

GPU Acceleration of the Sum-Check Protocol Over Towers of Binary Fields for Verifiable Computing

Andrew Fan*

*Palo Alto High School &
George Mason University*
Fairfax, US
afan4@gmu.edu

Harry Han*

*Thomas Jefferson High School for Science and Technology &
George Mason University*
Fairfax, US
harryhjhan@gmail.com

Yanze Wu

*Cyber Security Engineering Department
George Mason University*
Fairfax, US
ywu42@gmu.edu

Md Tanvir Arafin

*Cyber Security Engineering Department
George Mason University*
Fairfax, US
marafin@gmu.edu

Abstract—Emerging zero-knowledge proof protocols such as Binius and Binius-FRI operate over towers of binary fields, allowing for ultra-fast polynomial commitments over a base field. Sum-check, a key protocol in algebraic proof systems, is one of the key implementation bottlenecks for Binius and similar protocols. While sum-check is a massively parallel algorithm, GPU acceleration of sum-check has received little attention due to the lack of native GPU support for binary field multiplication. Hence, in this paper, we explore the key issues in existing GPU-based sum-check accelerators and present SumCATS - an efficient GPU implementation for sum-check acceleration. SumCATS leverages two fundamental improvements over the existing solutions. First, it adapts a CPU-based algorithmic improvement to sum-check proving and applies it to GPUs by recognizing the reduction pattern and shared memory optimizations. Secondly, SumCATS reduces the number of global memory accesses by precomputing products of random challenges and using base field operations to reconstruct extension field elements. When these optimizations are combined, SumCATS achieves a significant speedup ($1.81\times$ on NVIDIA RTX 3090 Ti, $1.62\times$ on NVIDIA A100) over the baseline GPU implementation (Binius-GPU) for sum-check over binary tower fields. The code and research artifacts for SumCATS design are available at https://github.com/SPIRE-GMU/sum_cats.

Index Terms—GPU Acceleration, Binary Fields, Sum-Check

I. INTRODUCTION

SNARKs (Succinct Non-interactive ARGument of Knowledge) are a type of cryptographic protocol that allows a prover to prove their knowledge of some witness that satisfies a certain property. SNARKs often allow the prover to do this without revealing information about the witness itself, and allow for very short (succinct) transactions. Most SNARK designs combine sum-check or other polynomial Interactive Oracle Proof (IOP) protocols with a polynomial commitment scheme to generate an interactive proof protocol, and then apply the Fiat-Shamir heuristic [1] to make them non-interactive.

*This work was performed at George Mason University, where Andrew Fan and Harry Han was a part of the Aspiring Scientists Summer Internship Program.

The sum-check protocol [2] is a fundamental building block in the design of many modern SNARKs such as Binius [3], [4]. It provides an efficient way for a prover to prove the sum of a polynomial without revealing the entire polynomial. Sum-check, through a recursive algorithm, reduces a claim to the evaluation of the polynomial at a single random point in the field. The commitment step of such SNARK protocols is fast, leaving sum-check as the main bottleneck in the operation.

Algorithms for accelerating the sum-check protocol have been explored thoroughly [2], [5], [6]. However, using GPUs for sum-check presents a fundamental challenge. For binary tower fields, addition is defined using bitwise XOR, but there is no efficient hardware implementation for multiplication. Additionally, modern GPUs lack built-in instructions for algebraic number theory-based computations. These factors limit the speedup of applying GPUs to sum-check over a tower of binary fields. Hence, in this paper, we present a high-performance GPU implementation of the sum-check protocol over binary fields. Our key contributions are:

- We draw inspiration from previous algorithmic improvements [7] to build SumCATS: two GPU-based sum-check accelerators. The first algorithmic improvement does more base field operations before introducing extension field multiplications each round. The second improvement leverages fast base-extension field multiplications to reconstruct field elements on GPU threads, thereby significantly reducing the number of global memory accesses and enhancing the throughput of faster local memory.
- We also reorganize the storage of field elements to take advantage of faster shared memory and the highly efficient parallel reduction pattern.
- We implement SumCATS using the CUDA framework. On the NVIDIA RTX 3090 Ti, we achieve a speedup of $1.81\times$ on the largest benchmark test case. On the NVIDIA A100, we achieve a speedup of $1.62\times$ over the largest benchmark implementation [6].

