Sumcheck 201

Ingonyama

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 discuss 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]. 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

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

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

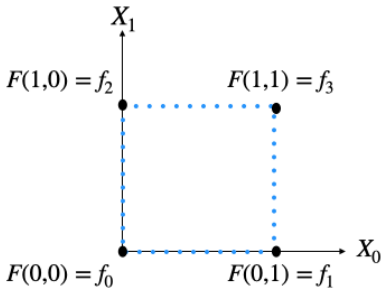

Figure 1.1. Evaluations of a Multilinear polynomial $F(X_2, X_1)$ on a boolean hypercube $\{0,1\}^2$.

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

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

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

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 proceeds 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

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

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 memory bound in the first round is 2^n and decreases by half in each subsequent round.
- 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.
- The main disadvantage of the method is that device utilization decreases as the number of rounds increases.

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

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,

$$F(X_3, X_2, X_1) = f_0 \cdot \bar{X}_3\bar{X}_2\bar{X}_1 + f_1 \cdot \bar{X}_3\bar{X}_2X_1 + f_2 \cdot \bar{X}_3X_2\bar{X}_1 + f_3 \cdot \bar{X}_3X_2X_1 + f_4 \cdot X_3\bar{X}_2\bar{X}_1 + f_5 \cdot X_3\bar{X}_2X_1 + f_6 \cdot X_3X_2\bar{X}_1 + f_7 \cdot X_3X_2X_1 \quad (1.2.7)$$

The round polynomial in the first round is given by

$$r_1(X) = \sum_{X_i \in \{0,1\}} F(X_3, X_2, X) = \sum_{i=0}^{3} r_1^{(i)}(X)$$
$$r_1^{(i)}(X) = f_{2i} \cdot \bar{X} + f_{2i+1} \cdot X \quad (1.2.8)$$

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

$$r_2(X) = \sum_{X_i \in \{0,1\}} F(X_3, X, \alpha_1) = \sum_{i=0}^{1} r_2^{(i)}(X)$$
$$r_2^{(i)}(X) = r_1^{(2i)}(\alpha_1) \cdot \bar{X} + r_1^{(2i+1)}(\alpha_1) \cdot X \quad (1.2.9)$$

