Adapting AlphaEvolve to Optimize Fully Homomorphic Encryption on TPUs

Shruthi Gorantala **Jianming Tong*** **Asra Ali** **Baiyu Li** **Jonathan Katz** **Jeremy Kun**
Thomas Steinke† **Abhradeep Thakurta†** **Julian Walker†** **Amir Yazdanbakhsh†**
Google *Georgia Institute of Technology †Google DeepMind
gshruthi@google.com, jianming.tong@gatech.edu,{baiyuli, asraa, jkun, jkcrypto}@google.com,
{steinke, athakurta, julianwa, ayazdan}@google.com

***Abstract*—Fully Homomorphic Encryption (FHE) protects data during computation, but its computational and memory overheads make deployment at scale difficult. Google TPUs provide leading throughput, yet high FHE performance requires optimizing JAX/Pallas implementations to trigger the right XLA lowering decisions and to fill the TPU's coarse-grained vector registers. This optimization loop is typically manual: developers edit frontend code, compile through an opaque stack, inspect hardware traces, and repeat. We adapt AlphaEvolve to automate this frontend design-space exploration for FHE kernels on TPUs. AlphaEvolve treats the kernel implementation as an evolutionary search problem; candidate programs are generated by an LLM, compiled and executed on real TPUv5e hardware, scored on their latency by XProf, and admitted only if they pass correctness/security checks and have a higher score.**

We evaluate AlphaEvolve on TFHE bootstrapping in Jaxite and on CKKS rotation and multiplication in CROSS. Within 24 hours, AlphaEvolve discovered JAX/Pallas-level modifications that lead to loop unrolling for parameter reuse, XLA-favorable tiling, and removal of redundant type conversions. These optimizations reduce TFHE blind-rotation latency during bootstrapping from 10.0 ms to 3.5 ms (2.86×), CKKS rotation latency from 4854 μs to 3714 μs (1.31×), and CKKS multiplication latency from 6340 μs to 5359 μs (1.18×) relative to expert baselines. The results show that guarded LLM evolution can effectively automate performance engineering of JAX/Pallas-based FHE kernel optimizations and identify optimizations that human experts might miss.

I. Introduction

Fully Homomorphic Encryption (FHE) [8], a powerful example of a privacy-enhancing technology, offers strong privacy guarantees as it enables computation on encrypted data. However, FHE incurs orders-of-magnitude computational and memory overheads [9], making efficiency a deployment bottleneck.

TPUs are attractive for FHE because they provide high memory bandwidth and industry-leading throughput per watt [17], [18], but they are not natively designed for cryptographic kernels. Unlike dense neural-network layers that TPUs were designed to handle, FHE primitives rely on modular arithmetic and layout-sensitive polynomial operations. Algorithmic transformations could lower modular arithmetic into kernels natively supported by TPUs [18], but efficient TPU execution also requires (1) aligning tensor shapes with the granularity of vector registers (VRegs), (2) avoiding data movement by removing implicit layout transformations, and (3) limiting off-chip memory accesses by tiling, scheduling, and unrolling. All three require modifications to the front-end JAX/Pallas kernels.

Modern compiler stacks make this optimization difficult. JAX and Pallas expose high-level programming interfaces, while underlying XLA compiler [16] performs hundreds of transformations before emitting TPU code. As a result, a small frontend change can have a large but non-obvious effect on VReg utilization, memory scheduling, and implicit layout conversion. Human developers often discover performant implementations only through a slow loop of code edits, compilation, profiling, and trace inspection.

AI coding agents such as AlphaEvolve [14] have demonstrated significant advances in multiple fields including algorithm design, improving code efficiency, architecture discovery, and enabling discovery across scientific domains. Further, the use of AI to improve cryptographic implementations remains largely unexplored. This may be because LLM hallucinations risk introducing security vulnerabilities which are unacceptable in cryptographic implementations.

