Efficient Verifiable Computation Made Easy

by

Chengyuan Ma

B.S. Computer Science and Engineering, MIT, 2025

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by:	Chengyuan Ma Department of Electrical Engineering and Computer Science May 9, 2025
Certified by:	Xuhao Chen Research Scientist, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair Master of Engineering Thesis Committee

Efficient Verifiable Computation Made Easy

by

Chengyuan Ma

Submitted to the Department of Electrical Engineering and Computer Science
on May 9, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Recent advancements in cloud computing, data privacy, and cryptography have sparked a growing interest in Verifiable Computation (VC) in both industry and academia. In particular, zero-knowledge proof (ZKP) algorithms are gaining rapid traction due to their strong privacy guarantees. However, they are notoriously computationally intensive, making performance a critical concern. Given the inherent data parallelism and heavy use of vector operations in ZKP computations, multicore CPUs and GPUs offer a promising acceleration path. Unfortunately, accelerated programming for ZKP remains challenging: ZKP algorithms evolve rapidly, their structures grow increasingly complex, and writing high-performance ZKP code is tedious, error-prone, non-portable, and unfriendly to algorithm developers.

We present an end-to-end compiler framework, Zera, that lowers ZKP algorithms to parallel hardware for efficient acceleration, with minimal programmer effort.

By effectively leveraging ZKP algorithm patterns and trends, we are able to automate the key performance optimizations, with a succinct linguistic extension and a set of practical compiler customizations. Consequently, with just 92 lines of trivial high-level annotation added to the original 7,000 lines of C++ code, our single-source code solution delivers 33.9× and 24.0× speedup on GPU over a highly optimized serial C++ implementation on CPU and an existing multithreaded Rust baseline on CPU, respectively. Compared to our hand-optimized GPU/CUDA implementation requiring an extra 2,000 lines of low-level code (roughly 60 programmer hours), our compiler-generated GPU implementation is only 58% slower (1.58× slowdown) on large inputs, demonstrating a compelling trade-off between performance and productivity.

Thesis supervisor: Xuhao Chen
Title: Research Scientist

Acknowledgments

I would like to express my deepest gratitude to my advisor, Dr. Xuhao Chen, for his invaluable mentorship throughout this research. His vision and insight were instrumental in keeping the project focused on meaningful impact, steering it away from devolving into engineering minutiae. Under his guidance, I learned how to better structure research around real-world problems and communicate my contributions effectively. I began my MEng journey simply wanting to build something cool; in retrospect, I accomplished far more than I expected, thanks in large part to his encouragement and guidance.

I am also sincerely grateful to TB Schardl and the Kitsune team at LANL. A large portion of the work presented in this thesis stemmed from a fortunate encounter when TB mentioned Kitsune in a group discussion. He generously introduced me to the team and invited me to their regular meetings. TB and the Kitsune team have been immensely helpful in deepening my understanding of OpenCilk and Kitsune internals, which became the foundation of the compiler portion of this thesis. It has been a true honor to learn from and work alongside such experienced compiler developers.

Many thanks as well to the SuperTech group. Their weekly meetings were an incredible learning experience, and their thoughtful feedback on my presentations was invaluable in refining my ideas.

I would also like to thank Simon Langowski. Although we only met once, his work on hardware acceleration for ZKPs was among the initial inspirations for this project. The perspective and experience he shared were very helpful.

Finally, I am grateful to my friends and family for their unwavering support throughout this journey. As an introvert who often finds comfort in solitude, their check-ins and conversations helped me stay grounded and resilient through the inevitable moments of stress.

The success of this work would not have been possible without the collective support, insight, and faith of all these remarkable individuals.

This thesis is adapted from joint work conducted with my advisor, Dr. Xuhao Chen, and is being prepared for conference submission. The use of the first-person plural ("we") throughout this thesis reflects the collaborative nature of this research. I was responsible for all implementation, experimentation, and drafting of the thesis. Dr. Chen provided structural guidance and editorial feedback throughout the writing process. Generative AI tools (ChatGPT and Grammarly) were used for proofreading assistance; all content was reviewed and finalized by me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Advancements in cloud computing, data privacy awareness, and cryptography have sparked a growing interest in Verifiable Computation (VC) in recent years. VC enables secure and reliable data processing using untrusted infrastructure. Among the various VC technologies, *Zero-knowledge Proof* (ZKP) has emerged as the most practical and widely adopted form of VC in real-world applications. Therefore, this thesis places a specific focus on ZKP.

Formally, ZKP is a cryptographic method of proving the validity of a statement without revealing anything other than the validity of the statement itself. This "zero-knowledge" property is attractive for many privacy-preserving applications. Despite its great potential, for a long time, ZKP has been impractical due to its compute intensity. Recent advances in cryptography, known as zk-SNARK, have brought ZKP closer to practical use. zk-SNARK can generate a *succinct* proof for any program — often from a few hundred bytes to kilobytes — sublinear to the complexity of the program. This allows fast verification and cheap distribution of the proof, and empowers zk-SNARK deployment in real-world blockchain and cryptocurrency systems [1, 2]. Therefore, ZKP is now getting lots of interests recently in both industry [1, 3, 4] and academia [5–16].

Despite the fast proof verification, proof generation in ZKP remains quite expensive and slow. To generate proofs for a program, the program is first arithmetized and translated into a constraint system, whose size is usually several times larger than the initial program (e.g., up to a few million constraints). The prover then performs N arithmetic operations over a large finite field, where N is super-linear in the number of constraints. As a result, proof generation is much (e.g., hundreds of times) slower than verification; it could be up to a few minutes just for a single payment transaction, in the context of trusted digital wallets. Other VC technologies, such as homomorphic encryption, suffer from very similar performance bottlenecks, as they often use related cryptographic constructs and require similar arithmetization.

Given this performance bottleneck, there has been a lot of research and engineering (e.g., ZPrize [3]) efforts that aim to accelerate VC using parallel hardware, e.g., multicore CPUs, GPUs, and FPGAs, and even proposing ASIC chips. Among them, GPU solutions are particularly attractive. Given the inherent data parallelism and heavy use of vector operations in VC computations, GPUs offer a promising acceleration path without modifying the hardware. Moreover, GPUs are widely available in modern computing systems, including mobile devices and desktops, powerful server machines, and even large-scale datacenters and

```
1  struct Scalar { /* modular (big) integer */ };
2  // Define identity and reduction functions
3  void id_fn(void *view) { *(Scalar*)view = 0; }
4  void reduce_fn(void *left, void *right) {
5      *(Scalar*)left += *(Scalar*)right;
6  }
7
8  Scalar sum_of_squares(const Scalar *A, size_t N) {
9      Scalar parallel_reducer(id_fn, reducer_fn) sum = 0;
10     [[zera::target("gpu")]]
11     parallel_for (size_t i = 0; i < N; i++) {
12         Scalar square = A[i] * A[i];
13         sum += square;
14     }
15     return sum;
16 }
```

Listing 1.1: Parallel reducer on modular integers in ZERA. The annotation at line 10 indicates GPU offloading.

cloud computing platforms.

In the meantime, cryptography researchers are constantly improving VC (specifically ZKP) algorithms to reduce computation and make them more practical, which motivates the need for a fast prototyping solution. However, since VC algorithms are becoming increasingly sophisticated, parallel (GPU) programming for VC is particularly time-consuming, tedious, and error-prone, as detailed in Section 3.1. Moreover, hand-optimized implementations are usually specific to certain hardware architectures, and this limited portability largely hampers the adoption of these algorithms in practice.

To support productive and high-performance prototyping in the field of VC, with a focus on ZKP, we propose ZERA, a compiler-based VC prototyping framework that efficiently leverages heterogeneous hardware acceleration for VC, but with extremely low programming burden on programmers. By extending the CPU-focused task parallel model to heterogeneous hardware, e.g., GPUs, ZERA offers a *single-source* solution that can enjoy both conventional multicore parallelism and massively parallel hardware acceleration. For example, the programmer annotates line 10 in Listing 1.1 to offload the reduction computation to GPU, while a hand-optimized GPU implementation would require an extra 100 lines of CUDA code.

To the best of our knowledge, ZERA is the first holistic compiler framework for ZKP computation. Note that one way to quickly prototype GPU acceleration is to use directive annotations, e.g., OpenACC or OpenMP. However, ZKP algorithms operate on finite field with complex data types, which makes existing directive based solutions inapplicable. For example, in OpenACC there is no trivial way to deal with parallel reduction of modular integers, a major operation in ZKP algorithms, e.g., line 9–15 in Listing 1.1. There also exist GPU libraries, e.g., Thrust [17] and CUB [18], which are limited as they are bound to specific GPU vendors and their use precludes certain optimizations like kernel fusion.

The key challenge in building ZERA lies in the following aspects. First, given the source language in the fork-join parallel model suited for multicore CPUs, optimizing for high efficiency on GPUs is nontrivial to automate. Second, existing compiler/runtime support for

GPU architectures [19] is very limited in both functionality and performance, which cannot meet the requirements of ZKP algorithms.

Therefore, instead of trying to build a generic system, our idea is to leverage ZKP domain knowledge to simplify and customize the system design. To this end, we analyze representative ZKP algorithms, summarize algorithm trends, and extract computational patterns in them. In addition to the abundant data parallelism, we identify that ZKP computation is dominated by specific computational patterns, e.g. `map-reduce`, while more sophisticated patterns appear much less frequently. ZKP algorithms often operate on complex data structures — such as ultra-wide vectors — which demand specialized system support. At the same time, they exhibit structured parallelism that can potentially simplify the design of parallelization strategies. On the other hand, modern ZKP algorithms include many computational steps and inherently involve plenty of data communication and synchronizations, which is particularly problematic in the CPU-GPU heterogeneous platform.

Based on these observations, we introduce three tailored compiler techniques for ZKP, to address the GPU automation challenges. First, to enable efficient GPU kernel code generation, We propose a *pattern-aware lowering* strategy, which enjoys the same simple, generic task programming interface for CPU, but can also efficiently perform ZKP patterns over modular integer data on the GPU. Second, to further improve code generation in the context of nested nested parallelism, we propose *collaborative heterogeneous parallelism*, which addresses the complexity of hierarchical thread mapping by using GPU multi-stream parallelism with assistance of CPU multithreading. We also expose easy-to-use synchronization control to the programmer and allow *asynchronous* multi-stream kernel launching, to minimize synchronization overhead. Lastly, to deal with frequent memory allocation and data copy across CPU and GPU, we propose *adaptive memory management*. It heuristically allocates unified virtual memory to reduce memory fragmentation and page fault overhead, and also mitigates data movement overhead by automatically inserting host-device data copy function calls.

We build Zera based on the OpenCilk [20] ecosystem, and implement a representative algorithm *Shockwave* [21] in Zera, with just 92 lines of trivial high-level annotation added to the original 7,000 lines of C++ code. We also hand-optimize Shockwave on GPU with 2,000 extra lines of CUDA code. Experiments demonstrate that our single-source Zera solution delivers a 24.0× speedup on GPU over an existing multithreaded Rust baseline on CPU, with 1/40 programming time and only 1.58× slowdown than hand-optimization.

This paper makes the following contributions.

- We build Zera, the first compiler-based accelerated programming framework for agile ZKP algorithm development. It offers a *single-source* solution that largely simplifies acceleration on heterogeneous parallel hardware.
- We holistically analyze recent ZKP algorithms in an end-to-end fashion, identify computational patterns, acceleration opportunities, and performance bottlenecks.
- We propose tailored compilation techniques that automate ZKP performance optimizations, including pattern-aware lowering, collaborative CPU-GPU parallelization, and adaptive heterogeneous memory management.

- We implement Zera, and show that with a few simple annotations, Zera can deliver significant speedups on multicore and GPU. Zera also largely reduces programming complexity compared to hand-optimized GPU programs, with only a small performance loss.

Chapter 2

Background and Related Work

2.1 Verifiable Computation

Verifiable Computing describes cryptographic protocols enabling one party to check the correctness of computations performed by another party without re-executing the entire computation. In many scenarios, one party is a computationally weak client outsourcing the expensive computation to an untrusted worker; in other applications, one party may possess private information that the other party must not know, in which case verifiable computing is performed with the added requirement of being zero-knowledge. Verifiable Computing has gained significant attention in recent years due to its potential applications in cloud computing, blockchain, and other distributed systems where trust, computational integrity, or data privacy are paramount (we give more examples in Section 2.2 below). The theoretical foundation of verifiable computing was established by the PCP theorem [22] in 1992. Since then, practical techniques for achieving verifiable computing have been developed, including:

1. **Homomorphic Encryption (HE)** allows a computationally powerful worker to compute directly on encrypted data provided by the other party.

2. **Succinct Non-interactive Argument of Knowledge (SNARK)** allows the prover to convince the verifier of the validity of an NP statement, where the proof can be efficiently validated (verifier runs in time sublinear to circuit size).

3. **Zero-knowledge Proof (ZKP)** allows one party (the prover) to convince another party (the verifier) of the validity of an NP statement, where the verifier learns nothing except the validity of the claim itself. A ZKP protocol can adopt a SNARK as the underlying proof scheme, in which case it is also referred to as **zk-SNARK**.

This thesis will primarily focus on ZKP. However, from a performance engineering perspective, we believe that VC algorithms in these directions are similar, so our work generalizes to all VC technologies.

2.2 Zero-knowledge Proof and Proof Generation

Zero-knowledge proof (ZKP) describes a family of cryptographic algorithms that has drawn great attentions in the recent years. In ZKP, one party (the prover) can prove to another party (the verifier) that a given NP statement is true, without revealing any information beyond the mere fact of the statement's validity. ZKP algorithms have demonstrated their applications in blockchain [23], verifiable outsourcing [24, 25], verifiable machine learning [12, 15, 26, 27], electronic voting [28], online auctions [29], static program verification [30], verifiable image editing [31], verifiable database [32, 33], electronic cash [34], and more.

Recent empirical studies consistently show that proof generation remains a significant performance bottleneck. Producing a proof of some computation remains 5–6 orders of magnitude slower than running the computation natively on CPU [35]. For instance, it takes recent ZKP implementations 10–100 seconds to prove millions of modular arithmetic [36], SHA-256 of kilobyte-sized preimages [36, 37], edits to 100–1000 kB images [38], or small neural net inference over a dozen images [39, 40]. In contrast, these tasks can be done near-instantly if executed natively.

Many algorithmic improvements have been made to improve the prover performance, such as optimized finite field arithmetic [41–44], smaller fields [45–48], application-specific prover specialization [12, 40], and avoiding performance bottlenecks such as elliptic curve operations [21, 49, 50] or NTT [21, 48, 51]. These optimizations, when combined, improved prover performance by an order of magnitude [36, 37]. Nevertheless, the performance gap remains wide and limits broader applications of the ZKP technology.

2.3 Acceleration for Proof Generation

ZKP algorithms involve inherent data parallelism and heavy use of vector operations. Algorithm 1 gives an example of typical ZKP prover computation — a batched sumcheck [52, 53] protocol, reducing the sum of evaluations of polynomials over the boolean hypercube to point evaluations of their multilinear extensions. The abundant parallelism (e.g., line 3) motivates acceleration using parallel hardware like GPUs.

There have been prior performance engineering attempts to parallelize and speed up ZKP proof generation on CPU SIMD [54], GPUs [5, 9–11, 13, 16, 55], FPGAs [8, 14, 56], TPU [57], or even custom ASIC accelerators [6, 7, 35, 58–60]. In particular, Multi-scalar Multiplication (MSM), a bottleneck of the Groth16 [61] ZKP algorithm, has attracted significant attention. GZKP [5] explored a novel GPU parallelization strategy that improved load imbalance, and DistMSM [13] leveraged multiple GPUs for large inputs; HardcamlMSM [14] presented an FPGA implementation of MSM. Nevertheless, Groth16 is an early algorithm and many newer ZKP protocols that address its limitations no longer include MSM as a core operation.

