Chimera: A Hybrid GPU Backend for Sumcheck Acceleration in Zero-Knowledge Provers

Kashfia Farheen and Nektarios Georgios Tsoutsos

University of Delaware
{farheen, tsoutsos}@udel.edu

Abstract. Sumcheck is increasingly used in proof systems based on multilinear polynomials, but its execution structure differs from that of conventional GPU targets such as MSM and NTT. Each round contains substantial data-parallel polynomial computation, but consecutive rounds depend on challenges derived from previous rounds and interact with the surrounding prover's proof-generation operations. Efficient acceleration therefore depends not only on GPU arithmetic throughput, but also on where polynomial data resides, what crosses the CPU–GPU boundary, and when execution should return to the CPU.

We present Chimera, a hybrid CPU–GPU backend for Sumcheck computations in the Spartan R1CS prover. Chimera keeps large polynomial data on the GPU across profitable rounds, returns only the compact polynomial values needed by Spartan's transcript, and sends the resulting challenge back to the GPU for folding. It switches sufficiently small late rounds to Spartan's native CPU implementation. When polynomial data exceeds GPU memory, Chimera processes it in smaller portions and transitions to resident GPU execution once the remaining data fits. Chimera preserves Spartan's proof format and verifier behavior while also reducing CPU-side overhead directly surrounding the accelerated Sumcheck computations.

On BN254 Spartan R1CS instances, Chimera reduces the measured Sumcheck-related region from 672.785 ms to 84.245 ms at $N = 2^{17}$, a 7.99× speedup. Complete proving time improves at every measured size, reaching 1.27×–1.64× speedup for $N = 2^{24}$–2^{27}. Separately, our standalone quadratic GPU Sumcheck executes in 28.7 ms at $N = 2^{22}$, compared with 84.2 ms for ICICLE's CUDA Sumcheck implementation, a 2.94× GPU-to-GPU improvement. These results show that effective Sumcheck acceleration depends on both efficient GPU execution and careful integration with the surrounding proof system.

Keywords: Zero-knowledge proofs · Sumcheck · GPU acceleration.

1 Introduction

Zero-knowledge proofs (ZKPs) allow a prover to demonstrate that a statement is true without revealing the private information that supports the claim [20]. This property has made ZKPs an important building block for applications that require both correctness and privacy, including private payments, verifiable computation, and privacy-preserving data processing [5, 15, 34]. Modern proof systems can produce compact proofs that are inexpensive to verify, but expensive to generate [11, 17, 22, 33, 36]. As ZKPs scale to larger computations, reducing prover time becomes an important systems challenge.

A large body of work has studied and accelerated computationally intensive operations in proof generation using GPUs, distributed systems, and specialized hardware [12–14, 29, 30, 32, 35, 38, 40, 41, 46]. Much of this work focuses on large arithmetic operations, such as multi-scalar multiplication (MSM) and polynomial transforms, such as FFTs and NTTs, which expose large amounts of regular parallel work. More recent proof systems, however, increasingly use multilinear polynomials and rely on the Sumcheck protocol [10, 31, 36, 43, 44]. Recent accelerator designs have consequently begun to target Sumcheck directly, including full-system and programmable hardware accelerators [12, 13]. These efforts demonstrate that Sumcheck is an important acceleration target, but also highlight that its execution structure differs substantially from other acceleration targets.

Sumcheck verifies a claimed sum of multivariate polynomial evaluations through a sequence of dependent rounds that progressively make the computation smaller [31]. In each round, the prover compresses the current computation into a smaller polynomial. The protocol then generates a new random value, called a challenge, which is used to fix one of the polynomial's variables. This reduces the amount of polynomial data that must be processed in the following round. For multilinear polynomials represented by their evaluations, the remaining data is reduced by half after each round.

The work within a round can be highly parallel, since many polynomial evaluations can be processed at the same time. However, the rounds cannot execute independently: the next round cannot begin until the random value from the current round has been generated and used to reduce the data. At the same time, every round contains less work than the one before it. The GPU therefore begins with a large parallel workload that continuously shrinks as Sumcheck progresses. This combination of parallel computation within rounds, sequential dependencies between rounds, and shrinking workloads makes Sumcheck attractive for GPU acceleration but difficult to accelerate efficiently.

These properties make Sumcheck acceleration different from offloading a standalone MSM or NTT. A GPU implementation that transfers the complete polynomial data between the host and device every round pays communication costs precisely at the boundary between dependent rounds. Conversely, simply keeping the entire computation on the GPU is not necessarily efficient because late rounds contain progressively less parallel work. Effective acceleration therefore depends not only on the throughput of a Sumcheck kernel, but also on where polynomial data resides across rounds, what information crosses the CPU–GPU boundary, and when execution should return to the CPU.

The problem becomes more difficult when Sumcheck is embedded in a complete proof system. We study this problem in the Spartan R1CS prover [36]. Spartan is a useful case study because Sumcheck is a central part of its proving procedure rather than an isolated arithmetic routine. The Spartan R1CS prover performs two Sumcheck computations, which together accounted for 52.8% of total proving time in an initial profiling run with 1,024 constraints. Each round of the Sumcheck is surrounded by proof-generation work that must remain consistent with the existing proof format and verifier. The Spartan prover must still construct and commit to the round polynomial, update the proof transcript, generate challenges needed for the next round and perform other proof-generation operations.

This interaction with the Spartan prover machinery prevents Sumcheck from being treated as a completely independent GPU operation. Moving the entire prover to the GPU would require reimplementing substantial portions of Spartan, including operations that expose much less parallelism than polynomial evaluation and folding. This leads to the central systems question in this work: *where should the CPU–GPU boundary be placed when accelerating Sumcheck inside an existing prover?* The design must preserve Spartan's proof behavior while reducing data movement, adapting to the shrinking Sumcheck workload, and using the GPU efficiently.

To this end, we introduce Chimera, a hybrid CPU–GPU backend that addresses these requirements for the two Sumcheck computations in the Spartan R1CS prover. Chimera places the accelerator boundary around the large polynomial computation while leaving Spartan's proof-facing operations on the CPU. At the beginning of each GPU-backed Sumcheck computation, Chimera transfers the required polynomial data to the GPU. For each round, the GPU processes this data and returns only the few values that Spartan needs to construct the round polynomial. Spartan then performs its existing commitment and transcript operations and generates the random challenge for that round. The challenge is returned to the GPU to reduce the current polynomial data for the next round. The reduced data remains on the GPU and becomes the input to the next GPU round. This preserves Sumcheck's required round ordering while avoiding repeated movement of the full polynomial data. Because Spartan continues to control the transcript and proof construction, Chimera does not change the proof format or verifier behavior.

Keeping the polynomial data on the GPU does not mean that every round should execute there. Because each round reduces the amount of remaining data, Chimera uses the GPU while there is enough parallel work to justify its overhead, then switches later rounds back to Spartan's native CPU implementation once the remaining computation becomes small. Chimera also integrates a memory-management mechanism for when the entire polynomial data cannot fit on the GPU. In this mode, portions of the current data are processed

separately and their contributions are combined before the round result is returned to Spartan. As folding reduces the amount of remaining polynomial data, Chimera transfers the remaining polynomial data into the resident GPU workspace once it fits and continues execution using the resident GPU path. Accelerating the main Sumcheck computation also exposes costs in the prover operations surrounding it. Chimera therefore optimizes several parts of the Spartan integration, including commitments to the small degree-two and degree-three round polynomials and construction of the polynomial data required by the second Sumcheck. These changes support the overall system design by preventing surrounding prover overheads from dominating after the main Sumcheck computation has been accelerated.

We additionally implement a standalone GPU Sumcheck engine to study the GPU computation independently of Spartan's proof-generation requirements. Unlike the integrated backend, the standalone engine allows the GPU to control the entire Sumcheck round, including challenge generation, memory management, and the transition to the next round. This provides a controlled environment for evaluating how efficiently Sumcheck itself can use the GPU, including the effects of persistent memory, parallel reduction, Montgomery-domain arithmetic, fused execution, CUDA Graphs, and asynchronous chunk processing. Separating the standalone and integrated implementations allows us to distinguish improvements to the GPU computation itself from improvements that result from the way the accelerator is integrated into the complete Spartan prover.

We evaluate Chimera on BN254 Spartan R1CS instances using an NVIDIA GeForce RTX 5090. At $N = 2^{17}$, Chimera reduces the measured Sumcheck-related region from 672.785 ms to 84.245 ms, corresponding to a 7.99× speedup. This acceleration does not translate directly into a 7.99× full-prover speedup because Sumcheck is only one part of the overall proving time. At this problem size, the accelerated region accounts for approximately 14.1% of the original 4.77-second CPU proving time. Accelerating that portion by 7.99× predicts an end-to-end speedup of approximately 1.14×, close to the measured 1.15×. Thus, once Sumcheck is accelerated, work outside the accelerated region becomes a larger fraction of the remaining prover time.