In this paper, we use AlphaEvolve, an agentic framework, to optimize JAX/Pallas-based FHE kernels on hardware accelerators for AI, such as Google's TPU, with rigorous correctness checking. We formulate kernel optimization as an evolutionary search problem using real TPU execution latency and execution traces from Xprof [15] as feedback. By automating the discovery of JAX/Pallas-level improvements, AlphaEvolve enables cryptographic developers to explore architectural ideas and co-optimize cryptographic primitives on specialized hardware like TPUs. Our contributions are:

- We formulate JAX/Pallas-level FHE-on-TPU kernel optimizations as a security-guarded evolutionary search problem.
- We design a hardware-in-the-loop evaluator that scores candidates using real TPUv5e latency from XProf traces while rejecting candidates that fail functional correctness, randomized ciphertext tests, or security parameter checks.
- We identify concrete JAX/Pallas optimizations discovered by search that outperform domain-expert baselines: loop unrolling to reuse common parameters, finer-grained scheduling to hide off-chip memory access latency, avoid datatype conversion, and XLA-favorable tiling that improves VReg utilization.
- We demonstrate end-to-end speedups over expert baselines: 2.85× for TFHE blind rotation, 1.31×/1.18× for CKKS rota-

tion/multiplication within 24 hours of automated exploration.

II. Background and Motivation

A. AlphaEvolve

AlphaEvolve [14] is an agentic framework designed for algorithmic and scientific discovery by combining LLMs with the evolutionary search. It could be used to automatically evolve code in any programming language through a flywheel of code generation and code evaluation.

The workflow begins with a human programmer configuring the system based on expert-guided, task specific instructions, an initial code example to seed the evolutionary database and an evaluator that scores the generated code.

Next, the system samples code database and a pool of prompts. These prompts are fed into LLMs to generate new code modification. The generated code blocks are then compiled and fed into the evaluators, where they are scored, ranked, and only top-rank correct programs are added back to the evolutionary database. The ranking and evolution employs structured evolutionary techniques, such as island-based models or MAP-Elites algorithms [14]. These techniques balance genetic diversity based on inherent metrics (e.g., program length, code complexity) and performance on the evaluator metrics (fitness score). This optimization loop improves the code to find better program based on the fitness score.

B. Fully Homomorphic Encryption

FHE [8] enables computation on encrypted data with holistic security. FHE schemes can be broadly categorized into scalar schemes and vector schemes based on the type of data they encrypt. We consider two schemes here:

1) TFHE: TFHE [4] encrypts bits or short integers and supports general-purpose encrypted computation. Its dominant cost is bootstrapping, especially blind rotation and negacyclic vector-matrix polynomial multiplication done as part of bootstrapping.

2) CKKS: CKKS [3] packs numeric values into vectors and is widely used for privacy-preserving machine learning and numerical workloads. We optimize two backbone CKKS operations from CROSS [18]: homomorphic rotation (`CKKS-Rot`) and homomorphic multiplication (`CKKS-Mult`). They combine automorphisms, key switching, rescaling, tensor multiplication, and modular reduction.

C. TPU Execution Model

Fig. 1 summarizes the TPU features. On-chip vector memory (VMEM) is partitioned across 128 lanes. Each lane contains 8 sublanes, and each sublane provides 32 instances of 32-bit registers. Registers are organized into $(8, 128)$ blocks, called VRegs, each formed by selecting one register from every sublane across all 128 lanes. The XLU performs data reorganization, while the VPU and MXU accelerate vector and matrix operations. Both computation and data movement execute in a single instruction, multiple data (SIMD) manner over one $(8, 128)$ VReg at a time. Consequently, programs that fully utilize VRegs achieve higher efficiency, whereas

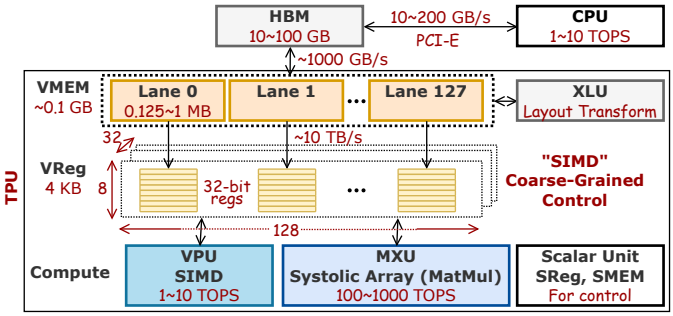

Fig. 1: TPU microarchitecture overview – SIMD machine.

operations requiring fine-grained or irregular data access incur significant overhead.