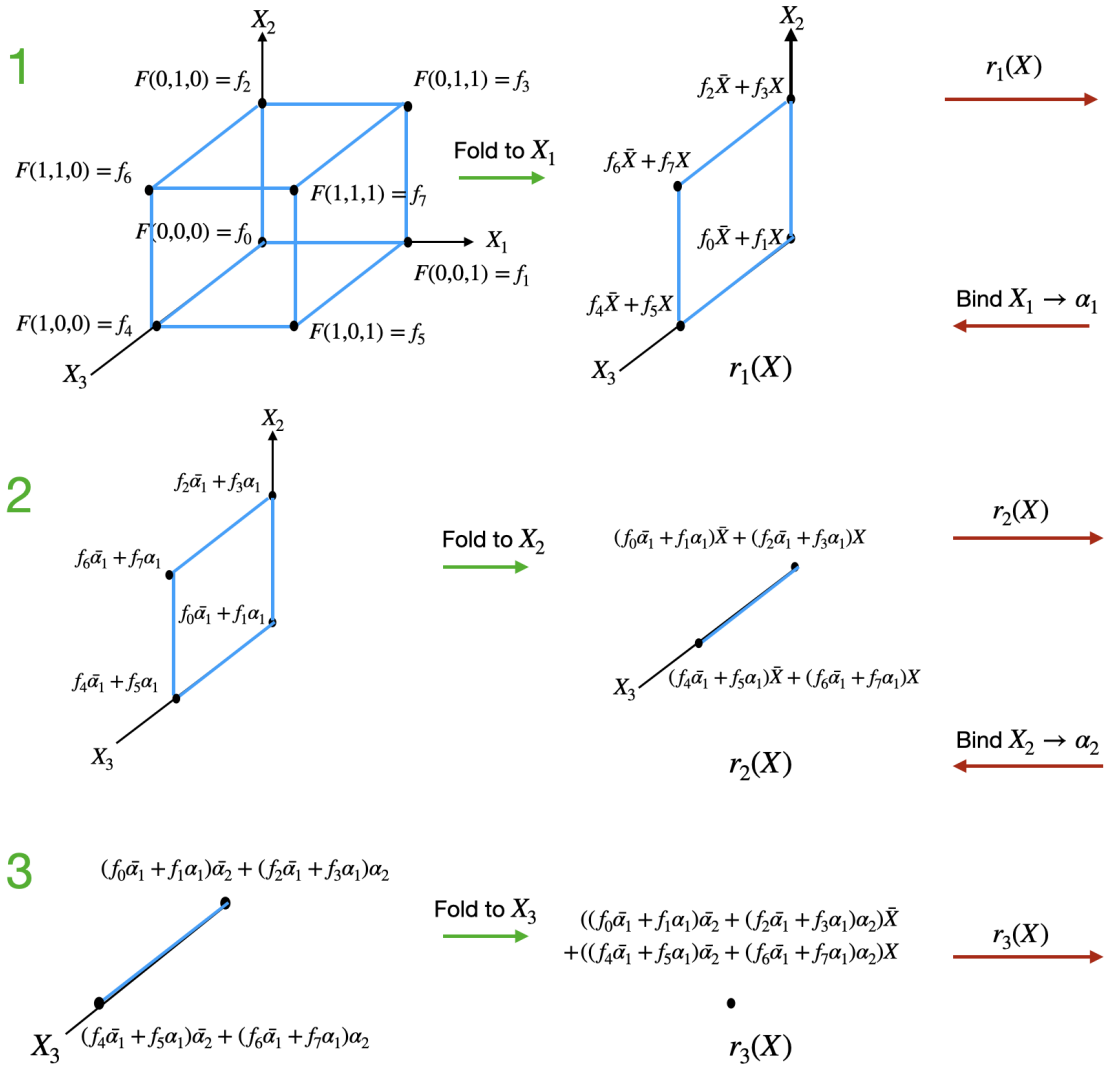

Figure 1.2. A schematic representation of algorithm 1 by recursion: Round polynomial computation for MLE $F(X_3, X_2, X_1)$. In each round, we trace over one dimension X_i in the boolean hypercube $\{0,1\}^3$, the round polynomial is the sum of all vertices of the folded cube in each round. The reduced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

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

$$\begin{aligned} r_3(X) &= F(X, \alpha_2, \alpha_1) \\ &= r_2^{(0)}(\alpha_2) \cdot \bar{X} + r_2^{(1)}(\alpha_2) \cdot X \end{aligned} \tag{1.2.10}$$

as explained in fig. 1.2.

1.2.1 Single MLE - Algorithm 2: Precomputations

We notice that the round polynomial in the p'th round (1.2.11)

$$\begin{aligned} r_p(X) &= \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-p}, X, \vec{\alpha}_{p-1}) \\ &= \sum_{r=1}^{2^p} \left(\chi_{\vec{s}_p(r-1)}(X, \vec{\alpha}_{p-1}) \circ \sum_{X_i \in \{0,1\}} F(\vec{X}_{n-p}, \vec{s}_p(r-1)) \right) \end{aligned} \tag{1.2.11}$$

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

$$r_p(X) = \sum_{r=1}^{2^p} \mathcal{F}_p[r] \circ \Xi_p[r] \tag{1.2.12}$$

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.

$$\begin{aligned} \mathcal{F}_{p-1}[r] = \sum_{X_i} F(\vec{X}_{n-p+1}, \vec{s}_{p-1}(r-1)) &= \sum_{X_{n-p+2}} \Bigg(\bar{X}_{n-p+2} \sum_{X_i} F(\vec{X}_{n-p-2}, 0, \vec{s}_{p-1}(r-1)) \\ &\quad + X_{n-p+2} \sum_{X_i} F(\vec{X}_{n-p-2}, 1, \vec{s}_{p-1}(r-1)) \Bigg) \\ &= \sum_{X_i} F(\vec{X}_{n-p-2}, \vec{s}_p(r-1)) + \sum_{X_i} F(\vec{X}_{n-p-2}, \vec{s}_p(r-1+2^{p-1})) \\ &= \mathcal{F}_p(r) + \mathcal{F}_p(r+2^{p-1}) \end{aligned} \tag{1.2.13}$$

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

$$\begin{aligned} \Xi_p[r]\big|_{r=1}^{2^p}(X, \vec{\alpha}_{p-1}) &= \left\{ \chi_{\vec{s}_p(r-1)}(X, \vec{\alpha}_{p-1}) \right\}_{r=1}^{2^p} \\ &= \left(\bar{X} \cdot \left\{ \chi_{\vec{s}_{p-1}(r-1)}(\vec{\alpha}_{p-1}) \right\}_{r=1}^{2^{p-1}}, X \cdot \left\{ \chi_{\vec{s}_{p-1}(r-1)}(\vec{\alpha}_{p-1}) \right\}_{r=1}^{2^{p-1}} \right) \\ &= \left(\bar{X} \cdot \Xi_{p-1}(\vec{\alpha}_{p-1}), X \cdot \Xi_{p-1}(\vec{\alpha}_{p-1}) \right) \end{aligned} \tag{1.2.14}$$

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 (1.2.13) and (1.2.14). The main features of algorithm (2) are

- The precompute, Hadamard accumulator and Lagrange updater phase are all parallelizable.
- 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
- The hadamard multiplications are 2^p per round, between the challenge vector and the coefficients vector.