However, high performance comes at the cost of escalated programming complexity. Manually implementing these optimizations for certain hardware requires the programmer to be familiar with both the algorithm and the hardware architecture. These implementations often turn out to be time-consuming, tedious, error-prone and unportable. Therefore, a key challenge in ZKP system design is how to improve performance while retaining high programming productivity.

Algorithm 1 A round of batched cubic sumcheck used in Spartan [53] zk-SNARK.

Require: size-b lists of width-2^n finite field vectors $\{A_i\}_{i=1}^b$, $\{B_i\}_{i=1}^b$, $\{C_i\}_{i=1}^b$; a width-b vector v; a cubic function f

$\pi_0 = 0, \pi_2 = 0, \pi_3 = 0$ ▷ reducer variables
for $i \leftarrow 1$ **to** b **do** ▷ b is around a dozen
  for $j \leftarrow 1$ **to** 2^{n-1} **do** ▷ e.g., $n = 20$, highly parallel
    $\pi_0 \mathrel{+}= v[i] \cdot f(A_i[j], B_i[j], C_i[j])$ ▷ reduction
    $\pi_2 \mathrel{+}= v[i] \cdot f(2A_i[j] - A_i[j + 2^{n-1}], 2B_i[j] - B_i[j + 2^{n-1}], 2C_i[j] - C_i[j + 2^{n-1}])$
    $\pi_3 \mathrel{+}= v[i] \cdot f(3A_i[j] - 2A_i[j + 2^{n-1}], 3B_i[j] - 2B_i[j + 2^{n-1}], 3C_i[j] - 2C_i[j + 2^{n-1}])$
  end for
end for ▷ synchronization point
$r \leftarrow g(\pi_0, \pi_2, \pi_3)$ ▷ dependency on π_0, π_2, π_3
for $i \leftarrow 1$ **to** b **do**
  for $j \leftarrow 1$ **to** 2^{n-1} **do**
    $A_i[j] = r \cdot A_i[j + 2^{n-1}] + (1 - r) \cdot A_i[j]$
    $B_i[j] = r \cdot B_i[j + 2^{n-1}] + (1 - r) \cdot B_i[j]$
    $C_i[j] = r \cdot C_i[j + 2^{n-1}] + (1 - r) \cdot C_i[j]$
  end for
  halve the widths of A_i, B_i, C_i ▷ parallelism varies
end for

2.4 Automatic Parallelization for GPU

Compiler-based solutions have been developed for automatic parallelization, such as OpenMP-target [62]; OpenACC [63], OpenMPC [64], OpenMP2GPGPU [65]. These solutions allow the programmers to annotate loops with pragma *directives*, which instructs the compiler frontend to lower parallel loops. The frontend thus handles thread mapping and data movement for the GPU. Although they provide sophisticated performance tuning directives [63], this frontend-based approach has its inherent limitations. For example, as parallel loops are lower in the frontend, they do not undergo loop optimizations such as hoisting. Due to the same reason, the directives depend on literal forms of their arguments, and support for custom containers like `std::vector` is not standardized. In addition, they are tightly bound to specific languages and require substantial redesign when porting to a new language. Moreover, OpenACC and OpenMP hardcode the supported reduction operators to `+`, `*`, `max`, etc, over primitive types. While this solution suffices for HPC workloads, we show in Chapter 3 that ZKP operates on modular integers and elliptic curve group elements and their reduction cannot be expressed under OpenACC/OpenMP directives.

In addition to these directive-based frameworks, specific compiler optimizations for GPU parallelization have been developed: Split-tiling [66] aims to generalize efficient kernels for stencil computations. TACO [67], TVM [68], XLA [69], and PyTorch [70] are domain-specific compilers and frameworks that automatically parallelize sparse linear algebra and tensor operations over primitive types. Unfortunately, stencil computation is not common in ZKP, and ZKP operates on modular integers, which makes TACO and PyTorch inapplicable. ADSM [71] and CGCM [72] aim to simplify and accelerate data transfer between the device and the host, which are largely superseded by Unified Virtual Memory [73] supported by

modern GPUs.

IMCompiler [11] is a custom compiler supporting ZKP computation. More specifically, it automatically generates optimized GPU kernels for big modular integer multiplications. However, the integers they target ($\geq$ 1024 bits) are mainly used for key exchanges and are too big for modern ZKP algorithms. More importantly, IMCompiler is specific to big integer multiplications, while our work aims to support end-to-end parallel solutions for modern ZKP algorithms.

There also exist libraries, e.g., Thrust [17], CUB [18], Gunrock [74], that simplify parallelization on GPU. Users call functions that are hand optimized by library developers. However, they are often bound to specific GPU vendors and their use precludes certain optimizations like kernel fusion.

2.5 OpenCilk and Kitsune

OpenCilk [75], originated from Cilk [76], is a task-parallel platform that makes parallel programming a simple extension of serial programming. It provides language abstractions for shared-memory parallel computations on multicore CPUs. Specifically, Cilk supports fork-join parallelism, and provides linguistic mechanisms for task spawning (with `cilk_scope` and `cilk_spawn`) and parallel loops (with `cilk_for`).

OpenCilk also supports parallel reduction through reducer hyperobject [77]. Unlike OpenMP or OpenACC, OpenCilk supports arbitrary associative reduction through user-defined identity and reduction functions. Conceptually, updates to reducer variables in parallel tasks are redirected to the tasks' "local views," looked up through a runtime call. OpenCilk's CPU runtime implements the call and then merges these views implicitly on task join.

Cilk programmers are only responsible for expressing the logical parallelism in the application, that is, which tasks may run in parallel. The OpenCilk compiler produces optimized parallel code, and the OpenCilk runtime system schedules and load-balances the computation onto the available processors. OpenCilk features the Tapir [78] extension to LLVM IR, which provides a first-class representation of fork-join parallelism. As an IR-level extension, Tapir enables parallelized code to undergo regular loop optimizations. After these optimizations, Tapir IR instructions are lowered to calls to OpenCilk's work-stealing runtime.

Based on the Tapir extension, Kitsune [19] extends the compiler backend to generate GPU code. Kitsune can only deal with flat `parallel_for` loops in the source program, which are lowered into regular LLVM loops with annotated backedges and surrounded by Tapir `sync` and `detach` instructions. These loops then undergo regular loop optimizations such as hoisting and unrolling, after which a *loop spawning* pass lowers loops into runtime calls, with a simple strategy of one iteration per GPU thread. Loop bodies are outlined and cloned to a separate LLVM module, together with all dependencies, and compiled to PTX with LLVM NVPTX backend. The loops are then replaced by kernel launch calls and prefetches. Kitsune uses Unified Virtual Memory (UVM) [73] for memory management and requires explicit use of specialized `__kitrt_malloc` and `__kitrt_free` functions, which wrap UVM functions and maintain a map from pointer to metadata. The Kitsune compiler identifies all pointers used in the kernel and inserts runtime calls to prefetch them to the GPU, with the size of

every prefetch obtained from the metadata map. However, we show in Section 4.3 that data copy is a major bottleneck and propose our optimizations to address it. In addition, Kitsune can not handle nested parallelism, and does not support OpenCilk programming interface, e.g., Cilk reducer, which makes it hard to support ZKP computation.

Chapter 3

Zera: Striking Performance and Productivity

This chapter presents our analysis of ZKP algorithms from a performance engineering perspective. Based on our analysis, we lay out our design for the programming interface of Zera, our compiler system. To automate performance optimizations, we also identify computational patterns in ZKP algorithms, which will motivate our customized compiler techniques in Chapter 4.

3.1 Characterization of ZKP Algorithms

Modern ZKP constructions [79] often consist of four components: arithmetization, interactive oracle proof (IOP), commitment schemes, and Fiat-Shamir transform, as detailed in Appendix A. We dive deep into the algorithms and make the following observations.

First, ZKP algorithms involve inherent data parallelism and heavy use of vector operations. Without loss of generality, every ZKP protocol connects a chain of cryptographic constructs through a series of mathematical transformations, which computationally involve operations on *structured* data such as finite field vectors, matrices, and polynomials, all of which include abundant data parallelism. Consequently, we observe structured parallelism due to the data regularity, and in particular, high parallelism present in the innermost loop. For example, in Algorithm 1, although the trip count b of the outer loop is small, the inner loop has high parallelism over width-2^n vectors. Additionally, operations on these data types have *high arithmetic intensity*, because multiplication and modular reduction have complexity quadratic in word limb count. On the other hand, operating on multi-word modular integers and elliptic curve group elements, rather than primitive data types, complicates automatic parallelization. This is particularly problematic for solutions like OpenACC, which have limited support for custom data types, especially for reduction.

We investigate ZKP algorithms listed in Table 3.1 and discover that they evolve rapidly and increasingly complex, while code reuse across different algorithms is impractical or extremely challenging. As shown in Table 3.1, the code complexity, in terms of lines of code, has increased dramatically over the past few years. ZKP algorithms are constantly evolving to support greater expressivity and performance. For example, Jolt [87] enables

	Year	LoC	Parallel patterns				
			MSM	FFT	map/zip-reduce	Merkle tree	SpMV
Groth16 [61]	2016	2100	✓	✓	✓	✗	✗
Bulletproof [80]	2017	4700	✓	✗	✓	✗	✗
Hyrax [81]	2017	7300	✓	✗	✓	✗	✗
Libra [82]	2019	11700	✓	✗	✓	✗	✗
Spartan [53]	2019	8400	✗	✗	✓	✗	✓
Plonky [83]	2019	15000	✓	✓	✓	✗	✗
Virgo [84]	2019	11500	✗	✓	✓	✓	✗
Virgo++ [85]	2020	8900	✗	✓	✓	✓	✗
Nova [86]	2021	26000	✓	✗	✓	✗	✓
Shockwave [21]	2021	8100	✗	✓	✓	✓	✓
Brakedown [21]	2021	8500	✗	✗	✓	✗	✓
Orion [49]	2022	9500	✗	✗	✓	✓	✓
Plonky2 [45]	2022	44400	✗	✓	✓	✓	✗
HyperPlonk [51]	2022	7300	✗	✗	✓	✓	✓
Jolt [87]	2023	52800	✓	✗	✓	✗	✓
Binius [48]	2023	93900	✗	✗	✓	✓	✗
Plonky3 [46]	2024	53000	✗	✓	✓	✓	✓

Table 3.1: Parallel patterns and implementation complexity of recent ZKP algorithms, ranked by release date. LOC (lines of code) column shows the growing algorithm complexity.

RISC-V instruction traces to be arithmetized directly without translating programs into traditional circuit-based representations; Binius [48] introduces new binary field-friendly arithmetization and commitment techniques that are more hardware-friendly; Orion [49] achieves linear prover time and succinct proofs with recursive proof composition. While such advances build on prior ideas, their implementations *rarely allow code reuse* and often require significant adaptation. Although a conceptual layering exists, in practice, different algorithm steps are closely coupled and specialized to each other for performance. Improvements in one component often necessitate changes in others, making code abstraction and reuse across ZKP systems extremely hard. A major challenge posed by algorithm complexity is that, unlike deep learning, where a few kernels (e.g., GEMM) dominate the performance, the ZKP algorithms involve many steps that involve dozens or hundreds of non-trivial kernels. This means a huge burden for hand optimization.

To better understand algorithm details and performance bottlenecks, we hand-optimize a representative ZKP algorithm, Shockwave, in an end-to-end fashion. We first faithfully re-implement Shockwave in 7,000 lines of C++ following a Rust baseline, with optimizations to reduce redundant copies and improve cache locality, and then parallelize it onto GPU using CUDA. We learn the following insights from this manual experience. First, despite code-sharing efforts and using libraries [18], it is extremely time-consuming to manually optimize this complex ZKP algorithm on GPU, which involves 2,000 lines of CUDA code and takes a highly experienced CUDA programmer about 60 hours, to achieve >50× speedup than serial C++. In addition, although the data parallelism in ZKP is very GPU friendly and leads to

tremendous speedups, especially compute kernel-wise, for nested parallelism, we have to flatten loops or adopt careful hierarchical thread mapping strategy to minimize synchronization overhead and improve load balance. Finally, despite GPU-friendly kernels, end-to-end performance is hampered by plenty of data communication between the CPU and GPU. Hence, we manually orchestrate data transfer to minimize communication overhead, which solely takes $\sim 20\%$ of the development time. Finally, we specifically optimize kernels such as NTT with more advanced techniques like tiling, but this leads to a marginal performance gain as they are not frequently called.

Overall, the characteristics of ZKP algorithms motivate the need for parallel hardware acceleration and a highly automated prototyping solution, e.g., compiler support, to facilitate the agile development of new ZKP algorithms/systems.

3.2 Data Parallel Patterns in ZKP

While generating optimized GPU kernels for arbitrary computations remains difficult, we find it tractable to build a *domain-specific* compiler targeting ZKP systems. This is because ZKP systems predominantly use a small set of structured *parallel patterns* amenable to compiler analysis and transformation. In particular, as discussed in Section 3.1, modern ZKP systems [21, 48–50, 53] connect their multiple components with transformations of mathematical objects with regular data representations, such as finite field vectors and polynomials. Several parallel patterns naturally arise in this setting:

1. `map`/`zip` over *finite field arrays* — basic vector and polynomial arithmetic.
2. `reduce` over *finite field* — polynomial evaluation, inner product, and reduction of proof claims (e.g., sumcheck [52]).
3. `Merkle tree` *construction* — for succinct commitments to large datasets where only a few positions are later opened.
4. `NTT` (*Number Theoretic Transform*) — Reed-Solomon code used by linear-code-based schemes [50, 88] for soundness.
5. `SpMV` (*finite-field sparse matrix-vector multiplication*) — in inner-product-like arguments [53] or linear-time encoders [49].
6. `MSM` (*multi-scalar multiplication*) [8], used by early KZG [89] commitments.

Table 3.1 summarizes the appearance of these patterns in ZKP algorithms. We observe from Table 3.1 that `map-reduce` and `zip-reduce` are dominant patterns that appear frequently, while `MSM` is less frequently used in newer ZKP protocols. We consider `map`/`zip` together with `reduce` because, as shown in Algorithm 1, `map`/`zip` and `reduce` operations often appear in fused forms which are especially amenable to optimization. By focusing on this small set of patterns and prioritizing the most frequent patterns, we can design an intuitive linguistic extension and a compiler that achieves reasonably high performance on a GPU.

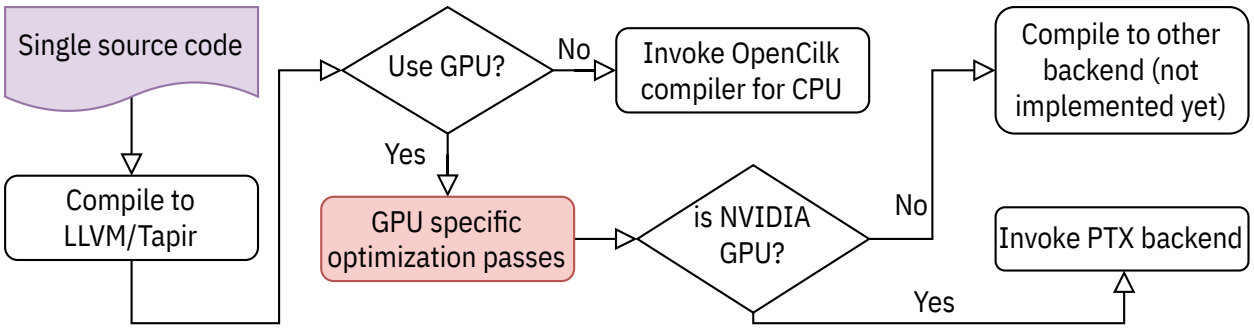


Figure 3.1: ZERA compilation workflow.

