EFFICIENT PARALLEL IMPLEMENTATION OF QRUOV POST-QUANTUM SIGNATURE SCHEME ON A GPU

BY

CHAI YAU YUEN

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONOURS)

Faculty of Information and Communication Technology

(Kampar Campus)

FEBRUARY 2026

COPYRIGHT STATEMENT

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Computer Science (Honours) at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Dr Lee Wai Kong, for his invaluable guidance, support, and encouragement throughout this project. His expertise and insightful feedback have greatly contributed to the development and completion of my work.

Furthermore, I also extend my heartfelt thanks to my friends and family for their unwavering support and understanding during this journey. Their encouragement and patience have been a constant source of motivation. Additionally, I appreciate everyone else who offered help and encouragement along the way.

ABSTRACT

The rapid advancement of quantum computing threatens the mathematical foundations of classical cryptographic systems, accelerating the urgent need for Post-Quantum Cryptography (PQC). The Quotient Ring Unbalanced Oil and Vinegar (QR-UOV) signature scheme offers robust quantum resistance and compact signature sizes, making it an excellent candidate for secure digital communications. However, its heavy reliance on dense matrix operations and multivariate polynomial evaluations introduces severe computational bottlenecks when executed on traditional CPU architectures. To address this, this project presents a highly optimized, massively parallel implementation of the QR-UOV signature scheme utilizing the NVIDIA CUDA framework on a Graphics Processing Unit (GPU). To overcome the sequential limitations of the CPU, the workload was redesigned using coarse-grained batching, allowing thousands of independent signatures to be processed concurrently. Furthermore, to resolve severe global memory bandwidth bottlenecks, the architecture introduced Kernel Fusion and Shared Memory Tiling, ensuring that intermediate finite-field operations bypassed high-latency memory transactions. A dynamic Hybrid CPU-GPU Fallback controller was also engineered to resolve rank-deficient matrices on the host, thereby guaranteeing strict cryptographic correctness without inducing warp divergence on the device. The system was extensively benchmarked across exponentially increasing workloads. The optimized fused GPU implementation achieved a peak throughput of 205,189 signatures per second and 320,082 verifications per second, delivering a remarkable 242-fold speedup over the baseline CPU implementation. An alternative hardware-accelerated variant utilizing Tensor Cores (WMMA) was also evaluated, providing valuable architectural insights into the latency trade-offs of mapping integer-based cryptography onto half-precision floating-point hardware. Lastly, this project proves that optimized heterogeneous computing is a strict prerequisite for deploying PQC schemes in high-volume and real-time environments such as cloud servers and IoT gateways.

Area of Study: Post-Quantum Cryptography, Parallel Computing
Keywords: QR-UOV, Graphics Processing Unit (GPU), CUDA, Kernel Fusion, Digital Signatures, Tensor Cores, Heterogeneous Computing

TABLE OF CONTENTS

TITLE PAGE **I**
COPYRIGHT STATEMENT **II**
ACKNOWLEDGEMENTS **III**
ABSTRACT **IV**
TABLE OF CONTENTS **V**
LIST OF FIGURES **VIII**
LIST OF TABLES **IX**
LIST OF ABBREVIATIONS **X**
CHAPTER 1 INTRODUCTION **1**
1.1 Problem Statement and Motivation 1
1.2 Objective 3
1.3 Project Scope 4
1.4 Contributions 5
1.5 Report Organization 7
CHAPTER 2 LITERATURE REVIEW **8**
2.1 Previous Studies 8
2.1.1 Optimization of Falcon for GPUs 8
2.1.2 GPU-based Parallel Acceleration of SPHINCS+ (GRASP) 10
2.1.3 High Throughput Lattice-based Signatures on GPUs 12
2.1.4 High Throughput Acceleration of Cryptography Using Tensor Core 14
2.2 Limitation of Previous Studies 16
CHAPTER 3 SYSTEM METHODOLOGY/APPROACH **17**
3.1 System Design Diagram/Equation 17
3.1.1 System Architecture Diagram 17
3.1.2 Overview of the Heterogeneous Architecture 19
3.1.3 Components and Functionalities of the Heterogeneous System Architecture 20
3.2 Use Case & Activity Diagrams 22
3.2.1 Use Case Diagram 22
3.2.2 Use Case Description 23
3.2.3 Activity Diagram 25
3.2.4 Activity Diagram Description 27
CHAPTER 4 SYSTEM DESIGN **28**

4.1 System Block Diagram 28
4.1.1 Signature Generation Block 28
4.1.2 Signature Verification Block 30
4.2 Parallelization Strategy 32
4.2.1 Coarse-Grained Parallelism (Batch Processing) 32
4.2.2 Fine-Grained Parallelism (Thread Cooperation) 33
4.3 Proposed Optimization Techniques 34
4.3.1 Moving External CPU Loop to GPU Kernel 34
4.3.2 Kernel Fusion to Reduce Memory Overhead 36
4.3.3 Shared Memory Tiling for Finite-Field Operations 38
4.3.4 Hardware-Accelerated Tensor Core Integration (WMMA) 42
CHAPTER 5 SYSTEM IMPLEMENTATION 44
5.1 Set-up and Key Expansion 44
5.2 Batched Signature Generation 46
5.2.1 Fused Kernel Operations 46
5.2.2 Batched LU Decomposition 48
5.2.3 Tensor Core Implementation 50
5.2.4 Hybrid CPU-GPU Fallback (Rare Case) 52
5.3 Batched Signature Verification 54
5.3.1 CUDA Core Verification Variant 54
5.3.2 Tensor Core Verification Variant 56
CHAPTER 6 SYSTEM EVALUATION AND DISCUSSION 57
6.1 Hardware and Software Setup 57
6.2 Performance Metrics Definition 58
6.3 Evaluation of Signature Generation Optimizations 60
6.3.1 Performance Gain Moving from CPU to Separated Kernel GPU 64
6.3.2 Performance Gain from Separated Kernel GPU to Kernel Fusion 66
6.3.3 Tensor Core Acceleration vs. Standard CUDA Cores 67
6.3.4 Impact of Hybrid CPU-GPU Fallback (Rare Case) 69
6.4 Signature Verification Results 71
CHAPTER 7 CONCLUSION AND RECOMMENDATION 74
7.1 Conclusion 74
7.2 Recommendations for Future Work 75
REFERENCES 77

APPENDIX A **78**
QRUOV_Sign_Separated 78
QRUOV_Sign_Tensor 100
QRUOV_Sign_Fused 124
QRUOV_Verify 145
QRUOV_Tensor 153
POSTER **165**

LIST OF FIGURES

Figure Number	Title	Page
Figure 3.1	System Architecture Diagram of the Hybrid GPU-Accelerated QR-UOV Scheme	17
Figure 3.2	Use Case Diagram of the GPU-Accelerated QR-UOV System	22
Figure 3.3	Signature Generation Activity Diagram	25
Figure 3.4	Signature Verification Activity Diagram	262
Figure 4.1	Signature Generation Block	28
Figure 4.2	Signature Verification Block	30
Figure 4.3	Algorithm 1: Batched CPU to GPU Kernel Transformation	35
Figure 4.4	Algorithm 2: Kernel Fusion for Finite-Field Matrix Operations	37
Figure 4.5	Algorithm 3: Shared Memory Tiling for Batched LU Decomposition	39
Figure 4.6	Data Flow Comparison of Naive vs Shared Memory Optimized LU Decomposition	40
Figure 4.7	Algorithm 4: Tensor Core (WMMA) Integration for Finite-Field Matrices	43
Figure 5.1	Algorithm 5: Warp-Level Parallel Reduction via __shfl_down_sync()	47
Figure 5.2	Algorithm 6: Hybrid CPU-GPU Fallback Controller for Singular Matrices	53
Figure 5.3	Algorithm 7: Batched Signature Verification Execution	54
Figure 6.1	Throughput Comparison between Baseline CPU and GPU_Separated Architectures	64
Figure 6.2	Throughput Comparison between Fused GPU and Separated GPU Implementations	66
Figure 6.3	Throughput Comparison between Fused CUDA Cores and Tensor Cores (WMMA)	67
Figure 6.4	Verification Throughput Comparison between Standard CUDA and Tensor Core Architectures	72

LIST OF TABLES

Table Number	Title	Page
Table 3.1	Components and Functionalities of the Heterogeneous System Architecture	21
Table 3.2	Use Case Description of Generate Batch of Signatures	23
Table 3.3	Use Case Description of Verify Batch of Signatures	24
Table 6.1	Specifications of laptop	57
Table 6.2	Summary of System Performance Metrics	59
Table 6.3	Overall Benchmark Results for baseline CPU	60
Table 6.4	Overall Benchmark Results for GPU Signature Generation without kernel fused	61
Table 6.5	Overall Benchmark Results for Fused GPU Signature Generation	62
Table 6.6	Overall Benchmark Results for Tensor Core Signature Generation	63
Table 6.7	Frequency and CPU Resolution Time of Rank-Deficient Matrices	69
Table 6.8	Benchmark Results for Signature Verification Architectures	71

LIST OF ABBREVIATIONS

AES-CTR	Advanced Encryption Standard in Counter Mode
ALU	Arithmetic Logic Unit
API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRBG	Deterministic Random Bit Generator
ECC	Elliptic Curve Cryptography
FLOPs	Floating-Point Operations Per Second
FP16	Half-Precision Floating-Point (16-bit)
FP32	Single-Precision Floating-Point (32-bit)
GPU	Graphics Processing Unit
INT8	8-bit Integer
IoT	Internet of Things
LU	Lower-Upper (Decomposition)
MAC	Multiply-Accumulate
MQTT	Message Queuing Telemetry Transport
NIST	National Institute of Standards and Technology
PCIe	Peripheral Component Interconnect Express
PQC	Post-Quantum Cryptography
PQC-DSA	Post-Quantum Cryptography Digital Signature Algorithm
PRG	Pseudo-Random Generator
QR-UOV	Quotient Ring Unbalanced Oil and Vinegar

