Sumcheck API for V3 ICICLE

In this document, we describe the criteria for sumcheck API for usage in ICICLE V3. The resources related to sumcheck are

- Sumcheck201
  - CPU Implementation in Rust
  - GPU implementation without Fiat Shamir (fake hasher)
  - Algorithm 1/3 in the note and this CPU implementation code is relevant for our discussion and focus on large fields with size ≥ 256 bits.
  - For small fields ≤ 64 bits, algorithm 1 or 3 should not be used as it leads to excessive extension field multiplications which is not recommended. A precomputation based algorithm: Algorithm 2/4 are relevant, this CPU implementation is relevant Code. While this algorithm can also be used for ≥ 256 bits - it will lead to memory bottlenecks in this case.
    - Further optimizations in this direction paper by @suyash and @yuvaldomb.

Encoding of Multilinear extensions

- Multi Linear Extensions (MLE) encode 2^n elements $a_i \in \mathbb{F}$ in a boolean hypercube defined by n Boolean vectors $X_i \in \{0, 1\}$.
- Every a_i can be defined as boolean evaluations of a polynomial $F(X_1, X_2, \ldots . X_n)$ linear in each variable X_i.
- Encoding can be done in two complementary ways. Both are equivalent in definitions. But have implementation differences as explained below.

We define the bits representation of n using the variables above as

$$Bit(\vec{n}) = (X_n, X_{n-1}, \ldots, X_2, X_1)$$

where $MSB = X_n$ and $LSB = X_1$.

- **Canonical encoding**: The first is following the definition in Sumcheck201 where we take the bit representation of $\vec{n}$ with the ordering and encode all field elements as evaluations on the boolean hypercube as follows

$F(0,0,0)$	$F(0,0,1)$	$F(0,1,0)$	$F(0,1,1)$	$F(1,0,0)$	$F(1,0,1)$	$F(1,1,0)$	$F(1,1,1)$
a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7

This is represented as the polynomial

$$F(X_3, X_2, X_1) = a_0 \cdot \bar{X}_3\bar{X}_2\bar{X}_1 + a_1 \cdot \bar{X}_3\bar{X}_2X_1 + a_2 \cdot \bar{X}_3X_2\bar{X}_1 + a_3 \cdot \bar{X}_3X_2X_1 + a_4 \cdot X_3\bar{X}_2\bar{X}_1 + a_5 \cdot X_3\bar{X}_2X_1 + a_6 \cdot X_3X_2\bar{X}_1 + a_7 \cdot X_3X_2X_1$$

Where $\bar{X} = 1 - X$, and then the evaluation can be understood as $F(bits(i))$ for $i = 0, 1, 2, ..7$ in canonical bit representation. The polynomial is encoded as a vector in the sequence. So if one needs to access odd or even elements it leads to strided memory access in a SIMD setting.

- **Bit reversed encoding** (recommended): In this encoding the vector is stored with the bit ordering $X_1, X_2, X_3 \ldots$ This encoding is used in practical implementation here

This is represented as the polynomial

$$F(X_1, X_2, X_3) = a_0 \cdot \bar{X}_1\bar{X}_2\bar{X}_3 + a_1 \cdot \bar{X}_1\bar{X}_2X_3 + a_2 \cdot \bar{X}_1X_2\bar{X}_3 + a_3 \cdot \bar{X}_1X_2X_3 + a_4 \cdot X_1\bar{X}_2\bar{X}_3 + a_5 \cdot X_1\bar{X}_2X_3 + a_6 \cdot X_1X_2\bar{X}_3 + a_7 \cdot X_1X_2X_3$$

Note that the vector of evaluations have the ordering

$F(0,0,0)$	$F(1,0,0)$	$F(0,1,0)$	$F(1,1,0)$	$F(1,0,0)$	$F(1,0,1)$	$F(0,1,1)$	$F(1,1,1)$
a_0	a_4	a_2	a_6	a_1	a_5	a_3	a_7

- Note that in this ordering the tensor structure stores the odd and even values as contiguous vectors.
- In the accumulation phase of sumcheck, the main computation is sum even and odd term of a given vector. The bit reversed encoding, leads to even and odd terms stored as a contiguous set each, and has coalesced access by construction for the sum-reduce sub kernels.

Arithmetic

We can define
Given a structure of the MLE:

- **Additions** are like vector additions in the respective elements

  $$F(X_1, X_2) + \gamma \cdot G(X_1, X_2) = H(X_1, X_2)$$

  where $\gamma \in \mathbb{F}$.