3.3 System Overview and Programming Interface

ZERA provides a simple task-parallel programming model and a single-source user interface for heterogeneous hardware. Like existing CPU-based task programming models, the ZERA programmer uses keywords, e.g., `parallel_for`, to indicate logical parallelism. We provide a simple linguistic extension, `target("gpu")` attribute, for users to indicate GPU offloading, as shown in Listing 1.1 line 10. This allows programmers to accelerate their code without understanding low-level architectural details. Our system guarantees that logically-parallel CPU tasks launch logically-parallel accelerator (e.g., GPU) tasks, which join as the CPU tasks join. The task structure on the accelerator mirrors the fork-join task graph on CPU, kept in `sync` by default, and we grant programmer control to `detach` and `re-sync` the two at times for better performance (see Section 4.2).

In particular, we support exactly the same reducer interface for both CPU and GPU settings. This way, little code change is required from the programmer side for reduction on the GPU. Listing 1.1 shows an example of how to use `parallel_reducer` that targets a GPU. The programmer is allowed to mark an arbitrarily typed variable or object as a `reducer` (line 9). The reducer can have identity and reduction functions of their choice, specified by (`id_fn`, `reduce_fn`) type decorator (line 1 to 6), which is why we are able to easily support reduction on modular integers and elliptic curve group elements. Line 10 is the only change needed to have ZERA compile the reduction to GPU.

Fig. 3.1 shows the compilation workflow in our framework. The programmer writes a single-source code (purple box) that can be executed on heterogeneous hardware architectures. The source code is first compiled by the OpenCilk frontend to Tapir/LLVM IR. We extend the frontend to support our linguistic extension for GPU. Our modified frontend attaches `target("gpu")` attribute to loop backedge as the loop is lowered to IR. The loop then undergoes LLVM's target-independent loop optimizations. After that, if targeting CPU, we invoke the original OpenCilk backend and generate the binary for the multicore CPU. When targeting GPU, the IR is then optimized by our GPU-specific optimization passes (red box), which ensure the high efficiency of our generated ZKP code on GPUs and are the major contributions of this work (see details in Chapter 4). We then extend Tapir/Kitsune's *loop spawning* pass to lower these loops to GPU kernels. For NVIDIA GPUs, it invokes LLVM's NVPTX backend, while for other GPUs, we need to invoke other corresponding backends

(e.g., AMD GPU), which have not been fully functioning yet. Our evaluation in this paper thus focuses on NVIDIA GPUs.

While our implementation is based on the OpenCilk [20] ecosystem, Zera is not specific to any particular language. Being agnostic to the source language is important as the programmer has more flexibility. As OpenCilk has ongoing support for Rust [90], our Zera design can be easily extended to support Rust, which is difficult for OpenACC/OpenMP.

Chapter 4

Compiler Support and Optimizations

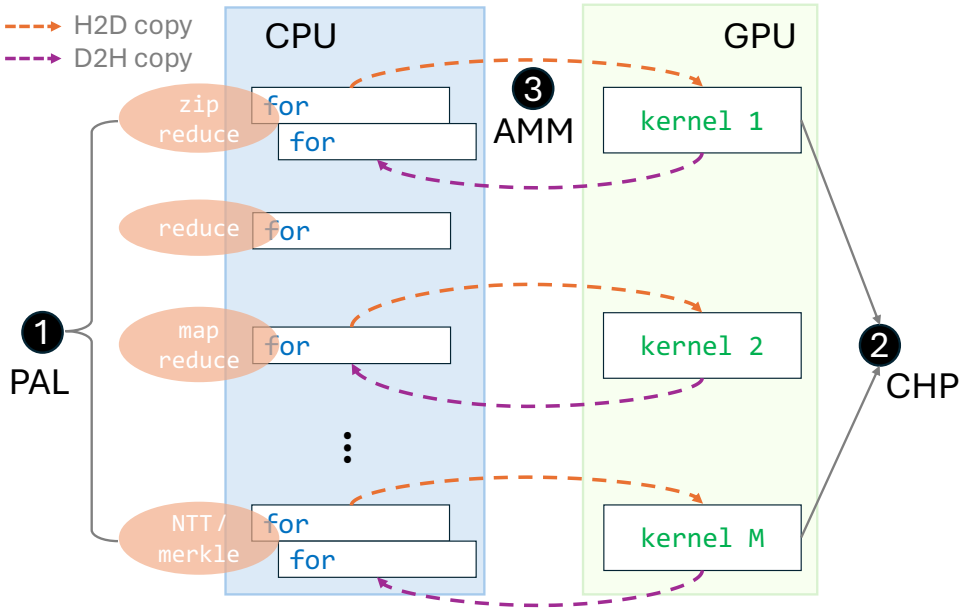


Figure 4.1: Overview of compiler support and optimizations: ❶ pattern-aware lowering, ❷ collaborative heterogeneous parallelism, and ❸ adaptive memory management. ❶ and ❷ generates efficient GPU kernels. ❸ optimizes data movement.

As shown in Fig. 3.1, our key contribution is the GPU-specific optimizations in our compiler. To enable high-performance ZKP on GPUs, we need two components: efficient GPU kernels and efficient management of CPU-GPU data movement. Therefore, as shown in Fig. 4.1, our compiler support includes ❶ (Section 4.1) and ❷ (Section 4.2), for generating efficient GPU kernels, and ❸ (Section 4.3) for managing CPU-GPU memory and data movement.

Automatic generation of performant GPU kernels is generally hard. First, mapping computation to the GPU hierarchy is non-trivial. CUDA programming involves managing *threads* (within *warps*), *blocks*, and *grids*, as well as different memory spaces (global, shared, constant, local, texture). A compiler has to make smart decisions about how many threads/blocks to use, which computations should go where, how to avoid bank conflicts

and divergence, etc. These choices depend heavily on the data access pattern and algorithm structure, which are hard to analyze statically. Additionally, GPU performance hinges on details like warp occupancy, memory coalescing, shared memory usage, register pressure, specialized instructions, and instruction-level parallelism. These are hard to reason about without hardware-specific knowledge. Compilers thus either make conservative guesses, which leads to suboptimal performance, or try to be aggressive at the risk of hurting correctness or portability.

4.1 Pattern-aware Lowering

Our first insight is that ZKP algorithms tend to follow a narrow set of computational patterns (see Section 3.1), which allows us to simplify kernel generation significantly. In particular, the dominant patterns are `map`, `zip`, and `reduce`, where loop iterations are typically independent and exhibit low data reuse. Consequently, scratchpad tiling, which is very difficult to automate, does not significantly impact performance in these patterns. Beyond these basic patterns, more complex patterns like `NTT`/`FFT` or `SpMV` benefit from tiling, but they are either infrequently used or no longer bottlenecks in modern ZKP schemes. Hence, these can be parallelized using suboptimal but simpler strategies without significantly affecting end-to-end performance. Additionally, ZKP kernels operate over modular integers, rendering hardware-specific instructions such as fused floating-point multiply-accumulate irrelevant. This further reduces the complexity of code generation.

Guided by these observations, we propose a *pattern-aware lowering* strategy in our compiler. For common flat-loop patterns like `map` and `zip`, we apply Kitsune's straightforward iteration-to-thread lowering (Section 2.5). For `reduce` patterns, which involve data dependencies, and thus critical for performance and benefit from careful handling, we introduce an optimized lowering path specialized for commutative reduction. Finally, nested parallelism is addressed in Section 4.2.

Automating the generation of generic `reduce` across various data types and operators is hard. Fortunately, we find in ZKP algorithms that reductions are performed over field operations and are thus *commutative* — they can be accumulated to global memory out-of-order. Furthermore, we observe that `reducer` variables (modular integers) are small, but reduction operations (modular arithmetic) are relatively complex, taking dozens of cycles. This means the user-defined *compute* dominates the compiler-generated *communication*, and thus, a suboptimal code generation is acceptable.

With the observations, we propose a compiler pass that lowers the reducers to high-performance reductions on GPU. Our approach adopts a typical hierarchical reduction scheme, as shown in Fig. 4.2, which leverages the GPU memory hierarchy to achieve high performance. Our compiler frontend identifies uses of reducer variables in the loop body and marks them with intrinsic calls returning task-local views. To lower to the GPU in the backend, we identify these lookup calls in the outlined kernel and instantiate them with `alloca` at kernel entry, effectively creating one local view per GPU thread for every reducer. We also generate block-wide tree reductions on these local views at the end of the kernel. Finally, block sums are accumulated atomically to global memory.

In the case of multiple reducers in the same kernel, instead of generating a separate

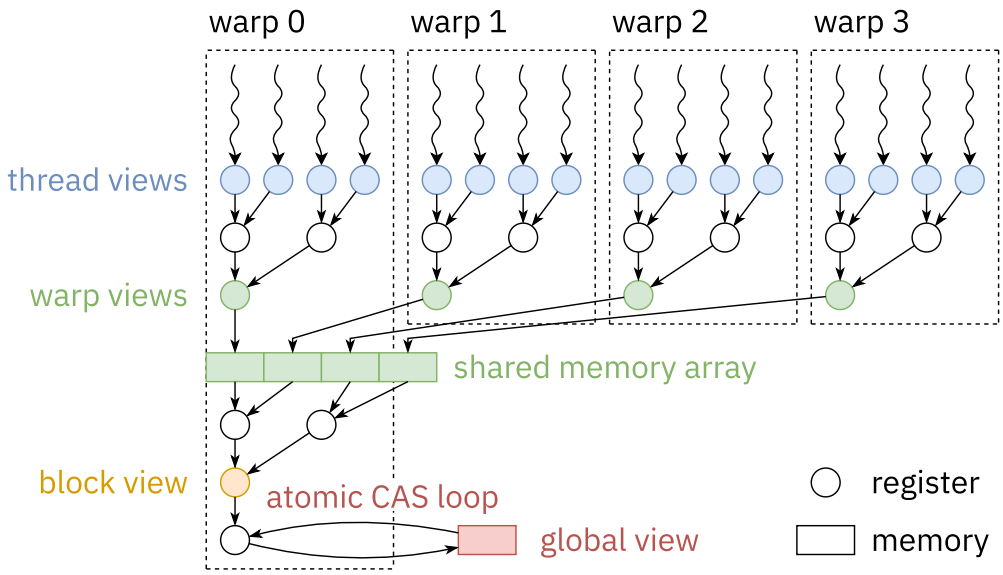


Figure 4.2: Hierarchical reduction generated by our compiler. A tree reduction is first performed within each warp to obtain warp-wide sums ("warp views"), written to a shared memory. All but the first warp exits and the first warp reads the warp views and does another tree reduction to obtain the block-wide sum ("block view"), subsequently accumulated to the global view via an atomic compare-and-exchange (CAS) loop.

reduction tree for each reducer variable, our compiler fuses all local reducer views into a single hierarchical reduction tree at the end of the kernel. This design eliminates redundant synchronization: all reducers are reduced together at each shuffle or aggregation step, amortizing warp shuffle and shared memory communication overhead across reducers. It also minimizes control divergence, avoids excess register pressure from serializing multiple reductions, and allows non-contributing warps to terminate earlier once their contributions are merged. In practice, since typical ZKP kernels have a small to moderate number of reducers, fusing reductions achieves lower intra-block synchronization cost without noticeably increasing register or memory pressure.

4.2 Collaborative Heterogeneous Parallelism

We observe that ZKP algorithms exhibit abundant opportunities for *nested parallelism*, e.g., nested loops at lines 2–3 and 11–12 in Algorithm 1. Mapping nested loops onto parallel hardware is a scheduling problem that aims for load balancing to achieve high hardware utilization. The optimal mapping strategy is highly *case-specific* and often *input-dependent*. While task-stealing runtimes [75] handle this problem gracefully on CPUs, it is more complex on GPUs.

Since GPUs organize hardware units hierarchically, they naturally expose a hierarchical threading model that requires the programmer to statically map threads in multiple levels. A static mapping thus can not handle input dependency or varying parallelism (e.g. line 17 in Algorithm 1). Assuming trip counts N and M for the outer and inner loops, both input dependent, the programmer may map the outer loop to thread blocks, and the inner loop to

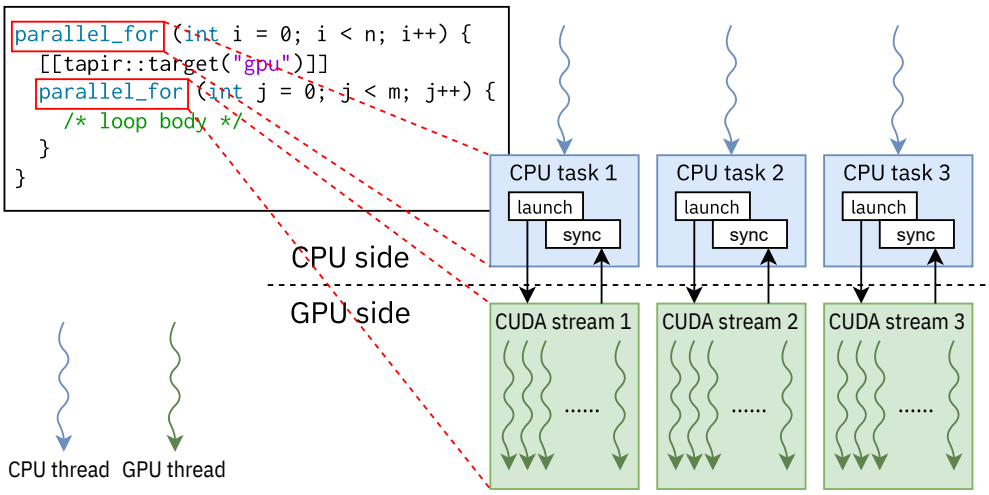


Figure 4.3: Collaborative CPU-GPU parallelism in Zera that handles nested parallelism automatically. The innermost loop with abundant parallelism is lowered to GPU. Outer parallel loops are lowered to parallel CPU tasks.

threads in a block. However, this strategy works poorly when N is small, due to insufficient thread blocks to saturate the GPU hardware units. Recall that the case of a small N and large M is common in ZKP, as demonstrated in Algorithm 1. Another common strategy is flattening the nested loop. Once flattened, a simple mapping strategy (e.g., one iteration per thread) would work well. However, loop flattening requires that the inner loop iteration M is invariant. Anyway, both strategies require a *holistic* understanding of the algorithm, which is hard for a compiler to infer. In addition, nested loops might cross function boundaries, while compilers often operate with limited context, further complicating the problem.

We address the challenge by exploiting multi-stream parallelism on GPU, with the assistance of CPU multithreading, namely, *collaborative heterogeneous parallelism*, as shown in Fig. 4.3. Multiple outer loop iterations may run on different CPU threads, each of which simultaneously launches the inner loop (likely with high parallelism M) as a GPU kernel, to keep the GPU busy. By leveraging CPU multithreading to launch GPU streams, our approach elegantly delegates dynamic behavior handling (e.g., input dependency) to the runtime schedulers, which near-optimally coordinate tasks to sustain GPU utilization. This strategy incurs additional kernel launch overhead, but since it achieves high GPU utilization, it is overall a compelling tradeoff. More importantly, this strategy enables an automatable solution for managing complex, data-dependent parallelism, as it is easy to reason by the compiler, and naturally extends to arbitrary fork-join parallelism outside the innermost loop.