For FHE, the implication is direct: mathematically equivalent implementations can differ sharply in performance depending on shapes, layout and scheduling choices of JAX/Pallas tensors.

D. The TPU Compiler Stack

We inherit the multi-layer TPU compilation stacks and profiling system to support FHE computations:

1) JAX: JAX expresses high-level numerical and cryptographic computation. Both expert-crafted TPU libraries, Jaxite [2] and CROSS [18], use JAX / Pallas as the frontend.

2) Pallas: it serves as a kernel language that allows for the creation of custom, low-level hardware kernels within JAX. Our system relies on Pallas kernels to establish fine-grained control over data movement and memory staging on the TPU [12].

3) XLA: it lowers JAX / Pallas programs to TPU executables and applies optimizations such as fusion and layout selection [10], [16]. XLA compilation passes are shape-sensitive.

4) Xprof: it's an integrated profiling tool within the XLA ecosystem that captures high-fidelity hardware execution traces. It extracts metrics such as compute and memory bandwidth utilization as well as on-chip and off-chip storage occupancy.

E. Challenge: Compiler-Obscured Performance Engineering

TPU performance engineering demands rigorous optimizations: (1) tiling workloads to strictly align with vector register (VReg) granularities, (2) navigating frontend abstractions to bypass implicit layout transformations, and (3) sizing tensors to effectively hide off-chip memory latency behind compute.

Developers are strictly bound to high-level programming interfaces such as JAX and Pallas, limiting the easy access to manipulating the low level schedule. At this level of abstraction, every source modification must traverse hundreds of opaque compilation passes within stacks like XLA [16] before finally materializing as a low-level TPU execution trace. Because these intermediate layers obscure critical microarchitectural details, adapting an algorithm for TPU execution devolves into a sluggish trial-and-error process. Developers are forced to experimentally deduce the specific shapes, casts, loops, and tiling parameters at frontend JAX / Pallas that cause XLA to produce a good TPU program.

This paper adapts AlphaEvolve to automate this compiler-steering loop while keeping correctness and security checks invisible to AlphaEvolve as hidden tests.

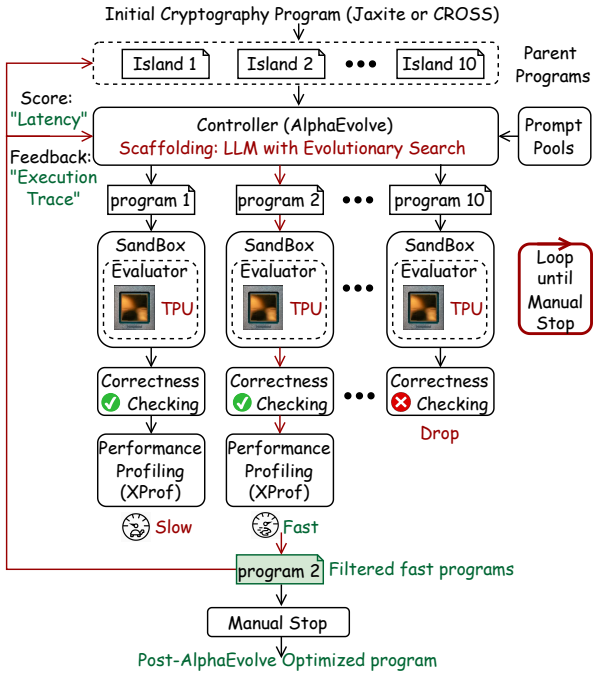

Fig. 2: Workflow for adapting AlphaEvolve to cryptographic kernel optimization. AlphaEvolve samples parent programs, prompts an LLM to generate candidate JAX/Pallas implementations, and evaluates each candidate in a sandbox on TPU hardware. Invalid candidates are rejected by correctness and security checks. Valid candidates are scored by latency and profiled through execution traces. Fast candidates are returned to the evolutionary population for further search.

