in \vec{X}$

$$P_m(\vec{X}) = \prod_{j=1}^{m} \left(\sum_{k_j=1}^{2^n} \chi^{(j)}_{\vec{s}_n(k_j-1)}(\vec{X}) \cdot F^{(j)}(\vec{s}_n(k_j-1)) \right) \tag{1.5.1}$$

Our discussion in this section is a generalization of the product sum-check algorithm 2 and 3 of [4], and the precompute algorithm 3 of [2] for product of 2 and 3 MLE's respectively.

In both cases, the round polynomials are represented in terms of the evaluations of the individual MLE's in the product. We generalize it to a product of m MLE's and express the computation as a tensor product, which improves parallelism significantly.

The round polynomial $r_p(X)$ for the m product sum-check is a univariate polynomial of degree m, and requires the prover to compute $m+1$ evaluations

$$\begin{aligned} r_p(X) &= \prod_{j=1}^{m} \sum_{X_i \in \{0,1\}} F^{(j)}(\vec{X}_{n-p}, X, \vec{\alpha}_{p-1}) \\ &= \prod_{j=1}^{m} \sum_{r_j=1}^{2^p} \left(\chi^{(j)}_{\vec{s}_p(r_j-1)}(X, \vec{\alpha}_{p-1}) \circ \sum_{X_i \in \{0,1\}} F^{(j)}(\vec{X}_{n-p}, \vec{s}_p(r_j-1)) \right) \end{aligned} \tag{1.5.2}$$

We note that the vectors that participate in the hadamard product are of dimension $2^{m \cdot p}$, so even for a few products, the size and memory can go to unmanageable levels on memory-constrained devices as the rounds proceed. Nevertheless, equation (1.5.2) has a tensor structure when re-expressed as follows, can be used to parallely compute several intermediate steps. First, we express the coefficient vector as a tensor $\in \mathbb{F}^{2^p \times 2^{n-p}}$ i.e, as p slices each of width 2^{n-p}

$$\mathcal{F}_p^{(j)} \equiv \sum_{X_i \in \{0,1\}} F^{(j)}(\vec{X}_{n-p}, \vec{s}_p(r_j-1))\Big|_{r_j=1}^{2^p} = \begin{pmatrix} F^{(j)}[0 : 2^{n-p}] \\ F^{(j)}[2^{n-p} : 2^{n-p+1}] \\ \vdots \\ F^{(j)}[2^n - 2^{n-p+1} : 2^n - 2^{n-p}] \\ F^{(j)}[2^n - 2^{n-p} : 2^n] \end{pmatrix}_{2^p \times 2^{n-p}} \tag{1.5.3}$$

and the challenge vector (since it is the same for each term of the j products in (1.5.2)) (1.3.4) as

$$\begin{aligned} \Gamma_p \equiv \Xi_p[r]\big|_{r=1}^{2^p}(X, \vec{\alpha}_{p-1}) &= \left\{ \chi_{\vec{s}_p(r-1)}(X, \vec{\alpha}_{p-1}) \right\}_{r=1}^{2^p} \\ &= (\bar{X}, X) \otimes \Xi_{p-1}(\vec{\alpha}_{p-1}) \end{aligned} \tag{1.5.4}$$

and rewrite (1.5.2) as

$$r_p(X) = \left([\mathcal{F}_p^{(1)} \circledast \mathcal{F}_p^{(2)} \circledast \ldots \circledast \mathcal{F}_p^{(m-1)}]_{2^{(m-1)p} \times 2^{n-p}} \cdot [\mathcal{F}_p^{(m)}]^t_{2^{n-p} \times 2^p} \right) \boxdot [\Gamma_p \otimes \Gamma_p \ldots \otimes \Gamma_p]_{2^{(m-1)p} \times 2^p} \tag{1.5.5}$$

where $\cdot$ represents matrix product, $\boxdot$ represents the inner product (element wise multiplication and sum) between the two matrices of dimension $2^{(m-1)p} \times 2^p$, $\otimes$ the tensor product,

and $\circledast : \mathbb{F}^{m_1 \times n} \times \mathbb{F}^{m_2 \times n} \longrightarrow \mathbb{F}^{m_1 m_2 \times n}$ defined as:

$$P \circledast Q := \begin{bmatrix} \dots \vec{p}_1 \dots \\ \dots \vec{p}_2 \dots \\ \vdots \\ \dots \vec{p}_{m_1} \dots \end{bmatrix} \circledast \begin{bmatrix} \dots \vec{q}_1 \dots \\ \dots \vec{q}_2 \dots \\ \vdots \\ \dots \vec{q}_{m_2} \dots \end{bmatrix} = \left[\begin{array}{c} \dots \vec{p}_1 \circ \vec{q}_1 \dots \\ \dots \vec{p}_1 \circ \vec{q}_2 \dots \\ \vdots \\ \dots \vec{p}_1 \circ \vec{q}_{m_2} \dots \\ \hline \dots \vec{p}_2 \circ \vec{q}_1 \dots \\ \dots \vec{p}_2 \circ \vec{q}_2 \dots \\ \vdots \\ \dots \vec{p}_2 \circ \vec{q}_{m_2} \dots \\ \hline \vdots \\ \hline \dots \vec{p}_{m_1} \circ \vec{q}_1 \dots \\ \dots \vec{p}_{m_1} \circ \vec{q}_2 \dots \\ \vdots \\ \dots \vec{p}_{m_1} \circ \vec{q}_{m_2} \dots \end{array}\right].$$

Note that $\circledast$ operation is highly parallelizable. As before the coefficient matrix product can be precomputed. We present the full computation in algorithm 4. The main features are

- The entire coefficient tensor product can be precomputed and stored in memory if possible.
- From a GPU point of view, the key to performance is efficient async operations, where while the Phase 1: hadamard product accumulator runs, the coefficient array for the next round should be updated in memory.
- From a compute point of view, every intermediate step, save the commit phase is GPU friendly due to extensive parallelism.
- The memory access patterns have a very unique FFT like structure (see fig. 1.3). Although it is not easy to visualize this in the product case. This structure could be perhaps used for efficient coalesced memory access.

1.6 Reference Implementation

As a proof of concept, we implemented Algorithm 3 and Algorithm 4 in Rust to ensure correctness and compare the relative performances. The code is open-source[2]. Our implementation allows us to run the sum-check prover for any "gate" equation in MLE language (see Section 1.4). As an example, we have a test that demonstrates how we can run the sum-check protocol for R1CS equation from the Spartan paper [3] (see Equation (1.4.2)).