A key runtime component of our approach is then stream management. As operations on the same stream are serialized, parallel kernel launches from different tasks must use different streams. In Zera, streams and the ordering of kernel launches match the fork-join structure of tasks:

- Every task gets assigned a private stream for the duration of its execution. This ensures kernels launched from different tasks may logically run in parallel.
- When tasks join, so do their streams. Any kernel launched in the joined task will be

```
// Scenario 1: for parallel tasks launching kernels, defer syncs to after join point
struct Scalar { /* modular (big) integer */ };
void fold(std::vector<Scalar> &P, const Scalar &r) {
    const size_t half = P.size() / 2;
    [[zera::target("gpu"), zera::deferred_sync]]
    parallel_for (size_t j = 0; j < half; j++)
        P[j] = r * P[j + half] + (1 - r) * P[j];
    // Otherwise, compiler inserts sync here.
    P.resize(half); // Need not observe kernel effect
}
void hyrax_round() {
    std::vector<std::vector<Scalar>> As, Bs, Cs;
    // ... omitted, see Algorithm 1
    const Scalar &r = /* ... */;
    parallel_for (size_t i = 0; i < n; i++) {
        spawn fold(As[i], r);
        spawn fold(Bs[i], r);
        fold(Cs[i], r); sync;
    }
    zera::sync_current_stream(); // sync deferred to here
}

// Scenario 2: remove syncs between sequential kernel launches
using Hash = std::uint8_t[32];
void merkle_tree(std::vector<Hash> hashes, size_t n) {
    // Assume hashes has size 2*n - 1 and has first n entries initialized as leaves for
    the merkle tree.
    std::span<Hash> prev_layer{hashes.data(), n};
    std::span<Hash> cur_layer = next_half(prev_layer);
    while (cur_layer.size() >= 1) {
        [[zera::target("gpu"), zera::deferred_sync]]
        parallel_for (size_t i = 0; i < cur_layer.size(); i++)
            blake3::hash_64(prev_layer[2 * i].data(),
                            cur_layer[i].data());
        // Otherwise, compiler inserts sync here.
        prev_layer = cur_layer;
        cur_layer = next_half(prev_layer);
    }
    zera::sync_current_stream();
}
template <typename T> auto next_half(std::span<T> s) {
    return std::span<T>{s.data() + s.size(), s.size() / 2};
}
```

Listing 4.1: Example of eager vs. lazy synchronization.

sequenced after all outstanding kernels launched from predecessor tasks.

To minimize stream management overhead, streams are joined asynchronously in Zera. We allocate streams on demand and recycle them whenever possible to minimize stream creation and destruction overhead. Consequently, the total number of streams used by our runtime is bounded by $3PD$ [77, 91], where P is the number of workers and D the maximum depth of the fork-join task graph.

In addition, we propose a hybrid *eager* and *lazy* synchronization to further reduce overhead. By default, we use the eager strategy, where every kernel is synchronized from the host immediately after launch, as we assume subsequent host-side code depends on the kernel's effect. This way, a nested parallel loop consequently involves one synchronization per outer loop iteration. In contrast, the lazy strategy allows kernels to be launched asynchronously within tasks, postponing synchronization until the corresponding join point. Our system allows the programmer to switch between eager and lazy mode. As shown in Listing 4.1, a `fold` function defined at line 1 is called in an outer `parallel_for` loop at line 13—16. Inside `fold` is an inner `parallel_for` loop at line 4 that targets GPU. By annotating with the loop attribute `deferred_sync` at line 3, we enable the lazy mode, and the kernels corresponding to this loop are launched asynchronously, without waiting for each other. All these kernels are eventually synchronized after the outer loop at line 18.

A further optimization is to allow programmers to annotate and instruct the compiler to do hierarchical thread mapping when it is beneficial. We expect it to be a minor impact for ZKP, as our current strategy already covers the common cases in ZKP. So we leave it as a future work.

4.3 Adaptive Heterogeneous Memory Management

Traditionally, GPU-accelerated code involves specialized functions to allocate, set, and copy memory between the host and the device. While important for performance, memory management is tedious and error-prone. In our design, the programmer writes a single-source code that can be targeted to either CPU or GPU without special annotation. Thus, our compiler and runtime system must collaborate to efficiently manage memory transfers between the CPU and the GPU.

Existing systems, e.g., Kitsune, employ the Unified Virtual Memory (UVM) [73] mechanism in modern GPUs to simplify data sharing between CPU and GPU. However, UVM requires explicitly prefetching to be performant, otherwise, data are migrated page-by-page with page faults, incurring high overhead. While Kitsune can issue prefetches, they only apply to pointers allocated with its custom allocator. This largely hampers productivity for ZKP algorithm developers, because annotating user code to use a different allocator is error-prone, especially given interaction with other libraries. For example, C++ STL containers must be supplied with extra allocator type arguments, which limits interoperability with their default-allocator counterparts and requires extensive type aliasing to be ergonomic. Furthermore, allocator customization may not even be possible for third-party libraries.

As a design choice, Zera removes the need for manual annotation by globally replacing the compiled program's heap allocator with UVM-backed allocation functions, requiring

minimal source code changes. For C++, this is done by overloading the global `new` and `delete` operators.[1]. However, a naïve realization of the design would suffer from two major performance issues. Below, we present two optimizations to address them.

4.3.1 Pooling Small UVM Allocations

The global use of UVM also applies to small objects that may never be sent to GPU, because they *might* be. A naïve allocator mapping `new` and `delete` one-to-one to UVM allocation calls, while easy to implement, has suboptimal performance on small memory object (e.g., <100 bytes) allocations, because UVM allocators guarantee chunk (e.g., 256-byte in CUDA) alignment, which causes unnecessary slowdown and high internal fragmentation.

One could address the problem by implementing a general-purpose UVM-backed memory allocator, at the cost of significantly complicating the runtime system. Instead, we simplify our design by leveraging this observation: in ZKP, large bulk allocations (for finite field polynomials, matrices, etc.) dominate in total size, while small allocations (for small vectors of pointers, strings for logging, etc.) exist fairly frequently. In response, we use a thread-local arena allocator in page-sized (4KB) chunks to pool the small allocations under 4KB, amortizing the cost of UVM allocation calls. Large allocations still invoke UVM allocations directly. Small object deallocation is a no-op by nature of arena allocation, meaning memory is not freed until the entire arena is deallocated at the end of the program. Fortunately, as large allocations dominate in ZKP, temporarily 'leaking' small objects has a negligible impact on memory usage.

4.3.2 Specializing Bulk Memory Operations

As we do not require in Zera GPU-specific annotation for memory operations, those like `memcpy`, `memmove`, and `memset` may perform poorly with UVM: if the memory region they operate on resides on GPU, the libc implementations of these functions will trigger a stream of page faults to migrate the pages back to host, which is both inefficient (due to context switching) and unnecessary (if the data is to be used on GPU again). Unfortunately, calls to these functions are common even without their presence in the source code. For example, they can arise in C++ from copy constructors, initializers, `std::copy`, `std::fill`, and more.

To remedy this issue, our compiler intercepts the specific intrinsics `llvm.mem{cpy`, `move`, `set}` used for these calls at the LLVM IR level, and redirects them to our runtime late in compilation. Our runtime specializes these calls to call the GPU version of these functions if any argument points to a UVM region and the size is larger than the page size. For `mem{cpy`, `move` and `set}` calls less than page size, our runtime still invokes the CPU counterpart, as single page fault overhead is low and the libc version is faster in the case of no page faults.

Overall, Zera uses an *adaptive* strategy that specializes memory management and operations based on object size and residency across CPU and GPU. We show in Chapter 6 that the two optimizations significantly improve the performance.

[1]Alternatively, one could perform fine-grained analysis of which memory/pointer could be passed to GPU. However, without language-native address space annotation, such analysis is complex and brittle across compile unit boundaries. Hence, we decide to opt for the blanket replacement approach, which is simple, correct, and well-supported by the C++ standard.

Finally, we acknowledge that libraries like Thrust [17] can also facilitate a smooth transition to a GPU-compatible codebase, by providing GPU-side STL-like containers and algorithm library support that operates on them. However, such library solutions are specific to the programming language (C++) and target platform (CUDA), and often do not achieve feature parity with the standard library. In comparison, we provide a language-agnostic solution that combines runtime optimizations and IR code transformation.

Chapter 5

Implementation Detail

5.1 Implementing Zera

We implement Zera by extending both the OpenCilk and Kitsune compiler and runtime codebases. This includes modifying the OpenCilk frontend to support `zera::target` attribute and adding reducer support (Section 4.1) to Kitsune's CUDA lowering. We also extend Kitsune's CUDA lowering code to support nested parallelism and our Collaborative Heterogeneous Parallelism (Section 4.2). As our changes are spread across two compiler codebases, we wrap our compilation workflow (see Fig. 3.1) with a Python script compatible with common compiler options, which can be set as the compiler launcher under CMake for integration with user codebase.

As was outlined in Section 4.1, we implement reducer support by extending Kitsune's CUDA lowering to identify `__cilkrts_reducer_lookup` calls the OpenCilk frontend generates for reducer use, replace them with thread-local views from `alloca`s, and generate a fused tree reduction at the end, followed by an atomic compare-and-swap (CAS) loop that accumulates the block-wide sum to a global memory view. For a reducer larger than what CAS natively supports, we allocate an additional four-byte spinlock next to its global view, which CAS operates on. Allocation, initialization, and copy-back of global views are inserted around the kernel launch call on the host side. Generating nontrivial algorithms, such as tree reduction, via LLVM's `IRBuilder` is a complex endeavor. Alternatively, using LLVM bitcode files, as was done in OpenCilk's lowering [75], could decouple the algorithmic structure from the compiler codebase and would be the preferred solution when extending support to more GPU platforms. However, we found that it was much more challenging to fuse reductions under the bitcode design. We leave a cleaner, bitcode-based implementation of fusion-capable reducer code generation as future work.

The runtime stream management of CHP is implemented by reusing reducer-related facilities in the OpenCilk CPU runtime, as the requirement of acquiring task-local stream and joining streams on task join is very similar to what is required for parallel reduction. In particular, we implement in our runtime a global reducer hyperobject, where:

1. The identity function claims an unused stream with an unused CUDA event from the stream recycle queue, or creates them by invoking the CUDA API if none is available.

2. The reduce/merge function captures outstanding computations in the right stream in the right event with `cuEventRecord`, and schedules the left stream to wait for the right event with `cuStreamWaitEvent`. This is followed by a host callback, which adds the right stream and right event back to the recycle queue for reuse.

In addition, we significantly optimized Kitsune's runtime to use concurrent data structures and reduce lock contention on hot paths, as it was previously not optimized for high CPU parallelism.

5.2 Implementing the Evaluation Test Bench

As elaborated in Chapter 6, we evaluate our system on Shockwave [21], a recent ZKP proof system with a Rust reference implementation. While much of Zera's backend optimization applies to LLVM IR and is thus language-independent, its linguistic extension is implemented on C++ only for now, so we reimplement Shockwave in single-threaded C++ following the Rust codebase. Then, we incrementally parallelize the single-threaded codebase with Zera to CPU and GPU.

CPU parallelization under Zera was straightforward, as we identified many logically parallel regions in our codebase. We replaced logically parallel `for` loops with `cilk_for` and surrounded other logically parallel code regions with `cilk_spawn`. Reductions across these logical parallel regions only require adding `cilk_reducer` type declaration to the reducer variable. We find that for a reasonably organized codebase, parallelization under Zera is almost entirely *additive*, in contrast to library-based solutions which enforce certain codestyles that may obscure the structure of the original algorithm. We also find scalability analysis and race detection tools in the OpenCilk ecosystem very helpful in identifying bottlenecks and reassuring correctness post-parallelization.

As there was no prior GPU-accelerated implementation of Shockwave, we manually parallelized our C++ codebase by incrementally writing custom CUDA kernels to establish a baseline for Zera-GPU. This process surfaced several key challenges of hand-written GPU parallelization:

- *Data transfer and memory management*: Incremental GPU parallelization is often performed at the function level, where function contracts implicitly specify not just data types, but also memory locations. In early stages, functions still accept inputs from CPU memory and produce outputs consumed by CPU code downstream. As a result, explicit CPU-GPU memory copies must be inserted around kernel calls. These copies become redundant as more of the pipeline is offloaded to the GPU, but removing them requires refactoring function signatures. Meanwhile, to preserve portability and enable a CPU-only build, function signatures must remain platform-independent. This tension persists throughout the parallelization process and is difficult to resolve cleanly. One common solution is to introduce platform-dependent type aliases or abstractions that encapsulate data transfer logic. For example, a core type in our codebase is `DensePoly`, which represents polynomial evaluations. To support both CPU and GPU backends, we eventually implemented two accessor functions, `.evals()` and `.evals_gpu()`, which

return CPU- and GPU-resident data, respectively, copying between them on demand. While workable, this pattern is tedious and error-prone.

- *Limited code reuse*: Despite our goal of maximizing code reuse, we found ourselves duplicating and adapting CPU loop bodies directly into GPU kernels. For short parallel loops, this was often the path of least resistance. Creating proper abstractions typically involves outlining the loop body into a standalone function with loop indices and dependencies as parameters — a mechanical but time-consuming process that obscures control flow. Higher-order constructs such as `par_loop(start, end, body)` can help reduce boilerplate, but are less intuitive than native loop syntax and see limited usage in C++.

- *Maintenance burden*: Manually managing data movement and writing high-performance CUDA code significantly increases maintenance overhead. Unit tests must be written to verify that GPU kernels match the behavior of their CPU counterparts. Algorithmic updates often require duplicating changes across CPU and GPU implementations. These burdens slow development and are particularly problematic given the fast-evolving nature of ZKP research.

Our manual GPU parallelization effort added approximately 2,000 lines of CUDA code — including custom kernels and tests — to an existing 7,000-line C++ codebase and took several days to complete. In contrast, parallelizing the same code with Zera-GPU took under an hour and required fewer than 100 lines of additive changes to the CPU-parallelized code. This stark contrast underscores the value of domain-specific compiler automation.

Chapter 6

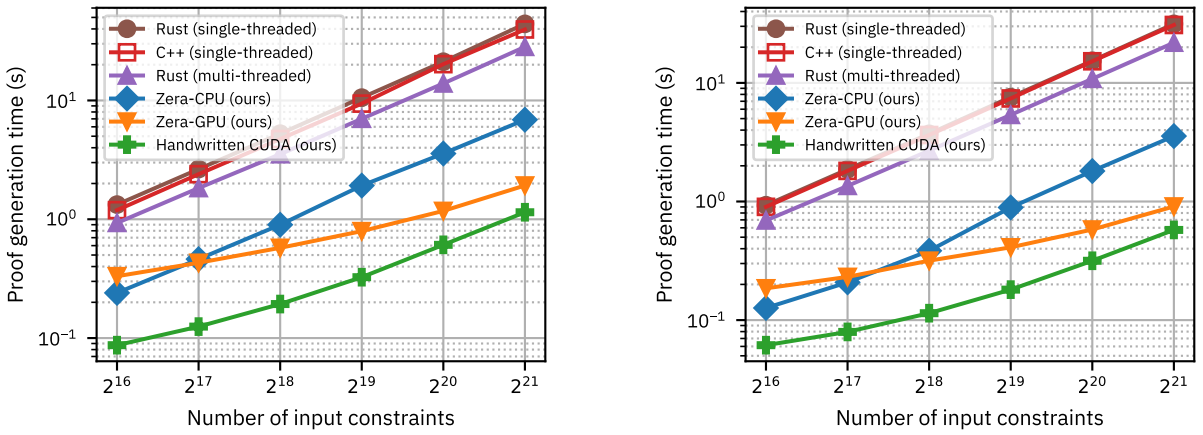
Evaluation

We evaluate the productivity and performance of ZERA by implementing Shockwave [21], a zk-SNARK combining Spartan IOP [53] and Reed-Solomon code. We chose Shockwave because it's recent and representative of the growing hash-based zk-SNARKs. We base our implementation on the BN254 scalar field, consistent with the Rust reference implementation. Table 6.1 gives an overview of the implementations we use. It shows clearly that ZERA largely improves productivity than manually parallelized code.