RAM	Random Access Memory
RSA	Rivest-Shamir-Adleman
SIMT	Single Instruction, Multiple Threads
SM	Streaming Multiprocessor
SRAM	Shared Random-Access Memory
TLS	Transport Layer Security
VRAM	Video Random-Access Memory
WMMA	Warp Matrix Multiply Accumulate

CHAPTER 1 INTRODUCTION

1.1 Problem Statement and Motivation

The rapid evolution of quantum computing has moved from theoretical curiosity to a systemic threat, casting a long shadow over the foundations of modern digital security. Existing classical cryptographic systems such as RSA and Elliptic Curve Cryptography (ECC) which heavily rely on mathematical structures can easily be solved by a sufficiently powerful quantum computer in a fraction of the time nowadays [1][2]. This vulnerability has given rise to the "Harvest Now, Decrypt Later" strategy where adversaries collect encrypted communications today with the intent of decrypting them once quantum hardware matures [3][4]. To counter this, the transition to Post-Quantum Cryptography (PQC) is no longer a distant goal but an urgent necessity [5]. Among the candidates for this new standard, the Quotient Ring Unbalanced Oil and Vinegar (QR-UOV) scheme stands out for its strong security, however, its practical adoption is currently stalled by the significant computational bottlenecks on traditional hardware.

The primary appeal of QR-UOV lies in its strong quantum resistance and its relatively compact signature sizes which are essential for maintaining efficiency in digital communications. Yet, these security benefits come at a high computational cost. The algorithm’s architecture is defined by the heavy reliance on dense matrix operations, multivariate polynomial evaluations, and complex linear algebra routines such as LU decomposition with partial pivoting and QR decomposition. When executed on general-purpose CPU-based platforms, these operations lead to excessive processing demands. For time-sensitive and high-volume environments such as Internet of Things (IoT) gateways, financial transaction systems, and cloud servers, this overhead results in unacceptable latency and throughput bottlenecks.

The challenge of deploying QR-UOV is further complicated by the reality of hardware limitations. While modern Graphics Processing Units (GPUs) offer the massive parallelism needed to accelerate such workloads, the mathematical structure of QR-UOV does not naturally map to a parallel architecture. Specifically, the scheme contains inherent sequential dependencies. Tasks like pivot selection and row swapping in LU decomposition create execution "synchronization points" that can leave GPU resources underutilized in accordance

with Amdahl's Law. Therefore, the successful deployment of QR-UOV requires more than just raw power, it demands an optimized and parallelized architecture capable of intelligently managing these structural bottlenecks.

To address these problems, this project explores the effective mapping of QR-UOV onto GPU platforms using the NVIDIA CUDA framework. By investigating both coarse-grain and fine-grain parallelism, this research identifies strategies to maximize hardware utilization and optimize memory management. Moreover, applying specialized hardware features like Tensor Cores are also able to speed up the dense matrix-multiply-accumulate operations at the core of the scheme. These optimizations are important for ensuring that organizations do not have to choose between strong security and system performance, avoiding a trade-off that often leads to delayed adoption or weakened security configurations.

In conclusion, the path to a quantum-secure future depends on the synergy between advanced mathematical schemes and high-performance hardware. By advancing parallel implementation techniques for multivariate-based cryptographic schemes like QR-UOV, it can enable faster, more practical, and sustainable security solutions. This project contributes to a vital area of digital infrastructure, ensuring that the next generation of secure connections can withstand the quantum threat without sacrificing the speed and efficiency required by the modern world.

1.2 Objective

This project aimed to utilize GPU parallelism via the NVIDIA CUDA framework to enhance the performance, speed and throughput of the QR-UOV signature generation and verification processes. By optimizing core operations such as finite-field polynomial arithmetic and dense matrix manipulations, the implementation targeted a minimum 5-fold speedup in throughput compared to sequential CPU baselines. To thoroughly evaluate and maximize system scalability, the architecture was designed to dynamically process exponentially increasing batch sizes. This allowed workloads to scale automatically up to the GPU's maximum available memory (VRAM) limit while continuously profiling execution latency.

These performance gains were driven by a highly efficient architecture that uses advanced CUDA programming paradigms including kernel fusion, warp-level reductions, and shared memory tiling alongside specialized hardware such as Tensor Cores. To maximize hardware utilization, the system was optimized to maintain at least 80% GPU multiprocessor occupancy and achieve a 40% reduction in average execution time for core operations by tuning thread block dimensions and minimizing global memory access overhead. Furthermore, a major architectural contribution of this project was the development of a dynamic hybrid CPU-GPU execution model designed to overcome algorithmic bottlenecks. This model processed optimal full-rank matrices natively on the GPU while offloading complex rank-deficient LU decomposition cases to the host CPU and thereby preventing stalls in the parallel execution pipeline.

Throughout the optimization process, strict validation was conducted to ensure that the GPU-accelerated implementation strictly maintained the mathematical exactness, cryptographic correctness, and post-quantum security guarantees of the original QR-UOV scheme. Lastly, the project delivered comprehensive benchmarking results, detailing throughput (operations per second) and processing latency (milliseconds per batch) providing a strong and practical foundation for future research and the integration of post-quantum cryptography into high-volume environments.

1.3 Project Scope

This project focused on accelerating the QR-UOV signature scheme by exploiting the parallel processing capabilities of modern GPU architectures. The scope encompassed a detailed analysis and architectural redesign of QR-UOV's signature generation and verification algorithms to support batched workload execution. Specifically, the project targeted their linear algebra operations over finite fields and quotient rings to identify parallelization opportunities and resolve major computational bottlenecks.

Building on insights from existing GPU-accelerated post-quantum schemes such as Falcon [6] [7], the primary technical effort involved translating QR-UOV's mostly sequential computations into parallel workloads executable by thousands of CUDA threads. A significant addition to this scope was the design of a hybrid CPU-GPU execution model. This model was implemented to manage inherent sequential bottlenecks, specifically by offloading the "rare case" of rank-deficient matrices during LU decomposition to the host CPU while the GPU concurrently processed full-rank matrices. Furthermore, a strong emphasis was placed on optimizing memory access patterns and efficiently utilizing GPU memory hierarchies including shared, global, and register memory to minimize conflicts, reduce latency, and maximize overall throughput.

Additionally, a comprehensive performance evaluation was conducted by benchmarking the batched GPU-accelerated implementation against CPU-based versions across various batch sizes and parameter sets. Metrics such as signature generation speed, verification time, overall throughput (signatures processed per second), and resource utilization were analyzed to ensure high efficiency and hardware compliance. Correctness and security validation involved thorough testing to confirm that the batched parallel execution did not compromise cryptographic integrity, produce incorrect outputs, or introduce vulnerabilities.

On the other hand, this project did not involve the development of new cryptographic primitives or mathematical modifications to the underlying QR-UOV algorithm itself. Moreover, the project did not address side-channel or fault attack resistance as the primary focus remained strictly on performance acceleration and throughput optimization. Broader integration with unrelated cryptographic libraries, user interface development, or deployment in commercial production environments also remained outside the project's boundaries.

1.4 Contributions

This project makes an important and timely contribution to the field of post-quantum cryptography by addressing the critical challenge of enhancing the practical performance of the QR-UOV digital signature scheme through GPU acceleration. In a world that rapidly moving toward the quantum computing era where present cryptographic standards face the imminent risk of being broken, this project offers a feasible route toward efficient and secure quantum-resistant signature schemes. This work benefits a diverse audience including cryptographers, cybersecurity practitioners, researchers and organizations that rely on secure digital communications, eventually helping to protect sensitive information and digital infrastructure with long-term societal impact.

The core problem tackled in this research revolves around the high computational overhead and latency inherent in QR-UOV implementations on traditional CPU architectures. Solving this bottleneck is highly relevant because it directly determines the practicability of deploying post-quantum cryptography in real-time and high-throughput environments such as financial systems, communication networks, and vital government infrastructure. To address this, this project explored the parallelization of QR-UOV using GPUs. This approach is particularly effective due to the algorithm's unique mathematical complexity which heavily involves multivariate polynomial evaluations and matrix operations that benefit immensely from GPU parallelism and optimized memory usage.

By bridging the gap between strong quantum-resistant algorithms and practical deployment constraints, this project provides pragmatic and implementable technical solutions. Among these is the development of a functional and highly optimized CUDA implementation which capable of executing both signature generation and verification across large concurrent batches. Furthermore, the project introduces a novel hybrid CPU-GPU execution architecture to overcome the severe sequential bottlenecks of LU decomposition with partial pivoting. By designing a system where the GPU processes bulk, full-rank matrix computations while dynamically offloading rare and rank-deficient mathematical edge cases to the host CPU, the implementation maximizes hardware utilization without sacrificing cryptographic correctness. These architectural strategies are further enhanced by kernel fusion techniques and shared memory optimizations that drastically reduce global memory latency.

Besides, this project is also highly valuable to the academic and industrial community because it delivers measurable performance improvements for a credible cryptographic scheme currently under consideration in the NIST PQC standardization process. It offers a comprehensive optimization framework that includes GPU memory mapping and thread block configurations as well as extensive performance benchmarks across various security levels and batch sizes. These comparative evaluations demonstrate clear throughput and efficiency gains, making them an important reference for future PQC implementations. Moreover, understanding how to apply modern GPU features to achieve high thread occupancy in complex cryptographic computations has broader implications beyond QR-UOV as it may influence the design of other multivariate schemes and parallel computation tasks.