- **Partial sums** over any number of directions

  $$r(X) = \sum_{X_i \in \{0,1\}} F(X, X_2, \ldots, X_n)$$

- **Evaluations** on any $\vec{\alpha_i} \in \mathbb{F}$

$$r(X, \vec{\alpha}) = F(X, \alpha_2, \ldots, \alpha_n)$$

- **Partial sums with Evaluations**

$$r(X) = \sum_{X_i \in \{0,1\}} F(\alpha_1, \alpha_2, X_2, X_3 \ldots, X_{n-1}, X)$$

If $F(\vec{X})$ is degree d in all variables, then $r(X)$ is a univariate polynomial of degree d.

- **Multiplications**: Multiplication of two MLE's gives a multivariate polynomial. In general $\prod_{i=1}^{m} F_i(X_1, X_2, \ldots X_n)$ is degree m in X_i.
    - We assume that each polynomial in the product have the same MLE structure, i.e they are of the same dimension 2^n.
    - We do not need to implement this as it is stated above and unwrap the tensor structure, since this approach is expensive with naive multiplication. The sumcheck algorithm computes these products efficiently in terms of evaluations of the individual MLE's in the product.

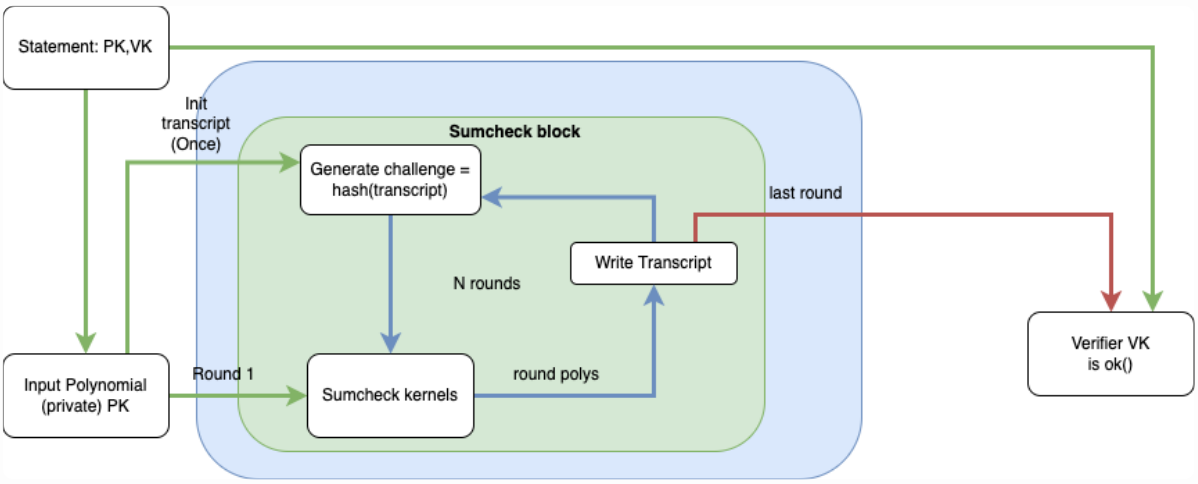
- We do not consider division of MLEs.

- **eq_poly**: This is a special indicator function:

$$eq_n(\vec{X}, \vec{Y}) = \prod_{i=1}^{n} \left(\bar{X_i} \cdot \bar{Y_i} + X_i \cdot Y_i \right)$$

  This has the property that $eq(\vec{X}, \vec{Y}) = 1$ iff $\vec{X} = \vec{Y}$. Note that this is just the Lagrange basis polynomial in 2 dimensions.

Overall architecture



Arbitrary products pseudo code for sumcheck

From Sumcheck201 Algorithm 3 is the most general algorithm for large fields and any size product.

Algorithm 3 Recursive Sumcheck for product MLE Algorithm 3

Define: $C = \sum_{0,1}^{n} \text{combine}(F_1, F_2, \ldots, F_z)$ ▷ Max degree of the combine is m
Input: z MLE's each of length $N = 2^n$, $F_1, F_2, \ldots F_z, C, \text{combine}(F_1, F_2, \ldots, F_z)$
let $T = [\text{public}]$
let $\alpha_0 \leftarrow \text{hash}(T[0], C)$
for $p = 0, 1, 2 \ldots, n-1$ **do** ▷ $\log_2 N$ rounds
  ▷ The degree of the round polynomial is the highest number of products in combine
  Phase 1: Accumulation
  let $r_p := [0 : m]$ ▷ $m+1$ 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**
      $r_p[k] += \text{combine}\left(\{F_l[i] \cdot (1-k) + F_l[i + 2^{n-p-1}] \cdot k\}_{l \in [z]}\right)$
    end for
  end for
  Phase 2: Challenge generation
  $T.\text{append}(r_p)$
  $\alpha_p \leftarrow \text{hash}(\alpha_{p-1}, r_p)$
  Phase 3: Intermediate representation
  for $i = 0, 1, \ldots, 2^{n-p-1} - 1$ **do**
    for $l = 0, 1, \ldots z - 1$ **do** ▷ $\bar{\alpha} = 1 - \alpha$
      $F_l[i] = \bar{\alpha}_p \cdot F_l[i] + \alpha_p \cdot F_l[i + 2^{n-p-1}]$ ▷ Update each polynomial array
    end for
  end for
end for
return T