II. PRELIMINARIES

A. Notation

This work involves binary fields denoted by $GF(2^m)$, where m is an integer. $\mathbb{B}, \mathbb{S}, \mathbb{E}$ denote the base, short extension, and extension fields (which are equivalent to $GF(2), GF(2^{2^2}), GF(2^{2^7})$ for the remainder of the text), and their elements are represented with $\mathbf{b}, \mathbf{s}, \mathbf{e}$. For this work, the \mathbb{S} and \mathbb{E} fields are defined using the towers of binary field extensions. We use the \parallel operator to denote bit concatenation and $t_{a:b}$ to denote the concatenation of all bits of an integer t , from the a -th bit to the b -th bit.

B. Towers of Binary Fields

We begin with an overview of a popular construction of towers of binary fields, [8], which is used in Binius. The base field \mathbb{B} has only two elements $\{0, 1\}$. Multiplication in \mathbb{B} is the binary AND operation, and addition is an XOR operation. To extend this field, we define a root of the irreducible polynomial α_0 , such that $\alpha_0^2 = \alpha_0 + 1$. Thus, we have an extended field $GF(2^{2^1})$ with elements $\{0, 1, \alpha_0, 1 + \alpha_0\}$ with the relation $\alpha_0^2 = \alpha_0 + 1$. Subsequently, we can extend this field further with $\alpha_1^2 = \alpha_1 \alpha_0 + 1, \alpha_2^2 = \alpha_2 \alpha_1 + 1$ etc. This construction is known as a tower of binary fields [8]. After h extensions, there will be 2^{2^h} elements in the resulting field $GF(2^{2^h})$.

An interesting property of binary tower fields is that elements of height h extension field $GF(2^{2^h})$ can be written as a linear combination of elements of the height $(h - 1)$ field. *i.e.*,

$$A = L + \alpha_{h-1}R \quad (1)$$

where $A \in GF(2^{2^h})$ and $L, R \in GF(2^{2^{h-1}})$. Thus, elements in a height- h extension field can be represented by a 2^h length string of bits; each bit represents the binary coefficient of the product of a subset of $\{\alpha_0, \alpha_1, \dots, \alpha_{h-1}\}$. For example, using Eq. 1, we can write elements of $GF(2^{2^1})$ as shown in Table I.

TABLE I
ELEMENT OF $GF(2^{2^1})$ AND THEIR BITSTRING REPRESENTATION

L	R	Element of $GF(2^{2^1})$	Bitstring Representation
0	0	$0 + 0\alpha_0 = 0$	00
1	0	$1 + 0\alpha_0 = 1$	10
0	1	$0 + 1\alpha_0 = \alpha_0$	01
1	1	$1 + 1\alpha_0 = 1 + \alpha_0$	11

C. Multiplying Elements in Binary Tower Fields

In an extended binary field, an addition is still performed with bitwise XOR. Here we give details on the multiplication of elements in an extended field. Assume we have two height- $(h + 1)$ extension field elements A and B . Using Eq. 1 we can write:

$$A = L_A + \alpha_h R_A, B = L_B + \alpha_h R_B$$

Then, using the tower construction property $\alpha_h^2 = \alpha_h \alpha_{h-1} + 1$, the product AB can then be expressed as

$$L_A L_B + \alpha_h (L_A R_B + L_B R_A) + \alpha_h^2 R_A R_B$$

$$= L_A L_B + R_A R_B + \alpha_h (L_A R_B + L_B R_A + \alpha_{h-1} R_A R_B).$$

This gives us a simple recursive algorithm for multiplying two extension field elements. We can also remove one multiplication from this expression and replace it with more additions using Karatsuba's trick:

$$AB = L_A L_B + R_A R_B + \alpha_h ((L_A + R_A)(L_B + R_B) - L_A L_B - R_A R_B + \alpha_{h-1} R_A R_B) \quad (2)$$

D. The Sum-Check Protocol

The sum-check protocol [2] is a fundamental building block in the design of many modern SNARKs. We start with an n -variate polynomial P of the form

$$P(x_1, \dots, x_n) = \prod_{i=1}^d p_i(x_1, \dots, x_n) \quad (3)$$

where each $p_i(x)$ is an n -variate multilinear polynomial, *i.e.*,

$$p_i(x_1, \dots, x_n) = \sum_{j_1 \in \{0,1\}} \sum_{j_2 \in \{0,1\}} \dots \sum_{j_n \in \{0,1\}} c_{j_1 j_2 \dots j_n} x_1^{j_1} x_2^{j_2} \dots x_n^{j_n}. \quad (4)$$

The sum-check protocol provides an efficient way for a prover to prove a claim (of the following form) to the verifier:

$$S = \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \dots \sum_{x_n \in \{0,1\}} P(x_1, \dots, x_n) \quad (5)$$

The sum-check protocol works recursively in multiple rounds, where each round reduces the polynomial by one variable. In this work, we focus on the cases where $c_* \in \mathbb{B}$ for $p_2 \dots p_d$ and $c_* \in \mathbb{E}$ for p_1 . This form is used in protocols such as Binius and Binius-FRI [3], [4].