In summary, by emphasizing the critical need for efficient cryptographic implementations, this project presents a meaningful step toward building a strong quantum-resistant cybersecurity ecosystem. It addresses a key challenge in securing digital communications against future quantum threats by drastically improving the efficiency of the QR-UOV scheme on parallel hardware. By providing practical and optimized solutions that make quantum-resistant cryptography more accessible and usable in real-world systems, this project offers valuable insights and tools that readers can apply to advance secure cryptographic technologies for the quantum era and hence protecting users from the unpredictable threats posed by future quantum computers.

1.5 Report Organization

This report is organized into seven chapters that systematically detail the research, design, implementation and evaluation of the project. Chapter 1 introduces the project by presenting the problem statement, motivation, objectives, scope and key contributions to the field of post-quantum cryptography. Following this, Chapter 2 provides a literature review that analyzes existing studies on the GPU acceleration of other post-quantum cryptographic schemes and identifying the current research gap regarding the QR-UOV algorithm. Chapter 3 outlines the system methodology and approach including the overall architectural design, system models, and use cases for the batched cryptographic operations. Next, Chapter 4 dives deeper into the system design describing the top-down architecture, GPU memory hierarchy mapping, thread block configurations, and the specific parallelization strategies formulated for both signature generation and verification. Chapter 5 then discusses the system implementation, translating the system design into working software by discussing the specific CUDA programming techniques utilized such as kernel fusion, batched LU decomposition, and the novel hybrid CPU-GPU fallback logic. Chapter 6 details the system evaluation and discussion which comprises a comprehensive analysis of the testing environment, performance metrics, execution times and throughput results gathered from the batched benchmarks. Finally, Chapter 7 concludes the report by summarizing the project's achievements in relation to its initial objectives and providing recommendations for future research and hardware optimizations.

CHAPTER 2 LITERATURE REVIEW

2.1 Previous Studies

2.1.1 Optimization of Falcon for GPUs

Li et al. [6] proposed a journal article in 2025 that presents a huge advancement in accelerating Falcon lattice-based digital signature scheme using GPU parallel computing. Falcon is a compact lattice-based signature scheme that requires computationally intensive which limits its practical deployment in applications requiring both security and efficiency. Prior implementations of Falcon, especially in CPU-based had suffered from low throughput and earlier GPU implementations have not fully exploited adaptive parallelism or optimized key Falcon components for GPU architectures.

To overcome such limitations, the authors introduce cuFalcon which applies adaptive parallel strategies that are modified from Falcon's algorithmic structure for GPU architecture. At the operational level, cuFalcon introduces a memory-efficient Fast Fourier Transform (FFT) that optimizes GPU memory hierarchies. The authors also propose an adaptive parallel Fast Fourier Sampling (ffSampling) to accelerate signature generation. A compact computation model is also applied across operations to reduce memory footprint. At the signature level, cuFalcon offers three implements variant versions which include raw key, expanded key and balanced version of cuFalcon that can offer trade-offs between speed and memory usage. For additional optimizations include batching and streaming mechanisms and secure memory pool is done to efficiently handle multiple concurrent signature requests.

The strengths of cuFalcon's approach are evident in its performance metrics. The raw key version achieves 172k signings per second while the expanded key version reaches 201k signatures per second. This result has outperformed the Falcon reference CPU implementation by 36.75 times and the prior best GPU implementation by nearly 3 times. Moreover, the balanced version of cuFalcon also reduces memory usage by 70% compared to the expanded key version while improving throughput by 7% which makes it suitable for deployment in memory-constrained environments.

Despite having these advantages, cuFalcon's approach is not without limitations. The trade-offs between memory consumption and throughput require careful tuning, which may complicate deployment in highly resource-constrained environments. The implementation is

tightly coupled with NVIDIA's CUDA architecture which limits its portability to other GPU platforms such as AMD or integrated GPUs. In addition, the complexity introduced by adaptive parallelism and multiple implementation variants may increase the maintenance burden and necessitate expert knowledge for optimal tuning.

In summary, cuFalcon shows an important step forward in the high-performance implementation of post-quantum digital signatures. By solving the computational restriction of Falcon on GPUs through adaptive parallel strategies and architectural optimizations, the authors have set a new benchmark for throughput and efficiency in PQC acceleration. While challenges remain in terms of portability, complexity and further optimization, cuFalcon's innovations provide a strong foundation for future research and deployment in secure, quantum-resistant systems.

2.1.2 GPU-based Parallel Acceleration of SPHINCS+ (GRASP)

The research paper from Ning et al [1] published in 2021 demonstrated that the optimized implementation of SPHINCS+ achieved a notable performance increase. SPHINCS+, is a post-quantum cryptography Digital Signature Algorithm (PQC-DSA) with distinct feature of very short public and private key length compared to other PQC. However, this PQC also has the same issue of slow performance in terms of signing and verification operations which limit its real-world applicability. To resolve this, the authors proposed a solution using GRASP, is a GPU-based parallel acceleration framework that utilizes CUDA for efficient parallelization of SPHINCS+ operations.

The authors approach those issues by analyzing the algorithm's structure of SPHINCS+ to identify the computationally intensive components which are suitable for parallel execution. It then implements adaptable parallelization strategies using CUDA, allowing the framework to be tuned for either maximizing throughput or minimizing latency depending on application requirements. The key optimizations include improved memory access patterns, bottom-up hypertree computation enhancements, and kernel fusion techniques that reduce GPU kernel launch overhead and improve resource utilization. Through these innovations, GRASP able achieves performance improvements ranging from 1.37 times to 3.45 times over prior GPU-based implementations and outperforms the NIST reference implementation by more than three orders of magnitude.

The strengths of GRASP's solution lie in its acceleration of SPHINCS+ operations, which makes the scheme more practical for deployment in environments demanding high-speed cryptographic operations. Its adaptable parallelization framework which offers flexibility enabling users to utilize the GPU thread configurations to their specific needs whether prioritizing throughput or latency. Furthermore, the comprehensive optimization approach that combines algorithmic insights with hardware-level enhancements ensures effective utilization of GPU resources.

Despite these notable strengths, GRASP has its certain limitations. Its performance gains are heavily dependent on accessing modern high-performance GPUs, which may not be available in all deployment scenarios. Additionally, achieving optimal performance requires careful tuning of parallelization parameters and memory layouts which make the process complex and highly specific to the underlying hardware. The framework's reliance on CUDA

also limits its portability to other parallel computing environments such as AMD GPUs or FPGA platforms, where re-engineering efforts would be necessary. Moreover, as workloads scale, GPU memory and resource constraints could become the limitations when increasing thread counts or handling very large data sets.

In conclusion, the GRASP framework shows that implementing PQC on GPU architectures able to bring advancement in accelerating. By effectively exploiting GPU parallelism and introducing adaptable, comprehensive optimizations, GRASP makes SPHINCS+ more viable for practical applications which require strong quantum-resistant digital signatures.

2.1.3 High Throughput Lattice-based Signatures on GPUs

The paper by Lee et al. [7] presents a pioneering study on high-throughput implementations of the lattice-based signature schemes using Falcon and Mitaka on Graphics Processing Units (GPUs) through targeting the acceleration of post-quantum cryptographic algorithms. Falcon, a compact lattice-based signature scheme standardized by NIST relies heavily on the Fast Fourier Sampling (ffSampling) algorithm for signature generation. However, ffSampling poses lot of challenges for GPU implementation due to its inherently recursive nature and data dependencies that limit parallelization. Previous implementations of Falcon were mostly optimized on CPUs using SIMD instructions (AVX2) and microcontrollers but fell short of leveraging the massive parallelism offered by GPUs.

To address these limitations, the authors proposed an iterative version of the ffSampling algorithm which cleverly emulating recursive stack management to eliminate costly dynamic parallelism overhead on GPUs. They parallelize key Falcon operations such as FFT, NTT, and polynomial arithmetic, and introduce a kernel-fusion technique to reduce GPU kernel launch overheads. The paper also pioneers the first GPU implementation of Mitaka, a recent parallelizable variant of Falcon that replaces ffSampling with a hybrid Gaussian sampler better suited for parallel execution. Mitaka's sampling process is divided into batch random sample generation and rejection sampling phases, enabling efficient GPU parallelism.

The authors evaluate their implementations on multiple GPU architectures and able to demonstrate that Mitaka can achieve up to 20 times higher signature generation throughput than Falcon due to its parallel-friendly design. Meanwhile, Falcon delivers superior verification performance because of its integer-domain computations using Number Theoretic Transform (NTT) which GPUs can execute more efficiently than Mitaka's floating-point FFT-based verification. The work also includes extensive micro-benchmarking of fused kernel executions and showing substantial reductions in execution time. Their implementations outperform existing CPU-based AVX2 and floating-point reference implementations by significant margins.

Despite these breakthroughs, the authors acknowledge that Mitaka's verification performance can be further enhanced by porting it entirely to the integer domain and replicating Falcon's efficient approach. The complex trade-offs in design decisions between sampling

algorithms and domain arithmetic highlight the intricate relationship between cryptographic scheme structure and parallel hardware efficiency.

In summary, this work establishes the first comprehensive GPU-accelerated implementations of Falcon and Mitaka by overcoming known parallelism bottlenecks in lattice-based PQC signatures. By innovating iterative ffSampling and batch sampling strategies combined with GPU architectural optimizations, the authors able to set a new performance benchmark for post-quantum digital signatures on massively parallel platforms. Their contribution is not only able to provide critical insights and foundations for accelerating PQC candidates on GPUs but also pointing toward future improvements in design portability and verification acceleration.