Across the measured range, Chimera improves end-to-end Spartan prover time, with the largest improvements appearing when the accelerated computation accounts for a larger fraction of the prover. For $N = 2^{24}$–2^{27}, Chimera achieves 1.27×–1.64× end-to-end speedup. Our evaluation also separates standalone GPU performance from Spartan-integrated performance, examines CUDA orchestration behavior, and studies how scheduling, memory residency, and surrounding proof-generation costs affect the achievable speedup.

Overall, this work makes the following contributions:

- **A systems characterization of prover-integrated Sumcheck.** We identify the round dependency, shrinking workload, polynomial-data movement, and surrounding proof-generation work that distinguish accelerating Sumcheck inside Spartan from accelerating a standalone GPU kernel. This characterization motivates the CPU–GPU boundary used by Chimera.
- **A protocol-aware hybrid Sumcheck backend.** We design and implement Chimera, which keeps large polynomial data resident on the GPU across profitable rounds while Spartan retains commitment, transcript, challenge-generation, and proof-construction logic. Chimera combines GPU-resident execution, CPU fallback for small rounds, memory-scalable execution for larger inputs, and GPU kernels designed to expose the parallelism available within each round, without changing Spartan's proof format or verifier behavior.
- **An evaluation separating GPU execution from prover integration.** We evaluate Chimera inside the complete Spartan prover and separately study an optimized standalone GPU Sumcheck engine. Chimera accelerates the measured Sumcheck-related region by 7.99× at $N = 2^{17}$ and improves complete proving time at every measured size, reaching 1.27×–1.64× end-to-end speedup for $N = 2^{24}$–2^{27}.

2 Background and Motivation

This section provides the background needed to motivate Chimera's design. We first describe the algebraic setting used by the Spartan R1CS prover and review the Sumcheck protocol and its role in Spartan. We then examine the execution characteristics of Sumcheck that are most relevant to GPU acceleration, including

parallelism within rounds, dependencies between rounds, the shrinking polynomial workload, and GPU memory movement.

2.1 Zero-Knowledge Proofs and R1CS

A zero-knowledge proof allows a prover to demonstrate that a computation was performed correctly without revealing the private inputs used in that computation [20]. Many proof systems first express the computation as algebraic constraints over a finite field [4, 7, 18, 21, 34, 36]. In a Rank-1 Constraint System (R1CS), these constraints take the form

$$\langle \mathbf{a}_i, \mathbf{z} \rangle \cdot \langle \mathbf{b}_i, \mathbf{z} \rangle = \langle \mathbf{c}_i, \mathbf{z} \rangle,$$

where $\mathbf{z}$ contains the public and private values associated with the computation. The prover demonstrates that it knows values satisfying all of the constraints without revealing the private portion of $\mathbf{z}$.

The cost of producing such a proof depends strongly on the proof system. Traditional SNARK implementations often spend a large fraction of proving time in operations such as multi-scalar multiplication and polynomial transforms [30, 32, 38]. Spartan instead represents the constraint system using multilinear polynomials and relies heavily on the Sumcheck protocol [31, 36]. This changes the structure of the prover workload: rather than repeatedly executing a few large standalone kernels, the prover performs a sequence of dependent Sumcheck rounds over polynomial evaluation data.

In this work, the Spartan prover operates over the BN254 scalar field. A field element occupies 32 bytes in the representation used by our GPU implementation, so the polynomial data processed by Sumcheck can quickly become large as the number of constraints increases.

2.2 The Sumcheck Protocol

Sumcheck verifies a statement about the sum of a multivariate polynomial over all Boolean assignments to its variables [31]. Sumcheck subsequently became a central component of interactive-proof systems and later proof systems based on multilinear polynomials [10, 19, 36, 43, 44]. Let $F(x_1, \ldots, x_\ell)$ be a polynomial over a finite field $\mathbb{F}$. The prover wishes to show that

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

The sum contains 2^ℓ evaluations of F. Sumcheck avoids treating these evaluations as one computation by reducing the problem one variable at a time.

For the first variable, the prover constructs a univariate polynomial

$$g_1(t) = \sum_{x_2, \ldots, x_\ell \in \{0,1\}} F(t, x_2, \ldots, x_\ell).$$

Evaluating g_1 at 0 and 1 separates the original sum according to the two possible values of x_1. The original value must therefore satisfy

$$S = g_1(0) + g_1(1).$$

After this relation is established, the protocol generates a random field element α_1. The first variable is fixed to this value, and the remaining computation becomes

$$g_1(\alpha_1) = \sum_{x_2, \ldots, x_\ell \in \{0,1\}} F(\alpha_1, x_2, \ldots, x_\ell).$$

The same process repeats for the remaining variables. Each round removes one variable until the original sum has been reduced to a single polynomial evaluation.

Figure 1 summarizes this round-by-round reduction and the dependency between the round polynomial, challenge, and subsequent fold.

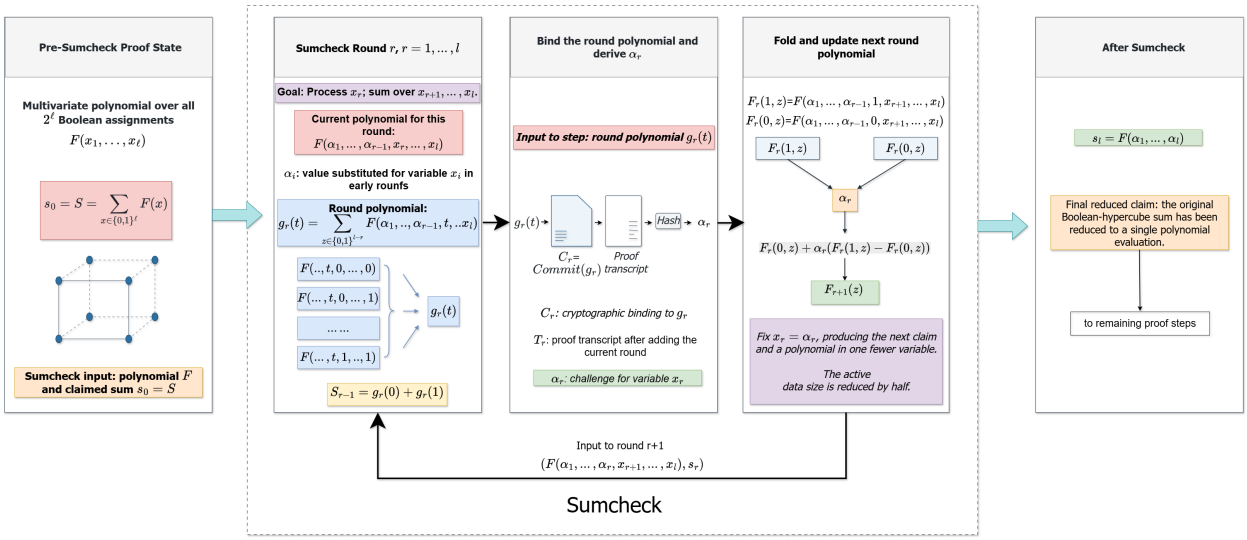

Fig. 1. Sumcheck protocol flow. In round r, the prover constructs the round polynomial g_r, incorporates it into the proof transcript, derives the challenge α_r, and fixes the current variable using that challenge. Repeating this process reduces the original sum claim to a final evaluation $F(\alpha_1, \ldots, \alpha_\ell)$.

For implementation purposes, the multilinear polynomials used by the prover are represented by their evaluations over the Boolean hypercube. Let E_r denote the evaluation table that remains at round r. A Sumcheck round can then be viewed as $E_r \to g_r \to \alpha_r \to E_{r+1}$. Here, g_r is the round polynomial and α_r is the challenge derived after g_r has been incorporated into the protocol.

The transition from E_r to E_{r+1} is implemented by a *fold*. Consider two evaluations v_0 and v_1, corresponding to the current variable taking values 0 and 1. Once α_r is known, the new value is $v' = v_0 + \alpha_r(v_1 - v_0)$. This operation is applied independently to every pair of evaluations. As a result, the length of the evaluation table is halved after every round.

This representation exposes the two properties that are central to GPU execution. First, the round evaluation and folding within a round contain substantial data parallelism because many evaluation pairs can be processed independently. Second, the consecutive rounds themselves remain sequential because the fold for round r cannot be performed until α_r has been generated.

2.3 Sumcheck in the Spartan R1CS Prover

Spartan is a general-purpose zkSNARK for R1CS satisfiability that does not require a trusted setup [36]. Its prover represents the R1CS instance using multilinear polynomials and uses Sumcheck to verify algebraic relations derived from those constraints.

The Spartan R1CS prover performs two Sumcheck computations [36]. The first evaluates a cubic relation that we write as $A(BC - D)$, while the second evaluates a quadratic relation AB. The symbols A, B, C, and D denote the multilinear polynomial data required by the corresponding Sumcheck computation. The cubic relation produces a degree-three round polynomial, while the quadratic relation produces a degree-two round polynomial.