Round 0. First, the prover commits the claimed sum S and sends it to the verifier. Then, the prover computes the polynomial

$$S_0(y) = \sum_{x_2 \in \{0,1\}} \dots \sum_{x_n \in \{0,1\}} P(y, x_2, \dots, x_n)$$

and sends S_0 to the verifier. The verifier checks that $S_0(0) + S_0(1) = S$. If the check fails, the verifier rejects the proof. Next, the verifier samples $r_0 \in \mathbb{E}$, and sends r_0 to the prover.

Round i . There are now i verifier-issued random challenges r_0, r_1, \dots, r_{i-1} . The prover calculates the polynomial

$$S_i(y) = \sum_{x_{i+2} \in \{0,1\}} \dots \sum_{x_n \in \{0,1\}} P(r_0, r_1, \dots, r_{i-1}, y, x_{i+2}, \dots, x_n) \quad (6)$$

and sends it to the verifier. The verifier checks that $S_i(0) + S_i(1) = S_{i-1}(r_{i-1})$, then generates a random challenge $r_i \in \mathbb{E}$, calculates $S_i(r_i)$ and sends r_i to the prover. The protocol then moves to round $i + 1$.

Final Round. After calculating $S_{n-1}(r_{n-1})$, the verifier evaluates $P(r_0, r_1, \dots, r_{n-1})$ using a single oracle query. This is typically achieved using a polynomial commitment scheme. The verifier checks that $S_{n-1}(r_{n-1}) = P(r_0, r_1, \dots, r_{n-1})$.

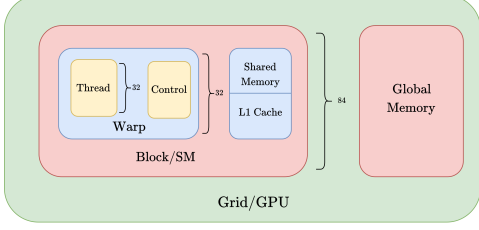


Fig. 1. Hierarchy of GPU memory and processing units, and their corresponding CUDA software components.

Thus, instead of computing the full 2^n sums to verify the value of S , the verifier can accept S via this sum-check protocol. The probability that a dishonest prover can pass this check is at most $dn/|\mathbb{E}|$. When we have a large extension field, such as $GF(2^{27})$, this probability becomes quite small.

The prover does not need to send the polynomial $S_i(y)$ at the i -th round. Instead, it suffices for the prover to evaluate $S_i(y)$ at $d+1$ points (typically at points $0, 1, \dots, d$) and send the evaluation points to the verifier. The verifier uses Lagrange interpolation on these points to discover $S_i(y)$. Sum-check on the verifier side requires a trivial amount of computation, but the prover requires significant computing to determine $S_i(y)$ as given in Eq. 6. Therefore, we will focus on accelerating the protocol on the prover side.

E. GPU Architecture and Programming

GPUs are built of multiple streaming multiprocessors (SMs), each with multiple cores. Each core has a control unit, multiple threads, and memory as shown in Fig. 1. Since each core only has a single control unit, the same instruction is run on all threads with different data. GPU memory architecture is split into multiple layers. The largest memory is the global memory, which is accessible by all threads but is very slow. Each SM has an L1 cache, which is faster than the global memory cache but can only be accessed by threads within an SM. Crucially, the movement of data between the L1 cache and global memory is hardware-managed, meaning the behavior of these data transfers might be unpredictable. To counteract this, the L1 cache memory can also be split into shared memory, which behaves similarly but can be allocated and accessed through code. This work focuses on leveraging the GPU memory architecture to accelerate the sum-check protocol.

III. EXISTING ALGORITHMS FOR SUM-CHECK ACCELERATION

In this section, we detail two key algorithms, BDT-1 and BDT-2, described in [7], for accelerating the sum-check protocol. We begin with the multilinear polynomials p_1, p_2, \dots, p_d from Eq. 3. Initially, each polynomial is represented with indexed arrays $A_1[\vec{x}], A_2[\vec{x}], \dots, A_d[\vec{x}]$, where $A_j[\vec{x}] = p_j(\vec{x})$, and $\vec{x} = \{x_1, x_2, \dots, x_n\}$. We define the following operations:

a) *Folding*: The folding procedure takes in array A_j and coefficient c and outputs an array A'_{jc} with half the length of A_j such that

$$A'_{jc}[\vec{x}] = c \cdot (A_j[0||\vec{x}] - A_j[1||\vec{x}]) + A_j[0||\vec{x}],$$

where $A_j[0||\vec{x}]$ represents the lower half of array A_j and $A_j[1||\vec{x}]$ represents the upper half of array A_j .

b) *Composition*: Composition takes in the arrays A_1, A_2, \dots, A_d and outputs an array H such that $H[\vec{x}] = \prod_{j=1}^d A_j[\vec{x}]$.