2.1.4 High Throughput Acceleration of Cryptography Using Tensor Core

The journal article "TensorCrypto: High Throughput Acceleration of Lattice-based Cryptography Using Tensor Core on GPU" from Lee et al [8] address the performance hold-up in lattice-based cryptography, specifically the computationally intensive polynomial convolution operation, which is central to many post-quantum cryptosystems. Traditional implementations on CPUs and even standard GPU integer units struggle to deliver the high throughput necessary for large-scale applications like IoT gateways and cloud servers, which must handle massive numbers of secure connections simultaneously. The core problem is that existing GPU implementations do not exploit the full potential of NVIDIA's tensor cores, which are specialized hardware units designed for fast matrix multiplication.

To solve this, the authors propose and implement a tensor-core-based approach to accelerate polynomial convolution on GPUs for the first time. They develop parallel algorithms to adapt polynomial operations into matrix multiplications suitable for tensor cores, overcoming challenges such as mismatched polynomial degrees using techniques like zero padding, sign conversion, and type casting. Their solution is validated on NTRU, LAC, and FrodoKEM cryptosystems, demonstrating outstanding throughput improvements compared to conventional GPU implementations.

One of the most important strengths of this work is the impressive performance gains realized by the tensor-core-based implementation. For polynomial convolution with degree 1024, the authors report speedups of up to 3.41 times on the RTX2060 GPU and 5.77 times on the RTX3080 compared to traditional integer-based GPU implementations. When applied to complete cryptographic operations such as encapsulation and decapsulation in NTRU, throughput improvements of up to 2.02 times and 1.56 times respectively were observed. These results highlight the effectiveness of harnessing tensor cores for lattice-based cryptography, marking a considerable leap forward in practical post quantum cryptographic performance. Furthermore, the author's method also achieves broad applicability of the approach. They demonstrate that their solution is not limited to a single cryptosystem but can accelerate a range of lattice-based schemes with small modulus values, including both ideal lattice constructions like NTRU and LAC, as well as generic lattice schemes such as FrodoKEM. The parallel algorithm design allows flexible handling of different polynomial sizes and ephemeral key pairs, making the solution adaptable to various cryptographic parameters.

Since everything is not perfect, the solution has some limitations. Primarily, it is restricted to lattice-based schemes with small modulus values where polynomial multiplication is not already efficiently handled by Number Theoretic Transform (NTT) methods. For popular schemes like KYBER and NewHope, which utilize NTT and often involve larger modulus sizes, the tensor core approach is less effective or not applicable due to the precision constraints of current tensor core hardware. NVIDIA tensor cores are optimized for half precision (16-bit) floating-point operations, whereas many cryptographic schemes require higher precision arithmetic to maintain security and correctness. This precision mismatch limits the direct applicability of tensor core acceleration to a subset of lattice-based cryptosystems. Additionally, the need to pad polynomials to multiples of 16 and perform sign conversions introduces overhead that can reduce efficiency, especially when polynomial degrees do not align well with tensor core matrix sizes. Lastly, this concludes that the approach is not universally applicable to all lattice-based cryptography.

In conclusion, the work presented by the authors represents a significant advancement in accelerating lattice-based cryptography by exploiting the specialized capabilities of GPU tensor cores. It delivers substantial throughput improvements for a class of post-quantum cryptosystems and opens new pathways for hardware-software co-design in cryptographic computing. While current limitations restrict its universal applicability, ongoing developments in hardware and algorithmic techniques definitely can expand the scope and effectiveness of this innovative approach.

2.2 Limitation of Previous Studies

Previous studies and published papers have demonstrated that PQC algorithms can be effectively implemented on GPU architecture. However, there is a notable gap in the existing research regarding the implementation of the QR-UOV signature scheme on GPU architecture. While QR-UOV has been studied and specified in detail including its key generation, signature generation, and verification algorithms, and some software implementations have been explored with CPU-based parallelization techniques such as OpenMP [9], however there is no prior work that has been explicitly addressed or reported on leveraging GPU architectures for accelerating QR-UOV computations.

Existing literature on GPU implementations in cryptography primarily focuses on other schemes such as SPHINCS+ and Falcon where GPU-based acceleration has shown notable performance improvements. Additionally, GPU implementations of QR decomposition algorithms which are mathematically related to some operations in QR-UOV have been studied extensively, highlighting the challenges and architectural considerations for efficient GPU utilization. However, these studies do not extend to the QR-UOV scheme itself.

Therefore, the absence of research on GPU-based implementations of QR-UOV represents a limitation in the current body of knowledge, indicating an opportunity for future work to explore this avenue for performance enhancement in post-quantum signature schemes.

CHAPTER 3 SYSTEM METHODOLOGY/APPROACH

3.1 System Design Diagram/Equation

The proposed system adopts a heterogeneous computing approach with the CPU providing sequential control and fallback mechanisms while the GPU performing massively parallel and compute-intensive cryptographic operations.

3.1.1 System Architecture Diagram

The system architecture is divided into two primary environments which are the Host (CPU) and the Device (GPU). The host acts as the central controller responsible for memory allocation, initializing Pseudo-Random Generators (PRG), and handling rare sequential bottlenecks. The device acts as a cryptographic accelerator with processing thousands of matrix operations concurrently through batched CUDA kernels. Communication between the host and device is facilitated via the Peripheral Component Interconnect Express (PCIe) bus using cudaMemcpy operations.

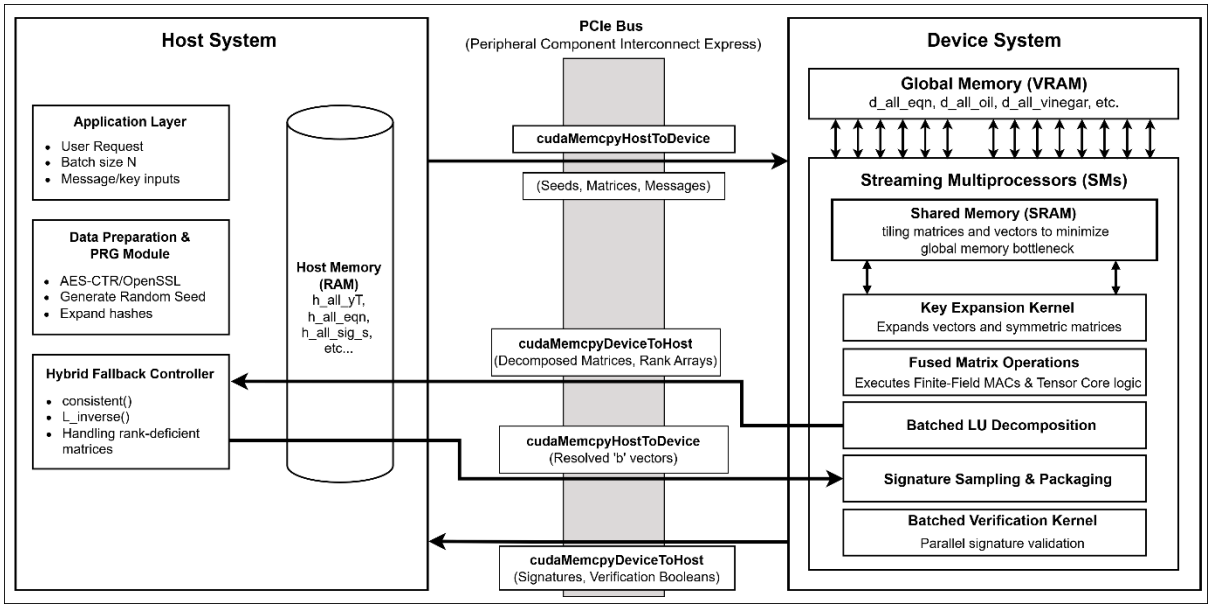


Figure 3.1: System Architecture Diagram of the Hybrid GPU-Accelerated QR-UOV Scheme

As illustrated in Figure 3.1, execution begins on the host side which prepares arrays of seeds, messages, and cryptographic variables based on the user-defined batch size (N). This data is transferred to the device's global memory. The execution is then divided into two synchronous phases to handle rank-deficient matrices efficiently without introducing GPU thread divergence.

In the first phase, the GPU executes batched kernels to generate the linear systems and perform parallel LU decomposition across grids and thread blocks. The rank statuses and decomposed matrices are then copied back to the host. At this point, the CPU acts as a hybrid fallback controller. It iterates through the batch, computing the target vector b for full-rank matrices. If the CPU identifies a rank-deficient matrix, it triggers a fallback loop repeatedly re-sampling the salt and hashing the message until a mathematically consistent vector b is found for that specific system. Once the host CPU has resolved the vectors for the entire batch, they are transferred back to the device. In the second phase, the GPU resumes parallel execution to sample the solutions and output the final packaged signatures.

3.1.2 Overview of the Heterogeneous Architecture

The proposed system architecture was designed using a heterogeneous computing model because the Quotient Ring Unbalanced Oil and Vinegar (QR-UOV) signature scheme involves a mix of highly parallelizable matrix multiplications and strictly sequential algebraic operations, hence a single-processor approach is insufficient. The architecture divides the computational workload between the Host (the Central Processing Unit) and the Device (the Graphics Processing Unit). This division ensures that massively parallel tasks are executed on the GPU while sequential and control-heavy tasks are managed by the CPU, preventing thread divergence and maximizing hardware occupancy.