	Acceleration	**Lines of code**	**Developer effort**
Rust reference	CPU (optionally multi-threaded)	8,100	N.A.
C++	CPU (single-threaded)	7,000	N.A.
ZERA-CPU	CPU (multi-threaded with Cilk)	<100 *	$\sim$ 1 hour
ZERA-GPU	GPU	<200 *	$\sim$ 0.5 hour
Handwritten CUDA	GPU	2,000 *	$\sim$ 1 week

Table 6.1: Various Shockwave implementations and their developer efforts needed. * means Δ from serial C++. ZERA-CPU needs less than 100 lines of additional code over serial C++. ZERA-GPU adds about another 100 lines of code over ZERA-CPU. Hand-written CUDA requires 10× of code and 40× programming time than ZERA GPU.

- *Rust reference implementation*: We take Shockwave's reference implementation from Github with 8,100 lines of code. This reference implementation can optionally be built with Rayon [92] to exploit CPU multithreading.
- *C++ baseline (single-threaded)*: We re-implement Shockwave in 7,000 lines of C++, which largely mirrors the Rust baseline in terms of structure and optimizations. Additionally, the C++ codebase includes our optimizations to reduce redundant copies and improve cache locality.
- *ZERA-CPU*: we fully parallelize the C++ baseline using OpenCilk [75] by replacing 49 `for` loops with `cilk_for` and inserting 28 `cilk_spawn`s. Less than 100 lines were changed, taking less than one programmer-hour.



(a) on the `low-end` machine

(b) on the `high-end` machine

Figure 6.1: End-to-end performance comparison. Each data point is measured as the mean running time of 10 runs.

- *Zera-GPU*: based on the OpenCilk baseline, we parallelize the codebase with Zera by attaching `target("gpu")` attribute to 30 `cilk_for` loops. Other `cilk_for` loops are outer loops and they remain parallelized on CPU. Conversion from Zera-CPU into Zera-GPU took less than 30 minutes.
- *Handwritten CUDA (parallel GPU)*: We hand-write CUDA kernels to accelerate the parallel loops in the Zera CPU baseline. Despite code-sharing efforts, an extra 2,000 lines of CUDA were added to the codebase. It took around a week of developer effort to develop and test the additional CUDA code.

As proof generation is typically a bottleneck at the client side, we use representative client computing platforms for our evaluation. We run our benchmark on two platforms to show the impact of different hardware specifications.

- `low-end`: a laptop with an 8-core AMD Ryzen CPU (hyperthreading on) and an Nvidia RTX 3080 Mobile GPU, representing consumer hardware.
- `high-end`: a desktop PC with a 20-core Intel CPU and an Nvidia RTX 5080 GPU representing workstations.

6.1 End-to-end Performance Comparison

We measured the end-to-end proof generation time of all implementations on input R1CS instances of various sizes. We used the same input synthetic R1CS instances for all implementations as the original Spartan [53] and Shockwave [21] papers. We exclude the running time of setup and circuit commitment, as they only need to be run once.

Figure 6.1 compares the end-to-end performance of various implementations on the two platforms. We make the following observations:

1. *CPU acceleration was underexploited*. While Rust baseline uses Rayon [92], it was only used on selected bottlenecks, leading to a modest speedup. Starting with comparable single-threaded performance, our OpenCilk-based Zera-CPU implementation offers a much better speedup (up to 6.13×) compared to the multithreaded Rust baseline, especially in workstation scenarios with powerful multi-core CPUs.

2. *GPU acceleration has even higher potential*. On both machines tested, the handwritten CUDA baseline achieves around 6× speedup over Zera-CPU. More importantly, for the largest input, the speedup goes up to 24.7× over the Rust multicore baseline on the `low-end` platform and 37.9× on the `high-end` platform. However, note that this speedup comes at a cost of significant extra developer effort and maintenance burden, as this GPU code is manually written.

3. *With significantly reduced programming effort, Zera-GPU delivers competitive speedups with the manual GPU implementation, especially on larger inputs*. Comparing the orange and green curves in Fig. 6.1, we can tell that our compiler-based approach for automatic GPU parallelization does incur some overhead — mostly synchronization and UVM page fault, which is expected. This overhead is reflected more clearly on the smaller inputs, Zera-GPU can be slower than Zera-CPU and the latter is more favorable. Despite the overhead, Zera-GPU is constantly much faster than Rust multicore baseline on all input sizes tested, with speedups from 2.8× to 24×. As input scales up, the overhead is amortized by increased computation, and the gap shrinks significantly. Our Zera GPU implementation performs within 1.68× (`low-end`) and 1.58× (`high-end`) the running time of the handwritten CUDA implementation on the largest input. Moreover, Zera-GPU is favorable over Zera-CPU for large instances on both machines tested. We note that the performance gap narrows even more on the workstation as its newer GPU architecture and faster PCIe lanes reduce synchronization overhead. This is exciting because it implies that our compiler-based solution would be even more attractive on future generations of GPUs.

6.2 Detailed Performance Comparison

We conduct zoomed-in performance comparison of Zera-GPU's auto parallelization against hand-optimized CUDA code for three small sections of the algorithm: *Batch NTT*, *Row combination* and *Hyrax/cubic sumcheck*. The results are presented in Fig. 6.2.

6.2.1 Batch NTT

In Shockwave's polynomial commitment scheme, NTT is performed for each row of a matrix of modular integers. This computation demonstrated nested parallelism as NTT itself is highly parallelizable. In this case, Zera adopts a different parallelization strategy from hand-optimized CUDA. Our hand-optimized CUDA implementation maps rows to thread blocks and parallelizes NTT within a single thread block using shared memory tiling and warp shuffles. Thus, it only requires a single kernel launch. In comparison, Zera generates device-wide kernels for one round of NTT butterfly, a simpler approach that respects the

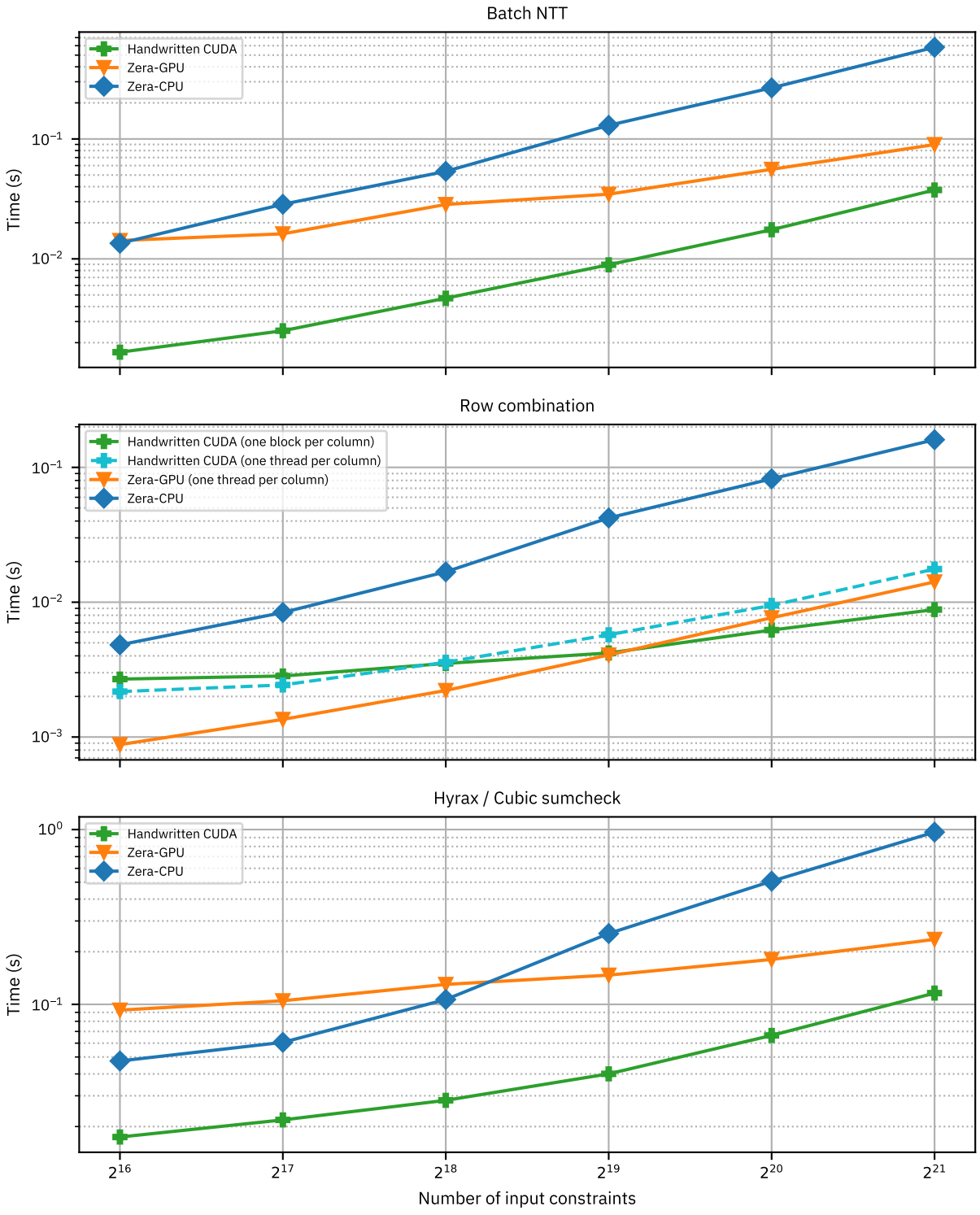


Figure 6.2: Zoomed-in GPU kernel performance comparison. Benchmarked on the `high-end` machine.

serial C++ code but requires $R \log_2 C$ kernel launches for a $R \times C$ matrix, where R parallel CPU tasks each sequentially launch $\log_2 C$ rounds of butterflies over different streams, as explained in Section 4.2.

As seen in Fig. 6.2, Zera's simpler strategy causes the batch NTT operation to be 2.4–3.2× slower than a hand-optimized CUDA kernel on the two largest input sizes (8.5× slower on the smallest input), with the gap attributable to synchronization and scheduling overhead. Nevertheless, the handwritten CUDA kernel requires GPU performance engineering experience and is significantly more time-consuming to write and debug.

6.2.2 Row Combination

This kernel computes a linear combination of rows in a flat (e.g., 512×65536) matrix, preprocessed to be in column-major order. In Zera, we mark the loop over column indices parallel, mapping every column to a CUDA thread in which the sum is computed sequentially. In hand-optimized CUDA, we explored two approaches: one being the same as Zera parallelization, and the other one mapping every column to a thread block where the sum over rows is computed with CUB `BlockReduce`. The second approach is more amenable to global memory coalescing and performs better at scale. We compare the performance of both approaches to Zera.

As shown in Fig. 6.2, Zera's automatic parallelization achieves performance comparable to the handwritten CUDA variants. For the CUDA kernel with the same parallelization strategy (one thread per column), Zera achieves a better running time by using better launch parameters — handwritten CUDA code hardcodes 256 threads per block whereas Zera dynamically sets block size with `cudaOccupancyMaxPotentialBlockSize` heuristic to maximize occupancy. Compared with the better-performing one-block-per-column + block reduction CUDA kernel, Zera's generated kernel is only 1.6× slower on the largest input tested.

6.2.3 Hyrax/Cubic Sumcheck

Shockwave uses Spartan IOP which uses Hyrax [81] as a sub-protocol. We are particularly interested in Hyrax because it involves fine-grained parallelism with many synchronization steps: Given an input circuit with N constraints, Hyrax uses $O(\log_2 N)$ rounds of sumcheck protocol, each requiring $O(\log_2 N)$ rounds of reduction and mapping (Algorithm 1). Altogether, Hyrax requires $O(\log_2^2 N)$ rounds (concretely, 256–441 in our benchmarks) of kernel launches with synchronization in between as required by the Fiat-Shamir transform, making it an interesting sample that amplifies synchronization overhead in its implementations.

As seen in Fig. 6.2, a considerable performance gap exists between Zera GPU and handwritten CUDA due to Zera's synchronization and UVM pagefault overhead on small inputs. However, as inputs get larger, these overhead are amortized by computation and our implementation is within 2× the running time of hand-optimized CUDA.

We note that the performance gap is lower when considering end-to-end performance. Zera-GPU achieved 1.58× performance gap on the `high-end` platform despite some of its kernels, shown above, having >1.6× performance gap from handwritten CUDA implementation due to different parallelization strategy and synchronization overhead. This is because

(1) end-to-end running time includes memory transfer, which both implementations spent comparable time on, and (2) as shown in the row combination kernel case, Zera-GPU can achieve equal or better performance when the handwritten CUDA kernel adopts the same parallelization strategy due to better launch parameters.

6.2.4 Multicore Scalability

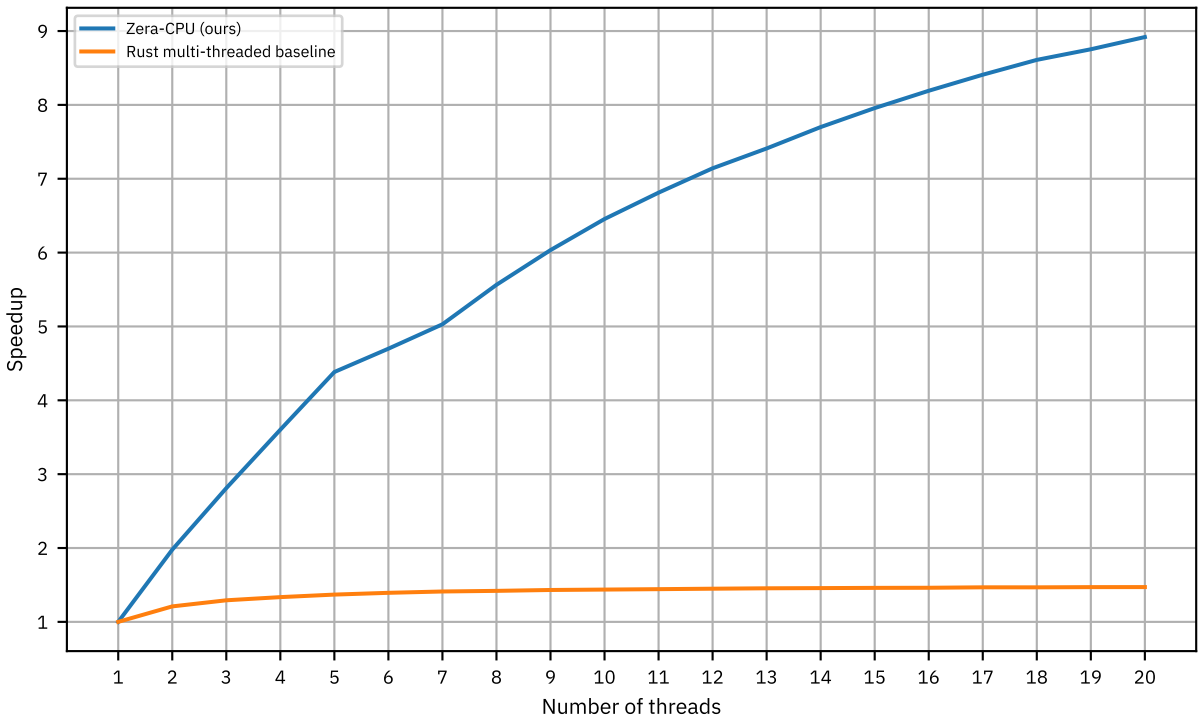


Figure 6.3: Performance scalability on multicore CPU, as measured on the `high-end` machine.