III. Adapting AlphaEvolve for Cryptography

Fig. 2 shows how we specialize AlphaEvolve for cryptographic kernels. The controller, prompt sampling, and population evolution are the standard AlphaEvolve mechanisms. The adaptation consists of four cryptography-specific components: an initial expert cryptographic implementation, a constrained editable region, an evaluator that enforces correctness and security, and real hardware latency reward with execution traces as feedback. The loop keeps going until manual stop. Final best candidate is selected for human review and integration.

A. Search Scope and Initial Programs

We initialize AlphaEvolve with existing optimized implementations rather than blank files. For TFHE, we feed Jaxite's blind-rotation implementation. For CKKS, we pick the `CKKS-Rot` and `CKKS-Mult` implementations in CROSS [18]. We limit exposure to kernels located far below the roofline [19] to AlphaEvolve. Security-critical parameters, hidden tests, and reference outputs are kept invisible. This separation narrows the search to hardware-performance choices while preventing the evaluator from rewarding weakened cryptographic settings.

B. Evaluation Engine

The evaluator is the core safeguard that makes AlphaEvolve applicable to cryptographic kernels. A candidate receives a performance score only after passing correctness and security checks. Candidates that fail compilation, violate functional correctness, change protected parameters, or produce invalid ciphertext results are dropped.

1) Correctness Checks: We use both local and end-to-end checks. Local tests compare intermediate computational submodules against reference outputs. End-to-end tests instantiate the cryptosystem with valid parameters, generate randomized plaintext inputs, encrypt them, run the candidate homomorphic computation, decrypt the result, and compare against the expected cleartext computation. The hidden test set prevents candidates from overfitting to visible kernels or reward-hacking the evaluator.

2) Security Checks: The evaluator rejects candidates that alter protected parameter choices or reduce the target security level. When parameterized paths are exercised, the system verifies that the resulting configuration remains consistent with at least 128-bit security using standard lattice-estimator methodology [1]. For FHE computations that may accumulate noise across repeated operations, the evaluator includes high-depth end-to-end tests to detect decryption failures that may not appear in a single primitive invocation.

3) Latency Reward and Trace Feedback: Valid candidates are compiled and executed on TPUv5e. The reward is the negative runtime latency of the target primitive or functional module, excluding XLA compilation overhead. For kernels whose local speedups may not translate into application speedups, AlphaEvolve can evolve multiple primitive implementations together and score the full module latency. In addition to scalar latency, XProf traces provide diagnostic feedback such as memory behavior, layout transformations, and compute utilization. This feedback helps the search identify when a frontend change improves or degrades performance.

IV. AlphaEvolve for optimizing FHE

A. Setup

1) Initial programs: For TFHE, we use the bootstrap implementation from jaxite [2] and allow AlphaEvolve to modify various parts of `blind_rotate`. For CKKS, we adopt `CKKS-Rot` and `CKKS-Mult` from the TPU-based CKKS library, CROSS [18] that offers state-of-the-art throughput, as the initial programs and apply AlphaEvolve to search for faster implementations. We expose the implementations of Rescale, TensorMultiply, KeySwitch, ApproxModDown, and AutoMorphism from both kernels to AlphaEvolve.

2) Prompt Setup: We use the system prompt together with a prompt sampled uniformly from §A. These prompts are designed to steer AlphaEvolve towards JAX / Pallas modifications for performance improvements, especially scheduling and layout changes that increase VReg utilization in TPU and minimize implicit layout transformation, rather than toward new algorithms discovery.

3) Correctness, and Performance Tests: For TFHE, we incorporate end-to-end tests where the cryptosystem is initiated with parameters offering 128-bit security and random inputs are encrypted and evaluated against the generated code. The outputs

are then decrypted and compared against the expected plaintext values. All values of the 3-bit lookup tables are evaluated for correctness.