In the above diagram $\bar{\alpha} = 1 - \alpha$

General API

In CPU we defined the API

```
pub fn prove<G, C>(
        prover_state: &mut ProverState<F>,
        combine_function: &C,
        transcript: &mut Transcript,
    ) -> SumcheckProof<F>
```

the output is a proof with the properties

```
        SumcheckProof {
            num_vars: prover_state.num_vars,
            degree: r_degree,
```

```
            round_polynomials: r_polys,
        }
```

Combine function

The combine function C is the lambda function applied in line 10, Arbitrary product and linear combinations of product is unfeasible for GPU due to non universality of phase 1. (more on this later) Thus optimization depends on the specific functional form of the API call.

A round in sumcheck is the sequence in the loop 5.

Prover State

The prover is initialized with the following data

1. MLE Polynomials involved in the sumcheck
2. Degree of the round poly (=Max degree of products in Sumcheck expression)

The intialization also defines a **Prover state**,

```
pub struct ProverState<F: PrimeField> {
    /// sampled randomness (for each round) given by the verifier
    pub randomness: Vec<F>,
    /// Stores a list of multilinear extensions
    pub state_polynomials: Vec<LinearLagrangeList<F>>,
    /// Number of variables
    pub num_vars: usize,
    /// Max number of multiplicands in a product
    pub max_multiplicands: usize,
    /// The current round number
    pub round: usize,
}
```

- the two quantities that are updated at the end of each round are
    - round number
    - state polynomials (line 19 in phase 3)

Transcript protocol

The transcript functionality implements the Fiat-Shamir transformation to make the proof non interactive. The implementation for Merlin can be used as a model.

- An important decision to be made is whether to implement this sub protocol in ICICLE
- It has two functionalities
    - Transcript Operations
        - Initialize transcript with prover state.
        - Append prover state - once per round.

    - Generating randomness based on a hasher.
- With the transcript data, verifier must be able to generate exactly the same randomness as generated by the prover, thus rendering the proof to be non interactive.

Kernels for API

1. **Phase 1: accumulation_kernel** depends on the combine function and is non-universal, is user specified, and API needs to be defined for specific combine functions for optimal use. However, this can be dealt with in a few ways we suggest below and still maintain some level of universality in API design.
2. **Phase 2: Challenge Generation** is universal subject to hash function specification of the API call, generally Keccak should be sufficient. We need a provide specification of transcript hasher, public data if any to be appended in transcript.
3. **Phase 3: Intermediate representation** is a fold function that updates the vector for the next round, which is universal, only depends on number of vectors and challenge in a round, but operates independently on each vector (inner loop 18-20)

Phase 1: Accumulation phase

- It is non universal, parallelizable.
- Specify combine function. Example R1CS:

$$eq(\vec{X}) \cdot \Big(A(\vec{X}) \cdot B(\vec{X}) - C(\vec{X}) \Big)$$

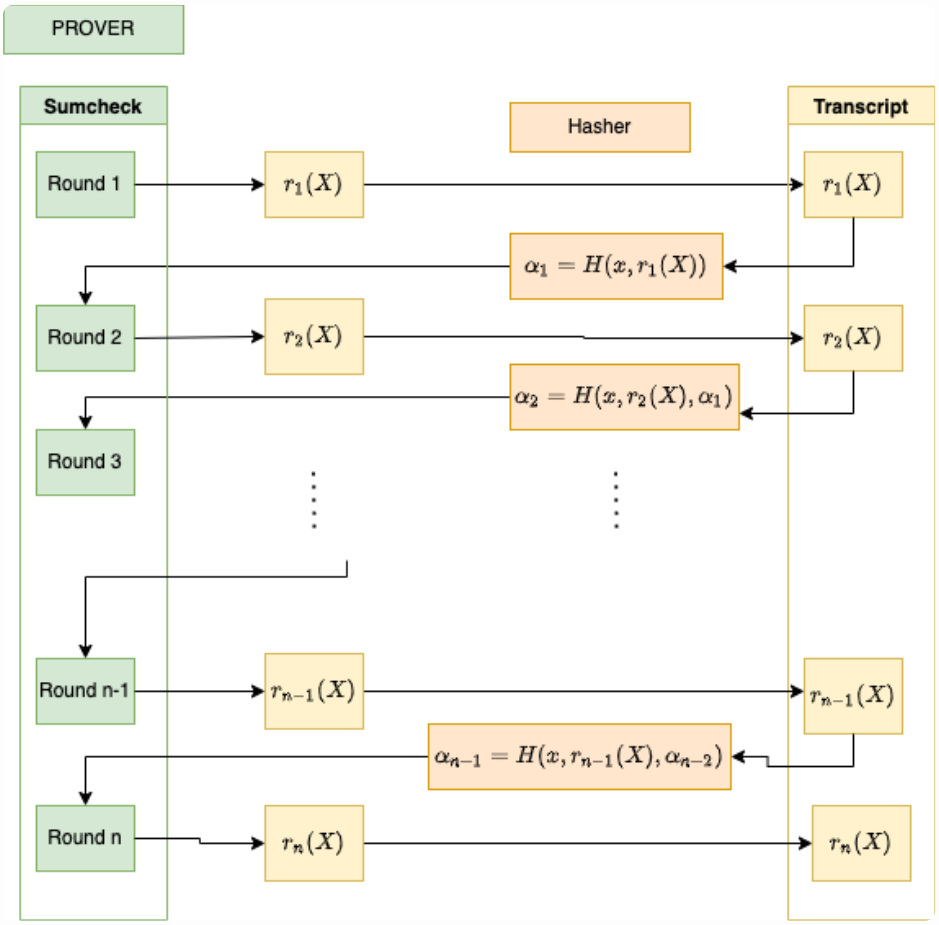
- Eg lets take n=3, i.e vector size 8, highest degree = 3
    - the highest degree is three, and thus the round polynomials are cubic, and required four evaluations, i.e k=0,1,2,3 in line 8
    - for round 1 :$p = 0$, and in line 9; $i = 0, 1, 2, 3$

```
for k=0,1,2,3 {
  for i = 0,1,2,3 {
   r_1[k] += (A[i] * (1-k) + A[i+4]* k) * (B[i] * (1-k) + B[i+4] * k)
               * (eq[i] *(1-k) +eq[i+4]*k)
             - (C[i] * (1-k) + C[i+4] * k)* (eq[i] *(1-k) +eq[i+4]*k)
  }
}
```

- For $k = 0, 1$ any $r_i(0), r_i(1)$ are straightforward, for $k \geq 2$ there are linear combinations of odd/even terms in each product. There can be many efficient ways to do this using karatsuba - consult @yuvaldomb @suyashbagad.
- Requires the initial evaluations of A, B, C, e in memory initially and then this is updated in line 19. The size decreases by 2 in each round. (see here)

Phase 2: Challenge generation and Transcript protocol

- It is universal,does not depend on the combine function, and is an essential functionality for sumcheck to be sound.
  In the case of sumcheck the transcript functionality is explained in the following picture.



1. It records the prover output of the sumcheck protocol per round in a sequential manner.
   - x is a public input
   - r_i are $d+1$ Field elements each, if the max degree of the sumcheck is d. Thus in the case of single MLE, each round output is two field elements. This is called a prover state, and the prover records the prover state in the transcript at the end of each round.

2. Using a hasher, generates a random field element such that the output per round is hash chained to all the inputs in the previous round.
   - The hasher can be any hash, though in general Keccak is commonly used eg: Jolt

3. If the sumcheck protocol has n rounds, the prover records n states and the hasher activates $n-1$ times for the prover.
4. The verifier code must use the same transcript and hasher, and also in addition generate α_n to do the final check.

Generally speaking Fiat Shamir implemented in the following way using hash chaining is secure.

$$\alpha_i = Hash(x, i, \alpha_{i-1}, r_i)$$

Example usage in supersumcheck

User needs to provide certain transcript/proof functionalities. Which we cover in FS_for_sumcheck.md

Phase 3: Intermediate representation (fold kernel)

This is extremely straightforward, and its only functionality is fold on randomness, and uniformly apply it for each polynomial seperately in the prover state.

```
// update prover state polynomials
  for j in 0..prover_state.state_polynomials.len() {
    prover_state.state_polynomials[j].fold_in_half(alpha);
  }
```

- It is universal, highly parallelizable and does not depend on the combine function.