3.1.3 Components and Functionalities of the Heterogeneous System Architecture

Environment / Component	Module / Sub-component	Functionality and Responsibility
Host Environment (CPU)	Data Preparation & PRG	Initialized the Pseudo-Random Generators (PRG) using AES-CTR via the OpenSSL library. Allocated contiguous memory blocks in system RAM to prepare data structures for GPU processing.
	Hybrid Fallback Controller	Processed the rank arrays returned from the GPU after batched LU decomposition. For rank-deficient (singular) matrices, the controller executed a fallback loop iteratively re-sampling salts and hashing the message until a mathematically consistent target vector was found, thereby maintaining cryptographic correctness without thread divergence
Communication Interface	PCIe Bus (PCI Express)	Acted as the high-speed bridge between Host and Device. Latency was minimized by utilizing bulk transfers (cudaMemcpy). During signature generation, data was transferred in two synchronous phases to support the hybrid fallback mechanism whereas batch verification required transfers only at the absolute beginning and end of the execution cycle.
Device Environment (GPU)	Global Memory (VRAM)	Stored large batched arrays (N x 54 x 54). The host calculated memory requirements dynamically to ensure the 75% VRAM threshold was not exceeded to prevent out-of-memory errors.

	Shared Memory (SRAM)	Utilized as a low-latency cache for finite-field matrix multiplications and LU decomposition. Data was loaded into shared memory tiles to allow threads to cooperatively compute dot products without global memory bottlenecks
	Compute Kernels (SMs)	The GPU executes customized CUDA kernels mapped to 1D and 2D grids. The Key Expansion Kernel expands the private and public keys concurrently. The Fused Matrix Operations combine multiple finite-field multiply-accumulate (MAC) operations to reduce kernel launch overhead. The Batched LU Decomposition kernel assigns individual thread blocks to solve independent equations simultaneously Following the host fallback phase, the Signature Sampling & Packaging kernels conclude the generation process by computing the final vectors in parallel. The Batched Verification Kernel bypasses sequential decomposition entirely, utilizing parallel reduction techniques at the warp level to verify thousands of signatures in a single, highly optimized pass.

Table 3.1: Components and Functionalities of the Heterogeneous System Architecture

3.2 Use Case & Activity Diagrams

3.2.1 Use Case Diagram

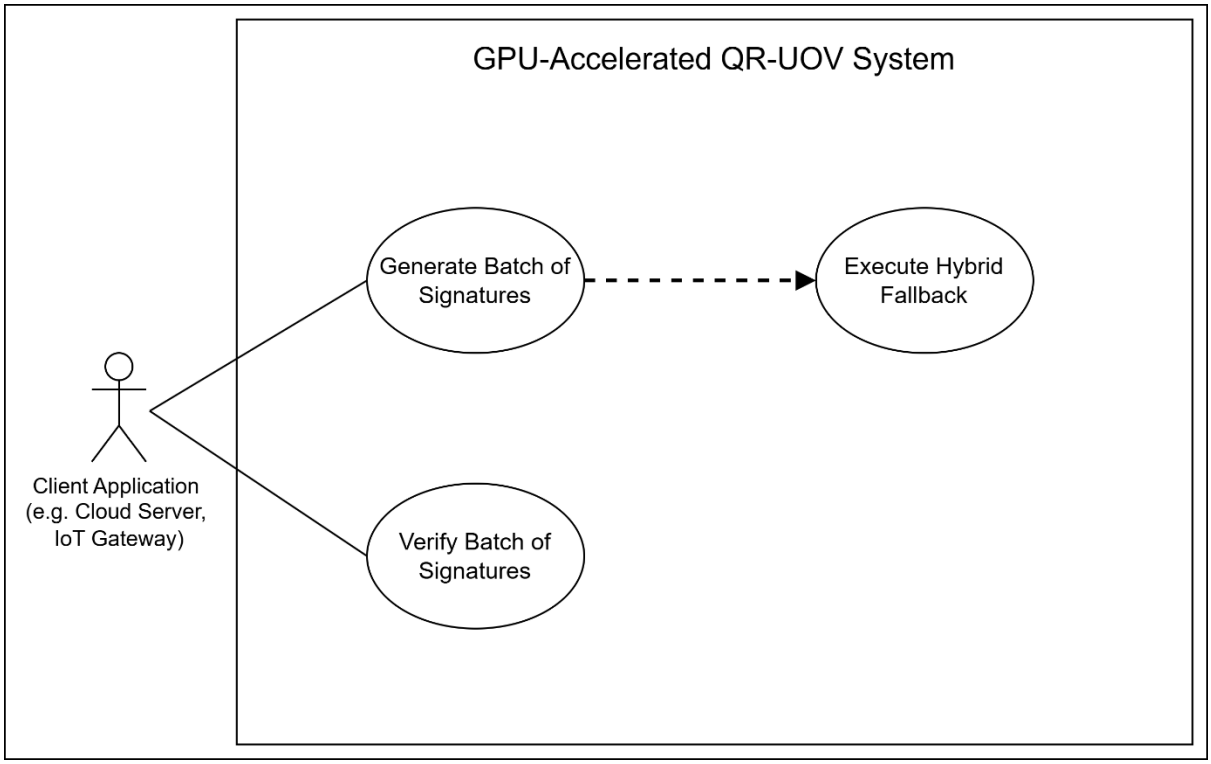


Figure 3.2: Use Case Diagram of the GPU-Accelerated QR-UOV System

The functional requirements of the GPU-accelerated QR-UOV system were modeled using use cases to define the interactions between the external client and the cryptographic acceleration engine. The system facilitated high-throughput signature operations through two primary use cases which are Batch Signature Generation and Batch Signature Verification. A critical internal use case (Execute Hybrid Fallback) was included to manage mathematical edge cases during the signing process.

3.2.2 Use Case Description

Use Case Name	**Generate Batch of Signatures**
Actor	Client Application
Description	The system receives a batch of *N* messages and private keys to generate a corresponding batch of QR-UOV digital signatures using a two-phase GPU execution model.
Pre-conditions	The GPU is initialized, and sufficient VRAM is available to accommodate the requested batch size without exceeding the 75% threshold.
Main Flow	1. Actor submits a batch size N, messages, and seed values. 2. CPU allocates memory and expands deterministic PRG seeds. 3. CPU transfers initialization data to the GPU (Phase 1). 4. GPU executes fused matrix multiplication and batched LU decomposition for the entire batch. 5. GPU returns the decomposed matrices and rank statuses to the CPU. 6. CPU iterates through the batch, computing standard target vectors for full-rank matrices. 7. CPU transfers the resolved target vectors back to the GPU (Phase 2). 8. GPU executes signature sampling and packaging. 9. System returns N valid signatures to the Actor.
Alternative Flow	6a. If the CPU identifies a rank-deficient matrix during its iteration in Step 6, it invokes the Execute Hybrid Fallback use case. 6b. The CPU iteratively re-samples the cryptographic salt and hashes the message until a mathematically consistent target vector is found. 6c. The consistent target vector is saved, and the CPU continues processing the remainder of the batch before proceeding to Step 7.
Post-conditions	*N* valid cryptographic signatures are generated and stored in host memory.

Table 3.2: Use Case Description of Generate Batch of Signatures

Use Case Name	**Verify Batch of Signatures**
Actor	Client Application
Description	The system receives a batch of *N* signatures, messages and public keys and verifies their authenticity concurrently.
Pre-conditions	Signatures and public keys must be properly formatted and available in host memory.
Main Flow	1. Actor submits a batch of signatures and original messages. 2. CPU hashes messages and transfers data to GPU. 3. GPU expands public key matrices. 4. GPU executes parallel matrix-vector verification operations. 5. System returns a boolean (Pass/Fail) array for the batch to the Actor.
Post-conditions	A boolean array indicating the validity of each signature is returned.