[2] https://github.com/ingonyama-zk/super-sumcheck

Algorithm 4 Algorithm 4: Precompute algorithm for product sum-check

Input: m MLE's $n = 2^k, F_1, F_2, \ldots F_m, C$
let $\alpha_0 \leftarrow \mathsf{hash}(T[0], C)$
let $A = [\ldots]_{2^{m-1} \times 2^{n-1}}, B = [\ldots]_{2^{n-1} \times 2}, \Gamma = [\ldots]_{2^{m-1} \times 2}$ ▷ Init for round 1

Phase 0: Precompute
for $l = 1, 2 \ldots m-1$ **do** ▷ Iterate over $m-1$ polynomials
    $A \leftarrow A \circledast \mathcal{F}_1^{(l)}$
end for
$B \leftarrow \mathcal{F}_1^{(m)}$
$AB \leftarrow A \cdot B^t$

for $p = 1, \ldots, n$ **do** ▷ Begin round polynomial computation
    let $r_p = [0; m+1]$ ▷ $m+1$ evaluations
    Phase 1: Hadamard product accumulator
    for $k = 0, 1, \ldots m$ **do**
        $\Gamma_p = \left(\otimes_{m-1} \begin{bmatrix} (1-k) \\ k \end{bmatrix}_{2\times 1} \right) \otimes \Gamma$ ▷ Challenge vector for the round
        $r_p[k] \leftarrow AB \boxdot \Gamma_p$
    end for
    $A.\mathsf{resizeBy}(2,2), B.\mathsf{resizeBy}(2,2)$

    for $l = 1, 2 \ldots, m$ **do** ▷ Update coefficients from memory, done in parallel
        $A \leftarrow A \circledast \mathcal{F}_{p+1}^{(l)}$
    end for
    $B \leftarrow \mathcal{F}_{p+1}^{(m)}$

    Phase 2: Challenge generation
    $T.\mathsf{append}(r_p)$
    $\alpha_p \leftarrow \mathsf{hash}(\alpha_{p-1}, r_p)$

    Phase 3: Lagrange updater
    $\Gamma.\mathsf{resizeBy}(2,2)$
    $\Gamma \leftarrow \Gamma \otimes \left(\otimes_{m-1} \right) \begin{bmatrix} (1-\alpha_p) \\ \alpha_p \end{bmatrix}_{2\times 1}$
end for
return T

The work on benchmarking the two algorithms with respect to speed and memory is still in progress. We also intend to release a multi-threaded version of the code to demonstrate parallelisability. We will be sharing both of these updates to the code in our next release.

Acknowledgments

We stand on the shoulders of giants. We would like to thank Yuval Domb and Tomer Solberg for discussions. We thank Omer Shlomovits for comments and suggestions on the draft.

Chapter 2

CheckFold: Folding Sumcheck Instances

Suyash Bagad
Ingonyama
suyash@ingonyama.com

Karthik Inbasekar
Ingonyama
karthik@ingonyama.com

Abstract. In this chapter, we construct a knowledge-reduction argument for the sum-check protocol that reduces the proving of a large sized sum-check to the proving of a folded small-sized sum-check. The folding argument is proven using a batched sum-check protocol that verifies the linear independence and satisfiability of each of the folded instances. The method is more suitable for large fields, where the problem is memory bound.

2.1 Introduction

In the previous chapter §1.1, we focused on identifying efficient parallizable compute components in the sum-check protocol. In implementations of algorithm 1 and its generalizations, we observed that the problem is memory bound by the instance size which implies worse communications complexity between host and device and has a significant impact on overall performance, especially in the case of large fields.

Using a folding scheme we can reduce memory bounds by folding {instance,witness} pairs such as R1CS folding in Nova/Hypernova [5, 6]. In our setting, the witness is the coefficients vector of a large MLE polynomial (or a product of many large MLE polynomials) and the instance is the sum-check statement on this polynomial (or products of polynomials). communications dealing with large instances, a typical problem is host-device communication. By folding the problem into sizes that minimize host-device communication or even reduces it to bare minimum required, one can achieve significant boost in performance. Using a suitable folding method it is possible to reduce large sum-check instances into smaller sum-check instances, thereby overcoming the instance size problem.

Folding schemes are a special case of application of algebraic knowledge reduction arguments (Definition 8 of [7]) which is a natural definition for our setting. Reductions in knowledge are defined over pairs (in general sequences) of relations $\mathcal{R}_1 = (u_1, w_1), \mathcal{R}_2 = (u_2, w_2)$ where u_i, w_i are statement/witness pairs. Starting from an initial statement u_1, the prover $\mathcal{P}$ and verifier $\mathcal{V}$ interact to generate statement u_2, with $\mathcal{P}$ providing a corresponding witness w_2 such that it satisfies $\mathcal{R}_2 = (u_2, w_2)$. The completeness property states that if $\mathcal{P}$ possesses a satisfying witness w_1 for the $\mathcal{V}$'s input statement u_1, $\mathcal{P}$ will output a satisfying witness w_2 for the $\mathcal{V}$'s statement u_2. The knowledge soundness property states that if the $\mathcal{P}$ provides a satisfying witness w_2 for the $\mathcal{V}$'s statement u_2, he/she knows a satisfying witness w_1 for the $\mathcal{V}$'s statement u_1 with overwhelming probability.

Below, we formally state the knowledge reduction argument for a MLE sum-check protocol.[1] Given an MLE sum-check instance,witness pair $\mathcal{R}_n = (u_n, w_n)$ where the suffix n is used to indicate a 2^n sized $\{instance, witness\}$ tuple. We define a knowledge reduced instance $\mathcal{R}_l^{folded} = (u_l^{folded}, w_l^{folded})$, where $0 < l < n$. An interactive public-coin protocol (which we call checkfold) reduces proving $\mathcal{R}_n$ to proving $\mathcal{R}_l^{folded}$. The interactive protocol is public coin and can be rendered non-interactive using the Fiat-Shamir protocol. Prover $\mathcal{P}$ sends to verifier $\mathcal{V}$ the folded witness w_l^{folded}. The completeness property guarantees that if the $\mathcal{P}$ has the witness w_n for the instance $\mathcal{R}_n$, the knowledge reduction argument will generate a valid witness w_l^{folded} for the instance $\mathcal{R}_l^{folded}$. Knowledge soundness property guarantees that a $\mathcal{P}$ who provided a satisfiable witness w_l^{folded} to the instance $\mathcal{R}_l^{folded}$ knows the witness w_n to $\mathcal{R}_n$ with high probability.

An important structure that facilitates this idea is the following. The sum-check protocol is known to be downward self-reducible (Lemma 4.1 of [8] and §2.4 of [9]). Hence large sumcheck instances can be written in terms of small sum-check instances using the union property

$$\mathcal{R}_n \equiv (u_n, w_n) = \bigcup_{i=1}^{2^l} (u_{n-l}^{(i)}, w_{n-l}^{(i)}) \tag{2.1.1}$$

Thus in order to apply a knowledge reduction argument it is sufficient to fold the instances

$$\mathcal{R}_l^{folded} \equiv (u_l^{folded}, w_l^{folded}) = \textbf{Fold}\left((u_{n-l}^{(1)}, w_{n-l}^{(1)}), \ldots, (u_{n-l}^{(2^l)}, w_{n-l}^{(2^l)})\right) \tag{2.1.2}$$

reducing the problem of verifying $\mathcal{R}_n$ to verifying $\mathcal{R}_l^{folded}$ where $0 < l < n$. The folding operation can be easily performed using a batched sum-check, since multiple MLE sum-check instances can be easily batched with a soundness error of $\frac{1}{|\mathbb{F}|}$ [10].

We can now explicitly realize the knowledge reducibility for sum-checks on a boolean hypercube in two steps. The first observation is that a MLE polynomial has a recursive structure as shown in Lemma 1 (2.1.3) that enables witness separability. Following which, an explicit realization of knowledge reduction (2.1.1) for MLE is illustrated by Lemma 2 (2.1.5).

[1] The MLE requirement can be relaxed here to have arbitrary multivariate polynomials, but more work is needed to prove that witness is separable as argued in the MLE case.