Importantly, these Sumcheck computations are not isolated from the rest of the Spartan prover. After the values for a round polynomial have been computed, Spartan constructs that polynomial and commits to it. It also updates a running record of the proof messages, called the transcript. The transcript is used by the Fiat–Shamir transformation [16] to generate the random value α_r for the current round. Only after these proof-generation steps produce the challenge can the polynomial data be folded for the following round.

The resulting execution order is therefore

$$\text{compute round result} \rightarrow \text{construct and commit to } g_r$$
$$\rightarrow \text{update transcript} \rightarrow \text{derive } \alpha_r \rightarrow \text{fold polynomial data.}$$

This ordering is fundamental to Chimera's design. The large polynomial computation is naturally parallel, but the challenge required for the next fold is produced by CPU-side proof-generation work surrounding that computation.

2.4 GPU Execution Characteristics

Round evaluation and folding are therefore natural candidates for GPU execution [30,32,38]. Early Sumcheck rounds have exactly this property: they process large polynomial evaluation tables, and the same field operations are applied independently across many entries.

The amount of available work, however, decreases after every round. If a Sumcheck computation begins with N entries in its current evaluation data, successive folds reduce the remaining data as

$$N, \quad \frac{N}{2}, \quad \frac{N}{4}, \quad \frac{N}{8}, \ldots$$

Consequently, later rounds expose progressively less parallelism. Kernel-launch and CPU–GPU synchronization costs that are small relative to the early rounds can become significant once only a small amount of polynomial data remains [38].

Memory movement creates a second concern. Copying the complete polynomial data back to the CPU after every round would repeatedly move the largest objects in the computation across PCIe. Keeping the data on the GPU avoids these repeated transfers, but requires it to remain available across the sequence of challenge-dependent rounds. For large inputs, polynomial data may exceed available GPU memory, requiring the computation to operate on smaller portions of the data.

These characteristics motivate Chimera's design questions: how long data should remain on the GPU, what should cross the CPU–GPU boundary, when later rounds should return to the CPU, and how to handle inputs exceeding GPU memory.

3 Chimera Design

Chimera is designed around the computational structure of Sumcheck inside the Spartan prover. The large polynomial computations within each round expose substantial data parallelism, whereas the surrounding commitments, transcript updates, challenge generation, and proof-construction operations contain much less parallel work and remain closely coupled to Spartan's CPU implementation. This difference motivates a hybrid execution model in which the GPU handles the large data-parallel Sumcheck computation while Spartan retains control of the proof-generation logic on the CPU.

This hybrid execution model must satisfy several requirements. The large polynomial computation should remain on the GPU across consecutive profitable rounds, while Spartan retains control of the proof-generation operations on the CPU. The interface between them must preserve the challenge dependency between rounds while avoiding repeated movement of the complete polynomial data. The execution policy must also account for the fact that the workload shrinks after every round, eventually reducing the benefit of GPU execution, and for inputs whose polynomial data exceeds available GPU memory. Finally, the GPU implementation itself must expose sufficient parallelism to use the accelerator effectively rather than merely reproducing the CPU computation on the device.

3.1 System Overview and Integration Boundary

The Spartan R1CS prover executes two Sumcheck computations during proof generation [36]. These computations are interleaved with proof-generation operations that remain part of the Spartan prover. Each

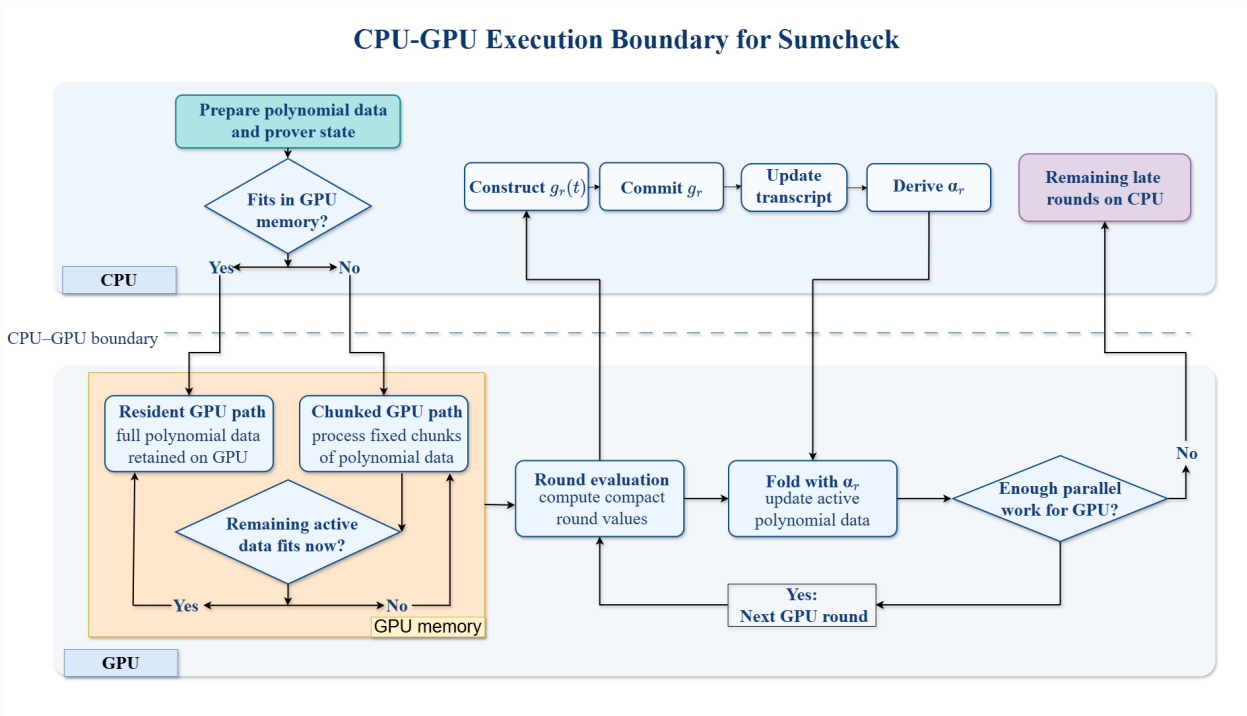

Fig. 2. Chimera's hybrid CPU–GPU execution model. Polynomial data that fits in GPU memory uses the resident path, while larger inputs use chunked execution until the remaining data fits in the resident GPU workspace. GPU-backed rounds compute compact round values, while Spartan constructs and commits to g_r, updates the transcript, and derives α_r. The challenge is returned to the GPU for folding. Once the remaining computation no longer provides sufficient parallel work, the remaining Sumcheck rounds execute on the CPU.

Sumcheck round is interleaved with several proof-generation steps, including committing to the round polynomial, updating the transcript, generating the challenge for the next round, and updating the commitments used by the proof. Chimera must therefore accelerate the large polynomial computations without changing how these surrounding prover steps are executed.

Chimera uses a Sumcheck-level CPU–GPU boundary. At the beginning of each GPU-backed Sumcheck computation, the polynomial data required for that computation is transferred to GPU memory. The GPU keeps this data across successive rounds instead of returning it to the CPU after every round. For round r, the GPU processes the current polynomial data and computes the compact values that Spartan needs to construct the round polynomial g_r. These values are sent to the CPU, where Spartan constructs g_r, commits to the round polynomial, updates the transcript, and derives the challenge α_r. The challenge is then sent back to the GPU and used to fold the current polynomial data for round $r + 1$.

Figure 2 summarizes this execution boundary and the transitions between chunked, resident GPU, and late-round CPU execution.

This division of work keeps the large, data-parallel portion of Sumcheck on the GPU while leaving proof construction under the control of the Spartan prover on the CPU. During resident GPU execution, only the compact round result and the corresponding challenge cross the CPU–GPU boundary between rounds; the polynomial data remains on the GPU. When the complete polynomial data does not fit in GPU memory, Chimera instead transfers and processes fixed-size portions within the current round, as described in Section 3.5. This reduces host–device communication while preserving Spartan's existing proof format and verifier behavior.

A finer-grained design that invokes the GPU separately for each operation would repeatedly incur synchronization, host orchestration, and data-movement overheads. At the other extreme, moving the entire

proof-generation process to the GPU would require re-implementing Spartan's transcript handling, commitments, and other proof-construction logic, even though much of this work exposes considerably less parallelism than the polynomial computation itself. Chimera instead keeps the large polynomial data on the GPU for the rounds in which GPU execution is beneficial, while Spartan continues to control the proof-generation operations on the CPU. Chimera reuses the allocated GPU workspace across these rounds so that the reduced polynomial data can be consumed directly by the following GPU-backed round.

3.2 Hybrid CPU–GPU Scheduling

Keeping the polynomial data on the GPU does not mean that every Sumcheck round should execute there. Each round reduces the amount of remaining polynomial data, so the amount of available parallel work decreases throughout the computation. Early rounds contain enough work to amortize GPU kernel-launch and synchronization overheads, while later rounds eventually become too small to use the GPU efficiently.