A. BDT-1: Sum-check over Arbitrary Multilinear Polynomials

For an arbitrary round i , the prover can compute each interpolation point $S_i(0), S_i(1), \dots, S_i(d)$ by folding each array about the interpolation point, composing the folded arrays, then adding the compositions. Therefore,

$$S_i(y) = \sum_{\vec{x}' \in \{0,1\}^{n-i-1}} \prod_{j=1}^d A'_{jy}[\vec{x}'] = \sum_{\vec{x}' \in \{0,1\}^{n-i-1}} H[\vec{x}']. \quad (7)$$

where $\vec{x}' = \{x_{i+1}, x_{i+2}, \dots, x_n\}$. To see why BDT-1 works, note that for an arbitrary multilinear polynomial $q_j(\vec{x}') = p_j(r_0, r_1, \dots, r_i, \vec{x}')$, we have

$$\begin{aligned} q_j(y, \vec{x}') &= (1-y)q_j(0, \vec{x}') + yq_j(1, \vec{x}') \\ &= (1-y)A_j[0||\vec{x}'] + yA_j[1||\vec{x}'] \\ &= y(A_j[0||\vec{x}'] - A_j[1||\vec{x}']) + A_j[0||\vec{x}'], \end{aligned} \quad (8)$$

which is the same as folding A_j with y as the coefficient.

To move on to the next round using the verifier's random challenge r_i , we can simply fold each array with r_i , setting each array A_i to A'_{ir_i} .

Let's calculate the number of multiplications needed in round i . After folding i times, each array in round i has 2^{n-i} entries. To calculate the initial claimed value, we need to perform $(d-1) \cdot 2^{n-i}$ **ee** multiplications ($d-1$ multiplications for each element). To fold over each interpolation point, we perform $(d+1)(d-1)2^{n-i-1}$ **se** multiplications in total. To compose these folded arrays and find each interpolation value, we perform $(d+1)(d-1)2^{n-i-1}$ **ee** multiplications. Finally, to fold for the next round, we perform $(d-1) \cdot 2^{n-i-1}$ **ee** multiplications.

B. BDT-2: Optimizing For Small Fields

In the 1st round of BDT-1, each array A_i has binary values and the interpolation points are from the short extension field; therefore, computation in Eq. 7 is dominated by cheap **bb** multiplications. However, after the first round, since the random challenge $r \in \mathbb{E}$ is introduced, the prover can no longer take advantage of **bb** operations. The key idea for BDT-2 is to separate base field operations from the extension field elements to take advantage of **bb** operations.

We now explain BDT-2 for an arbitrary sum-check round. To calculate $S_i(y)$, we first split p_j into a linear combination of multiple components:

$$p_j(r, y, \vec{x}) = \sum_{k \in \{0,1\}^{i+1}} \left(\prod_{l=1}^i V(k_l, r_l) \right) \cdot A_j[k||\vec{x}], \quad (9)$$

where $V(1, a) = a$ and $V(0, a) = 1 - a$. For example, when $i = 1$, we have

$$\begin{aligned} p_j(r_1, y, \vec{x}) &= (1-r_1)(1-y)A_j[0|0||\vec{x}] + \\ & r_1(1-y)A_j[1|0||\vec{x}] + (1-r_1)yA_j[0|1||\vec{x}] \\ & \quad + r_1yA_j[1|1||\vec{x}] \end{aligned} \quad (10)$$

Let

$$B(k_1, k_2, \dots, k_d) = \sum_{\vec{x} \in \{0,1\}^{n-i-1}} A_1[k_1|\vec{x}]A_2[k_2|\vec{x}] \cdots A_d[k_d|\vec{x}].$$

And $g(j, k) = j_{(k-1)(i+1)+1:k(i+1)}$. Then, we can express $S_i(y)$ as a linear combination of evaluations of B :

$$S_i(y) = \sum_{j \in \{0,1\}^{d(i+1)}} \left[\prod_{k=1}^d \left(\prod_{l=1}^i V(j_{k(i+1)+l}, r_l) \right) \cdot V(j_{k(i+1)+(i+1)}, y) \right] \cdot B(g(j, 1), g(j, 2), \dots, g(j, d)). \quad (11)$$

Since each $p(r, y, x)$ has 2^{i+1} terms, there are $(2^{i+1})^d = 2^{di+d}$ terms in the linear combination. Furthermore, each evaluation of B multiplies 2^{n-i-1} binary field elements.