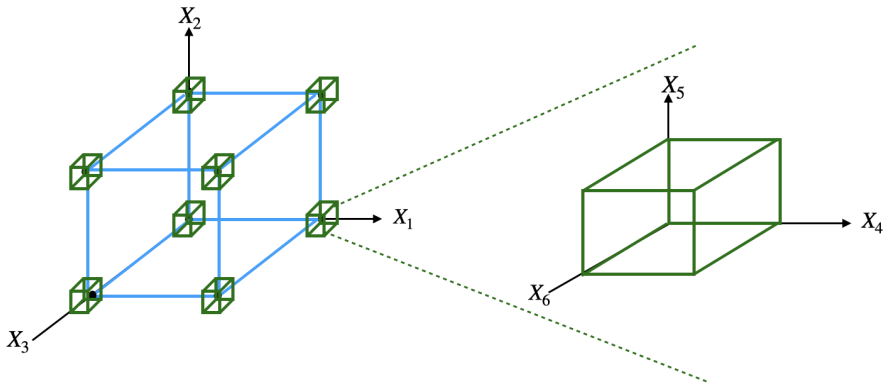

Figure 2.1: Expressing $\{0,1\}^6 \equiv \bigcup_{2^3}\{0,1\}^3$ we can write $\mathbb{F}^{2^6} \equiv \bigcup_{2^3}\mathbb{F}^{2^3}$. The polynomial $F(X_6,\dots X_1)$ is represented as $F(X_3,X_2,X_1)$ (blue cube) with polynomials $G_{k-1}(X_6,X_5,X_4)$ (green cubes) with $k=1,\dots,8$ on the vertices of the blue cube.

Lemma 1: Evaluations $\mathbb{F}^{2^n}$ on $\{0,1\}^n$ can be represented as evaluations $\bigcup_{2^l}\mathbb{F}^{2^{n-l}}$ on $\bigcup_{2^l}\{0,1\}^{n-l}$

$$F(\vec{X}) = \sum_{k=1}^{2^l} \chi_{\vec{s}_l(k-1)}(\vec{X}_l)\cdot G_{k-1}(\vec{X}_{n-l}) \tag{2.1.3}$$

where $\vec{X}_{a-b} = (X_a, X_{a-1},\dots,X_{b+1})$, the lagrange polynomials $\chi_{\vec{s}}(X))$ are defined in (1.1.1) and

$$G_{k-1}(\vec{X}_{n-l}) \equiv F(\vec{X}_{n-l}, \vec{s}_l(k-1)) \tag{2.1.4}$$

is defined on $\{0,1\}^{n-l}$ for each $k=1,2,\dots 2^l$. The proof follows from induction, by recursively applying the folding lemma 1 [2] on (1.1.3). As an example, see fig. 2.1 for a representation that writes $\mathbb{F}^{2^6} \equiv \bigcup_{2^3}\mathbb{F}^{2^3}$ represented on $\{0,1\}^6 \equiv \bigcup_{2^3}\{0,1\}^3$.

Lemma 2: A sumcheck size 2^n defined as $C \stackrel{?}{=} \sum_{\vec{X}\in\{0,1\}^n} F(\vec{X})$ can be represented as 2^l little sumcheck instances of size 2^{n-l} such that

$$C \stackrel{?}{=} \sum_{\vec{X}\in\{0,1\}^l}\sum_{k=1}^{2^l} \chi_{\vec{s}_l(k-1)}(\vec{X}_l)\cdot g_{k-1} \tag{2.1.5}$$

where

$$g_{k-1} \stackrel{?}{=} \sum_{\vec{X}\in\{0,1\}^{n-l}} G_{k-1}(\vec{X}_{n-l}) \;\forall\; k \;\in \{1,2,\dots,2^l\} \tag{2.1.6}$$

Proof of Lemma 2 follows directly by tracing over the $\{0,1\}^n$ boolean hypercube on (2.1.3). See fig 2.2 for a visual description of Lemma 2.

In the following sections, we propose a simple folding protocol "Checkfold" §2.2.1 that realizes the knowledge reducibility argument by folding. The extension to arbitrary product sum-checks is straightforward and is proposed in §2.2.2. In the simple algorithm the verifier

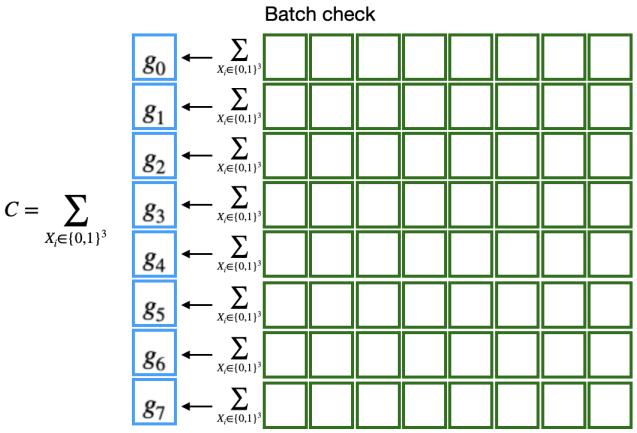

Figure 2.2: The sum-check representation of the folding, where each coefficient in the little sum-check (blue boxes) are themselves a result of the sum-checks (green boxes). The 8 green sum-checks can be batched together to verify the sums as well as their linear independence.

cost increases by $\mathcal{O}(2^l)$ for an l folding by the prover. In the case of arbitrary products of m MLE's this cost goes up by $\mathcal{O}(2^{lm})$. In §2.3.1 and §2.3.2 we propose an optimized version that reduces verifier cost to the original sum-check verifier $\mathcal{O}(n)$ and to that of checking a single pairing equation. The prover cost gain is essentially in the memory

2.2 Checkfold: Sum-Check folding

Lemma 2 reduces the claim (2.1.5) of instance size 2^n into 2^l claims (2.1.6) of instance size 2^{n-l} each. In this section we present the simplest version of the protocol that folds the 2^l instances into a single instance of size 2^{n-l}. The folding argument itself is proven using a sum-check protocol of instance size 2^{n-l}. Thus, the prover and verifier engage in a sum-check protocol with $n - l$ rounds, which is logarithmic in the folded instance size instead of the original instance size. Furthermore, the original instance is verified by using the proof elements of the folding, which increases the verifier work by $\mathcal{O}(2^l)$. The basic version of this protocol is a starting point for exploring the trade-off between proof size and number of rounds, which can be important if the sum-check is part of proof recursion. For eg, at the end of every round in a sum-check, there is an associated Fiat-Shamir challenge generation. Proving the challenge generation from a hash function in a recursive setting is expensive in general. Thus, reducing the number of rounds helps reduce the computational complexity in recursive proofs. The protocol is summarized below in §2.2.1

2.2.1 Protocol 1:

1. **Setup**

    - $\mathcal{P}$ sends the purported sum C to $\mathcal{V}$

- $\mathcal{P}$ arranges the coefficients of $F(\vec{X})$ into little MLE's $G_{k-1}(\vec{X}_{n-l})$ for $k = 1, 2, 3, \ldots 2^l$.

$$F(\vec{X}) = \sum_{k=1}^{2^l} \chi_{\vec{s_l}(k-1)}(\vec{X}_l) \cdot G_{k-1}(\vec{X}_{n-l}) \tag{2.2.1}$$

2. **Folding**

- $\mathcal{P}$ computes the partial sums

$$g_{k-1} = \sum_{\vec{X} \in \{0,1\}^{n-l}} G_{k-1}(\vec{X}_{n-l}) \ \forall \ k \ \in \{1, 2, \ldots, 2^l\} \tag{2.2.2}$$

  and sends the coefficients g_{k-1} to $\mathcal{V}$. We can modify this step later using commitments.