For CKKS, functional correctness is enforced using a hidden test set with fixed ciphertext inputs and expected ciphertext outputs.

For both schemes, performance is measured from XProf traces, and we return the negative kernel latency as the score so that lower latency corresponds to a higher score.

We run the evolution process with 10 controllers and 10 evaluators per controller, for a total of 100 TPUv5e chips.

TABLE I: Latency Comparison to SoTA [2], [18]

Workload	Latency Improvement	Code/Analysis
Blind Rotation (TFHE)	9.4 ms (TPUv5e-1)	Code: jaxite [2]
+ Unrolling	10 → 7.8 ms	Code: [5] Reason: §IV-B1
+ Fine-grained scheduling	7.8 → 6.3 ms	Code: [6] Reason: §IV-B2
+ Type Cast Removal	6.3 → 3.5 ms	Code: [7] Reason: §IV-B3
Post-AlphaEvolve blind Rotation	**9.4 ms → 3.5 ms**	**2.85× speedup**
Post-AlphaEvolve bootstrap	**10 → 4 ms**	**2.5× speedup**
`CKKS-Mult` (CKKS)	6340 µs (TPUv5e-1)	Code ↗ [18]
+ XLA-favor tiling choice	5359 µs (TPUv5e-1)	Code: ↗ Reason: §IV-C
Post-AlphaEvolve `CKKS-Mult`	**6340 → 5359 µs**	**1.18× speedup**
`CKKS-Rot` (CKKS)	4854 µs (TPUv5e-1)	Code: ↗ [18]
+ XLA-favor tiling choice	3714 µs (TPUv5e-1)	Code: ↗ Reason: §IV-C
Post-AlphaEvolve `CKKS-Rot`	**4854 → 3714 µs**	**1.3× speedup**

B. AlphaEvolve for TFHE

AlphaEvolve explored several distinct classes of optimizations, ranging from standard loop transformations to complex kernel scheduling and data type optimizations.

1) Loop Unrolling for Parameter Reuse [5]: In the negacyclic vector-matrix polynomial multiplication (`negacyclic_vector_matrix_polymul`) routine [2], the system independently discovered that unrolling the primary `for` loop by a factor of 8 yielded significant latency reductions. In the baseline trace, the loop executes in serial, loading the common parameters repeatedly. By unrolling the loop, common parameters get reused across multiple loop iterations. Because the target parameter size (3.51 MB) fit within the available on-chip memory, this transformation reduced latency from 10 ms to 7.8 ms. This highlights that AlphaEvolve could automatically identify and exploit loop unroll optimizations.

2) Fine-Grained Memory and Compute Scheduling [6]: The system optimized data access patterns by adjusting the tiling and scheduling choice of the kernel. Specifically, it tiled one big JAX tensor into two blocks with shape halved. This modification allowed the XLA compiler to schedule more fine-grained data loading from off-chip memory, and hiding it behind computation. This optimization reduced the overall bootstrapping latency from 7.8 ms to 6.3 ms.

3) Elimination of Redundant Type Casts [7]: The most significant single optimization involved the removal of an explicit bitcast operation. The baseline implementation explicitly cast a right-hand side (`rhs`) operand to 8-bit integers (`i8`). AlphaEvolve identified that the dynamic range of the input data (values from 0 to 255 due to RLWE decomposition) was already sufficiently captured by the existing `bf16` precision. By removing the redundant and expensive bitcast operator, the system reduced the execution time from 6.29 ms to 3.35 ms. This optimization requires a deep understanding of both the mathematical dynamics of TFHE and the specific hardware execution costs of bitcast operations, making it difficult for human engineers to consistently identify.

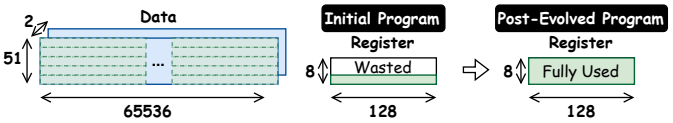

Fig. 3: Effect of the discovered layout changes. The initial program leaves substantial VReg capacity unused because operations are expressed at a granularity that does not match VReg. The evolved program exposes fuller VReg utilization.