C. GPU Implementation of Bitsliced BDT-1

Currently, only BDT-1 for sum-check over tower fields has been implemented on GPUs [6]. It achieved a $5\times$ speedup over the same algorithm on CPU. This implementation uses bitslicing that stores the arrays A_1, A_2, \dots, A_d using batches of 128 unsigned integers, with each integer representing the same bit within 32 field elements. This allows for 32 multiplications in parallel instead of just 1 by using bitwise operations on the integers in the array.

Folding and composition are also implemented on the GPU. For folding, one thread is assigned to each output batch, and folding is applied to the two input batches that are folded into the output batch. Additionally, a grid-stride technique is used to accommodate large arrays. The composition-then-add algorithm similarly assigns a thread to each batch in the output composition. For each corresponding input batch, the thread does a running multiplication using the unrolled Karatsuba algorithm. Then, each thread accumulates its result with an atomic XOR operation. We start with this baseline GPU implementation to accelerate the sum-check protocol.

IV. DESIGN OF SUMCATS

A. SumCATS-1: Optimized GPU Implementations of Composite BDT-1 and BDT-2

SumCATS-1 is built upon three key components: (1) combining BDT-1 and BDT-2 on GPU, (2) parallel reduction, and (3) improving BDT-1 on base-field polynomials.

1) *Combining BDT-1 and BDT-2*: While CPU implementations of BDT-1 and BDT-2 have been explored and have been successful [7], these algorithms are highly parallel. A complexity analysis of the two algorithms reveals that BDT-2 becomes increasingly expensive as the number of rounds increases, whereas BDT-1 becomes less expensive as the rounds progress. Therefore, it is optimal to switch from BDT-2 to BDT-1 when the number of operations in BDT-2 for that round exceeds the number of operations in BDT-1. To find the optimal switchover point, we plot the estimated number of operations for each round using the complexity calculations for BDT-1 and BDT-2. We find that optimal switch rounds are 3, 2, and 2, for $d = 2, 3, 4$ respectively.

Algorithm 1 Optimized GPU Kernel for BDT-2

Require: Unbitsliced array of extension field elements A_1 , representing coefficients of p_1 . Bitsliced arrays of base field elements A_2, \dots, A_d , representing coefficients of p_2, \dots, p_d . Round index i , number of polynomial variables n , composition size d . thread index t .

Ensure: $B = \sum_{x \in \{0,1\}^{n-i-1}} A[g(j, 1)|x] \cdots A[g(j, d)|x]$ for specific j .

$B \leftarrow 0$ $\triangleright B \in \text{GF}(2^{128})$

for all thread t do

$\text{idx} \leftarrow t$

while $\text{idx} < \text{Number of Batches}$ **do**

Get corresponding base field batch product

$p \leftarrow A_2[g(j, 2)|\text{idx}] \cdots A_d[g(j, d)|\text{idx}]$

$\text{acc} \leftarrow 0$ $\triangleright \text{acc} \in \text{GF}(2^{128})$

for $k = 0 \dots 31$ **do**

Multiply bit of $p[k]$ by corresponding A_1 element

$\text{acc} \leftarrow \text{acc} + p[k] \cdot A_1[g(j, 1)|32 \cdot \text{idx} + k]$

end for

Store acc in shared memory $s[t]$

Perform reduction on s , obtaining final sum S

$B \leftarrow B + S$

$\text{idx} \leftarrow t + \text{Number of threads}$

end while

end for

return B

2) *Parallel Reduction*: A useful parallel algorithm for optimizing both BDT-1 and BDT-2 is reduction [9]. Given some array of data a_1, a_2, \dots, a_n and some associative operator \oplus , we want to compute $a_1 \oplus a_2 \oplus \dots \oplus a_n$. A binary tree structure can be used; each thread computes the operation \oplus for two elements, halving the number of elements added together until there is just one element. Furthermore, we can take advantage of shared memory by loading segments of the array into shared memory, then performing reduction within a block on the shared memory, and finally accumulating the results of each block using atomic operations. Furthermore, we can rearrange the active threads so that the active threads correspond to the leftmost active array elements, reducing control divergence.