Chimera therefore uses a hybrid execution policy rather than assigning an entire Sumcheck computation permanently to either the CPU or GPU. Large rounds continue through the GPU-backed path, while sufficiently small rounds are returned to Spartan's native CPU implementation. The switching decision is based on the size of the remaining Sumcheck computation and GPU characteristics, allowing Chimera to use GPU execution only while the available parallelism is large enough to justify its overhead.

This policy is particularly well suited to Sumcheck because the workload shrinks predictably from round to round. Once execution crosses the point where CPU processing becomes preferable, the remaining rounds continue on the CPU instead of repeatedly switching between processors.

3.3 Integrating into Spartan Prover Efficiently

Accelerating Sumcheck changes the performance balance of the surrounding Spartan prover. Operations that were relatively small compared with round evaluation and folding become a more significant part of the execution after the main Sumcheck computation is accelerated on the GPU. Chimera therefore also removes unnecessary work immediately surrounding Sumcheck, while leaving the mathematical operations or the proof unchanged.

One source of overhead is the commitment made to the small polynomial produced in every Sumcheck round. These polynomials contain only a few coefficients, but Spartan's existing commitment path is a general-purpose routine designed to handle much larger inputs. Using that path for every round introduces allocation, data preparation, and group-operation overhead that is disproportionate to the size of the object being committed. Chimera therefore uses a specialized path for these small commitments and reuses already available commitment values when possible. The resulting commitments and their position in the proof remain unchanged.

A second source of overhead appears before the second Sumcheck begins. At this point, Spartan needs one polynomial data set obtained by combining information from the three R1CS matrices A, B, and C. The original implementation first constructs a complete evaluation array for A, another for B, and another for C. It then scans those three arrays again to construct the single combined array that the second Sumcheck actually uses.

Chimera removes these intermediate arrays. As Spartan processes the nonzero entries of A, B, and C, their contributions are accumulated directly into the one final array required by the second Sumcheck. In other words, the original path constructs three large intermediate results and then combines them, whereas Chimera constructs the required result directly when needed. The mathematical result is identical, but this avoids storing three temporary dense arrays and the additional full pass needed to combine them.

3.4 GPU Execution of a Sumcheck Round

The CPU–GPU boundary determines which work is sent to the GPU, but the computation within a Sumcheck round must also be organized so that the GPU can process many independent pieces of work in parallel. The structure of multilinear polynomials provides this parallelism naturally.

Within a Sumcheck round, one polynomial variable is processed at a time. For each fixed assignment to all of the other variables that remain in the computation, the evaluation data contains two corresponding values: one obtained when the current variable is set to 0, and one obtained when it is set to 1. These two values form a pair. Because the two entries in a pair differ only in the value of the current variable, and different pairs correspond to different assignments of the remaining variables, a round contains many independent pairs that can be processed simultaneously.

For an evaluation pair (X_0, X_1), multilinearity allows the univariate slice along the active coordinate to be evaluated at any point t as $X(t) = X_0 + t(X_1 - X_0)$. Chimera distributes these pairs across GPU threads. Each thread uses the two values in its assigned pair to compute that pair's contribution to the current round result according to the polynomial relation being evaluated. Since the pairs are independent, these contributions can be computed concurrently across the polynomial data.

The contributions from all pairs must then be combined to obtain the result needed by Spartan for the current round. A serial aggregation would limit the benefit of parallel pair processing, so Chimera performs this reduction on the GPU. Threads first combine contributions within each GPU block of the kernel execution, producing block-level partial results, and these partial results are then reduced to the completed round result. Only this compact result is returned to Spartan.

After Spartan performs the proof-generation steps for the round and returns the challenge α_r, the same pairwise organization is used to fold the polynomial data for the next round. For each pair (X_0, X_1), the fold computes $X' = X_0 + \alpha_r(X_1 - X_0)$ for the challenge point α_r. Because this operation is independent across pairs, folding is executed in parallel across threads. Each pair produces one new entry, halving the amount of polynomial data while leaving the folded data on the GPU for the following round.

3.5 GPU Memory Management for Large Inputs

Chimera normally keeps the polynomial data for a Sumcheck computation resident in GPU memory across successive rounds. This avoids repeatedly transferring the same large data between the CPU and GPU. For sufficiently large inputs, however, the polynomial data required for a round may exceed the available GPU memory.

To handle these cases, Chimera uses a memory management scheme. Instead of requiring the complete polynomial data to reside on the GPU at once, the current data is divided into fixed-size chunks. Each chunk is transferred to the GPU and processed independently to compute its contribution to the current Sumcheck round. These partial contributions are accumulated until all chunks have been processed. This is what we refer to as chunked execution.

Only after the entire current round has been processed is the completed round result returned to Spartan. Spartan then performs its normal proof-generation steps and generates the challenge for that round. Chunking therefore changes how the polynomial data is processed within a round, but it does not change the dependency between rounds: the following round still cannot begin until the current challenge has been generated. As Sumcheck progresses, each round reduces the amount of polynomial data that remains. Chunked execution is needed only while the current data exceeds the amount that can be held in GPU memory. Once the remaining data becomes small enough to fit, the entire polynomial can be held in GPU for the remaining rounds. In the current Spartan-integrated implementation, this chunked execution is used for the quadratic second Sumcheck.

4 Implementation

Section 3 described the design decisions that determine how Chimera divides Sumcheck execution between Spartan and the GPU. We now describe how those decisions are realized in the implementation. We begin with the interface between Spartan and the GPU backend, then describe round evaluation, challenge-driven folding, hybrid execution, GPU memory management, and the changes made to the surrounding Spartan prover. We finally describe the standalone GPU Sumcheck engine used to study GPU execution independently of Spartan's proof-generation requirements.

4.1 Spartan–GPU Interface and Round Flow

Chimera is invoked directly from the Sumcheck computations in the Spartan R1CS prover. Rather than exposing individual GPU kernels to the prover, the backend provides an interface that follows the lifetime of a Sumcheck computation. The main operations are `state_new`, which initializes the GPU-side polynomial data, `round_evals`, which computes the result needed for the current round, and `fold`, which applies Spartan's challenge to produce the polynomial data for the following round.

At the beginning of a GPU-backed Sumcheck computation, `state_new` receives the polynomial data required by that computation and transfers it from CPU memory to GPU memory. The GPU buffers are retained across successive rounds rather than recreated or repopulated after every round. This initial transfer therefore establishes the GPU-resident data that is subsequently consumed by both round evaluation and folding. This amortizes the cost of data transfer across rounds.

Each round begins with `round_evals`. The GPU processes the current polynomial data and computes the compact result required for Spartan to construct the round polynomial g_r. Only this result is copied to the CPU; the polynomial data itself remains on the GPU. Spartan then constructs g_r, performs the commitment and transcript operations required by the proof, and derives the Fiat–Shamir challenge α_r.

Spartan passes α_r back to Chimera through `fold`. The GPU uses the challenge to reduce the current polynomial data by half, and the resulting data is retained in GPU memory as the input to the next invocation of `round_evals`. A GPU-backed round therefore follows `round_evals` $\rightarrow g_r \rightarrow$ Spartan proof-generation steps $\rightarrow \alpha_r \rightarrow$ `fold`.

This interface preserves the ordering required by Spartan while keeping the large polynomial data inside the GPU backend. During the round loop, the CPU and GPU exchange only the information needed to cross the protocol boundary: the result of the current round in one direction and the resulting challenge in the other. When the Sumcheck computation completes, the final values needed by the Spartan prover are copied back to the CPU.

4.2 GPU Round Evaluation

The `round_evals` operation implements the main arithmetic work of a Sumcheck round. Chimera provides separate GPU kernels for the two polynomial relations required by the Spartan prover. The first Sumcheck evaluates $A(BC - D)$, using the current evaluation data for A, B, C, and D, while the second evaluates AB, using the current data for A and B.

The kernels operate directly on the pairwise representation described in Section 3.4. For each polynomial X, a thread reads the pair (X_0, X_1) associated with the current Sumcheck variable. Values required at a point t are computed directly from the pair as $X(t) = X_0 + t(X_1 - X_0)$. The kernel therefore does not construct an intermediate polynomial for each pair. Instead, it interpolates the required values from the two stored evaluations and immediately uses them in the Sumcheck relation.

For the quadratic kernel, the corresponding A and B pairs are interpolated and combined as $A(t)B(t)$. For the cubic kernel, the corresponding values from all four polynomial data sets are interpolated and combined as $A(t)\big(B(t)C(t) - D(t)\big)$. Each thread performs this computation for its assigned portion of the current evaluation data and produces a contribution to the values needed to construct the round polynomial.

Because a round contains many such contributions but Spartan requires only a small completed result, Chimera performs the aggregation on the GPU. Threads within a block first combine their contributions into block-level partial results. A subsequent reduction combines these partial results into the final round values. Performing both stages on the GPU avoids serially accumulating the per-pair outputs on the CPU and prevents the reduction from becoming a bottleneck after the parallel kernel computation.