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 (1.2.13) , (1.2.14) 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 for $p = 1$ in terms of coefficients of $p = 2$

$$\begin{aligned}
\mathcal{F}_1[r]\big|_{r=1}^{2} = \sum_{X_i} F(\vec{X}_2, \vec{s}_1(r-1)) &= \left\{ \sum_{X_i} F(\vec{X}_2, 0), \sum_{X_i} F(\vec{X}_2, 1) \right\} \\
&= \Big\{ \sum_{X_2} \bar{X}_2 \cdot \sum_{X_i} F(\vec{X}_1, 0, 0) + \sum_{X_2} X_2 \cdot \sum_{X_i} F(\vec{X}_1, 1, 0) \\
&\quad , \sum_{X_2} \cdot \bar{X}_2 \sum_{X_i} F(\vec{X}_1, 0, 1) + \sum_{X_2} X_2 \cdot \sum_{X_i} F(\vec{X}_1, 1, 1) \Big\} \\
&= \{\mathcal{F}_2(1) + \mathcal{F}_2(3), \mathcal{F}_2(2) + \mathcal{F}_2(4)\} \qquad (1.2.15)
\end{aligned}$$

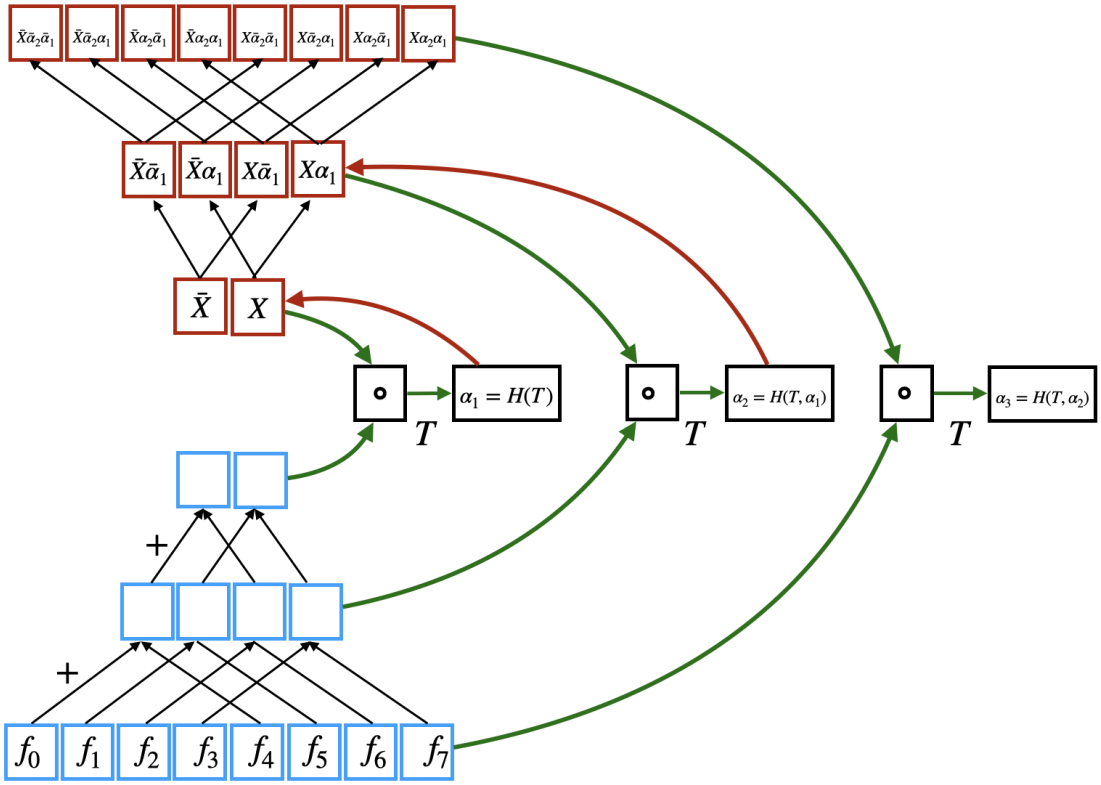

Figure 1.3. A schematic representation of algorithm 2 for the sum-check prover computation for $F(X_3, X_2, X_1)$. The blue boxes represent the precompute data, the arrows connecting other blue boxes represent the memoization which can parallelize the process. The red boxes, represent the challenge vectors, 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.