- $\mathcal{V}$ samples $\beta \leftarrow \mathbb{F}$ and sends it to $\mathcal{P}$.

- $\mathcal{P}$ receives β and folds (batches) the 2^l instances into a single instance by constructing the batched relations defined as

$$\begin{aligned} \tilde{F}(\vec{X}_{n-l}) &\equiv \sum_{k=1}^{2^l} \beta^{k-1} \cdot G_{k-1}(\vec{X}_{n-l}) \\ \tilde{C} &\equiv \sum_{k=1}^{2^l} g_{k-1} \cdot \beta^{k-1} \end{aligned} \tag{2.2.3}$$

  $\mathcal{V}$ gets oracle access to $\tilde{F}(\vec{X}_{n-l})$ (needed for last round verification). Note that $\mathcal{V}$ can construct $\tilde{C}$ on its own.

3. **Little sum-check**

- $\mathcal{P}$ and $\mathcal{V}$ engage in a "little sum-check protocol" that essentially generates a proof of the folding on $\{0,1\}^{n-l}$ in step 2 above. The protocol checks

$$\tilde{C} \stackrel{?}{=} \sum_{\vec{X} \in \{0,1\}^{n-l}} \tilde{F}(\vec{X}_{n-l}) \tag{2.2.4}$$

  using the standard sum-check protocol on $\{0,1\}^{n-l}$ which essentially checks the satisfiability of each of the 2^l sum-checks and their linear independence in one step

$$\sum_{k=1}^{2^l} g_{k-1} \cdot \beta^{k-1} \stackrel{?}{=} \sum_{\vec{X} \in \{0,1\}^{n-l}} \sum_{k=1}^{2^l} \beta^{k-1} \cdot G_{k-1}(\vec{X}_{n-l}) \tag{2.2.5}$$

- The number of rounds in the instance (2.2.4) is lesser due to the folding, for each $p = 1, 2, \ldots n - l$th round, $\mathcal{P}$ sends a round polynomial $r_p(X)$ to $\mathcal{V}$, and $\mathcal{V}$ responds with a random challenge $\alpha_p \leftarrow \mathbb{F}$.

4. **Soundness:** The folding and the little sum-check together have a soundness error of $\delta_1 \bigcup \delta_2 \leq \frac{(n-l)d}{|\mathbb{F}|} \bigcup \frac{\mathcal{O}(2^l)}{|\mathbb{F}|}$ where the $\frac{\mathcal{O}(2^l)}{|\mathbb{F}|}$ soundness error (which is negligible still for large fields) is due to the batching and d is the degree of the round polynomial. Note that if in step 2 folding if we require $\mathcal{V}$ to send a string of random variables $\{\beta_0, \ldots, \beta_{2^l}\}$, suitably generated after sending the coefficients g_k, then the batching error can be brought down to $\frac{1}{\mathbb{F}}$. However for large fields eg 256 bit prime fields, $\frac{\mathcal{O}(2^l)}{|\mathbb{F}|}$ is still negligible, so we do not pursue this direction. [2] This is yet another trade-off space in sum-check protocols.

   Since we consider sumcheck over linear combination of MLE polynomials, we have $d = 1$ in this case.

5. **Proof size:** The proof consists of

$$\pi = \{C, g_0, g_1, \ldots, g_{2^l-1}, r_1(X), r_2(X), \ldots, r_{n-l}(X)\} \tag{2.2.6}$$

   In the above C is the claimed sum, the folded coefficients g_{k-1} for $k = 1, 2 \ldots 2^l$ and the degree d round polynomials $r_p(X)$ for $p = 1, 2, \ldots, n-l$. The total proof consists of $(1 + 2^l + (d+1) \cdot (n-l))$ field elements $\mathbb{F}$, which seems to have an interesting trade-off space.

6. **Verifier**

   - The first step is to verify the folding. This is done by following the standard verifier algorithm for the "little sum-check protocol".
   - $\mathcal{V}$ takes the coefficients g_{k-1} from the proof and constructs

$$\tilde{C} = \sum_{k=1}^{2^l} g_{k-1} \cdot \beta^{k-1} \tag{2.2.7}$$

     $\mathcal{V}$ uses the above to check the round polynomials in the "little sum-check protocol"

$$\begin{aligned} \sum_{X \in 0,1,\ldots,d} r_1(X) &\stackrel{?}{=} \tilde{C} \\ \sum_{X \in 0,1,\ldots,d} r_i(X) &\stackrel{?}{=} r_{i-1}(\alpha_{i-1}) \quad \forall i = 2, 3, \ldots n-l \\ r_i(\alpha_i) &\stackrel{?}{=} \tilde{F}(\vec{\alpha}_{n-l}) \end{aligned} \tag{2.2.8}$$

     In the last round, i.e. $i = n - l$, $\mathcal{V}$ makes a single oracle query to $\tilde{F}(\vec{X}_{n-l})$.

[2] Basically the batching check $\sum_{i=0}^{d} \beta^i a_i \stackrel{?}{=} \sum_{i=0}^{d} \beta^i b_i$ has the soundness error $\leq d/|\mathbb{F}|$ from Schwarz-Zippel Lemma since it is equivalent to opening a polynomial at a random point β. Whereas $\sum_{i=0}^{d} \beta_i a_i \stackrel{?}{=} \sum_{i=0}^{d} \beta_i b_i$ has a soundness error of $1/|\mathbb{F}|$.

Data	**Sumcheck** algorithm (1)	**Checkfold** §2.2.1
Total Memory	$2 \cdot (2^n - 1)$	$3 \cdot 2^{n-l} + 2^{l+1} - 1$
Mult	$4 \cdot (2^{n-1} - 1)$	$2^{n-l} + 2^{l+1} + 2^n - 6$
Add	$3 \cdot 2^n - 2n - 4$	$2^n - 5 + 2^{n-l+1}(2^{l-1} + 1) - 2(n-l)$

Table 2.1: Prover cost comparison with the naive sumcheck protocol in terms of field elements

Step	**Mult**	**Add**	**Memory**
Folding: compute g_{k-1}	–	$2^l \cdot (2^{n-l} - 1)$	2^l
Folding: powers of β	$2^l - 1$	–	2^l
Folding: $\tilde{C}$	$2^l - 1$	$2^l - 1$	1
Folding: $\tilde{F}(\vec{X}_{n-l})$	$2^{n-l} \cdot (2^l - 1)$	$2^{n-l} \cdot (2^l - 1)$	2^{n-l}
little sumcheck	$4 \cdot (2^{n-l-1} - 1)$	$3 \cdot 2^{n-l} - 2(n-l) - 4$	$2 \cdot (2^{n-l} - 1)$

Table 2.2: Prover Cost breakdown for checkfold §2.2.1

- $\mathcal{V}$ still needs to check the original instance. For this, $\mathcal{V}$ interpolates $\{g_{k-1}\}$ on $\{0,1\}^l$ and does $\mathcal{O}(2^l)$ (naive) work to check the original instance

$$C \stackrel{?}{=} \sum_{\vec{X} \in \{0,1\}^l} \sum_{k=1}^{2^l} \chi_{\vec{s}_l(k-1)}(\vec{X}_l) \cdot g_{k-1} \tag{2.2.9}$$

  The verifier accepts if the check passes, and rejects if the check fails. Note that this part is just the partial sum computation and can be easily parallelized using methods discussed in previous chapter.