Once the reduction is complete, only the resulting round values are copied back to Spartan prover. The full polynomial data remains in GPU memory. Spartan uses the returned values to construct g_r, performs the required commitment and transcript operations, and generates the challenge that will be passed back to the GPU for folding.

4.3 Challenge-Driven Folding and GPU Memory Reuse

After Spartan constructs the round polynomial, updates the transcript, and derives the challenge α_r, the challenge is passed to Chimera through `fold`. The GPU then reduces the current polynomial data to the data required by the following Sumcheck round. For every pair (X_0, X_1), the fold computes $X' = X_0 + \alpha_r(X_1 - X_0)$. The operation is applied independently across all pairs and across each polynomial used by the current Sumcheck. Each pair produces one output value, so the amount of polynomial data is reduced by half after every fold.

Chimera reuses the GPU memory allocated when the Sumcheck computation begins. The output of the fold is written into reusable GPU buffers and becomes the input to the next call to `round_evals`. Consequently, advancing from one round to the next does not require allocating a new set of buffers or copying the reduced polynomial data back through CPU memory. The allocated workspace continues to be reused as the amount of polynomial data decreases throughout the computation.

Field arithmetic inside the GPU backend is performed over the BN254 scalar field using Montgomery representation. Polynomial values are converted into the representation used by the GPU computation and remain in that representation across round evaluation and folding. Keeping the data in the same arithmetic representation avoids repeatedly converting the complete polynomial data between successive rounds.

The fold is launched only after Spartan has returned the challenge for the current round. Chimera therefore preserves the protocol dependency between rounds: the polynomial data cannot be reduced for round $r + 1$ until α_r has been generated, and the next round evaluation operates only on the data produced by that challenge-dependent fold.

GPU field arithmetic.. To improve arithmetic throughput during round evaluation and folding, Chimera implements BN254 finite-field operations using eight 32-bit limbs. Montgomery multiplication is implemented using inline PTX instructions, allowing the multi-limb arithmetic to use the GPU's carry condition code during multiplication and accumulation. Polynomial values remain in Montgomery representation across successive round-evaluation and folding operations, avoiding repeated conversion of the complete polynomial data between canonical and Montgomery representations.

4.4 Hybrid Execution and GPU Memory Management

Because each fold reduces the amount of remaining polynomial data by half, the amount of parallel work available to the GPU decreases from round to round. Before continuing GPU execution, Chimera checks the size of the remaining computation. Large rounds continue through the GPU backend, while sufficiently small rounds are returned to Spartan's native CPU implementation. This avoids incurring GPU kernel-launch and synchronization overhead once the remaining work is too small to use the accelerator effectively.

GPU memory capacity creates the opposite problem for very large inputs. The normal execution path keeps the polynomial data resident on the GPU across rounds, but some inputs may require more memory than is available on the device. For this particular issue, Chimera therefore provides a chunked execution path that processes the current polynomial data in fixed-size portions rather than requiring the complete data set to be resident at once.

Within a round, each memory chunk contains a portion of the polynomial pairs required by the Sumcheck computation. The chunk is transferred to the GPU and processed using the same round-evaluation computation as the resident path. Its result is accumulated with the contributions from the other chunks. After all chunks for the current round have been processed, the combined result is returned to Spartan, which performs the normal proof-generation steps and produces the challenge for that round. Chunking therefore changes the memory organization of the computation without changing the ordering of the Sumcheck protocol.

As Sumcheck progresses, each fold reduces the amount of remaining polynomial data. An input that initially requires chunked execution may therefore become small enough to fit entirely in GPU memory after several rounds. When this occurs, Chimera transfers the remaining polynomial data into the resident GPU workspace and completes the remaining rounds using the normal resident path. Chunked execution is therefore used only while required by the size of the current polynomial data.

The reduction in GPU memory usage depends on the selected chunk size. For example, at $N = 2^{16}$, the quadratic Sumcheck contains two polynomials with 131072 field elements each. With 32-byte BN254 field elements, storing both complete polynomials requires $2 \cdot 131072 \cdot 32 = 8.0$ MiB. Using `chunk_pairs`=8192 requires only the low and high portions of the two polynomials for the current chunk, $4 \cdot 8192 \cdot 32 = 1.0$ MiB, reducing the working memory required by the round evaluator by $8\times$.

The Spartan-integrated implementation uses chunking primarily as a GPU memory management mechanism. All chunks belonging to a round are completed before Spartan generates the corresponding challenge, so the implementation does not pipeline work across challenge-dependent rounds. The standalone engine described later extends chunked execution with asynchronous transfer and computation overlap.

4.5 Spartan-Specific Prover Optimizations

Once round evaluation and folding are accelerated, smaller operations surrounding the Sumcheck computations account for a larger fraction of their execution time. Chimera therefore modifies two parts of the Spartan prover that directly feed into or consume the accelerated Sumcheck computation.

The first modification targets commitments to the round polynomials. These polynomials are small (degree three in the first Sumcheck and degree two in the second), but Spartan's existing commitment routine follows a general-purpose path that performs allocation, data preparation, and group operations intended for larger inputs. Chimera provides a specialized path for these small round-polynomial commitments and reuses commitment values that are already available when possible. The optimization changes only how the commitments are computed internally; the commitment values, transcript order, and proof data expected by the verifier remain unchanged.

The second modification reduces the work required to construct the polynomial data used by the second Sumcheck. In the original prover, Spartan first materializes three dense evaluation arrays, $A(r_x, \cdot)$, $B(r_x, \cdot)$, and $C(r_x, \cdot)$, and then performs another pass over those arrays to form $r_A A(r_x, \cdot) + r_B B(r_x, \cdot) + r_C C(r_x, \cdot)$, which is the combined polynomial data consumed by the second Sumcheck.

Chimera eliminates this intermediate representation. As the contributions from the A, B, and C matrices are processed, they are multiplied by their corresponding coefficients r_A, r_B, and r_C and accumulated directly into the final combined array. The implementation therefore constructs the data that the second Sumcheck actually needs rather than first constructing three complete dense arrays and then scanning them again to combine them. This removes the temporary arrays and the additional full pass over their contents before GPU execution.

Neither modification changes the mathematical computation performed by Spartan. They reduce CPU-side allocation, data movement, and prover overhead that becomes more significant once the main Sumcheck computation has been accelerated.

4.6 Standalone GPU Sumcheck Engine

In addition to the Spartan-integrated backend, we implement a standalone GPU Sumcheck engine. The standalone experiments use the quadratic relation $F(A, B) = AB$. The standalone engine serves as a controlled environment for studying the GPU execution of Sumcheck independently of Spartan's commitments, transcript, and proof-construction requirements. Its purpose is to determine which GPU execution techniques are useful before adapting them to Chimera and to separate limitations of the GPU computation from limitations introduced by prover integration.

The resident path keeps the polynomial data and working buffers allocated on the GPU throughout the Sumcheck computation. The engine reuses these buffers as the data shrinks and keeps polynomial values in Montgomery representation across round evaluation, reduction, and folding. Together, these choices avoid repeated allocation, host–device movement, and representation conversion between rounds.

The engine also parallelizes the aggregation of the contributions produced during round evaluation. Block-level partial results are reduced on the GPU to form the completed round result, avoiding a serial aggregation step on the critical path. After the challenge is available, the standalone implementation can additionally

fuse the fold with work for the following round, allowing newly folded values to be consumed immediately rather than writing the entire reduced polynomial data and reading it again in a separate pass.

The absence of Spartan's proof-generation boundary also allows the complete round transition to remain on the GPU. The standalone engine constructs the transcript input on the device, computes the Keccak hash, derives the BN254 challenge, and supplies that challenge directly to the fold. This provides a useful comparison with Chimera, where the round result must return to Spartan so that the existing prover can perform its commitment and transcript operations before returning the challenge.

For inputs larger than GPU memory, the standalone engine uses chunked execution with asynchronous transfer and computation. Reusable GPU buffers are double-buffered so that transfer of one chunk can overlap computation on another chunk within the Sumcheck round. All chunks contribute to the completed round result before the challenge is generated. As folding reduces the remaining polynomial data, execution switches from chunked processing to the resident path once the remaining data fits in GPU memory.

These experiments establish which improvements come from GPU execution of Sumcheck and which costs arise only when that computation is embedded inside the Spartan prover. Section 5 evaluates these mechanisms separately from end-to-end Chimera results.

5 Experimental Evaluation

We evaluate Chimera from both the prover and GPU perspectives. First, we measure the effect of Chimera on complete Spartan proof generation and on the Sumcheck-related portion of the prover. We then examine the contribution of the main mechanisms in the integrated backend. Finally, we evaluate the standalone Sumcheck engine to separate the behavior of the GPU computation itself from the additional requirements imposed by integration with Spartan.