We also evaluate the performance scalability on the multicore CPU. Fig. 6.3 shows how the performance speedups over single-thread changes as we increase the number of threads. We observe that Zera-CPU (blue curve) scales much better than the Rust baseline (orange curve). The good scalability of Zera-CPU is expected as our framework provides a single-source programming solution with intuitive syntactic extensions, allowing programmers to parallelize their codebase with little effort and fully enjoy OpenCilk's advantages, e.g., its IR-level optimizations and provably close to optimal runtime scheduler. In comparison, the Rust baseline is multithreaded with Rayon. While Rayon also uses a work-stealing scheduler, it as a library solution incurs higher overhead user friction. The Rust multithreaded baseline is partially parallelized (only its polynomial commitment scheme), which we hypothesize is due to the inconvenience and user friction of applying Rayon.

6.3 Impact of Compiler Optimizations

Here, we evaluate the performance impact of our compiler's optimizations, including memory optimizations, CPU-GPU collaborative parallelism, and reduction fusion.

6.3.1 Memory Optimizations

We compared the end-to-end running time of our Zera-GPU implementation with and without the two memory optimizations we proposed in Section 4.3. As shown in Fig. 6.4, ablation of both optimizations leads to a 50%–80% slowdown in end-to-end running time. Of the two optimizations, the relative effect of our small allocation optimization diminishes as input size grows, because increased compute intensity generally amortizes memory allocation overhead given that the number of small allocations stays relatively constant; In contrast, our specialization of common bulk memory operations such as `memset` and `memcpy` becomes more impactful on larger inputs due to the increased inefficiency of page-fault driven migration of larger memory regions in the absence of these specializations. Overall, the two optimizations are mostly orthogonal, and their effects compound.

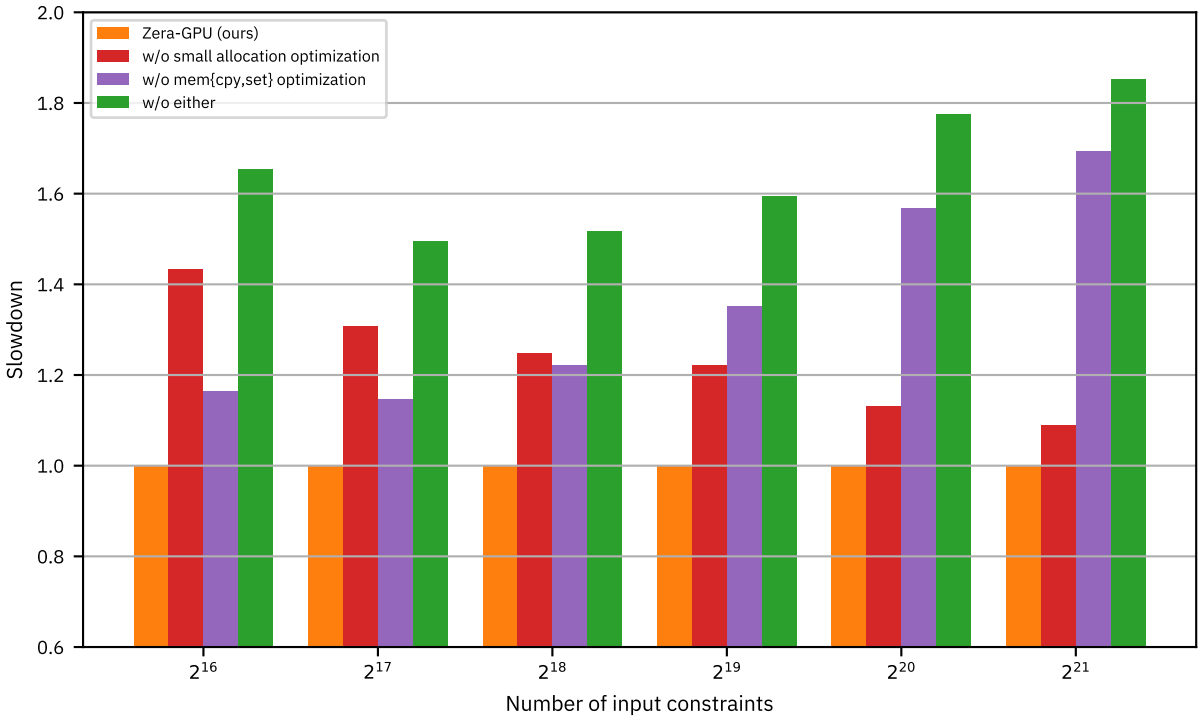


Figure 6.4: Effect of memory optimizations on end-to-end proof time of Zera-GPU programs, on the `high-end` machine.

6.3.2 CPU-GPU Collaborative Parallelism

As discussed in Section 4.2, our system handles nested parallelism by lowering outer loops to parallel CPU tasks, which launch inner loop kernels simultaneously for high GPU utilization. Our approach implies that the degree of CPU parallelism (i.e., the number of CPU workers) should affect the performance of our GPU-accelerated code. We investigate this effect by varying the CPU worker counts. As shown in Fig. 6.5, performance improved as the number of CPU workers increased from 1 to 8, achieving nearly 2× speedup for small input instances. This confirms that our system partially relies on simultaneous kernel launches to effectively utilize the GPU. As the input size increases, the speedup from having more workers diminishes, as larger grid size lead to better utilization even if kernels are launched serially.

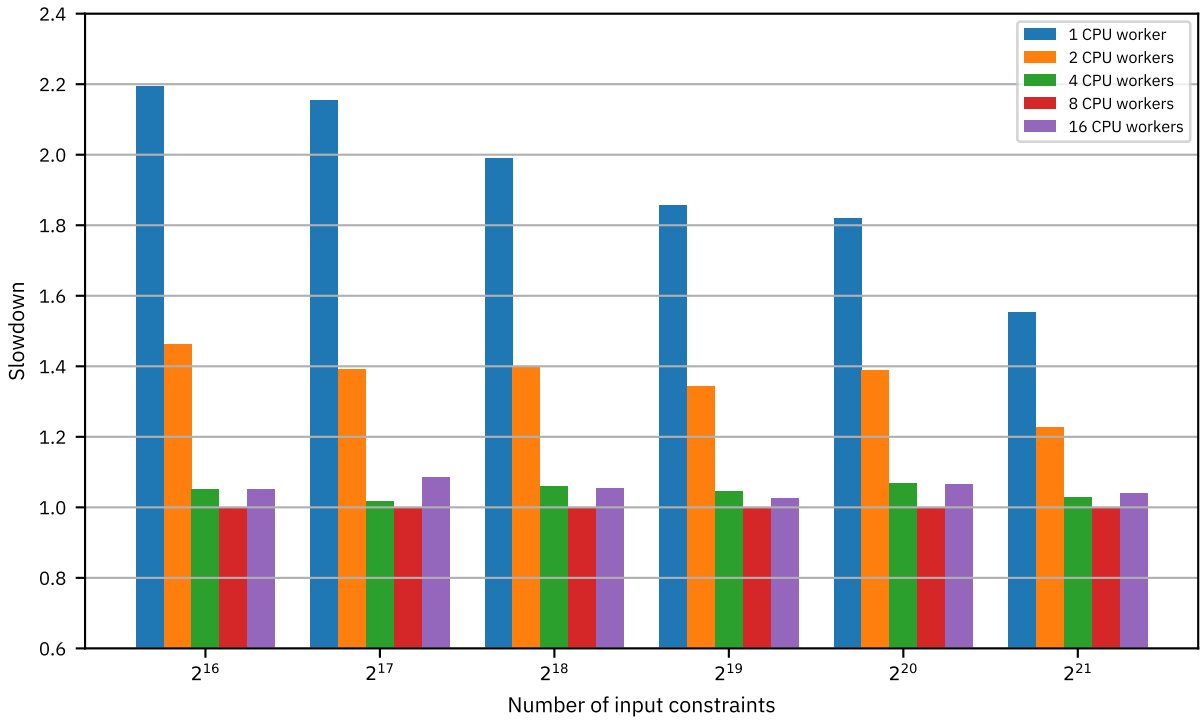


Figure 6.5: Effect of CPU parallelism on end-to-end proof time of Zera-GPU programs, on the `high-end` machine.

However, blindly adding CPU workers does not always increase the performance of our GPU-accelerated code. We observe performance regression when increasing the number of workers from 8 to 16. From Nsight Systems CUDA API and system call traces, we hypothesize that this is due to increased lock contention within the CUDA API itself. Overall, 8 CPU workers present an empirical sweet spot that maximizes our system's performance on both machines tested and was used in other evaluations.

Input size	Deferred sync	Eager sync	Speedup	p-value
Hyrax/Sumcheck				
2^{16}	91.2 ms	97.2 ms	6.15%	3.6×10^{-6}*
2^{17}	101.5 ms	110.3 ms	7.99%	1.0×10^{-5}*
2^{18}	120.6 ms	135.6 ms	11.04%	1.6×10^{-8}*
2^{19}	146.7 ms	160.7 ms	8.71%	7.8×10^{-9}*
2^{20}	180.5 ms	193.2 ms	6.55%	8.1×10^{-10}*
2^{21}	234.3 ms	253.9 ms	7.71%	3.8×10^{-10}*
Batch NTT				
2^{16}	14.8 ms	14.1 ms	-4.73%	6.8×10^{-1}
2^{17}	18.7 ms	16.6 ms	-12.61%	2.4×10^{-3}*
2^{18}	37.1 ms	28.4 ms	-30.36%	5.5×10^{-11}*
2^{19}	42.7 ms	34.8 ms	-22.81%	4.1×10^{-11}*
2^{20}	66.1 ms	56.0 ms	-18.21%	8.2×10^{-11}*
2^{21}	103.1 ms	89.4 ms	-15.32%	3.0×10^{-11}*
Merkle tree construction				
2^{16}	0.6 ms	0.7 ms	17.16%	3.0×10^{-11}*
2^{17}	0.9 ms	1.1 ms	16.20%	2.4×10^{-10}*
2^{18}	2.5 ms	1.2 ms	-111.54%	1.7×10^{-5}*
2^{19}	3.1 ms	2.0 ms	-55.27%	7.7×10^{-2}
2^{20}	3.7 ms	2.6 ms	-40.89%	9.5×10^{-7}*
2^{21}	5.1 ms	4.5 ms	-11.77%	5.1×10^{-4}*

Table 6.2: Effect of `deferred_sync` under CHP (Section 4.2) on synchronization overhead. Median running time reported over 30 runs for every input size on the `high-end` machine; p-values from Mann-Whitney test.

6.3.3 Deferred Synchronization

Our Collaborative Heterogeneous Parallelism (Section 4.2) enables a kernel to be launched asynchronously in one task, but its synchronization is deferred until after joining with other tasks. We expect this optimization to reduce synchronization overhead and evaluate its effectiveness on three parts of our Shockwave implementation. The result is presented in Table 6.2.

1. Hyrax/sumcheck involves folding N (10–20) vectors in parallel at the end of each round. Zera-GPU parallelizes the fold as a GPU kernel and the parallel loop over vectors as CPU tasks (see Listing 4.1). Deferred sync reduces N syncs in the loop body down to one after the join, leading to around 8% speedup in running time.

2. Batched NTT involves applying NTT independently over rows of an $R \times C$ matrix ($R \in [16, 512]$, $C \in [4096, 65536]$). Under Zera-GPU, R parallel CPU tasks each launch $\log_2 C$ NTT butterfly kernels sequentially. Deferred sync reduces $R \log_2 C$ syncs down to one. However, we found that it leads to performance regression of 20–30%.

3. Merkle tree construction accepts a vector of N leaf hashes and uses $\log_2 N$ many

kernel launches to build the Merkle tree layer-by-layer. Deferred sync reduces $\log_2 N$ syncs after every launch to one at the end. Interestingly, we observed that it leads to performance improvements for small instances but to regression on large ones.

As seen in Table 6.2, deferred synchronization does not always improve performance. We hypothesize that it only improves performance if the number of in-flight deferred-synchronized kernels is small, as in the case of Hyrax and a small Merkle tree. Too many kernels launched without synchronization might create a backlog on CUDA's scheduler, raising internal scheduling overhead that outweighs the benefit of reduced synchronization. This happened for batched NTT and a large Merkle tree. The exact threshold depends on GPU architecture and kernel complexity. Hence, deferred synchronization is offered as an advanced feature that should be empirically evaluated on a case-by-case basis.

6.3.4 Fusing Reduction

As discussed in Section 4.1, our compiler generates a fused tree reduction at the end of a kernel: all reducers are reduced together at each shuffle or aggregation step, amortizing warp shuffle and shared memory communication overhead across reducers. To evaluate this approach against the naive approach of one tree reduction per reducer, we compare the running time of the Hyrax/sumcheck portion of our Shockwave implementation. We choose Hyrax because it's one of the most reducer-heavy portions of Shockwave, requiring launching hundreds of kernels sequentially, each involving three reducers. We report the median running time of Hyrax on a range of input sizes compiled with fused and separate tree reductions in Table 6.3 as well as p-values obtained from Mann-Whitney test. We observe that generating fused reduction leads to a 2% speedup in Hyrax running time. The speedup is moderate as the running time includes noise and much synchronization overhead. However, it's statistically significant for large inputs as the amount of reduction work increases and demonstrates the effect of our optimization.

Input size	**Unfused**	**Fused**	**Speedup**	**p-value**
2^{16}	91.2 ms	92.3 ms	1.19%	1.7×10^{-1}
2^{17}	101.5 ms	104.1 ms	2.53%	5.7×10^{-2}
2^{18}	120.6 ms	124.1 ms	2.80%	1.6×10^{-2}
2^{19}	146.7 ms	150.4 ms	2.49%	4.4×10^{-6}*
2^{20}	180.5 ms	184.1 ms	1.95%	5.6×10^{-4}*
2^{21}	234.3 ms	239.8 ms	2.31%	1.1×10^{-3}*

Table 6.3: Effect of generating fused tree reduction for all reducers in a kernel over separate tree reductions on Hyrax running time. Median running time reported over 30 runs for every input size on the `high-end` machine; p-values from Mann-Whitney test.

Chapter 7

Conclusion and Future Work

In this thesis, we build a compiler framework, Zera, for ZKP algorithm designers to fast prototype new algorithms with high performance on modern hardware, e.g., multicore CPUs and GPUs. The key challenge in building such a domain-specific optimizing compiler lies in the mismatch of programming models across different hardware architectures. We therefore propose a pattern-based approach to preferentially support dominant parallel patterns that we identify in ZKP algorithms. This allows a single-source programming interface across different hardware architectures, without compromising performance. Subsequently, we propose a series of compiler optimizations to address architecture-specific performance bottlenecks in the generated computation. Experiments show that our framework significantly simplify ZKP programming, while retaining comparable performance than the hand-written implementation, which takes orders of magnitude more time and effort to develop.