Table 3.3: Use Case Description of Verify Batch of Signatures

3.2.3 Activity Diagram

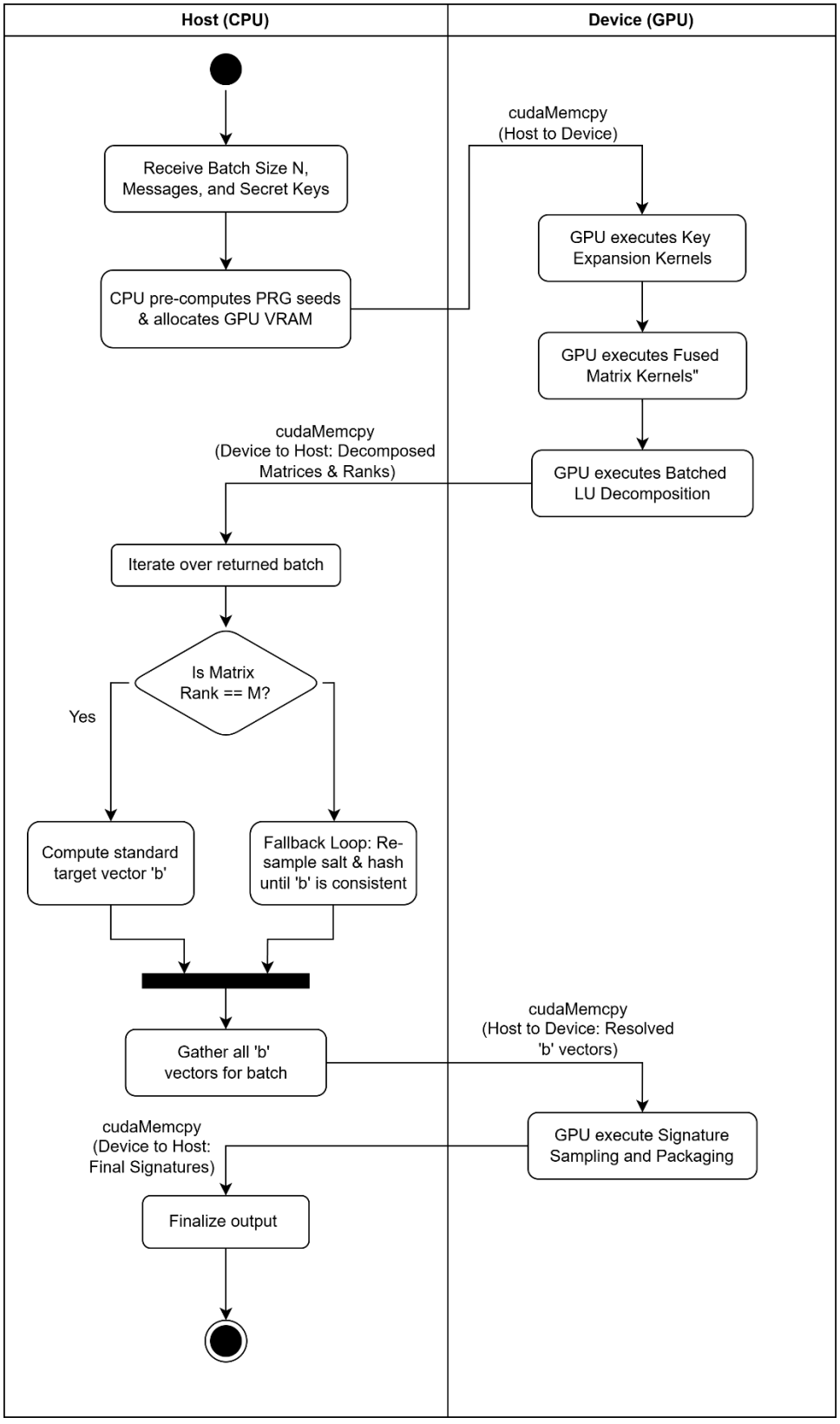


Figure 3.3: Signature Generation Activity Diagram

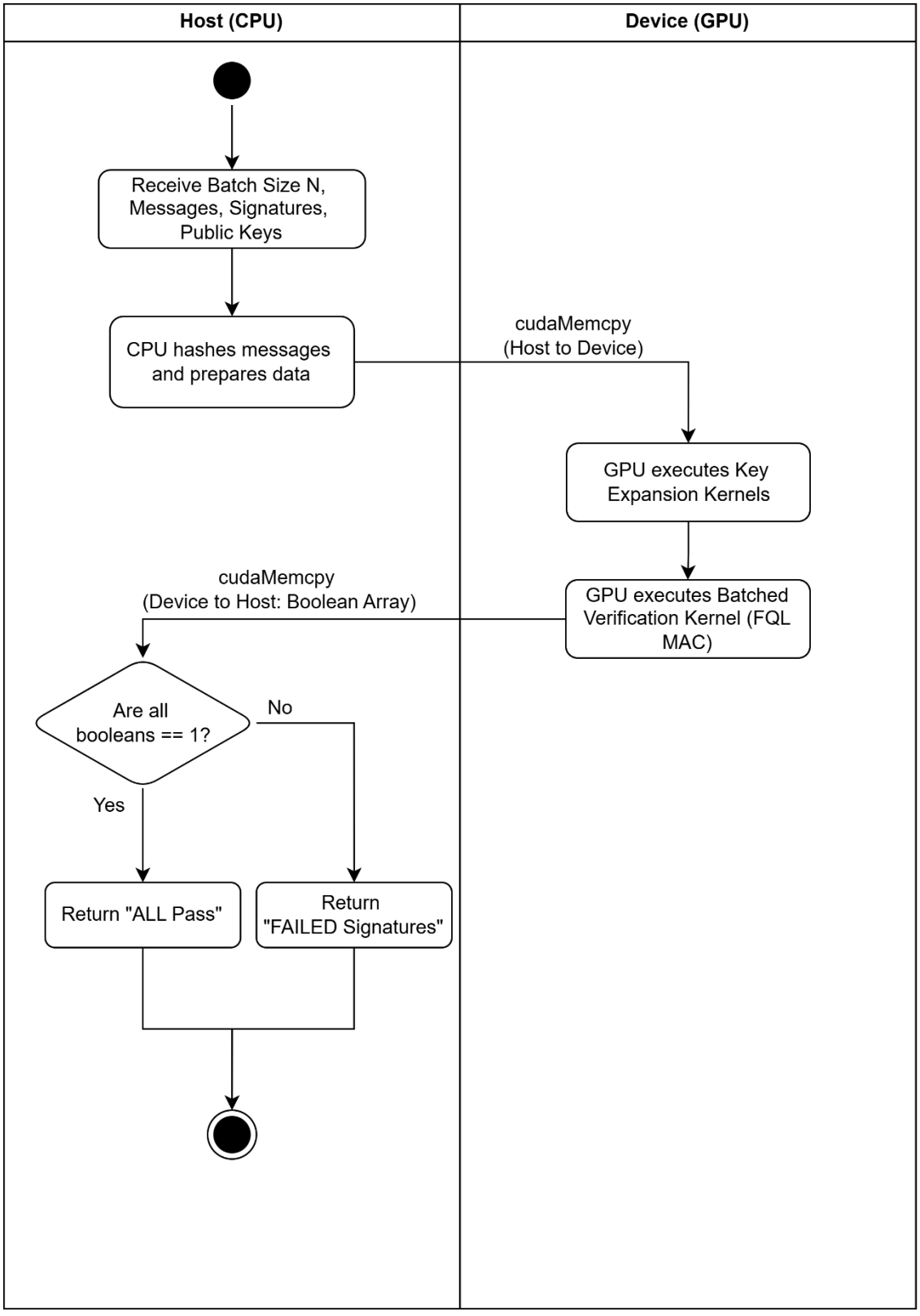


Figure 3.4: Signature Verification Activity Diagram

3.2.4 Activity Diagram Description

As shown in the activity diagrams, the execution flow was designed to maximize parallel occupancy on the GPU by processing data in batches. In the signature generation flow (Figure 3.3), the execution is divided into two synchronous phases to prevent severe GPU thread divergence. Following the batched LU decomposition on the device, the resulting rank array is transferred back to the host. Instead of forcing the GPU to execute the indeterminate, iterative do-while loop required to resolve target vectors for singular matrices which would cause severe warp divergence and penalty stalls within the Streaming Multiprocessors hence the architecture strategically shifts this fallback workload to the host CPU. The CPU handles standard vector computation for full-rank instances and executes the fallback loop for rank-deficient instances. Once the entire batch is mathematically resolved, the data is returned to the GPU for final signature sampling. This hybrid approach guarantees stability and correctness while optimizing the GPU's parallel throughput.

In contrast, the verification activity flow (Figure 3.4) was highly streamlined because verification in QR-UOV relied solely on multivariate polynomial evaluations (matrix-vector multiplications) over finite fields, it did not require the computationally intensive system solving complex LU decomposition that is necessary for signature generation. This simplified flow allowed the GPU to process the verification kernels in a single uninterrupted parallel pass, resulting in significantly higher overall throughput compared to signature generation.