Note that in order to be efficient in the verifier, the parameter l needs to be chosen carefully. For every l fold, the prover can save memory up to 2^l since the instance are reduced by that amount. The prover cost breakdown for the checkfold protocol is summarized in table 2.2. The cost comparison to the original sum-check algorithm is summarized in table 2.1. As we can see from fig 2.3 the memory gain becomes significant with folding size l, the gain is exponential, with the optimal value for folding at $l \simeq n/2$. There is also a reduction in the multiplications and additions as the folding size increases as we can see from table 2.1, but the gain is less significant with the folding size.

Moreover, the number of rounds is reduced with the folding to $n-l$. Protocol §2.2.1 can be made non-interactive using the Fiat-Shamir transformation. On the last round of the "little sum-check" the verifier queries a polynomial of $n-l$ variables, instead of n variables, which reduces the verifier cost for the last round. On the other hand, for l folded rounds, the proof size and verifier work increases by $\mathcal{O}(2^l)$.

An important point to note is that if one uses small fields ≤ 64 bits for the witness, then the β sent by the verifier in step 2 is from an extension field (for soundness). This

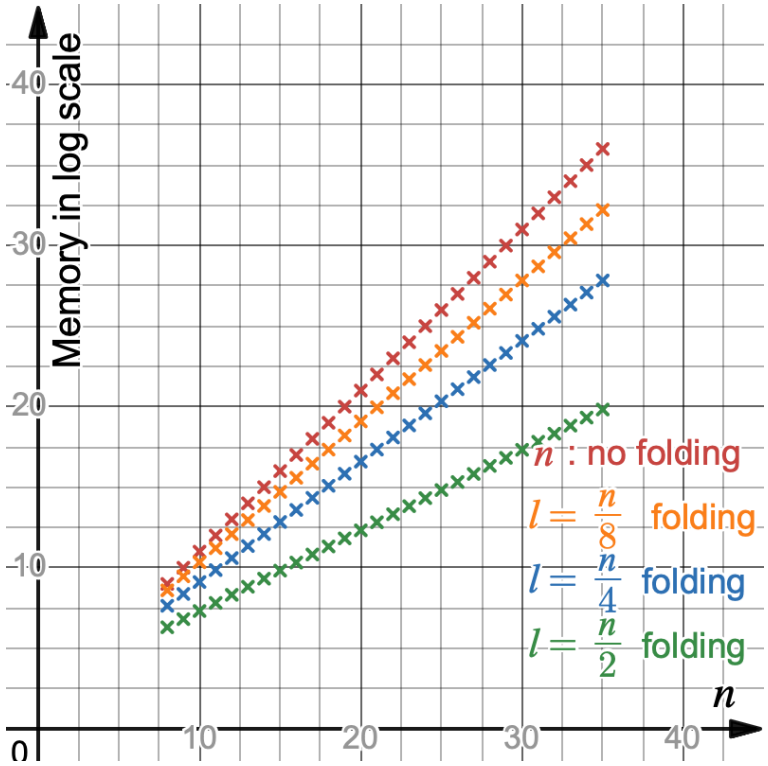

Figure 2.3: Total memory required for the Sumcheck and the Checkfold algorithms for different values of l. Note that the original instance size is 2^n, the number of folds is 2^l, and the size of the folded instance is 2^{n-l}

translates the entire folding step 2 and little sum-check step 3 into extension field domain, which is undesirable as it increases compute complexity, and the gains due to memory is not significant for small fields. For large fields, we don't need challenge generation from extension fields, and since the problem is memory bound to start with, the folding effectively resolves the bottleneck.

2.2.2 Protocol 1 for product MLE's:

In this section, we extend the folding scheme of §2.2.1 for arbitrary product MLE's. In this case for a product of m MLE's of size 2^n each, the memory bound due to instance size grows as $\mathcal{O}(2^{ml})$. The folding approach is significantly useful in these cases.

1. **Setup**

   - $\mathcal{P}$ sends the purported sum C to $\mathcal{V}$ such that

   $$C := \sum_{\vec{X} \in \{0,1\}^n} \prod_{j=1}^{m} F_j(\vec{X}).$$

   where m refers to the number of MLE's in the product.

- For each $j \in [m]$, the prover arranges the coefficients of $F_j(\vec{X})$ into little MLE's $G_{j,k-1}(\vec{X}_{n-l})$ for $k = 1, 2, 3, \ldots 2^l$.

$$F_j(\vec{X}) = \sum_{k=1}^{2^l} \chi_{\vec{s}_l(k-1)}(\vec{X}_l) \cdot G_{j,k-1}(\vec{X}_{n-l}) \tag{2.2.10}$$

2. **Folding**

- $\mathcal{P}$ computes the partial sums

$$g_{p-1} = \sum_{\vec{X} \in \{0,1\}^{n-l}} \prod_{j=1}^{m} G_{j,k_{p,j}}(\vec{X}_{n-l}) \tag{2.2.11}$$

for each $p \in \{1, 2, 3, \ldots, 2^{ml}\}$ and $k_{p,j} = \mathsf{bits}(p-1)[(j-1)l, jl]$ where $\mathsf{bits}(a)[i, j]$ denotes the bit-slice of a from the index i up to index j. $\mathcal{P}$ then sends the coefficients g_{p-1} to $\mathcal{V}$. We can modify this step later using commitments.

- $\mathcal{V}$ samples $\beta \leftarrow \mathbb{F}$ and sends it to $\mathcal{P}$.
- $\mathcal{P}$ receives β and folds the 2^{ml} instances into a single instance by constructing the batched relations defined as

$$\tilde{F}(\vec{X}_{n-l}) \equiv \sum_{p=1}^{2^{ml}} \beta^{p-1} \cdot \left(\prod_{j=1}^{m} G_{j,k_{p,j}}(\vec{X}_{n-l}) \right),$$

where $k_{p,j} = \mathsf{bits}(p-1)[(j-1)l, jl]$,

$$\tilde{C} \equiv \sum_{p=1}^{2^{ml}} g_{p-1} \cdot \beta^{p-1}, \tag{2.2.12}$$

the $\mathcal{V}$ gets oracle access to $\tilde{F}(\vec{X}_{n-l})$ (needed for last round verification). Note that the $\mathcal{V}$ can construct $\tilde{C}$ on its own.

3. **Little sum-check**

- $\mathcal{P}$ and $\mathcal{V}$ engage in a "little sum-check protocol" that essentially generates a proof of the folding on $\{0,1\}^{n-l}$ in step 3. The protocol checks

$$\tilde{C} \stackrel{?}{=} \sum_{\vec{X} \in \{0,1\}^{n-l}} \tilde{F}(\vec{X}_{n-l}) \tag{2.2.13}$$

using the standard sum-check protocol on $\{0,1\}^{n-l}$ which essentially checks the satisfiability of each of the 2^{ml} sum-checks and their linear independence in one step

$$\sum_{p=1}^{2^{ml}} g_{p-1} \cdot \beta^{p-1} \stackrel{?}{=} \sum_{\vec{X} \in \{0,1\}^{n-l}} \sum_{p=1}^{2^{ml}} \beta^{p-1} \cdot \left(\prod_{j=1}^{m} G_{j,k_{p,j}}(\vec{X}_{n-l}) \right) \tag{2.2.14}$$

where $k_{p,j} = \mathsf{bits}(p-1)[(j-1)l, jl]$.