While Zera demonstrated the possibility of productive high-performance ZKP prototyping, it was developed in a limited time frame, and future extensions are desirable. We conclude this thesis by recognizing some present limitations in our design or implementation and suggesting directions for future work.

7.1 Support for More Languages

At present, Zera only has a C++ frontend. While many high-performance ZKP algorithms are developed in C++ (e.g., libsnark [93]), a considerable proportion of the modern cryptography ecosystem is built on the Rust programming language (e.g., arkworks [94]), as the memory safety guarantee by Rust aligns well with general computer security research. For Rust developers, using Zera as is requires rewriting the codebase to C++ (as we did for Shockwave), which defeats the very productivity benefit Zera offers.

We remark that much of Zera is language-agnostic. Our GPU-oriented optimizations and lowering code operate on LLVM IR, which the Rust compiler also targets. Our adaptive memory management requires the ability to globally replace the language's heap allocation and deallocation functions, which Rust also supports with the `#[global_allocator]` attribute; Prior work has also demonstrated extension of the Rust language to express fork-join parallelism natively and to target Tapir IR [90]. There is no theoretical barrier to adding a Rust frontend to Zera, which would allow evaluation of Zera on more codebases.

7.2 Support for More Parallel Patterns

While Zera already supports most common parallel patterns used by ZKP algorithms such as `map`, `zip`, and `reduce`, support for more parallel patterns could cover more corner cases and enable more use cases. In particular, Zera does not currently support parallel `scan` or `sort`. We here specifically discuss the possibility of supporting parallel `scan`, as it would, by extension, enable support for parallel `filter` (stream compaction), both of which could be handy to developers.

Zera's current implementation builds on the OpenCilk ecosystem, which has yet to provide a linguistic primitive to express parallel "scanners." Scanner support from the OpenCilk upstream is a prerequisite for scanner support in our current Zera implementation. Fortunately, as of this writing, the OpenCilk developers have started experimenting with parallel scan support by extending reducer [77]: the prevailing is to enable fork-join parallelism within reduction operators and make scanners a special form of reducers with Blelloch's algorithm [95] embedded in the reduction operation. On the GPU side, we also prototyped primitives for lowering parallel scan to the GPU with the decoupled lookback algorithm [96]. The remaining work involves designing the front-end linguistic extension that unifies the two approaches, and we are hopeful that this can be completed shortly after this thesis.

7.3 Support for More Accelerator Targets

While Zera's design is not bound to any specific GPU vendor, we have only implemented support for lowering to Nvidia GPUs through PTX as it's the most mature GPGPU platform. Being able to lower to other GPU vendors or GPGPU targets, such as AMD, Intel, OpenCL, or even WebGPU will allow Zera to practically demonstrate its performance portability and present a concrete edge over vendor-specific libraries such as Thrust [17] or CUB [18].

As Zera's current implementation builds on Kitsune. The most hopeful direction so far is to polish Kitsune's AMD GPU target (`HIPABI.cpp`), which leverages LLVM's AMDGPU backend. However, the Kitsune team has found maintaining that target more challenging due to scarce documentation and API instability. On our side, our changes to Kitsune (reduction support and runtime optimizations) currently contain much Nvidia-specific code, and we'll need to explore better abstraction and modularization as more targets are added.

7.4 Improvement in Memory Management

In Section 4.3, we presented a pooled memory allocation strategy that speeds up UVM allocation for small objects. However, as we use an arena allocator, deallocations of these small objects are deferred until the end of the program when the arena is destroyed. While such leaks have negligible impact on memory usage for typical ZKP algorithms (Section 4.3), a general-purpose UVM-backed memory allocator is still desirable.

A natural idea is to modify existing high-performance general-purpose memory allocators such as mimalloc [97], replacing `mmap` with `cudaMallocManaged` and `munmap` with `cudaFree`.

Upon close scrutiny, however, this idea does not stand: A key insight of modern high-performance memory allocators is the colocation of allocation metadata and the allocation itself. On CPU, such colocation is cache-friendly due to better spatial locality. In UVM context, if the allocation and its metadata share the same page, the latter will be prefetched to GPU along with the former due to page-level false-sharing. Subsequent access to the metadata on CPU will then cause a page fault. More fundamentally, CPU-oriented allocators assume that allocation metadata and allocations are accessed by the same device, which is not true under UVM. Tight allocation-metadata colocation can be detrimental to performance in UVM context, and a general UVM-backed allocator needs a different design.

In addition to generalizing heap memory management under UVM, another interesting possibility is moving the stack to UVM. Currently, Zera errors at run time when a stack variable is used in the kernel because the stack is not accessible from the GPU side. Using Zera as is thus requires occasional hand-holding to avoid unintended stack capture. A practical example is manually hoisting out the use of `std::vector` in a parallel loop as a `std::span` and using the `std::span` in the loop instead — this nudges LLVM to only capture the heap data pointer instead of the object on stack containing it, although this is not always necessary. These workarounds are no longer necessary if we allocate the stack on UVM in the first place. However, doing so naïvely would have a catastrophic performance impact because the stack is tightly packed and vulnerable to page-level UVM false sharing. A potential solution would be to use an instrumentation pass like LLVM SafeStack [98] to identify stack variables that can be captured by GPU kernels and allocate them on a specific UVM memory region.

7.5 Evaluation with More VC Technologies

While titled "Efficient Verifiable Computation Made Easy," this thesis focuses on ZKP, which we believe is the most practically adopted form of verifiable computation. In theory, we predict that Zera can be easily adapted to accelerate other VC technologies such as Homomorphic Encryption, because they use similar cryptographic constructs (HE also operates on finite field/big modular integers). Concrete evaluations of Zera on these non-ZKP technologies will be able to validate our prediction and probe the boundary of our system's capabilities.

7.6 Extending Zera to Other Application Domains

Broadly, our research aspires to advance the automation of Accelerated Computing. While such automation remains a formidable challenge in general-purpose computing, our strategy is to approach it through domain specialization — starting with privacy-preserving computation. Mature domains like machine learning and tensor algebra already boast robust compiler and runtime ecosystems (e.g., PyTorch [70], TensorFlow XLA [69], TACO [67], TVM [68], MLIR [99], GraphIt [100]). In contrast, many emerging domains still rely heavily on manual optimization, hardware-specific tuning, or lack a unified compiler infrastructure.

Emerging domains that still heavily rely on manual optimization or ad hoc compiler support include Real-Time SLAM, Edge AI/ML, Quantum Simulation, Digital Twins, and

Physics-Informed Neural Networks (PINNs). For example, vector databases are rapidly gaining prominence due to embedding-based retrieval (e.g., in LLM-powered RAG systems). Despite appearing "application-layer," they pose unresolved systems and compiler-level challenges. Popular systems such as FAISS [101], ScaNN [102, 103], Milvus [104], Weaviate [105], Qdrant [106], and Pinecone [107] offer some CPU/GPU acceleration (e.g., FAISS via CUDA), but much of the tuning (such as quantization choices and graph parameters) remains manual. Index creation and query execution lack compiler-driven cost models or auto-tuning, opening up interesting research opportunities.

Our next step would be to investigate more emerging domains, e.g., vector database, and extend ZERA to those domains. Ultimately, our vision is to build a hybrid solution/framework that brings together compiler, machine learning (e.g., LLM) and human insight (domain knowledge specialization) to support efficient, scalable, and developer-friendly Accelerated Computing in a wide range of applications. We see this as a critical enabler of continued computing innovation in the post-Moore's Law era.

Appendix A

A Brief Summary of Modern ZKP Constructions

Modern ZKP constructions consist of four components: arithmetization, IOP, commitment schemes, and Fiat-Shamir transform.

1. *Arithmetization* helps formulate the computation in a representation with good mathematical properties. This typically involves tracing and expressing the computation as an arithmetic circuit or a constraint system under a finite field. This is done once for every circuit/program and does not count toward proof generation time. Popular arithmetization includes R1CS [108] and PLONK-style constraint systems [83]. Computationally, the goal to have good mathematical properties often aligns with *regularity* in data structure, e.g., R1CS uses three (sparse) finite field matrices.

2. Under an arithmetization, an *interactive oracle proof* (IOP) protocol reduces proving to the verifier the result of the computation under an input to allowing the verifier a small number of queries ("oracle") to functions that the prover holds. These functions are usually evaluations of polynomials or structured vectors over a finite field, enabling succinct verification through algebraic checks. Notable IOPs include Spartan [53] and HyperPlonk [51].

3. IOP oracle queries are reliably realized with *commitment schemes* that enable the prover to commit to the functions before receiving verifier queries and convince the verifier of claimed values of the oracles with little communication. Advance commitments prevent the prover from adapting responses based on the verifier's queries. Notable commitment schemes include KZG [89], FRI [88], and Ligero [50]. As the committed objects (e.g., polynomials) are highly regular, commitment schemes boast great data parallelism. E.g., KZG uses Multi-Scalar Multiplication (MSM), while FRI and Ligero use Number Theoretic Transform (NTT), both proven to be highly parallelizable.

4. To make the above interactive proof protocol non-interactive, the *Fiat-Shamir transform* [109] mechanically replaces the randomness with which the verifier generates challenges with a deterministic RNGs seeded with the cryptographic hash of current proof transcript. This enables the prover to generate the entire proof without an actual

verifier. Computationally, Fiat-Shamir introduces sequential dependencies across proof stages since each challenge is derived from the transcript so far. This creates synchronization points in a parallel implementation. Consequently, ZKP requires fine-grained parallelization beyond naïve multi-threading.

References

[1] D. Hopwood, S. Bowe, T. Hornby, N. Wilcox, et al. "Zcash protocol specification". In: *GitHub: San Francisco, CA, USA* 4.220 (2016), p. 32.

[2] G. Kappos, H. Yousaf, M. Maller, and S. Meiklejohn. "An empirical analysis of anonymity in zcash". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 463–477.

[3] The ZPrize Competition. *https://github.com/z-prize*. 2024. URL: https://www.zprize.io/.

[4] G. Botrel and Y. El Housni. "EdMSM: Multi-Scalar-Multiplication for recursive SNARKs and more". In: (2022).

[5] W. Ma, Q. Xiong, X. Shi, X. Ma, H. Jin, H. Kuang, M. Gao, Y. Zhang, H. Shen, and W. Hu. "GZKP: A GPU Accelerated Zero-Knowledge Proof System". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 340–353. ISBN: 9781450399166. DOI: 10.1145/3575693.3575711. URL: https://doi.org/10.1145/3575693.3575711.

[6] C. Wang and M. Gao. "UniZK: Accelerating Zero-Knowledge Proof with Unified Hardware and Flexible Kernel Mapping". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 1101–1117. ISBN: 9798400706981. DOI: 10.1145/3669940.3707228. URL: https://doi.org/10.1145/3669940.3707228.

[7] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun. "PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 416–428. DOI: 10.1109/ISCA52012.2021.00040.

[8] P. Qiu, G. Wu, T. Chu, C. Wei, R. Luo, Y. Yan, W. Wang, and H. Zhang. "MSMAC: Accelerating Multi-Scalar Multiplication for Zero-Knowledge Proof". In: *Proceedings of the 61st ACM/IEEE Design Automation Conference*. DAC '24. San Francisco, CA, USA: Association for Computing Machinery, 2024. ISBN: 9798400706011. DOI: 10.1145/3649329.3655672. URL: https://doi.org/10.1145/3649329.3655672.

[9] T. Lu, Y. Chen, Z. Wang, X. Wang, W. Chen, and J. Zhang. "BatchZK: A Fully Pipelined GPU-Accelerated System for Batch Generation of Zero-Knowledge Proofs". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 1101–1117. ISBN: 9798400706981.

[10] Z. Ji, J. Zhao, P. Gao, X. Yin, and L. Ju. "Accelerating Number Theoretic Transform with Multi-GPU Systems for Efficient Zero Knowledge Proof". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 1–14. ISBN: 9798400706981. DOI: 10.1145/3669940.3707241. URL: https://doi.org/10.1145/3669940.3707241.

[11] Z. Ji, J. Zhao, Z. Zhang, J. Xu, S. Yan, and L. Ju. "A Compiler-Like Framework for Optimizing Cryptographic Big Integer Multiplication on GPUs". In: *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2024, pp. 380–392.

[12] B. Feng, Z. Wang, Y. Wang, S. Yang, and Y. Ding. "ZENO: A Type-based Optimization Framework for Zero Knowledge Neural Network Inference". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 450–464. ISBN: 9798400703720. DOI: 10.1145/3617232.3624852. URL: https://doi.org/10.1145/3617232.3624852.

[13] Z. Ji, Z. Zhang, J. Xu, and L. Ju. "Accelerating Multi-Scalar Multiplication for Efficient Zero Knowledge Proofs with Multi-GPU Systems". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 57–70. ISBN: 9798400703867. DOI: 10.1145/3620666.3651364. URL: https://doi.org/10.1145/3620666.3651364.

[14] A. Ray, B. Devlin, F. Y. Quah, and R. Yesantharao. "Hardcaml MSM: A High-Performance Split CPU-FPGA Multi-Scalar Multiplication Engine". In: *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '24. Monterey, CA, USA: Association for Computing Machinery, 2024, pp. 33–39. ISBN: 9798400704185. DOI: 10.1145/3626202.3637577. URL: https://doi.org/10.1145/3626202.3637577.

[15] T. Liu, X. Xie, and Y. Zhang. "zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2968–2985. ISBN: 9781450384544. DOI: 10.1145/3460120.3485379. URL: https://doi.org/10.1145/3460120.3485379.

[16] N. Ni and Y. Zhu. "Enabling zero knowledge proof by accelerating zk-SNARK kernels on GPU". In: *J. Parallel Distrib. Comput.* 173.C (Mar. 2023), pp. 20–31. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2022.10.009. URL: https://doi.org/10.1016/j.jpdc.2022.10.009.

[17] Nvidia. *Thrust: The C++ Parallel Algorithms Library*. URL: https://nvidia.github.io/cccl/thrust/.

[18] Nvidia. *CUB 3.1 documentation*. URL: https://nvidia.github.io/cccl/cub/.

[19] Los Alamos National Laboratory. *Kitsune: LANL LLVM Fork*. URL: https://github.com/lanl/kitsune.

[20] T. B. Schardl and I.-T. A. Lee. "OpenCilk: A modular and extensible software infrastructure for fast task-parallel code". In: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 2023, pp. 189–203.

[21] A. Golovnev, J. Lee, S. Setty, J. Thaler, and R. S. Wahby. *Brakedown: Linear-time and field-agnostic SNARKs for R1CS*. Cryptology ePrint Archive, Paper 2021/1043. 2021. URL: https://eprint.iacr.org/2021/1043.

[22] S. Arora and S. Safra. "Probabilistic checking of proofs; A new characterization of NP". In: *33rd Annual Symposium on Foundations of Computer Science, FOCS 1992*. IEEE Computer Society. 1992, pp. 2–13.

[23] R. Lavin, X. Liu, H. Mohanty, L. Norman, G. Zaarour, and B. Krishnamachari. *A Survey on the Applications of Zero-Knowledge Proofs*. 2024. arXiv: 2408.00243 [cs.CR]. URL: https://arxiv.org/abs/2408.00243.

[24] Y. Xia, X. Yu, M. Butrovich, A. Pavlo, and S. Devadas. "Litmus: Towards a practical database management system with verifiable acid properties and transaction correctness". In: *Proceedings of the 2022 international conference on management of data*. 2022, pp. 1478–1492.

[25] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. "vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 863–880.

[26] J. Zhang, Z. Fang, Y. Zhang, and D. Song. "Zero knowledge proofs for decision tree predictions and accuracy". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 2039–2053.