5.1 Experimental Setup

We evaluate Chimera on an NVIDIA GeForce RTX 5090 with 32 GB of device memory connected through PCIe Gen5 x16, using NVIDIA driver 591.86 and CUDA 13.1. All experiments use the BN254 scalar field described in Section 2. The Spartan experiments use synthetic R1CS instances with equal numbers of constraints and variables. Because the core Sumcheck computation operates over dense multilinear evaluation tables, the GPU round-evaluation and folding kernels primarily process the resulting polynomial data rather than the original sparse R1CS representation. Synthetic instances therefore provide a controlled benchmark for evaluating the accelerated Sumcheck computation, although other circuit-dependent costs in the Spartan prover may vary across applications. GPU measurements use warmed-up executions and exclude CUDA context initialization.

We first evaluate Chimera as an integrated backend for the Spartan prover. The baseline for these experiments is the original CPU implementation of Spartan. This comparison measures the effect of introducing Chimera into the existing prover while preserving Spartan's proof construction and verifier behavior. It is therefore a system-level comparison between the original prover and the same prover with its Sumcheck computation accelerated by Chimera.

For the integrated experiments, we report both end-to-end proving time and the time spent in the *Sumcheck-related region*. End-to-end time measures the complete Spartan proof-generation operation. The Sumcheck-related region includes the two Sumcheck computations together with the prover work directly surrounding them, including preparation of the polynomial data, round evaluation and folding, transcript interaction, small round-polynomial commitments, target updates, and associated proof-construction work. Reporting both measurements allows us to distinguish the acceleration achieved within Chimera's target region from its effect on the complete prover.

We also evaluate the standalone GPU Sumcheck engine against ICICLE [23], a general-purpose GPU cryptography library that provides CUDA implementations of finite-field and proof-system primitives, including Sumcheck. This comparison is performed only at the standalone Sumcheck level, using matched GPU workloads, and is separate from the Spartan-integrated experiments above. It allows us to evaluate the

Table 1. End-to-end Spartan prover time in seconds. Spartan is the original CPU Spartan prover; Chimera is Spartan with the GPU Sumcheck backend.

N	Spartan	Chimera	Speedup	N	Spartan	Chimera	Speedup
2^{12}	1.19	0.81	1.46×	2^{21}	19.00	17.83	1.07×
2^{14}	2.36	2.02	1.17×	2^{22}	36.63	34.13	1.07×
2^{16}	3.91	3.78	1.03×	2^{23}	45.41	40.03	1.13×
2^{17}	4.77	4.14	1.15×	2^{24}	105.23	78.12	1.35×
2^{18}	7.96	7.34	1.08×	2^{25}	163.39	99.41	1.64×
2^{19}	8.50	7.85	1.08×	2^{26}	260.73	200.89	1.30×
2^{20}	16.62	15.16	1.10×	2^{27}	364.62	287.28	1.27×

efficiency of our GPU Sumcheck implementation against an existing GPU implementation without conflating that result with end-to-end Spartan performance.

5.2 Spartan End-to-End Performance

Chimera preserves Spartan's proof interface: the transcript, commitments, proof objects, and verifier behavior remain unchanged. During the two Sumcheck computations, large rounds use the GPU backend while sufficiently small later rounds return to Spartan's native CPU implementation.

Table 1 shows complete proving time from 2^{12} through 2^{27} constraints. Chimera improves end-to-end performance at every measured size, with speedups ranging from 1.03× to 1.64×. The speedup is not monotonic because Sumcheck is only one component of the complete Spartan prover, and the relative cost of the remaining prover operations changes with the instance size.

At $N = 2^{17}$, for example, Chimera reduces the measured Sumcheck-related region from 672.785 ms to 84.245 ms, corresponding to a 7.99× speedup. The original region accounts for approximately $\frac{0.672785}{4.77} \approx 14.1\%$ of the 4.77-second CPU proving time. Accelerating this fraction by 7.99×, while leaving the remaining prover work unchanged, predicts an end-to-end speedup of approximately $\frac{1}{(1-0.141)+0.141/7.99} \approx 1.14\times$, which closely matches the measured 1.15×.

The same effect explains why a large Sumcheck speedup does not translate into a similarly large full-prover improvement at larger inputs. As the R1CS instance grows, other operations in Spartan also grow substantially. In particular, the prover's polynomial-commitment computation, `polycommit`, becomes an increasingly large portion of proof generation and is not accelerated by Chimera. Once the Sumcheck computation has been reduced, this unaccelerated work can dominate the remaining prover time. Chimera therefore changes the performance balance of Spartan rather than accelerating every component of the prover.

5.3 Sumcheck-Related Performance

Figure 3 isolates the portion of Spartan most directly affected by Chimera. Chimera accelerates this region across the full measured range, but the speedup becomes less pronounced for some of the larger instances.

There are two reasons for this behavior. First, the measured region contains more than the parallel round-evaluation and folding kernels. It also includes construction of the polynomial data used by Sumcheck, transcript interaction, small round-polynomial commitments, target updates, and other proof-generation work. These operations do not receive the same GPU acceleration as the large polynomial computation.

Second, increasing the initial Sumcheck size increases the amount of polynomial data processed during the early rounds. These rounds expose more parallel work, but they also require more GPU memory traffic. At the same time, every Sumcheck computation continues to shrink by half from round to round, eventually reaching late rounds for which the remaining parallelism is too small to use the GPU efficiently. Chimera returns these rounds to the CPU, but their cost and the surrounding prover work remain part of the measured region.

Consequently, the Sumcheck-related speedup reflects more than arithmetic throughput. It is determined by the combination of GPU computation, memory traffic, shrinking round sizes, and the proof-generation operations that surround each Sumcheck round.

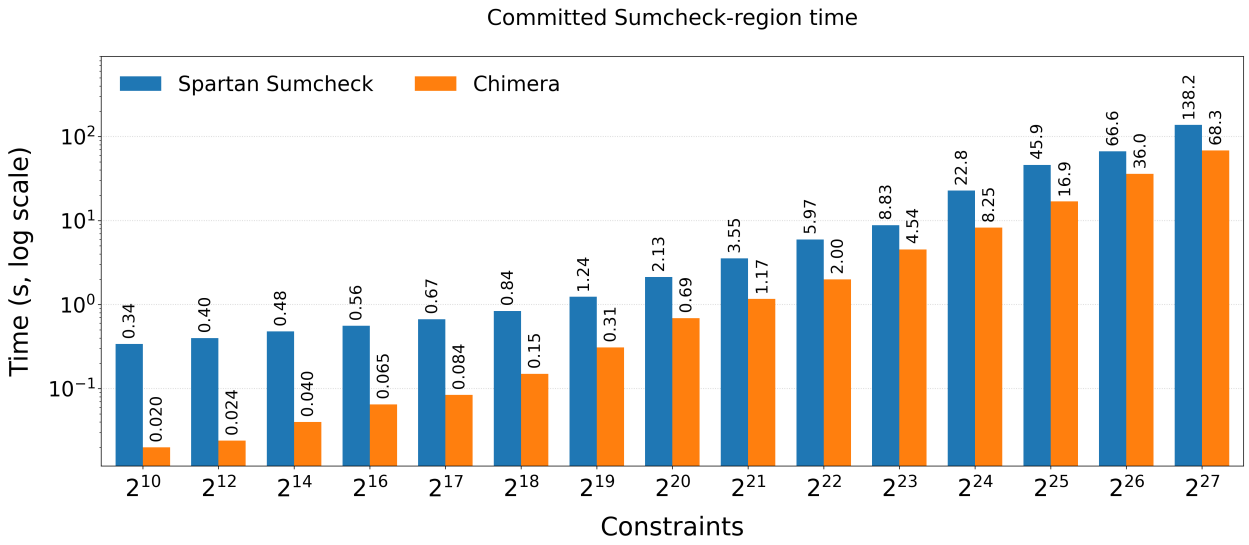

Fig. 3. Sumcheck-related execution time for the original Spartan prover and Chimera. Times are shown on a logarithmic scale.

Table 2. Attribution of Chimera's Sumcheck-related acceleration at $N = 2^{17}$, relative to the original Spartan implementation.

Mechanism	Role	Acceleration	Share
GPU-resident round kernels	Run Sumcheck round evaluation and folding on the GPU for large rounds, keeping the main polynomial computation on the device	1.35×	14.4%
Hybrid CPU–GPU scheduling	Use the GPU while the remaining computation is large, then return launch-dominated late rounds to Spartan's native CPU implementation	1.05×	2.3%
Spartan-side Sumcheck integration	Reduces CPU-side work surrounding GPU-backed rounds, including small commitments, target-commitment arithmetic, redundant commitment work, and small dot-product proof operations	4.57×	73.1%
Second-Sumcheck input construction	Constructs the polynomial data required by the second Sumcheck directly, avoiding extra dense intermediate arrays and a separate combine pass	1.17×	7.6%
Smaller optimizations	Remaining implementation changes not isolated as separate rows	1.05×	2.4%
Total Chimera speedup	Original Spartan 672.785 ms to Chimera 84.245 ms	7.99×	–

5.4 Integrated-System Ablation

Table 2 attributes the measured Sumcheck-related acceleration at $N = 2^{17}$ to the main mechanisms in the integrated Chimera implementation. The mechanisms interact, so the individual factors should be interpreted as an attribution of the final system rather than as independent microbenchmark speedups.