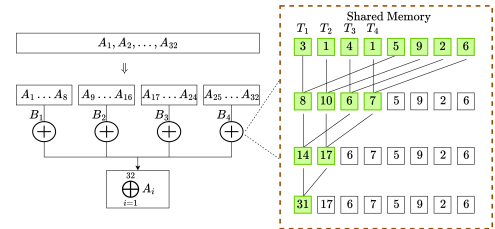


Fig. 2. Example of parallel reduction for addition for a list of 32 integers.

3) *GPU Implementation of BDT-2 On Base Field Polynomials*: Recall that we want to compute the values of $S_i(y)$ for all $y \in \{0, 1, \dots, d\}$ using Equation 11. To do so, we iterate on each term of the summation with an index $j \in \{0, 1, \dots, 2^{di+d}\}$. For each term, we precompute the

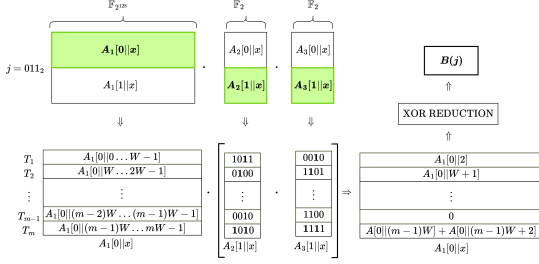


Fig. 3. Diagram of parallelized and optimized BDT-2, where W is the bit width of the polynomial evaluation batches, and m is the number of batches.

partial product of the random challenges and the value of $B(g(j, 1), g(j, 2), \dots, g(j, d))$. We note that while there are only 2^{di+d} **ee** computations per round to calculate random challenge products, there are $2^{di+d} \cdot 2^{n-i-1}$ **bb** operations for calculating B , which incurs a much larger cost. Therefore, it suffices to parallelize the products and sum operations over the base field elements in the arrays A_1, A_2, \dots, A_d , and do the rest on the CPU.

Similar to the GPU optimization of BDT-1, we batch the base field polynomial evaluations, packing each bit into a batch of 32 bits in a single 32-bit unsigned integer. Each thread in a block corresponds to a composition of a single batch in each array. A thread calculates the element-wise product of each batch using a single bitwise AND. Then, a reduction is applied using bitwise XOR to calculate the sum for the thread's current assigned B value.

Finally, after computing each B array element, we use the CPU to compute the linear combinations for each interpolation point. For each interpolation point, the CPU uses the precomputed random challenge products to calculate the coefficient of each B array element, then computes the linear combination. We use BDT-2 for the first rounds, then switch to the GPU implementation of BDT-1 at the switchover round. To switch to BDT-1 in round i , it suffices to fold each array A_1, A_2, \dots, A_d i times. Overall, SumCATS-1 uses Alg. 1 and the GPU implementation of BDT-1.

B. SumCATS-2: Optimizing BDT-1 For Base Fields

We further optimize the composition and folding operations in BDT-1 to use more base field operations during earlier rounds [5] and to improve the compute throughput and occupancy of BDT-1's kernel.

1) *Constructing Folded Batches Using Small Field Multiplications*: The key idea is that instead of using costly global memory accesses to load 128-bit extension field elements, we can instead load multiple 1-bit base field elements, then construct the 128-bit field element using the base field elements and random challenges. Recall the formula for folding an array of field elements using a random challenge r : $A'_{jr_i}[k] = (1-r_i)A_j[0][k] + r_iA_j[1][k]$. By repeating the folding operation and distributing, we can get a general form for i folds and

Algorithm 2 *get_batch*

Require: Bitsliced array of base field elements A_j representing coefficients of p_j . Batch index idx. Round index i . Random challenge products $R[0] \dots R[2^i - 1]$ where $R[k] = V(k_1, r_1)V(k_2, r_2) \dots V(k_i, r_i)$. Number of batches m . Number of polynomial variables n .

Ensure: Reconstructed batch $A_{jr_1, r_2, \dots, r_i}^i[\text{idx}]$.

function GET_BATCH(A_j , idx, R , i)

$a \leftarrow 0$; $l \leftarrow 0$; $k \leftarrow \text{idx}$

while $k \leq m$ **do**

$a \leftarrow a + R[l] \cdot A_j[k]$; $l \leftarrow l + 1$; $k \leftarrow k + 2^{n-i}/32$

end while

return a

end function

Algorithm 3 Optimized BDT-1 Using *get_batch*

Require: Bitsliced array of extension field elements A_1 , representing coefficients of p_1 . Bitsliced array of base field elements $A_2 \dots A_d$, representing coefficients of $p_2 \dots p_d$. Precomputed random challenge products $R[0] \dots R[2^i - 1]$. Round index i , number of polynomial variables n , composition size d . thread index t . Interpolation point y . Number of batches m .