- The number of rounds in the instance (2.2.13) is lesser due to the folding, for each $s = 1, 2, \dots n - l$th round $\mathcal{P}$ sends a round polynomial $r_s(X)$ to $\mathcal{V}$, and $\mathcal{V}$ responds with a random challenge $\alpha_s \leftarrow \mathbb{F}$.

4. **Soundness:** The folding and the little sum-check together have a soundness error of $\delta_1 \bigcup \delta_2 \leq \frac{(n-l)d}{|\mathbb{F}|} \bigcup \frac{\mathcal{O}(2^l)}{|\mathbb{F}|}$ where the $\frac{\mathcal{O}(2^l)}{|\mathbb{F}|}$ soundness error (which is negligible still for large fields) is due to the batching and d is the degree of the round polynomial. Since we are considering the product of m MLE polynomials, we have $d = m$.

5. **Proof size:** The proof consists of

$$\pi = \{C, g_0, g_1, \dots, g_{2^{ml}-1}, r_1(X), r_2(X), \dots, r_{n-l}(X)\} \tag{2.2.15}$$

   In the above C is the claimed sum, the folded coefficients g_{p-1} for $p = 1, 2 \dots 2^{ml}$ and the degree m round polynomials $r_s(X)$ for $s = 1, 2, \dots, n - l$. Thus the total proof consists of $(1 + 2^{ml} + (d + 1) \cdot (n - l))$ field elements in $\mathbb{F}$.

6. **Verifier**

   - The first step is to verify the folding, as before this is done by following the standard verifier algorithm for the "little sum-check protocol".
   - $\mathcal{V}$ takes the coefficients g_{p-1} from the proof and constructs

$$\tilde{C} = \sum_{p=1}^{2^{ml}} g_{p-1} \cdot \beta^{p-1} \tag{2.2.16}$$

     $\mathcal{V}$ uses this to check the round polynomials in the "little sum-check protocol"

$$\begin{aligned} \sum_{X \in 0,1,\dots,d} r_1(X) &\stackrel{?}{=} \tilde{C} \\ \sum_{X \in 0,1,\dots,d} r_i(X) &\stackrel{?}{=} r_{i-1}(\alpha_{i-1}) \quad \forall i = 2, 3, \dots n - l \\ r_i(\alpha_i) &\stackrel{?}{=} \tilde{F}(\vec{\alpha}_{n-l}) \end{aligned} \tag{2.2.17}$$

     For the last round, i.e. $i = n - l$, $\mathcal{V}$ makes a single oracle query to $\tilde{F}(\vec{X}_{n-l})$.

   - $\mathcal{V}$ still needs to check the original instance. For this, $\mathcal{V}$ interpolates $\{g_{p-1}\}$ on $\{0, 1\}^{ml}$ and does $\mathcal{O}(2^{ml})$ (naive) work to check the original instance

$$C \stackrel{?}{=} \sum_{\vec{X} \in \{0,1\}^l} \sum_{p=1}^{2^{ml}} \left(\prod_{j=1}^{m} \chi_{k_{p,j}}(\vec{X}_l) \right) \cdot g_{p-1} \tag{2.2.18}$$

     where $k_{p,j} = \mathsf{bits}(p-1)[(j-1)l, jl]$ and accepts if the check passes, and rejects if the check fails. Note that this part is just the partial sum computation and can be easily parallelized using methods discussed in the earlier chapter.

2.3 Checkfold Verifier optimization

In this and the next subsection we discuss an optimization of checkfold for the verifier, where we can reduce the $\mathcal{O}(2^l)$ interpolation costs to the original sum-check verification cost with an additional pairing check. The optimization stems from the fact that we can simply use KZG commitment scheme to prove the knowledge of the partial sums g_k instead of sending them to the verifier. The proposed protocol is summarized in §2.3.1. The generalization to product sumcheck is also straightforward and is presented in §2.3.2..

2.3.1 Protocol 2:

In protocol 1 §2.2.1. We modify the part where $\mathcal{P}$ sends the coefficients g_{k-1} to $\mathcal{V}$. Instead, we require the $\mathcal{P}$ to commit to a univariate polynomial with coefficients g_{k-1} whose opening can be used by the $\mathcal{V}$ to check $\tilde{C}$.. Unfortunately step 2 and step 3 together require the array of size 2^n is effectively passed over twice, and may be inefficient in terms of data transfer to an external device. Our observation is to do the partial sums (2.3.10) in CPU since it is just field additions and proceed with the folding in GPU.

1. **Setup** Same as step 1 in §2.2.1
2. **Commit to partial sums**
   - $\mathcal{P}$ computes the partial sums
   $$g_{k-1} = \sum_{\vec{X} \in \{0,1\}^{n-l}} G_{k-1}(\vec{X}_{n-l}) \; \forall \; k \; \in \{1, 2, \ldots, 2^l\} \tag{2.3.1}$$
   - $\mathcal{P}$ constructs $P(X) = \sum_{k=1}^{2^l} g_{k-1} X^{k-1}$ on domain H^{2^l} and commits using $\texttt{KZGCommit}(P(X))$ [11].
   - $\mathcal{V}$ samples $\beta \leftarrow \mathbb{F}$ and sends to $\mathcal{P}$.
   - $\mathcal{P}$ computes the KZG opening proof of $\texttt{KZGOpen}(P(\beta)) = \tilde{C}$ and sends the KZG proof $\{[P(X)]_1, [Q(X)]_1, P(\beta)\}$ where $Q(X)$ is the KZG quotient.
3. **Folding**
   - $\mathcal{P}$ uses β and folds the 2^l instances into a single instance by constructing the batched relations defined as
   $$\tilde{F}(\vec{X}_{n-l}) \equiv \sum_{k=1}^{2^l} \beta^{k-1} \cdot G_{k-1}(\vec{X}_{n-l})$$
   $$P(\beta) \equiv \tilde{C} \equiv \sum_{k=1}^{2^l} \beta^{k-1} \cdot g_{k-1} \tag{2.3.2}$$
   Furthermore, $\mathcal{V}$ gets oracle access to $\tilde{F}(\vec{X}_{n-l})$ (needed for last round verification). In this way $\tilde{C}$ will be checked by $\mathcal{V}$ using KZG opening proof later. Note that $\mathcal{V}$ has no access to g_{k-1}. The prover can construct the accumulator (2.3.2) by streaming 2^{n-l} slices into the device.

4. **Little sum-check** : Same as step 3 in §2.2.1

5. **Final sum-check**: $\mathcal{P}$ interpolates g_{k-1} on $\{0,1\}^l$ [3]

$$K(\vec{X}_l) = \sum_{k=1}^{2^l} \chi_{\vec{s}_l(k-1)}(\vec{X}_l) \cdot g_{k-1} \tag{2.3.4}$$

and $\mathcal{P}$ and $\mathcal{V}$ engage in a sum-check protocol on $\{0,1\}^l$ that checks the original instance sum.

$$C \stackrel{?}{=} \sum_{\vec{X} \in \{0,1\}^l} K(\vec{X}_l) \tag{2.3.5}$$

$\mathcal{P}$ sends l round polynomials $\tilde{r}_1(X), \ldots, \tilde{r}_l(X)$. For the round $i = l$, the $\mathcal{V}$ needs oracle access to $K(\vec{X}_l)$.