[27] T. Xie, T. Lu, Z. Fang, S. Wang, Z. Zhang, Y. Jia, D. Song, and J. Zhang. *zkPyTorch: A Hierarchical Optimized Compiler for Zero-Knowledge Machine Learning*. Cryptology ePrint Archive, Paper 2025/535. 2025. URL: https://eprint.iacr.org/2025/535.

[28] Z. Zhao and T.-H. H. Chan. "How to vote privately using bitcoin". In: *Information and Communications Security: 17th International Conference, ICICS 2015, Beijing, China, December 9–11, 2015, Revised Selected Papers 17*. Springer. 2016, pp. 82–96.

[29] H. S. Galal and A. M. Youssef. "Verifiable sealed-bid auction on the ethereum blockchain". In: *Financial Cryptography and Data Security: FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curacao, March 2, 2018, Revised Selected Papers 22*. Springer. 2019, pp. 265–278.

[30] S. Chen, J. H. Cheon, D. Kim, and D. Park. *Verifiable Computing for Approximate Computation*. Cryptology ePrint Archive, Paper 2019/762. 2019. URL: https://eprint.iacr.org/2019/762.

[31] A. Naveh and E. Tromer. "Photoproof: Cryptographic image authentication for any set of permissible transformations". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 255–271.

[32] T. Derei, B. Aulenbach, V. Carolino, C. Geren, M. Kaufman, J. Klein, R. Islam Shanto, and H. F. Korth. "Scaling Zero-Knowledge to Verifiable Databases". In: *Proceedings of the 1st Workshop on Verifiable Database Systems*. VDBS '23. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 1–9. ISBN: 9798400707759. DOI: 10.1145/3595647.3595648. URL: https://doi.org/10.1145/3595647.3595648.

[33] X. Li, C. Weng, Y. Xu, X. Wang, and J. Rogers. "ZKSQL: Verifiable and Efficient Query Evaluation with Zero-Knowledge Proofs". In: *Proc. VLDB Endow.* 16.8 (Apr. 2023), pp. 1804–1816. ISSN: 2150-8097. DOI: 10.14778/3594512.3594513. URL: https://doi.org/10.14778/3594512.3594513.

[34] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. "ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash." In: *USENIX Security Symposium*. Vol. 10. 2010, pp. 193–206.

[35] N. Samardzic, S. Langowski, S. Devadas, and D. Sanchez. "Accelerating Zero-Knowledge Proofs Through Hardware-Algorithm Co-Design". In: *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2024, pp. 366–379. DOI: 10.1109/MICRO61859.2024.00035.

[36] J. Ernstberger, S. Chaliasos, G. Kadianakis, S. Steinhorst, P. Jovanovic, A. Gervais, B. Livshits, and M. Orrù. "zk-bench: A toolset for comparative evaluation and performance benchmarking of snarks". In: *International Conference on Security and Cryptography for Networks*. Springer. 2024, pp. 46–72.

[37] D. Jin and ZKValidator. *Benchmarking ZKP Development Frameworks: The Pantheon of ZKP*. Accessed: 2025-04-11. 2024. URL: https://ethresear.ch/t/benchmarking-zkp-development-frameworks-the-pantheon-of-zkp/14943/1.

[38] T. Datta, B. Chen, and D. Boneh. "VerITAS: verifying image transformations at scale". In: *Cryptology ePrint Archive* (2024).

[39] P. Network. *Scaling Trust in the Age of Artificial Intelligence*. Accessed: 2025-04-11. 2024. URL: https://medium.com/polyhedra-network/scaling-trust-in-the-age-of-artificial-intelligence-cab076d6164e.

[40] B.-J. Chen, S. Waiwitlikhit, I. Stoica, and D. Kang. "ZKML: An Optimizing System for ML Inference in Zero-Knowledge Proofs". In: *Proceedings of the Nineteenth European Conference on Computer Systems*. 2024, pp. 560–574.

[41] P. L. Montgomery. "Modular multiplication without trial division". In: *Mathematics of computation* 44.170 (1985), pp. 519–521.

[42] R. El Khatib, R. Azarderakhsh, and M. Mozaffari-Kermani. "Optimized algorithms and architectures for montgomery multiplication for post-quantum cryptography". In: *International Conference on Cryptology and Network Security*. Springer. 2019, pp. 83–98.

[43] P. Barrett. "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor". In: *Conference on the Theory and Application of Cryptographic Techniques*. Springer. 1986, pp. 311–323.

[44] N. Emmart, F. Zheng, and C. Weems. "Faster modular exponentiation using double precision floating point arithmetic on the GPU". In: *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*. IEEE. 2018, pp. 130–137.

[45] P. Z. Team. *Plonky2: Fast Recursive Arguments with PLONK and FRI*. URL: https://docs.rs/crate/plonky2/latest/source/plonky2.pdf.

[46] P. Z. Team. *Plonky3: A toolkit for polynomial IOPs (PIOPs)*. URL: https://github.com/Plonky3/Plonky3.

[47] J. Bruestle, P. Gafni, and the RISC Zero Team. *RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity*. URL: https://dev.risczero.com/proof-system-in-detail.pdf.

[48] B. E. Diamond and J. Posen. *Succinct Arguments over Towers of Binary Fields*. Cryptology ePrint Archive, Paper 2023/1784. 2023. URL: https://eprint.iacr.org/2023/1784.

[49] T. Xie, Y. Zhang, and D. Song. *Orion: Zero Knowledge Proof with Linear Prover Time*. Cryptology ePrint Archive, Paper 2022/1010. 2022. URL: https://eprint.iacr.org/2022/1010.

[50] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. *Ligero: Lightweight Sublinear Arguments Without a Trusted Setup*. Cryptology ePrint Archive, Paper 2022/1608. 2022. DOI: 10.1145/3133956. URL: https://eprint.iacr.org/2022/1608.

[51] B. Chen, B. Bünz, D. Boneh, and Z. Zhang. *HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates*. Cryptology ePrint Archive, Paper 2022/1355. 2022. URL: https://eprint.iacr.org/2022/1355.

[52] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. "Algebraic methods for interactive proof systems". In: *Journal of the ACM (JACM)* 39.4 (1992), pp. 859–868.

[53] S. Setty. *Spartan: Efficient and general-purpose zkSNARKs without trusted setup*. Cryptology ePrint Archive, Paper 2019/550. 2019. URL: https://eprint.iacr.org/2019/550.

[54] R. Jiang, C. Peng, M. Luo, R. Chen, and D. He. "SimdMSM: SIMD-accelerated Multi-Scalar Multiplication Framework for zkSNARKs". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025.2 (2025), pp. 681–704.

[55] T. Lu, C. Wei, R. Yu, C. Chen, W. Fang, L. Wang, Z. Wang, and W. Chen. "cuZK: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on gpus". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.3 (2023), pp. 194–220.

[56] Y. Yang, Z. Lu, J. Zeng, X. Liu, X. Qian, and Z. Yu. "Falic: An FPGA-based Multi-Scalar Multiplication Accelerator for Zero-Knowledge Proof". In: *IEEE Transactions on Computers* (2024).

[57] R. Karanjai, S. Shin, W. Xiong, X. Fan, L. Chen, T. Zhang, T. Suh, W. Shi, V. Kuchta, F. Sica, et al. "TPU as Cryptographic Accelerator". In: *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy 2024*. 2024, pp. 37–44.

[58] A. Daftardar, J. Mo, J. Ah-kiow, B. Bünz, R. Karri, S. Garg, and B. Reagen. "Need for zkSpeed: Accelerating HyperPlonk for Zero-Knowledge Proofs". In: *arXiv preprint arXiv:2504.06211* (2025).

[59] Z. Yang, L. Zhao, P. Li, H. Liu, K. Li, B. Zhao, D. Meng, and R. Hou. "LegoZK: A Dynamically Reconfigurable Accelerator for Zero-Knowledge Proof". In: *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2025, pp. 113–126.

[60] C. Wang and M. Gao. "SAM: A scalable accelerator for number theoretic transform using multi-dimensional decomposition". In: *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE. 2023, pp. 1–9.

[61] J. Groth. *On the Size of Pairing-based Non-interactive Arguments*. Cryptology ePrint Archive, Paper 2016/260. 2016. URL: https://eprint.iacr.org/2016/260.

[62] C. Liao, Y. Yan, B. R. De Supinski, D. J. Quinlan, and B. Chapman. "Early experiences with the OpenMP accelerator model". In: *OpenMP in the Era of Low Power Devices and Accelerators: 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings 9*. Springer. 2013, pp. 84–98.

[63] S. Wienke, P. Springer, C. Terboven, and D. an Mey. "OpenACC—first experiences with real-world applications". In: *Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings 18*. Springer. 2012, pp. 859–870.

[64] S. Lee and R. Eigenmann. "OpenMPC: Extended OpenMP programming and tuning for GPUs". In: *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2010, pp. 1–11.

[65] S. Lee, S.-J. Min, and R. Eigenmann. "OpenMP to GPGPU: a compiler framework for automatic translation and optimization". In: *ACM Sigplan Notices* 44.4 (2009), pp. 101–110.

[66] T. Grosser, A. Cohen, P. H. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. "Split tiling for GPUs: automatic parallelization using trapezoidal tiles". In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. 2013, pp. 24–31.

[67] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. "The tensor algebra compiler". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–29.

[68] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. "{TVM}: An automated {End-to-End} optimizing compiler for deep learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594.

[69] A. Sabne. "XLA: Compiling machine learning for peak performance". In: *Google Res* (2020).

[70] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, et al. "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2024, pp. 929–947.

[71] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. "An asymmetric distributed shared memory model for heterogeneous parallel systems". In: *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*. 2010, pp. 347–358.

[72] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. "Automatic CPU-GPU communication management and optimization". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 142–151. ISBN: 9781450306638. DOI: 10.1145/1993498.1993516. URL: https://doi.org/10.1145/1993498.1993516.

[73] T. Allen and R. Ge. "In-depth analyses of unified virtual memory system for GPU accelerated computing". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3480855. URL: https://doi.org/10.1145/3458817.3480855.

[74] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. "Gunrock: a high-performance graph processing library on the GPU". In: *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '16. Barcelona, Spain: Association for Computing Machinery, 2016. ISBN: 9781450340922. DOI: 10.1145/2851141.2851145. URL: https://doi.org/10.1145/2851141.2851145.

[75] OpenCilk. *Write fast code with C/C++ and OpenCilk*. URL: https://www.opencilk.org/.

[76] M. Frigo, C. E. Leiserson, and K. H. Randall. "The implementation of the Cilk-5 multithreaded language". In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 1998, pp. 212–223.

[77] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. "Reducers and other Cilk++ hyperobjects". In: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 2009, pp. 79–90.

[78] T. B. Schardl, W. S. Moses, and C. E. Leiserson. "Tapir: Embedding fork-join parallelism into LLVM's intermediate representation". In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2017, pp. 249–265.

[79] N. Sheybani, A. Ahmed, M. Kinsy, and F. Koushanfar. "Zero-Knowledge Proof Frameworks: A Systematic Survey". In: *arXiv e-prints* (2025), arXiv–2502.

[80] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. "Bulletproofs: Short proofs for confidential transactions and more". In: *2018 IEEE symposium on security and privacy (SP)*. IEEE. 2018, pp. 315–334.

[81] R. S. Wahby, I. Tzialla, abhi shelat, J. Thaler, and M. Walfish. *Doubly-efficient zkSNARKs without trusted setup*. Cryptology ePrint Archive, Paper 2017/1132. 2017. URL: https://eprint.iacr.org/2017/1132.

[82] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. "Libra: Succinct zero-knowledge proofs with optimal prover computation". In: *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer. 2019, pp. 733–764.

[83] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge". In: *Cryptology ePrint Archive* (2019).

[84] J. Zhang, T. Xie, Y. Zhang, and D. Song. "Transparent polynomial delegation and its applications to zero knowledge proof". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 859–876.

[85] J. Zhang, T. Liu, W. Wang, Y. Zhang, D. Song, X. Xie, and Y. Zhang. "Doubly efficient interactive proofs for general arithmetic circuits with linear prover time". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 159–177.

[86] A. Kothapalli, S. Setty, and I. Tzialla. *Nova: Recursive Zero-Knowledge Arguments from Folding Schemes*. Cryptology ePrint Archive, Paper 2021/370. 2021. URL: https://eprint.iacr.org/2021/370.

[87] A. Arun, S. Setty, and J. Thaler. *Jolt: SNARKs for Virtual Machines via Lookups*. Cryptology ePrint Archive, Paper 2023/1217. 2023. URL: https://eprint.iacr.org/2023/1217.

[88] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. "Fast reed-solomon interactive oracle proofs of proximity". In: *45th international colloquium on automata, languages, and programming (icalp 2018)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 2018, pp. 14–1.

[89] A. Kate, G. M. Zaverucha, and I. Goldberg. "Constant-size commitments to polynomials and their applications". In: *International conference on the theory and application of cryptology and information security*. Springer. 2010, pp. 177–194.

[90] J. Hilton. "Enabling the Rust Compiler to Reason about Fork/Join Parallelism via Tapir". MEng thesis. Massachusetts Institute of Technology, 2024.

[91] R. D. Blumofe and C. E. Leiserson. "Scheduling multithreaded computations by work stealing". In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[92] *Rayon: A data parallelism library for Rust*. URL: https://github.com/rayon-rs/rayon.

[93] SCIPR lab. *libsnark: a C++ library for zkSNARK proofs*. URL: https://github.com/scipr-lab/libsnark.

[94] arkworks contributors. *`arkworks` zkSNARK ecosystem*. 2022. URL: https://arkworks.rs.

[95] G. E. Blelloch. "Scans as primitive parallel operations". In: *IEEE Transactions on computers* 38.11 (1989), pp. 1526–1538.

[96] D. Merrill and M. Garland. "Single-pass parallel prefix scan with decoupled look-back". In: *NVIDIA, Tech. Rep. NVR-2016-002* (2016).

[97] D. Leijen, B. Zorn, and L. De Moura. "Mimalloc: Free list sharding in action". In: *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*. Springer. 2019, pp. 244–265.

[98] C. developers. *SafeStack — Clang 21.0.0 documentation*. URL: https://clang.llvm.org/docs/SafeStack.html.

[99] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. "MLIR: Scaling compiler infrastructure for domain specific computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 2–14.

[100] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. "Graphit: A high-performance graph dsl". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–30.

[101] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou. "The Faiss library". In: *arXiv preprint arXiv:2401.08281* (2024).

[102] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar. "Accelerating Large-Scale Inference with Anisotropic Vector Quantization". In: *International Conference on Machine Learning*. 2020. URL: https://arxiv.org/abs/1908.10396.

[103] P. Sun, D. Simcha, D. Dopson, R. Guo, and S. Kumar. "SOAR: Improved Indexing for Approximate Nearest Neighbor Search". In: *Neural Information Processing Systems*. 2023. URL: https://arxiv.org/abs/2404.00774.

[104] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, et al. "Milvus: A purpose-built vector data management system". In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 2614–2627.

[105] Weaviate. *Weaviate: The AI-native database developers love*. URL: https://weaviate.io/.

[106] Qdrant. *Qdrant - Vector Database*. URL: https://qdrant.tech/.

[107] Pinecone. *Pinecone: The vector database to build knowledgable AI*. URL: https://www.pinecone.io/.

[108] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. "Succinct {Non-Interactive} zero knowledge for a von neumann architecture". In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 781–796.

[109] A. Fiat and A. Shamir. "How to prove yourself: Practical solutions to identification and signature problems". In: *Conference on the theory and application of cryptographic techniques*. Springer. 1986, pp. 186–194.