C. AlphaEvolve for CKKS

- `CKKS-Rot`: On TPUv5e, AlphaEvolve reduces the latency of `CKKS-Rot` from 4874 µs to 3754 µs, yielding a 1.31× speedup. This performance gain is primarily achieved by partitioning a tensor into two smaller subtensors. This strategic split triggers specialized XLA optimizations that significantly improve Vector Register (VReg) utilization during the AutoMorphism stage. Consequently, AutoMorphism's contribution to the total end-to-end latency drops from 12% to just 4%. These results demonstrate AlphaEvolve's ability to expose and exploit non-intuitive, compiler-specific performance "sweet spots" that are exceedingly difficult for human engineers to discover manually.
- `CKKS-Mult`: On TPUv5e, AlphaEvolve reduces latency of `CKKS-Mult` from 6340 µs to 5359 µs, corresponding to a 1.18× speedup. Such performance improvement comes from the scheduling optimization to reuse common parameters. The modest improvement indicates that `CKKS-Mult` is less constrained by underlying compiler inefficiencies.

These results show that AlphaEvolve can improve CKKS kernel performance on TPU by discovering implementation changes that increase effective VReg utilization and reduce inefficiencies in the execution schedule.

D. Takeaways

The three workloads show that AlphaEvolve is most effective for cryptographic kernels when the bottleneck is caused by a mismatch between cryptographic data layout and the TPU compiler/hardware execution grain. In TFHE, the search discovers both conventional locality improvements and a non-obvious cast elimination that depends on the ciphertext decomposition range. In CKKS, the search finds tensor shapes that steer XLA toward higher VReg utilization. Across all cases, the successful edits preserve the correctness and security while improving how the algorithm is presented to the compiler. By using AlphaEvolve to expose the performance headroom in the current TPU deployment flow, we also identify a key software bottleneck: existing programmable APIs force tensor reshapes and layout reorganizations that are not inherent to the hardware, introducing avoidable overhead.

V. Related Work

To our knowledge this is the first application of AI technologies to improve cryptographic performance. The closest work for improving kernel generation is done by autocomp [11] and EvoX [13] where agentic infrastructure has been used to improve machine learning kernels.

VI. Conclusion

This paper presents the first application of agentic AI to optimize FHE primitives on TPUs. We detail the design of using AlphaEvolve, a closed-loop system that combines LLMs with real-world hardware execution feedback and rigorous correctness checks. We demonstrate the framework's ability to autonomously discover performance improvements on TPUs.

Our results demonstrate that AI-driven search uncovers optimizations frequently missed by domain experts.

A. Experience and Lessons Learned using AlphaEvolve

Some of our experiences and learnings in adapting AlphaEvolve for cryptographic optimization are described below:

1) Granularity of Optimization: Targeting isolated operations, such as polynomial multiplication (`polymul` [2]), failed to produce end-to-end speedups. We found that the agent requires visibility into broader operational scopes to make a meaningful impact on overall latency.

2) Co-evolution and System-Level Scoring: Building on the need for broader visibility, shifting to the co-evolution of the `external_product` and scoring it against the complete bootstrapping execution successfully reduced latency from 10 ms to 5.6 ms.

3) Guided Domain Constraints: While evolving the `blind-rotate` operation [2], the system discovered that RLWE decomposition allowed input tensors to safely use `int8` rather than `int32`, and that unrolling the loop by a factor of eight yielded significant benefits. This underscores the importance of domain experts selectively granting the AI permission to modify high-impact code blocks.

4) Comprehensive Correctness Tests: Initial attempts to evolve the AND-gate failed to cover all values of a 3-input lookup table (`lut3` [2]). We had to update the evaluator to exhaustively verify all lookup table values, demonstrating that strict, comprehensive validation is essential to prevent the model from reward hacking.