6. **Soundness:** The folding, the little sum-check and final sumcheck together have a soundness error of $\delta_1 \bigcup \delta_2 \bigcup \delta_3 \leq \frac{(n-l)\cdot d}{|\mathbb{F}|} \bigcup \frac{l \cdot d}{|\mathbb{F}|} \bigcup \frac{2^l}{|\mathbb{F}|}$ where the $\frac{2^l}{|\mathbb{F}|}$ soundness error (which is still negligible for large fields) is due to the batching and d is the degree of the round polynomial. Since we are considering MLE polynomials, degree of round polynomials is $d = 1$.

7. **Proof size:** The proof consists of

$$\pi = \left\{ C, [P(X)]_1, [Q(X)]_1, P(\beta), \{r_1(X), r_2(X), \ldots, r_{n-l}(X)\}, \{\tilde{r}_1(X), \tilde{r}_2(X), \ldots, \tilde{r}_l(X)\} \right\} \tag{2.3.6}$$

In the above C is the claimed sum, two $\mathbb{G}_1$ elements and one $\mathbb{F}$ of the KZG proof $[P(X)]_1, [Q(X)]_1, P(\beta)$, the degree d round polynomials for little sum-check $r_i(X)$ for $i = 1, 2, \ldots, n-l$ and the degree d round polynomials of the final-sumcheck $\tilde{r}_j(X)$ for $j = 1, 2, \ldots, l$. Thus the total proof consists of $(2 + (d+1) \cdot n)$ field elements $\mathbb{F}$, and $2\mathbb{G}_1$ elements.

8. **Verifier**

- **KZG verification:** The first step is to verify the pairing equation for the KZG proof.

$$e(P(\beta) \cdot [1]_1 - [P(X)]_1, [1]_2] \stackrel{?}{=} e([Q(X)]_1, [\beta - X]_2) \tag{2.3.7}$$

[3] Actually there is no need to interpolate, may be a clever way to store the array since the coefficients of $K(\vec{X}_l)$ are partial sums in $F(\vec{X}_n)$

$$\begin{aligned} K(\vec{X}_l) &= \sum_{\vec{X} \in \{0,1\}^{n-l}} F(\vec{X}_n) \\ &= \sum_{k=1}^{2^l} \chi_{\vec{s}_l(k-1)}(\vec{X}_l) \cdot \sum_{\vec{X} \in \{0,1\}^{n-l}} G_{k-1}(\vec{X}_{n-l}) \\ &= \sum_{k=1}^{2^l} \chi_{\vec{s}_l(k-1)}(\vec{X}_l) \cdot g_{k-1} \end{aligned} \tag{2.3.3}$$

- **Little sumcheck:** The main difference here is, $\mathcal{V}$ sets $\tilde{C} = P(\beta)$ (which has been checked in the opening proof above) and uses this to check the round polynomials in the "little sum-check protocol"

$$\begin{aligned}\sum_{X\in 0,1,\ldots,d} r_1(X) &\stackrel{?}{=} \tilde{C}\\ \sum_{X\in 0,1,\ldots,d} r_i(X) &\stackrel{?}{=} r_{i-1}(\alpha_{i-1}) \quad \forall \quad i = 2,3,\ldots n-l\\ r_i(\alpha_i) &\stackrel{?}{=} \tilde{F}(\vec{\alpha}_{n-l})\end{aligned} \tag{2.3.8}$$

  For $i = n-l$, $\mathcal{V}$ makes a single oracle query to $\tilde{F}(\vec{X}_{n-l})$ at $\vec{\alpha}_{n-l}$.

- **Final sum-check:** $\mathcal{V}$ still needs to check the original instance following the sum-check protocol

$$\begin{aligned}\sum_{X\in 0,1,\ldots,d} \tilde{r}_1(X) &\stackrel{?}{=} C\\ \sum_{X\in 0,1,\ldots,d} \tilde{r}_i(X) &\stackrel{?}{=} r_{i-1}(\gamma_{i-1}) \quad \forall \quad i = 2,3,\ldots l-1\\ r_l(\gamma_l) &\stackrel{?}{=} K(\vec{\gamma}_l)\end{aligned} \tag{2.3.9}$$

  For $i = l$, $\mathcal{V}$ makes a single oracle query to $K(\vec{X}_l)$ at $\vec{\gamma}_l$.

The $\mathcal{V}$ accepts if the check passes and rejects if the check fails. The cost of the verifier is the same as the sum-check protocol $\mathcal{O}(n)$ and an additional pairing cost for the KZG opening proof.

In summary, we realized a knowledge reduction argument by folding a 2^n sized sum-check instance into 2^l instances of size 2^{n-l}. The folding is proven using batched sum-check. Using the 2^l "partial-sums" g_{k-1} the original sum-check instance is proven, but at a reduced instance size of 2^l. Compared to §2.2.1, $\mathcal{P}$ has to engage in a sub-protocol for KZG opening. The verifier has the original sum-check protocol cost, with an additional cost due to a pairing check.

2.3.2 Protocol 2 for product MLEs

In this section, we generalize the verifier optimization protocol §2.3.1 for arbitrary product sumchecks. This is also quite straightforward.

1. **Setup** Same as step 1 in §2.2.2
2. **Commit to partial sums**
   - $\mathcal{P}$ computes the partial sums

$$g_{p-1} = \sum_{\vec{X}\in\{0,1\}^{n-l}} \left(\prod_{j=1}^{m} G_{j,k_{p,j}}(\vec{X}_{n-l})\right) \tag{2.3.10}$$

     for each $p \in \{1,2,3,\ldots,2^{ml}\}$ and $k_{p,j} = \mathsf{bits}(p-1)[(j-1)l, jl]$.

- $\mathcal{P}$ constructs $P(X) = \sum_{p=1}^{2^{ml}} g_{p-1} X^{p-1}$ on domain $H^{2^{ml}}$ and commits using $\texttt{KZGCommit}(P(X))$ [11].
- $\mathcal{V}$ samples $\beta \leftarrow \mathbb{F}$ and sends to $\mathcal{P}$.
- $\mathcal{P}$ computes the KZG opening proof of $\texttt{KZGOpen}(P(\beta)) = \tilde{C}$ and sends the KZG proof $\{[P(X)]_1, [Q(X)]_1, P(\beta)\}$ where $Q(X)$ is the quotient polynomial in KZG commitment scheme.

3. **Folding**

   - $\mathcal{P}$ uses β and folds the 2^{ml} instances into a single instance by constructing the batched relations defined as

   $$\tilde{F}(\vec{X}_{n-l}) \equiv \sum_{p=1}^{2^{ml}} \beta^{p-1} \cdot \left(\prod_{j=1}^{m} G_{j,k_{p,j}}(\vec{X}_{n-l}) \right),$$

   where $k_{p,j} = \mathsf{bits}(p-1)[(j-1)l, jl]$ and

   $$\tilde{C} \equiv \sum_{p=1}^{2^{ml}} \beta^{p-1} \cdot g_{p-1}. \tag{2.3.11}$$

   Furthermore, $\mathcal{V}$ gets oracle access to $\tilde{F}(\vec{X}_{n-l})$ (needed for last round verification). In this way $\tilde{C}$ will be checked by verifier using KZG opening proof later. Note that verifier has no access to g_{k-1} as before.