CHAPTER 4 SYSTEM DESIGN

4.1 System Block Diagram

4.1.1 Signature Generation Block

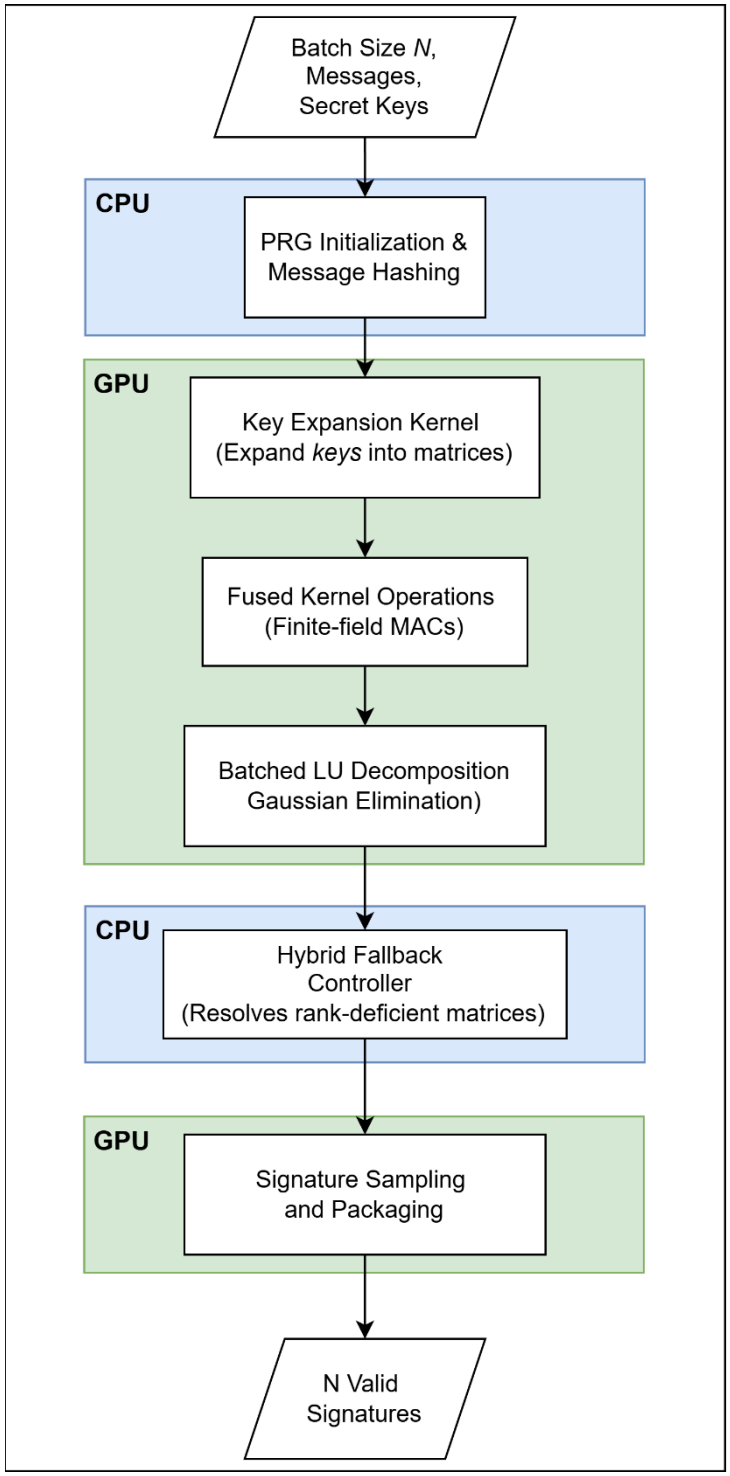


Figure 4.1: Signature Generation Block

Figure 4.1 illustrates the sequential execution pipeline for batched QR-UOV signature generation. The process begins on the host system, which receives the user-defined batch size *N* alongside arrays of original messages and secret keys. The CPU handles the initial data preparation which involves hashing the messages utilizing SHAKE-256 to generate the cryptographic digest and initializing the Pseudo-Random Generator (PRG) via AES-CTR. The AES-CTR PRG is utilized to derive randomized target vectors and cryptographic salts. Following this, the raw generated byte arrays are transferred to the GPU via a bulk PCIe transfer. To conserve PCIe bandwidth and accelerate formatting, the Key Expansion Kernel unpacks and expands these byte arrays into the massive evaluation matrices (for Secret key and Public Key) directly on the device's global memory.

Once the data is staged, the GPU executes the Fused Kernel Operati ons. This block merges multiple finite-field multiply-accumulate (MAC) operations into a single kernel, utilizing low-latency shared memory to bypass global memory bottlenecks. The generated linear equations are then passed to the Batched LU Decomposition kernel where Gaussian elimination is performed concurrently across *N* matrices using parallel thread blocks. After decomposition, a Hybrid Fallback Controller is triggered. The GPU returns the matrix rank statuses to the host CPU. If any matrix is identified as rank-deficient (singular), the CPU temporarily takes over to re-sample the salt and resolve the mathematical edge case via AES-CTR and SHAKE-256, thereby preventing severe warp divergence on the GPU. Finally, the resolved vectors are returned to the device, where the Signature Sampling and Packaging kernel finalizes the computations, outputting an array of *N* valid cryptographic signatures.

4.1.2 Signature Verification Block

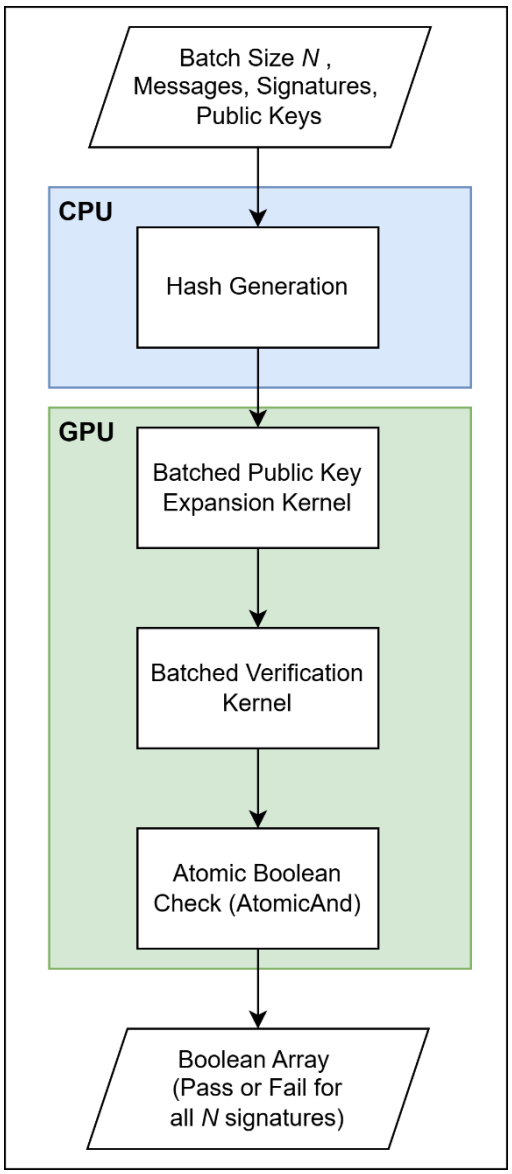


Figure 4.2: Signature Verification Block

Figure 4.2 shows the streamlined execution for batched QR-UOV signature verification. Unlike signature generation, verification does not require complex system solving like LU decomposition or CPU fallback logic, so it is allowed for a highly uninterrupted parallel flow. The pipeline initiates with the host receiving *N* messages, their corresponding signatures, and public keys. The CPU computes the SHAKE-256 hashes of the messages and utilizes AES-CTR to initialize and expand the public key seeds, then transferring the required arrays to the GPU in a single bulk transaction.

On the device, the Batched Public Key Expansion Kernel dynamically expands the raw PRG bytes into the massive public key evaluation matrices. The core computational workload is then handled by the Batched Verification Kernel which evaluates the multivariate polynomial equations by performing highly parallel matrix-vector multiplications over finite fields. Following the evaluations, an Atomic Boolean Check is executed. A designated thread within the warp utilizes atomicAnd hardware primitive to compare the computed polynomial evaluation against the original message hash, safely writing a 1 (Pass) or 0 (Fail) to the result array. Finally, the completed boolean array is transferred back to the host, indicating that the cryptographic validity of all *N* signatures simultaneously.

4.2 Parallelization Strategy

The challenges in accelerating the QR-UOV scheme lies in effectively mapping the sequential mathematical structures such as multivariate polynomial evaluations and dense linear algebra onto the massively parallel architecture of a GPU. To maximize the hardware occupancy of the Streaming Multiprocessors (SMs), a dual-level parallelization strategy was formulated. This strategy divides the cryptographic workload into coarse-grained and fine-grained parallelism.

4.2.1 Coarse-Grained Parallelism (Batch Processing)

Coarse-grained parallelism is achieved by processing multiple independent digital signatures concurrently across the GPU. In a traditional sequential implementation, the CPU processes one message at a time, repeatedly launching individual GPU kernels and triggering constant data transfers across the PCIe bus. This creates a severe bottleneck that underutilizes the massive computational throughput of modern graphics cards. To eliminate this overhead, the external processing loop is migrated entirely to the GPU. The workload is treated as a continuous batch of size N, where coarse-grained parallelism is strictly defined by mapping one independent signature to one independent computing unit on the device.

For computationally intensive operations involving dense linear algebra, this parallelism is implemented using a one block equals one signature mapping strategy. In high-complexity kernels such as LU_decompose_batched_kernel, the CUDA block index is mapped directly to a specific signature identifier. This architectural choice guarantees that if a batch of N=1024 signatures is submitted to the system, exactly 1024 independent thread blocks are launched across the Streaming Multiprocessors (SMs). As a result, the GPU is able to solve thousands of independent linear systems simultaneously without any inter-block dependencies or synchronization stalls.

Conversely, for lower-complexity and highly vectorized operations, dedicating an entire thread block to a single signature is computationally wasteful and lowers overall hardware occupancy. To optimize these stages, one thread equals one signature mapping strategy is deployed. In post-processing kernels such as fused_pack0_SIG_GEN_batched_kernel, a global thread index is utilized. This ensures that exactly one GPU thread handles the entirety of a single signature's final sampling and packaging. Under this configuration, a single block consisting of 256 threads can independently process 256 distinct signatures concurrently, maximizing resource utilization for lighter workloads.

4.2.2 Fine-Grained Parallelism (Thread Cooperation)

While coarse-grained parallelism effectively scales the workload across the entire GPU at a macro level, fine-grained parallelism operates at the micro level, strictly within the boundaries of a single signature. It relies on the threads inside a single block to cooperatively execute the underlying mathematical operations required for that specific signature. Rather than assigning a single thread to compute a complex equation sequentially, the workload is distributed across the warp and block, accelerating the heavy finite-field arithmetic intrinsic to the QR-UOV scheme.

A primary example of this thread cooperation is observed during the batched LU decomposition. When an entire block is assigned to process one $QRUOV\_m \times QRUOV\_m$ matrix, the threads within that block divide the matrix rows and columns among themselves. Instead of a single thread looping through the matrix to perform Gaussian elimination, the threads simultaneously execute row-elimination and finite-field scaling steps. This micro-level parallelism significantly reduces the execution time of solving the linear system as independent array elements are calculated and updated concurrently.

Furthermore, fine-grained parallelism is heavily utilized during the dense matrix-vector multiplications required for signature generation and verification. As multiple threads compute partial dot products concurrently using the finite-field multiply-accumulate (fql_mac) function, these partial sums must be aggregated. To achieve this efficiently, the architecture employs warp-level parallel reductions using the __shfl_down_sync() hardware primitive. This technique allows 32 threads within a warp to securely share and reduce their sums directly through the GPU's hardware registers. By keeping the reduction tree at the warp level, the implementation completely bypasses the severe latency and thread-stalling associated with shared or global memory atomic additions.

4.3 Proposed Optimization Techniques

4.3.1 Moving External CPU Loop to GPU Kernel

In traditional sequential implementations of the QR-UOV scheme, the host processes digital signatures one at a time. To generate a batch of N signatures, the CPU utilizes an external sequential for loop. Inside this loop, the CPU independently allocates local memory, expands the cryptographic keys, computes the dense LU decomposition, and samples the final signature for each message sequentially.

This pure CPU approach introduces a severe performance bottleneck. Because the execution is bound to sequential host threads, it completely fails to leverage massive data-level parallelism. The computational workload of dense matrix operations such as Gaussian elimination severely bottlenecks the CPU's Arithmetic Logic Units (ALUs), causing the total execution time to grow linearly and unsustainably as the batch size N increases.