Ensure: Evaluation $S = \sum_{x \in \{0,1\}^{n-i-1}} P(r_0, r_1, \dots, y, x)$

for all thread t **do**

idx $\leftarrow t$

while idx $< m/2$ **do**

$p \leftarrow A_1[\text{idx}]$

for $k = 2 \dots d$ **do**

upper $\leftarrow \text{get_batch}(A_k, \text{idx} + m/2, R, i)$

lower $\leftarrow \text{get_batch}(A_k, \text{idx}, R, i)$

folded $\leftarrow y \cdot (\text{upper} + \text{lower}) + \text{lower}$

$p \leftarrow p \cdot \text{folded}$

end for

Store p in shared memory $s[t]$

end while

Perform reduction on s , obtaining final sum S

end for

return S

arbitrary random challenges:

$$A_{jr_1, r_2, \dots, r_i}^i[k] = \sum_{l \in \{0,1\}^i} \left(\prod_{u=1}^i V(l_u, r_u) \right) \cdot A[l_j || l_{j-1} || \dots || l_2 || l_1][k]. \quad (12)$$

Furthermore, the products of each combination of $V(l_u, r_u)$ can be precomputed before launching a kernel, leading to negligible overhead from **ee** multiplications. Overall, this operation is computationally inexpensive because it only uses **be** multiplications and extension field additions. This operation can easily be extended to batches of 32 elements in the arrays. Furthermore, note that this operation requires significantly fewer global memory accesses than directly folding and obtaining the batch; we only access 2^j 32-bit integers to construct a single batch, instead of accessing 128 32-bit integers. Similar to BDT-2 for

TABLE II

TIMING RESULTS ON THE RTX 3090 AND A100 GPUS COMPARING BDT-2 WITH AND WITHOUT BASE FIELD/SHARED MEMORY OPTIMIZATIONS, AND THE BASE GPU IMPLEMENTATION.

n	d	RTX 3090					A100				
		Base (ms)	SumCATS-1 (ms)	SumCATS-2 (ms)	Speedup SumCATS-2	Speedup SumCATS-1	Base (ms)	SumCATS-1 (ms)	SumCATS-2 (ms)	Speedup SumCATS-2	Speedup SumCATS-1
20	2	30.824	38.873	24.415	0.79×	1.26×	19.500	58.690	40.620	0.33×	0.48×
20	3	49.796	55.172	40.917	0.90×	1.22×	34.878	59.935	65.280	0.57×	0.52×
20	4	79.356	73.778	63.388	1.08×	1.25×	57.927	79.979	98.470	0.72×	0.59×
24	2	159.94	114.99	109.72	1.39×	1.46×	131.32	119.63	108.65	1.10×	1.21×
24	3	277.01	183.18	187.11	1.51×	1.48×	223.91	183.86	181.15	1.22×	1.24×
24	4	440.99	279.00	304.88	1.58×	1.45×	352.46	272.60	320.13	1.29×	1.10×
28	2	2056.6	1191.6	1336.2	1.73×	1.54×	1819.7	1165.9	1350.5	1.56×	1.35×
28	3	3751.1	2070.6	2366.5	1.81×	1.59×	3154.9	1941.4	2239.5	1.62×	1.41×
28	4	5891.6	3406.1	3880.4	1.73×	1.52×	4783.1	3020.8	3999.2	1.58×	1.20×

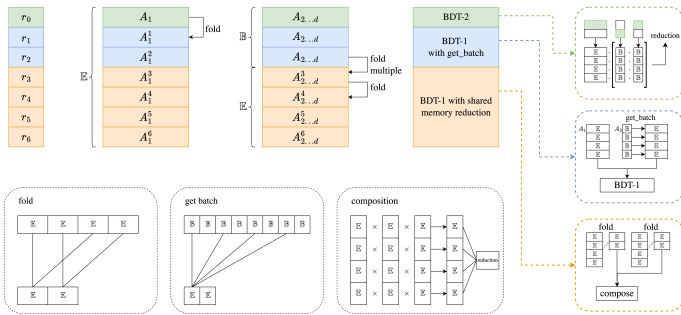


Fig. 4. High-level overview all the discussed optimizations in SumCATS-2. In this example, there are 7 rounds. We switch to the *get_batch* BDT-1 (Alg. 3) in Round 1, then switch to the optimized base BDT-1 in Round 3.

TABLE III

SYSTEM SPECIFICATIONS FOR THE THREE ENVIRONMENTS WE TESTED OUR OPTIMIZATIONS ON.