The largest attributed contribution comes from reducing the Spartan-side work surrounding GPU-backed Sumcheck rounds. This result reinforces that accelerating the round kernels alone is insufficient: once round evaluation and folding become faster, the proof-generation operations surrounding those rounds become a substantial part of the measured region. Because these mechanisms interact, the percentages in Table 2 should not be interpreted as independent additive contributions.

5.5 Standalone GPU Sumcheck Comparison

The integrated results above compare Chimera with the original Spartan prover and therefore measure the effect of accelerating Sumcheck inside a complete proof system. To evaluate the GPU Sumcheck implementation itself, we separately compare the standalone engine against ICICLE [23].

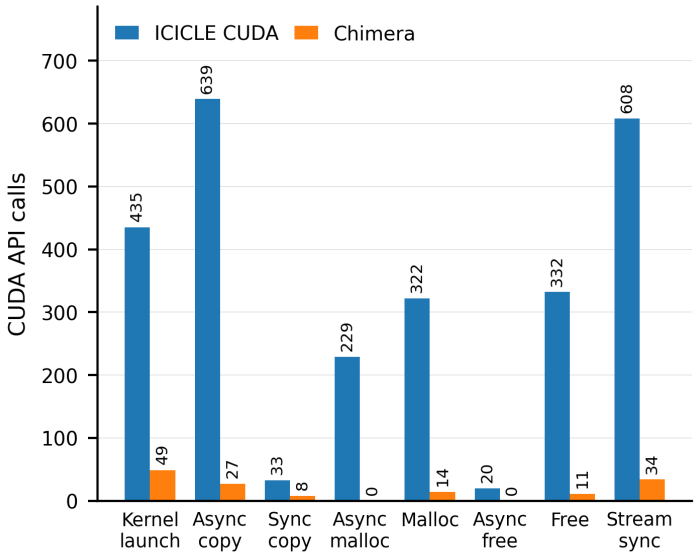

Fig. 4. Host-visible CUDA API activity in the profiled Sumcheck traces. The traces characterize orchestration behavior rather than end-to-end prover performance.

ICICLE is a general-purpose cryptographic acceleration library that provides GPU implementations of finite-field and proof-system primitives, including a CUDA Sumcheck implementation. We use this implementation as a GPU baseline for the standalone quadratic Sumcheck experiment. This comparison is separate from the Spartan results: both implementations execute Sumcheck directly on the GPU, without including Spartan's commitments, transcript handling, or other proof-generation work.

For the matched $N = 2^{22}$ quadratic Sumcheck experiment, our optimized standalone engine executes in approximately 28.7 ms, compared with 84.2 ms for ICICLE's CUDA Sumcheck implementation. This corresponds to a 2.94× speedup over an existing GPU implementation of Sumcheck. Unlike our standalone engine, which reuses its GPU workspace, keeps polynomial data in Montgomery representation across rounds, and fuses selected round-transition work, the profiled ICICLE execution exhibits substantial host-side orchestration, synchronization, and data movement.

Figure 4 further characterizes the orchestration costs surrounding GPU Sumcheck execution. In the profiled ICICLE trace, we observe 435 kernel launches, 639 asynchronous memory-copy calls, 229 asynchronous allocation calls, and 608 stream synchronizations.

These operations introduce orchestration overhead around the Sumcheck arithmetic. Chimera instead reuses its GPU workspace and keeps the polynomial data resident across rounds, reducing repeated allocation, movement, and synchronization on the critical path.

6 Discussion and Limitations

The evaluation shows that accelerating Sumcheck inside a complete prover is different from accelerating Sumcheck as an isolated arithmetic primitive. The standalone results demonstrate that GPU execution itself can be improved substantially through persistent GPU data, parallel reduction, efficient field representation, and reduced orchestration. The Spartan results show a second constraint: those improvements matter only to the extent that Sumcheck remains a significant part of complete proof generation.

This effect is visible after Chimera accelerates the two Sumcheck computations. Operations that previously accounted for a smaller portion of proving time become much more prominent. In particular, Spartan's larger polynomial commitment computation, `polycommit`, is outside Chimera's current acceleration bound-

ary and becomes one of the dominant remaining costs at large problem sizes. Consequently, the gap between the Sumcheck-related speedup and the end-to-end prover speedup is expected. Chimera changes the bottleneck of the prover rather than eliminating it. Further improvements to complete Spartan proving time will therefore require accelerating operations beyond Sumcheck, particularly the larger polynomial-commitment path.

The results also reinforce the importance of the CPU–GPU boundary. Moving only isolated arithmetic kernels to the GPU leaves repeated allocation, transfer, synchronization, and surrounding prover work on the critical path. At the other extreme, moving the entire prover to the GPU would require re-implementing proof-generation operations that do not expose the same degree of parallelism as the polynomial computation. Chimera instead keeps the large polynomial data on the GPU while Spartan continues to perform the commitments, transcript operations, and proof construction required between rounds. The standalone engine illustrates what becomes possible when those compatibility requirements are removed, while the integrated results show the cost of preserving them.

Chimera's hybrid scheduling is also specific to Sumcheck's shrinking workload. Increasing the initial problem size creates larger early rounds, but every execution eventually reaches later rounds with substantially less parallel work. GPU profitability therefore changes within a single Sumcheck computation. Likewise, an input that initially requires chunked execution may later fit entirely in GPU memory. These properties motivate both transitions used by Chimera: from chunked to resident GPU execution as the polynomial data shrinks, and from GPU to CPU execution when the remaining computation becomes too small to use the accelerator efficiently.

Our evaluation has several limitations. All integrated experiments use the Spartan R1CS prover over BN254 and synthetic R1CS instances with equal numbers of constraints and variables. We have not evaluated Chimera on other multilinear proof systems or finite fields, and the exact CPU–GPU switching points and relative costs may change with the prover, field representation, and hardware. The experiments also use a single NVIDIA RTX 5090. We do not claim that the measured thresholds or speedups transfer unchanged on lower-memory devices, or multi-GPU systems.

The current evaluation additionally focuses on single-proof latency. It does not study concurrent proof generation, power consumption, or multi-GPU execution. Chunked execution allows polynomial data larger than available GPU memory to be processed on one device, but it should not be interpreted as a distributed scaling result. Studying batch proving, multiple GPUs, and the interaction between large transfers and GPU memory bandwidth remains future work.

These limitations do not change the main systems result: Sumcheck performance depends not only on the throughput of its arithmetic kernels, but also on how the accelerator is incorporated into the surrounding prover. Chimera demonstrates one such integration for Spartan, while the standalone engine separates the performance of the GPU primitive from the constraints imposed by the complete proof system.

7 Related Work

A large body of work accelerates zero-knowledge proof generation using GPUs, distributed systems, and specialized hardware [14, 28–30, 32, 35, 40–42, 46, 47]. Much of this work focuses on highly parallel primitives such as MSM, NTT/FFT, polynomial computation, and batched proof generation. These systems demonstrate the importance of arithmetic throughput, memory movement, and scheduling in high-performance provers. Chimera addresses a different execution pattern: Sumcheck consists of a sequence of dependent rounds whose polynomial data shrinks after every round, making both data placement and the CPU–GPU boundary important to performance.

A separate line of work explores proof systems with different setup, transparency, and prover–verifier tradeoffs. Bulletproofs and related inner-product arguments provide discrete-log-based constructions without a trusted setup [6, 9], while Ligero, Aurora, and STARK-based systems study transparent proof generation [1, 3, 4]. Hyrax further develops efficient ZK arguments without trusted setup and introduces techniques for multilinear polynomial commitments [39]. Other general-purpose proof systems for arithmetic and R1CS computations explore different prover-efficiency and deployment tradeoffs, including Gemini and

Brakedown [7,21]. Polynomial-commitment techniques are also fundamental building blocks in modern proof systems, including pairing-based polynomial commitments such as KZG and discrete-log-based inner-product constructions [6,24].

Recent accelerator designs also target Sumcheck directly [12, 13, 25] and establish Sumcheck as an important acceleration target in modern multilinear proof systems. Chimera is complementary to these efforts: rather than designing a new hardware accelerator, it studies how GPU Sumcheck acceleration can be incorporated into an existing Spartan prover while retaining its commitments, transcript, proof construction, and verifier behavior.

Sumcheck is a central primitive in Spartan and other proof systems based on multilinear polynomials, including Libra, HyperPlonk, and Orion [10, 36, 43, 44]. Virgo similarly develops efficient zero-knowledge proofs using transparent polynomial-delegation techniques [45]. Lasso further develops lookup arguments built around multilinear polynomial commitments, illustrating the broader use of multilinear polynomial techniques in modern proof systems [37]. Jolt builds on Lasso to construct SNARKs for virtual-machine execution using lookup-based techniques [2]. In a complete prover, however, Sumcheck does not execute as an isolated arithmetic kernel. Its rounds interact with challenge generation, commitments, and other proof-generation operations. Chimera focuses on this integration problem: the large polynomial computation executes on the GPU, while the proof-generation operations that determine the transition between rounds remain with Spartan.