5) Security Checks: Because performance cannot come at the expense of cryptographic integrity, we introduced a standardized parameter check using a lattice estimator to ensure at least 128-bit security when co-optimizing scheme parameters.

6) Feedback and Prompts: To effectively steer the evolutionary search, we integrated hardware profiling tools (`xprof`) directly into the feedback loop and provided hardware-specific optimization guidelines as automated compilation prompts.

B. Future Work

Our initial results focus on TPUs but the framework is extensible and can potentially optimize FHE primitives as well as any cryptographic primitives on a variety of hardware accelerators. It can also be integrated with simulators to guide specialized hardware-software co-design. To ensure security guarantees are maintained during hardware optimization, future iterations can integrate parameter selection with noise tracking. Furthermore, the bottleneck of manual code reviews can be reduced by adding functional equivalence checkers directly into the validation pipeline. Finally, a fundamental open question remains in the design of reward functions to safely allow AI agents to discover novel cryptographic algorithms and protocols while preserving security guarantees.

Acknowledgments

This work represents an equal contribution between the first two authors, with the remaining co-authors listed alphabetically. The authors express their gratitude to Charles Hong, Andrew Ferraiuolo, Ben Kreuter, and Cindee Madison for their valuable feedback during various phases of this work. We thank Po-Sen Huang, Ngân (NV) Vũ, and the AlphaEvolve team for their assistance in establishing the experimental framework. We thank Drew Angeloff, Ankita Malviya Bairaria, Naomi Black, Elie Burzstein, Bryant Gipson, Pankaj Rohatgi, Amanda Walker, Moti Yung and the Safeworks leadership team for their guidance throughout this project. Finally, we also thank our extended team at Google DeepMind for their support of this research direction.

We also acknowledge the use of large language models in the preparation of this manuscript. Specifically, the models were used for proofreading, summarizing text, and generating LaTeX outlines. The authors have reviewed and verified all generated content and take full responsibility for the final manuscript.

References

[1] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," Cryptology ePrint Archive, Paper 2015/046, 2015. [Online]. Available: https://eprint.iacr.org/2015/046

[2] A. Ali, E. Astor, B. Gipson, S. Gorantala, M. Guevara, J. Kun, W. Lam, R. Misoczki, R. Springer, J. Takeshita, J. Tong, C. Tew, and C. Yun, "Jaxite: A fully homomorphic encryption backend targeting TPUs and GPUs, written in JAX," https://github.com/google/jaxite, 2024.

[3] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, ser. Lecture Notes in Computer Science, vol. 10624. Springer, 2017, pp. 409–437.

[4] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, Jan 2020.

[5] code perspective, "Add an unroll factor of 8 (reduces bootstrap speedup on tpu by 2 ms)," https://github.com/google/jaxite/pull/88, 2026, gitHub Pull Request #88, Accessed: 2026-05-06.

[6] code perspective, "Improvements to the `vector_matrix_polymul` provides a speedup of 1.7 ms," https://github.com/google/jaxite/pull/89, 2026, gitHub Pull Request #89, Accessed: 2026-05-06.

[7] code perspective, "Update _i32_matmul_unreduced_cggi to use single bfloat16 (instead of 4) for int32 multiplications," https://github.com/google/jaxite/pull/90, 2026, gitHub Pull Request #90, Accessed: 2026-05-06.

[8] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 169–178. [Online]. Available: https://doi.org/10.1145/1536414.1536440

[9] S. Gorantala, R. Springer, and B. Gipson, "Unlocking the potential of fully homomorphic encryption," *Communications of the ACM*, vol. 66, no. 6, pp. 68–75, jun 2023. [Online]. Available: https://doi.org/10.1145/3582500