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 $p = 3$

$$\begin{aligned}\mathcal{F}_2[r]\big|_{r=1}^{2^2} = \sum_{X_i} F(\vec{X}_1, \vec{s}_2(r-1)) = \Big\{&\sum_{X_i} F(\vec{X}_1, 0, 0), \sum_{X_i} F(\vec{X}_1, 0, 1) \\ &, \sum_{X_i} F(\vec{X}_1, 1, 0), \sum_{X_i} F(\vec{X}_1, 1, 1)\Big\} \\ &= \{\mathcal{F}_3(1) + \mathcal{F}_3(5), \mathcal{F}_3(2) + \mathcal{F}_3(6), \mathcal{F}_3(3) + \mathcal{F}_3(7), \mathcal{F}_3(4) + \mathcal{F}_3(8)\}\end{aligned} \tag{1.2.16}$$

The last round coefficients are just the input data

$$\mathcal{F}_3[r]\big|_{r=1}^{2^3} = (f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7) \tag{1.2.17}$$

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

$$\Xi_1(X)[r]\big|_{r=1}^{2} = \left\{\chi_{\vec{s}_1(0)}(X), \chi_{\vec{s}_1(1)}(X)\right\} = (\bar{X}, X) \tag{1.2.18}$$

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

$$\begin{aligned}\Xi_2(X, \alpha_1)[r]\big|_{r=1}^{4} &= \left\{\chi_{\vec{s}_2(0)}(X, \alpha_1), \chi_{\vec{s}_2(1)}(X, \alpha_1), \chi_{\vec{s}_2(2)}(X, \alpha_1), \chi_{\vec{s}_2(3)}(X, \alpha_1)\right\} \\ &= \left(\bar{X} \cdot \bar{\alpha}_1, \bar{X} \cdot \alpha_1, X \cdot \bar{\alpha}_1, X \cdot \alpha_1\right) \\ &= \left(\bar{X} \cdot \Xi_1(\alpha_1), X \cdot \Xi_1(\alpha_1)\right) \end{aligned} \tag{1.2.19}$$

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

$$\begin{aligned}\Xi_3(X, \vec{\alpha}_2)[r]\big|_{r=1}^{8} &= \left\{\chi_{\vec{s}_3(0)}(X, \vec{\alpha}_2), \chi_{\vec{s}_3(1)}(X, \vec{\alpha}_2), \chi_{\vec{s}_3(2)}(X, \vec{\alpha}_2), \chi_{\vec{s}_3(3)}(X, \vec{\alpha}_2)\right\} \\ &= \left(\bar{X} \cdot \bar{\alpha}_2\bar{\alpha}_1, \bar{X} \cdot \bar{\alpha}_2\alpha_1, \bar{X} \cdot \alpha_2\bar{\alpha}_1, \bar{X} \cdot \alpha_2\alpha_1, X \cdot \bar{\alpha}_2\bar{\alpha}_1, X \cdot \bar{\alpha}_2\alpha_1, X \cdot \alpha_2\bar{\alpha}_1, X \cdot \alpha_2\alpha_1\right) \\ &= \left(\bar{X} \cdot \Xi_2(\vec{\alpha}_2), X \cdot \Xi_2(\vec{\alpha}_2)\right) \end{aligned} \tag{1.2.20}$$

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

The discussion in the previous sections §1.2, §1.2.1 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

$$\begin{aligned}\mathsf{combine}(F_1(\vec{X}), F_2(\vec{X}), F_3(\vec{X}), F_4(\vec{X})) =& F_1(\vec{X}) \cdot F_2(\vec{X}) \cdot F_3(\vec{X}) \cdot F_4(\vec{X}) + F_1(\vec{X}) \cdot F_2(\vec{X}) \cdot F_3(\vec{X}) \\ &+ F_1(\vec{X}) \cdot F_2(\vec{X}) + F_1(\vec{X})\end{aligned} \tag{1.3.1}$$

another example which is used in Spartan [3] is

$$\mathsf{combine}(F_1(\vec{X}), F_2(\vec{X}), F_3(\vec{X}), F_4(\vec{X})) = F_1(\vec{X}) \cdot F_2(\vec{X}) \cdot F_4(\vec{X}) - F_3(\vec{X}) \cdot F_4(\vec{X}) \tag{1.3.2}$$