A related line of work reduces proving cost through recursive proof composition and folding rather than accelerator offload. Halo develops recursive proof composition without a trusted setup, while Nova introduces recursive arguments based on folding schemes and SuperNova extends this approach to more general non-uniform incremental computations [8, 26, 27]. These systems address a different source of prover cost than Chimera, but illustrate the broader effort to restructure proof computation for more efficient proving.

General-purpose GPU cryptography libraries provide another approach to accelerator integration. ICICLE [23], for example, provides reusable GPU implementations of finite-field and proof-system primitives, including Sumcheck. We use ICICLE as the comparison point for our standalone GPU Sumcheck experiments, rather than as a baseline for the Spartan-integrated system. Chimera differs in that its integrated backend is designed around the lifetime of Spartan's Sumcheck computation, allowing the polynomial data to remain on the GPU across rounds while exposing only the values required by Spartan.

8 Conclusion

This work presents Chimera, a hybrid CPU–GPU backend for Sumcheck computations in the Spartan R1CS prover. Chimera keeps large polynomial data on the GPU across profitable rounds, performs round evaluation and challenge-driven folding on the GPU, and returns only the compact results required by Spartan. As the computation shrinks, later rounds return to CPU execution, while chunked execution allows larger inputs to be processed when the polynomial data cannot initially fit entirely in GPU memory. These mechanisms preserve Spartan's existing proof format and verifier behavior.

On BN254 Spartan R1CS instances, Chimera reduces the measured Sumcheck-related region from 672.785 ms to 84.245 ms at $N = 2^{17}$, a 7.99× speedup. Chimera improves complete proving time at every measured size and achieves 1.27×–1.64× end-to-end speedup for $N = 2^{24}$–2^{27}. Separately, in the standalone $N = 2^{22}$ quadratic Sumcheck experiment, our GPU engine runs in approximately 28.7 ms compared with 84.2 ms for ICICLE's CUDA Sumcheck implementation, a 2.94× GPU-to-GPU improvement.

The results show that Sumcheck acceleration requires more than faster GPU arithmetic. Performance depends on retaining polynomial data across rounds, adapting execution as the workload shrinks, managing GPU memory for large inputs, and placing the CPU–GPU boundary so that acceleration does not conflict with the surrounding prover. They also reveal a bottleneck shift: once Sumcheck is accelerated, unaccelerated Spartan operations, particularly larger polynomial-commitment work, account for an increasingly important portion of complete proof generation. Chimera therefore improves one major component of the prover while also identifying where further end-to-end acceleration must come from.

Acknowledgment

The authors would like to acknowledge the support of the National Science Foundation (Award 2453861).

References

1. Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *ACM CCS*, pages 2087–2104, 2017.
2. Arasu Arun, Srinath T. V. Setty, and Justin Thaler. Jolt: SNARKs for virtual machines via lookups. In *EUROCRYPT 2024*, pages 3–33. Springer, 2024.
3. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. In *IACR Cryptology ePrint Archive*, 2018. Report 2018/046.
4. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT 2019*, pages 103–128. Springer, 2019.
5. Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.
6. Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *EUROCRYPT 2016*, pages 327–357. Springer, 2016.
7. Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. Gemini: Elastic SNARKs for diverse environments. In *EUROCRYPT 2022*, pages 427–457. Springer, 2022.
8. Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. In *IACR Cryptology ePrint Archive*, 2019. Report 2019/1021.
9. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334, 2018.
10. Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In *EUROCRYPT 2023*, pages 499–530. Springer, 2023.
11. Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *EUROCRYPT 2020*, pages 738–768. Springer, 2020.
12. Alhad Daftardar, Jianqiao Mo, Joey Ah-kiow, Benedikt Bünz, Siddharth Garg, and Brandon Reagen. zkPHIRE: A programmable accelerator for ZKPs over high-degree, expressive gates. In *IEEE International Symposium on High-Performance Computer Architecture*, 2026.
13. Alhad Daftardar, Jianqiao Mo, Joey Ah-kiow, Benedikt Bünz, Ramesh Karri, Siddharth Garg, and Brandon Reagen. Need for zkSpeed: Accelerating HyperPlonk for zero-knowledge proofs. In *ISCA*, 2025.
14. Alhad J. Daftardar, Brandon Reagen, and Siddharth Garg. SZKP: A scalable accelerator architecture for zero-knowledge proofs. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 1–13, 2024.
15. Electric Coin Company. Zcash: Privacy-protecting digital currency. `https://z.cash/`, 2016.
16. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology—CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1987.
17. Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. In *IACR Cryptology ePrint Archive*, 2019. Report 2019/953.
18. Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT 2013*, pages 626–645. Springer, 2013.
19. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. In *ACM STOC*, pages 113–122, 2008.
20. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.
21. Alexander Golovnev, Jonathan Lee, Srinath Setty, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In *CRYPTO 2023*, pages 193–226. Springer, 2023.
22. Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT 2016*, pages 305–326. Springer, 2016.

23. Ingonyama. ICICLE: GPU acceleration library for cryptography. `https://github.com/ingonyama-zk/icicle`, 2024.
24. Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT 2010*, pages 177–194. Springer, 2010.
25. Sunchae Kim, Taewoon Kang, Sangwon Shin, Taeweon Suh, Yibin Yang, and Gunjae Koo. SumcheckPIM: An efficient HBM-based PIM architecture for linear complexity zero knowledge proofs. In *Proceedings of the 40th ACM International Conference on Supercomputing (ICS)*, pages 1177–1192. ACM, 2026.
26. Abhiram Kothapalli and Srinath Setty. Nova: Recursive zero-knowledge arguments from folding schemes. In *CRYPTO 2022*, pages 359–388. Springer, 2022.
27. Abhiram Kothapalli and Srinath Setty. SuperNova: Proving universal machine executions without universal circuits. In *IACR Cryptology ePrint Archive*, 2022. Report 2022/1758.
28. Muyang Li, Yueteng Yu, Bangyan Wang, Xiong Fan, Mingyu Gao, and Shuwen Deng. MegaZK: A memory efficient GPU system accelerating end-to-end zero-knowledge proof. In *Proceedings of the 40th ACM International Conference on Supercomputing (ICS)*, pages 1165–1176. ACM, 2026.
29. Tao Lu, Yuxun Chen, Zonghui Wang, Xiaohang Wang, Wenzhi Chen, and Jiaheng Zhang. BatchZK: A fully pipelined GPU-accelerated system for batch generation of zero-knowledge proofs. In *ACM ASPLOS*, 2025.
30. Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. cuZK: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on GPUs. In *International Conference on Parallel and Distributed Systems*, 2022.
31. Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. The complexity of theorem-proving procedures. In *ACM STOC*, pages 2–13, 1990.
32. Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. GZKP: A GPU accelerated zero-knowledge proof system. In *ACM ASPLOS*, pages 340–353, 2023.
33. Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In *ACM CCS*, pages 2111–2128, 2019.
34. Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, 2013.
35. Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. Accelerating zero-knowledge proofs through hardware-algorithm co-design. In *IEEE/ACM MICRO*, 2024.
36. Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *CRYPTO 2020*, pages 704–737. Springer, 2020.
37. Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Unlocking the lookup singularity with Lasso. In *EUROCRYPT 2024*, pages 180–209. Springer, 2024.
38. Tarunesh Verma, Yichao Yuan, Nishil Talati, and Todd Austin. ZKProphet: Understanding performance of zero-knowledge proofs on GPUs. In *IEEE International Symposium on Workload Characterization*, 2025.
39. Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Hyrax: Proofs of data possession and retrievability with efficient zero-knowledge. In *IEEE S&P*, pages 705–722, 2018.
40. Cheng Wang and Mingyu Gao. UniZK: Accelerating zero-knowledge proof with unified hardware and flexible kernel mapping. In *ACM ASPLOS*, 2025.
41. Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In *USENIX Security*, pages 675–692, 2018.
42. Charles F. Xavier. PipeMSM: Hardware acceleration for multi-scalar multiplication, 2022. Preprint.
43. Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *CRYPTO 2019*, pages 733–764. Springer, 2019.
44. Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero Knowledge Proof with Linear Prover Time. Cryptology ePrint Archive, Paper 2022/1010, 2022.
45. Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Virgo: Transparent polynomial delegation and its applications to zero knowledge proof. `https://eprint.iacr.org/2019/1482`, 2019. IACR Cryptology ePrint Archive, Report 2019/1482.
46. Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. PipeZK: Accelerating zero-knowledge proof with a pipelined architecture. In *ISCA*, 2021.
47. Zhiyuan Zhang, Yanxin Cai, Wenhao Yin, Xueyu Wu, Yi Wang, Lei Ju, and Zhuoran Ji. Pipelonk: Accelerating end-to-end zero-knowledge proof generation on GPUs for PLONK-based protocols. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 439–451. ACM, 2026.