[10] X. He, "Accelerated linear algebra compiler for computationally efficient numerical models: Success and potential area of improvement," *PLOS ONE*, vol. 18, no. 2, p. e0282265, 2023.
[11] C. Hong, S. Bhatia, A. Cheung, and Y. S. Shao, "Autocomp: A powerful and portable code optimizer for tensor accelerators," 2025. [Online]. Available: https://arxiv.org/abs/2505.18574
[12] JAX Developers, "Pallas quickstart — JAX documentation," https://docs.jax.dev/en/latest/pallas/quickstart.html, 2024.
[13] S. Liu, S. Agarwal, M. Maheswaran, M. Cemri, Z. Li, Q. Mang, A. Naren, E. Boneh, A. Cheng, M. Z. Pan, A. Du, K. Keutzer, A. Cheung, A. G. Dimakis, K. Sen, M. Zaharia, and I. Stoica, "Evox: Meta-evolution for automated discovery," 2026. [Online]. Available: https://arxiv.org/abs/2602.23413
[14] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. Ruiz, A. Mehrabian *et al.*, "Alphaevolve: A coding agent for scientific and algorithmic discovery," *arXiv preprint arXiv:2506.13131*, 2025.
[15] OpenXLA Project, "Profiling JAX computations with XProf," https://openxla.org/xprof/jax_profiling, 2024.
[16] OpenXLA Project, "OpenXLA: Accelerated linear algebra," https://github.com/openxla/xla, 2026.
[17] J. Tong, J. Dang, S. Langowski, T. Huang, A. Ali, J. Kun, J. Jiang, S. Devadas, and T. Krishna, "Enabling ai asics for zero knowledge proof," 2026. [Online]. Available: https://arxiv.org/abs/2604.17808
[18] J. Tong, T. Huang, J. Dang, L. de Castro, A. Itagi, A. Golder, A. Ali, J. Jiang, J. Kun, Arvind, G. E. Suh, and T. Krishna, "Leveraging asic ai chips for homomorphic encryption," ser. HPCA'26. Australia: 2026 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2026.
[19] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

APPENDIX

A. LLM Instruction-Prompt Pool

We uniformly sample prompt from the following pool.

• **Prompt 1:** Suggest a new way to minimize data reorganization in the program.

• **Prompt 2:** Suggest a new way to keep the logical shapes of JAX arrays invariant throughout execution.

• **Prompt 3:** Suggest a new way to hide off-chip memory latency behind computation.

• **Prompt 4:** Suggest a new optimization idea based on your expert knowledge of TPU performance tuning.

B. Role and Hardware Architecture Prompts

• **Role:** You are an expert hardware-aware compiler optimizer specializing in TPU architectures.

• **Objective:** Optimize the input computational graph for maximum performance on the TPU v5e architecture. You must make strategic decisions regarding operation mapping (Vector vs. Matrix units), memory hierarchy management, and parallelism.

• **Target Hardware Specifications:**

- **Compute Capabilities (Peak):**
  - Clock Frequency: 1.50 GHz
  - Matrix Units (MXU): 197 TFLOPS (`bf16`/`fp8`), 394 TOPS (`int8`), 788 TOPS (`int4`).
  - Vector Units (VPU): `vmatmul` (1.54 TB/s), `vmatpush` (6.16 TB/s).
- **Memory Constraints (CRITICAL):**
  - `CMEM`: None (0 bytes). Do not tile data into CMEM.
  - `Vmem` (Vector Memory): 128 MB. Read: 18.5 TB/s, Write: 6.16 TB/s.
  - `HBM` (High Bandwidth Memory): ∼17.2 GB at 820 GB/s.
- **Interconnect:** 2D Torus, 4 links/chip, 45 GB/s per link.

• **Optimization Strategies Required:**

- **Compute Mapping:** Map matrix multiplications to the Systolic Array (MXU). Map element-wise, reduction, and non-matmul ops to the Vector Unit (VPU).
- **Memory Tiling & Fusion:** All functional tiling must target the 128 MB Vmem. Perform loop fusion to avoid spilling back to HBM.
- **Precision Tuning:** Evaluate suitability for `int8` quantization for 2× throughput.
- **Pipeline Parallelism:** Overlap HBM-to-Vmem DMA transfers with VPU/MXU execution.

• **Output Requirements:** Generate optimized code/IR and summarize: theoretical peak utilization, memory bandwidth utilization, and a bottleneck analysis.