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

1.4 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 $X_i \in \vec{X}$

$$P_m(\vec{X}) = \prod_{j=1}^{m} \left(\sum_{k_j=1}^{2^n} \chi^{(j)}_{\vec{s}_n(k_j-1)}(\vec{X}) \cdot F^{(j)}(\vec{s}_n(k_j - 1)) \right) \tag{1.4.1}$$

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

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

$$\begin{aligned} r_p(X) &= \prod_{j=1}^{m} \sum_{X_i \in \{0,1\}} F^{(j)}(\vec{X}_{n-p}, X, \vec{\alpha}_{p-1}) \\ &= \prod_{j=1}^{m} \sum_{r_j=1}^{2^p} \left(\chi^{(j)}_{\vec{s}_p(r_j-1)}(X, \vec{\alpha}_{p-1}) \circ \sum_{X_i \in \{0,1\}} F^{(j)}(\vec{X}_{n-p}, \vec{s}_p(r_j - 1)) \right) \end{aligned} \tag{1.4.2}$$

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 (1.4.2) has a tensor structure when re-expressed as follows, can be used to parallely compute several interme-

diate 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}

$$\mathcal{F}_p^{(j)} \equiv \sum_{X_i \in \{0,1\}} F^{(j)}(\vec{X}_{n-p}, \vec{s}_p(r_j - 1))\big|_{r_j=1}^{2^p} = \begin{pmatrix} F^{(j)}[0 : 2^{n-p}] \\ F^{(j)}[2^{n-p} : 2^{n-p+1}] \\ \vdots \\ F^{(j)}[2^n - 2^{n-p+1} : 2^n - 2^{n-p}] \\ F^{(j)}[2^n - 2^{n-p} : 2^n] \end{pmatrix}_{2^p \times 2^{n-p}} \tag{1.4.3}$$

and the challenge vector (since it is the same for each term of the j products in (1.4.2)) (1.2.14) as

$$\begin{aligned} \Gamma_p \equiv \Xi_p[r]\big|_{r=1}^{2^p}(X, \vec{\alpha}_{p-1}) &= \left\{\chi_{\vec{s}_p(r-1)}(X, \vec{\alpha}_{p-1})\right\}_{r=1}^{2^p} \\ &= (\bar{X}, X) \otimes \Xi_{p-1}(\vec{\alpha}_{p-1}) \end{aligned} \tag{1.4.4}$$

and rewrite (1.4.2) as

$$r_p(X) = \left([\mathcal{F}_p^{(1)} \circledast \mathcal{F}_p^{(2)} \circledast \ldots \circledast \mathcal{F}_p^{(m-1)}]_{2^{(m-1)p} \times 2^{n-p}} \cdot [\mathcal{F}_p^{(m)}]_{2^{n-p} \times 2^p}^t\right) \boxdot [\Gamma_p \otimes \Gamma_p \ldots \otimes \Gamma_p]_{2^{(m-1)p} \times 2^p} \tag{1.4.5}$$

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:

$$P \circledast Q := \begin{bmatrix} \ldots \vec{p}_1 \ldots \\ \ldots \vec{p}_2 \ldots \\ \vdots \\ \ldots \vec{p}_{m_1} \ldots \end{bmatrix} \circledast \begin{bmatrix} \ldots \vec{q}_1 \ldots \\ \ldots \vec{q}_2 \ldots \\ \vdots \\ \ldots \vec{q}_{m_2} \ldots \end{bmatrix} = \begin{bmatrix} \ldots \vec{p}_1 \circ \vec{q}_1 \ldots \\ \ldots \vec{p}_1 \circ \vec{q}_2 \ldots \\ \vdots \\ \ldots \vec{p}_1 \circ \vec{q}_{m_2} \ldots \\ \hline \ldots \vec{p}_2 \circ \vec{q}_1 \ldots \\ \ldots \vec{p}_2 \circ \vec{q}_2 \ldots \\ \vdots \\ \ldots \vec{p}_2 \circ \vec{q}_{m_2} \ldots \\ \hline \vdots \\ \hline \ldots \vec{p}_{m_1} \circ \vec{q}_1 \ldots \\ \ldots \vec{p}_{m_1} \circ \vec{q}_2 \ldots \\ \vdots \\ \ldots \vec{p}_{m_1} \circ \vec{q}_{m_2} \ldots \end{bmatrix}.$$

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.

1.5 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[1]. Our implementation allows us to run the sum-check prover for any "gate" equation in MLE language (see Section 1.3). 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 (1.3.2)). 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.

[1] 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

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.