4. **Little sum-check** : Same as before

5. **Final sum-check**: It remains for $\mathcal{P}$ to prove that the partial sums g_{p-1} are related to one another as

   $$K(\vec{X}_l) = \sum_{p=1}^{2^{ml}} g_{p-1} \cdot \left(\prod_{j=1}^{m} \chi_{k_{p,j}}(\vec{X}_l) \right) \tag{2.3.12}$$

   where $k_{p,j} = \mathsf{bits}(p-1)[(j-1)l, jl]$ and $\mathcal{P}$ and $\mathcal{V}$ engage in a sum-check protocol on $\{0,1\}^l$ that checks the original instance sum.

   $$C \stackrel{?}{=} \sum_{\vec{X} \in \{0,1\}^l} K(\vec{X}_l) \tag{2.3.13}$$

   $\mathcal{P}$ sends l round polynomials $\tilde{r}_1(X), \ldots, \tilde{r}_l(X)$. For the round $i = l$, the $\mathcal{V}$ needs oracle access to $K(\vec{X}_l)$.

6. **Soundness:** The folding, the little sum-check and final sumcheck together have a soundness error of $\delta_1 \bigcup \delta_2 \bigcup \delta_3 \leq \frac{(n-l)\cdot d}{|\mathbb{F}|} \bigcup \frac{l \cdot d}{|\mathbb{F}|} \bigcup \frac{2^l}{|\mathbb{F}|}$ where the $\frac{2^l}{|\mathbb{F}|}$ soundness error (which is still negligible for large fields) is due to the batching and d is the degree of the round polynomial. Since we are considering the product of m MLE polynomials, $d = m$.

7. **Proof size:** The proof consists of

$$\pi = \left\{ C, [P(X)]_1, [Q[\beta)]_1, P(\beta), \{r_1(X), r_2(X), \ldots, r_{n-l}(X)\}, \{\tilde{r}_1(X), \tilde{r}_2(X), \ldots, \tilde{r}_l(X)\} \right\} \tag{2.3.14}$$

In the above C is the claimed sum, two $\mathbb{G}_1$ elements and one $\mathbb{F}$ of the KZG proof $[P(X)]_1, [Q[\beta)]_1, P(\beta)$, the degree d round polynomials for little sum-check $r_i(X)$ for $i = 1, 2, \ldots, n-l$ and the degree d round polynomials of the final-sumcheck $\tilde{r}_j(X)$ for $j = 1, 2, \ldots, l$. Thus the total proof consists of $(2 + (d+1) \cdot n)$ field elements $\mathbb{F}$, and $2\mathbb{G}_1$ elements.

8. **Verifier**

   - **KZG verification:** The first step is to verify the pairing equation for the KZG proof.

$$e(P(\beta) \cdot [1]_1 - [P(X)]_1, [1]_2] \stackrel{?}{=} e([Q(X)]_1, [\beta - X]_2) \tag{2.3.15}$$

   - **Little sumcheck:** The main difference here is, $\mathcal{V}$ sets $\tilde{C} = P(\beta)$ and uses this to check the round polynomials in the "little sum-check protocol"

$$\begin{aligned} \sum_{X \in 0,1,\ldots,d} r_1(X) &\stackrel{?}{=} \tilde{C} \\ \sum_{X \in 0,1,\ldots,d} r_i(X) &\stackrel{?}{=} r_{i-1}(\alpha_{i-1}) \quad \forall \quad i = 2, 3, \ldots n-l \\ r_i(\alpha_i) &\stackrel{?}{=} \tilde{F}(\vec{\alpha}_{n-l}) \end{aligned} \tag{2.3.16}$$

     For $i = n - l$, $\mathcal{V}$ makes a single oracle query to $\tilde{F}(\vec{X}_{n-l})$ at $\vec{\alpha}_{n-l}$.

   - **Final sum-check:** $\mathcal{V}$ still needs to check the original instance following the sum-check protocol

$$\begin{aligned} \sum_{X \in 0,1,\ldots,d} \tilde{r}_1(X) &\stackrel{?}{=} C \\ \sum_{X \in 0,1,\ldots,d} \tilde{r}_i(X) &\stackrel{?}{=} r_{i-1}(\gamma_{i-1}) \quad \forall \quad i = 2, 3, \ldots l-1 \\ r_l(\gamma_l) &\stackrel{?}{=} K(\vec{\gamma}_l) \end{aligned} \tag{2.3.17}$$

     For $i = l$, $\mathcal{V}$ makes a single oracle query to $K(\vec{X}_l)$ at $\vec{\gamma}_l$.

   The verifier accepts if the check passes and rejects if the check fails. The cost of the verifier is the same as the sum-check protocol $\mathcal{O}(n)$ and an additional pairing cost for the KZG opening proof.

Acknowledgments

We stand on the shoulders of giants. We thank Srinath Setty and Justin Thaler for helpful clarifications and discussions. We thank Giacomo Fenzi for pointing out the correct soundness error for the batched sum-check as used in this paper. We thank Yuval Domb, and Tomer Solberg for discussions. We thank Giacomo Fenzi and Omer Shlomovits for carefully reviewing the draft.

Bibliography

[1] J. Thaler, *Proofs, arguments and zero knowledge*, . https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf.

[2] J. Thaler, *The sum-check protocol over fields of small characteristic*, . https://people.cs.georgetown.edu/jthaler/small-sumcheck.pdf.

[3] S. Setty, "Spartan: Efficient and general-purpose zksnarks without trusted setup." Cryptology ePrint Archive, Paper 2019/550, 2019. https://eprint.iacr.org/2019/550.

[4] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation." Cryptology ePrint Archive, Paper 2019/317, 2019. https://eprint.iacr.org/2019/317.

[5] A. Kothapalli, S. Setty, and I. Tzialla, "Nova: Recursive zero-knowledge arguments from folding schemes." Cryptology ePrint Archive, Paper 2021/370, 2021. https://eprint.iacr.org/2021/370.

[6] A. Kothapalli and S. Setty, "Hypernova: Recursive arguments for customizable constraint systems." Cryptology ePrint Archive, Paper 2023/573, 2023. https://eprint.iacr.org/2023/573.

[7] A. Kothapalli and B. Parno, "Algebraic reductions of knowledge." Cryptology ePrint Archive, Paper 2022/009, 2022. https://eprint.iacr.org/2022/009.

[8] L. Trevisan and S. Vadhan, *Pseudorandomness and average-case complexity via uniform reductions*, . https://people.seas.harvard.edu/~salil/research/uniform-cc.pdf.

[9] A. Chiesa, M. A. Forbes, and N. Spooner, "A zero knowledge sumcheck and its applications." Cryptology ePrint Archive, Paper 2017/305, 2017. https://eprint.iacr.org/2017/305.

[10] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "Hyperplonk: Plonk with linear-time prover and high-degree custom gates." Cryptology ePrint Archive, Paper 2022/1355, 2022. https://eprint.iacr.org/2022/1355.

[11] A. Kate, G. M. Zaverucha, and I. Goldberg, *Constant-size commitments to polynomials and their applications*, in *Advances in Cryptology - ASIACRYPT 2010* (M. Abe, ed.), (Berlin, Heidelberg), pp. 177–194, Springer Berlin Heidelberg, 2010.