Environment	RTX 3090	A100
Device	GeForce RTX 3090 Ti	A100 SXM4 80 GB
SM Count	84	108
Max Clock Speed	1695 MHz	1410 MHz
Host CPU	Intel i9 13th Gen 13900K	AMD EPYC 7543
Core count	32	32
Max Clock Speed	5.8 GHz	3.7 GHz
OS	Ubuntu 22.04 LTS	Rocky Linux 8.10

finding interpolation points, it is optimal to switch from using *get_batch* operations to directly folding and accessing 128-bit batches at a certain switchover round; however, the optimal round was instead experimentally determined.

The *get_batch* operation can also be applied to efficiently fold an array multiple times. Instead of repeating the folding operation j times, we can simply replace each element of A with its constructed batch using the *get_batch* operation.

2) *Improving Memory Usage of BDT-1*: We further improve the memory usage of BDT-1 by utilizing shared memory. Currently, threads stride through the entire array A of polynomial evaluations, compute compositions, and accumulate the composition to a locally stored sum. However, among other stored field elements, this uses a very high number of registers that overflows into the L1 cache and GPU global memory. Therefore, we move this stored accumulated sum to block-level shared memory; then, we run a block-level reduction and a final

atomic XOR using the reduction algorithm previously discussed on batch addition. We call SumCATS-1 combined with these optimizations (Alg. 2 and 3) SumCATS-2.

V. RESULTS

We tested SumCATS-1 and SumCATS-2 and compared run times on two GPU systems: an NVIDIA RTX 3090 Ti and an NVIDIA A100, and report it in Table II. We find that for large testcases with more variables, our algorithm with both optimizations performs much better (1.81× on RTX 3090 and 1.62× on A100) than the base implementation of only BDT-1. However, we find that on low-degree polynomials of only 20 variables, the optimized algorithm often performs worse than the base algorithm. We attribute this to the amount of CPU computations needed for both optimizations. For BDT-2, the coefficients of each term are not highly parallelizable because of the low number of terms; therefore, this work must be done on a CPU. For the *get_batch* optimization in BDT-1, the folding coefficients must similarly be precomputed on CPU. Because there are only $2^{20} \approx 10^6$ elements in the arrays, there is less room for parallelization for lower n . So, the extra CPU cost incurred actually makes the algorithm slower.

VI. CONCLUSION

In this work, we present SumCATS, a combination of multiple GPU-based optimizations to efficient algorithms for sum-check over binary tower fields. We adapt previous algorithmic improvements to sum-check to a highly parallel computing paradigm and apply well-known GPU patterns to further optimize our algorithm. Our work addresses a crucially overlooked gap in sum-check algorithms: parallelization. Future work may look at adapting the algorithms for more advanced GPU optimizations and patterns, such as warp-level primitives and tensor cores.

REFERENCES

- [1] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 186–194.
- [2] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, "Algebraic methods for interactive proof systems," *J. ACM*, vol. 39, no. 4, p. 859–868, Oct. 1992. [Online]. Available: <https://doi.org/10.1145/146585.146605>

- [3] B. E. Diamond and J. Posen, "Succinct arguments over towers of binary fields," Cryptology ePrint Archive, Paper 2023/1784, 2023. [Online]. Available: <https://eprint.iacr.org/2023/1784>
- [4] B. E. Diamond and Posen, "Polylogarithmic proofs for multilinear over binary towers," Cryptology ePrint Archive, Paper 2024/504, 2024. [Online]. Available: <https://eprint.iacr.org/2024/504>
- [5] S. Bagad, Q. Dao, Y. Domb, and J. Thaler, "Speeding up sum-check proving," Cryptology ePrint Archive, Paper 2025/1117, 2025. [Online]. Available: <https://eprint.iacr.org/2025/1117>
- [6] J. Posen and D. Yasny, "Slicing up binary towers: Accelerating sumcheck on gpu," Nov. 2024. [Online]. Available: <https://www.irreducible.com/posts/slicing-up-binary-towers>
- [7] S. Bagad, Y. Domb, and J. Thaler, "The sum-check protocol over fields of small characteristic," Cryptology ePrint Archive, Paper 2024/1046, 2024. [Online]. Available: <https://eprint.iacr.org/2024/1046>
- [8] D. Wiedemann, "An iterated quadratic extension of $gf(2)$," *The Fibonacci Quarterly*, vol. 26, no. 4, pp. 290–295, 1988.
- [9] M. Harris *et al.*, "Optimizing parallel reduction in cuda," *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.