To resolve this limitation and fully harness hardware acceleration, the proposed optimization completely removes the external for loop from the host CPU and translates it into a batched grid-stride architecture on the GPU. In this optimized kernel combined architecture, the CPU first aggregates the initialization data for all N signatures into flattened 1D contiguous memory arrays. Then, A single bulk cudaMemcpy transfers the entire batch to the device's global memory. The cryptographic kernels are then launched just once, with the batch dimension N mapped directly to the CUDA grid dimensions. The logic of this transformation is illustrated in Figure 4.3.

Algorithm 1: Batched CPU to GPU Kernel Transformation
Input: Batch size *N*, Secret/Public keys, Array of *N* messages
Output: Array of *N* valid signatures

1: // **Baseline CPU Approach**
2: **for** *(n = 0; n < N; n++)* **do**
3: *Expand_keys(data[n])* ▷ Processed sequentially on host CPU
4: *LU_decompose(eqn[n])* ▷ CPU sequential matrix solver
5: *sample_a_solution(result[n])* ▷ Output single signature
6: **end for**

7: // **Proposed Sign_Fused Batched Approach**
8: Flatten memory: *h_all_eqn = malloc(N * m * m * sizeof(Fq))*
9: *cudaMemcpy(HostToDevice, h_all_data)* ▷ Single bulk PCIe transfer
10: *kernel_fused_batched<<<grid_fused, 64>>>(..., N)*
11: *LU_decompose_batched_kernel<<<N, 128>>>(d_all_eqn, ..., N)*
12: Inside LU Kernel Execution:
13: *sig_id* ← *blockIdx.x* ▷ 1 Block mapped to 1 Signature
14: **if** *sig_id* ≥ *N* **then return**
15: Solve *d_all_eqn[sig_id]* cooperatively utilizing threadIdx.x
16: *cudaMemcpy(DeviceToHost, d_all_results)* ▷ Single bulk PCIe transfer

Figure 4.3: Algorithm 1: Batched CPU to GPU Kernel Transformation

By migrating the execution loop to the GPU and utilizing the kernel fusion architecture, the optimization achieves massive performance gains. It eliminates strict sequential dependency and reduces PCIe I/O operations from *O(N)* to *O(1)*. By mapping sig_id equal to blockIdx.x, the hardware scheduler instantly dispatches *N* thread blocks concurrently across all available SMs. This ensures that the arithmetic logic unit resources of the GPU are fully saturated, allowing the throughput to scale dynamically and efficiently as the batch size *N* increases.

4.3.2 Kernel Fusion to Reduce Memory Overhead

In complex cryptographic schemes like QR-UOV, generating a signature involves a deep sequence of dependent mathematical operations. In the initial parallelized implementation, these operations were modularized into six distinct, sequentially executed CUDA kernels including kernel_VxV, kernel_VxM_Pi1Sd, kernel_VxM_Pi2, kernel_M_SUB, kernel_EQN_GEN, and kernel_C_GEN.

While this separated approach successfully utilized GPU threads, it introduced a severe memory bandwidth bottleneck. In the CUDA architecture, when one kernel finishes execution, it must write its output to the device's global memory. The following kernel must then fetch this data from global memory before it can begin its computations. Consequently, the separated implementation required allocating massive intermediate buffers in VRAM. The continuous cycle of reading from and writing to high-latency global memory severely throttled the arithmetic throughput of the Streaming Multiprocessors (SMs), making memory input and output the limiting factor. Furthermore, dispatching six separate kernels per signature introduces cumulative kernel launch overheads.

To eliminate these memory bottlenecks, the proposed optimization utilizes a technique known as Kernel Fusion. The fused kernel implementation merges all six algebraic operations into a single and highly optimized monolithic kernel named kernel_fused_batched.

By fusing the kernels, intermediate cryptographic variables no longer need to be written back to the slow global VRAM. Instead, they are temporarily held in the GPU's ultra-fast, on-chip Shared Memory (SRAM) and local hardware registers. For example, arrays such as __shared__ uint64_t s_yT_Pi1[QRUOV_V] and s_yT_Fi2[QRUOV_M] are declared directly inside the fused kernel. Threads cooperatively compute the vector-matrix multiplications, store the intermediate results in SRAM, synchronize and immediately feed those results into the next mathematical step. The structural difference between the two approaches is illustrated in Figure 4.4.

Algorithm 2: Kernel Fusion for Finite-Field Matrix Operations
Input: Target vector (*yT*), Public Key Matrices (*Pi1*, *Pi2*), Secret Key (*Sd*)
Output: Final equation matrix (*eqn*), generated vector (*c)*

1: // **Separated Kernel Approach (High Memory Latency)**
2: *kernel_VxV<<<...>>>* ▷ Writes *d_yT_Pi1* to Global VRAM
3: *kernel_VxM_Pi1Sd<<<...>>>* ▷ Reads *d_yT_Pi1,* writes *d_yT_Pi1_Sd* to VRAM
4: *kernel_VxM_Pi2<<<...>>>* ▷ Writes *d_yT_Pi2* to VRAM
5: *kernel_M_SUB<<<...>>>* ▷ Reads VRAM, writes *d_yT_Fi2* to VRAM
6: *kernel_EQN_GEN<<<...>>>* ▷ Reads VRAM, writes final *eqn* to VRAM
7: *kernel_C_GEN<<<...>>>* ▷ Reads VRAM, writes final *c* to VRAM

8: // **Proposed Fused Kernel Approach (Low Memory Latency)**
9: *kernel_fused_batched<<<grid, 64>>>(...)*
10: Inside Kernel Execution:
11: Allocate Shared Memory: *s_yT_Pi1, s_yT_Pi2, s_yT_Fi2*
12: Compute $yT \times Pi1 \rightarrow$ Store in SRAM (*s_yT_Pi1*)
13: *__syncthreads()*
14: Compute $(yT \times Pi2) - (s\_yT\_Pi1 \times Sd) \rightarrow$ Store in SRAM (*s_yT_Fi2*)
15: *__syncthreads()*
16: Generate final *eqn* array from *s_yT_Fi2*
17: Compute parallel reduction for *c* using *__shfl_down_sync()*
18: Write final *eqn* and *c* to Global VRAM (Only once)

Figure 4.4: Algorithm 2: Kernel Fusion for Finite-Field Matrix Operations

Kernel fusion achieves immense performance gains by replacing hundreds of thousands of slow global memory transactions with low-latency shared memory and register accesses. It shrinks the memory footprint by removing the need to allocate large intermediate global arrays, freeing up VRAM to allow larger batch sizes *N* to be processed concurrently. Additionally, reducing the execution from six kernel launches down to a single launch significantly decreases driver overhead.

4.3.3 Shared Memory Tiling for Finite-Field Operations

In GPU architectures, Global Memory (VRAM) offers massive storage capacity but suffers from exceptionally high access latency. Cryptographic algorithms like QR-UOV heavily rely on dense linear algebra, specifically LU decomposition with partial pivoting, which operates at $O(m^3)$ complexity. During Gaussian elimination, the same matrix elements are repeatedly read, scaled by finite-field inverses, subtracted, and overwritten. Directly executing these operations in global memory introduces a severe bandwidth bottleneck, forcing the Streaming Multiprocessors (SMs) to remain frequently idle while waiting for high-latency memory transactions rather than performing arithmetic.

To overcome this, the architecture utilizes Shared Memory Tiling. Shared memory (SRAM) is a highly specialized and low-latency memory space located directly on the SM shared exclusively among the threads within a single block. By utilizing shared memory, data is brought physically closer to the arithmetic logic units, operating at near-register speeds.

In the LU_decompose_batched_kernel, rather than computing directly on the global matrix, the threads first cooperatively fetch the entire $QRUOV\_m \times QRUOV\_m$ matrix from VRAM into a statically allocated shared memory tile. Once the matrix resides in SRAM, the threads synchronize and perform the entire Gaussian elimination process on the GPU.

Additionally, physical row swapping during the pivoting phase is notoriously expensive and computationally intensive. To optimize this, a shared memory index array is utilized to track logical row permutations. Instead of rearranging the actual finite-field elements in memory, threads update the pointers within this index array to reflect logical swaps. Once the matrix is fully decomposed, the threads cooperatively write the final solved matrix back to global memory through a single coalesced transaction to maximize hardware throughput.

This optimization was similarly applied in the VERIFY_batched_kernel where intermediate evaluation vectors are cached in shared arrays while threads accumulate finite-field dot products which drastically reducing global memory fetching. The specific transformation for the LU decomposition kernel is outlined in Figure 4.5.

Algorithm 3: Shared Memory Tiling for Batched LU Decomposition
Input: Global matrix array *d_all_eqn* in VRAM
Output: Decomposed matrices in VRAM

```
// Naive Approach (High Latency)
for each pivot i do
    Read pivot row i from Global VRAM
    Compute finite-field inverse: inv = d_Fq_inv(A[i][i])
    for each target row j > i do
        Read A[j][k] and A[i][k] from Global VRAM
        A[j][k] ← Fq_sub(A[j][k], Fq_mul(A[j][i], A[i][k]))
        Write A[j][k] back to Global VRAM
    end for
end for

// Proposed Shared Memory Approach (Low Latency)
Allocate __shared__ Fq s_A[m][m] and __shared__ int p[m]
Threads cooperatively load d_all_eqn[sig_id] → s_A
__syncthreads()
for each pivot i do
    if pivoting required then update logical pointer p[i]
    Compute inverse from s_A
    Threads cooperatively perform row elimination directly on s_A
    __syncthreads()
end for
Threads cooperatively write s_A → d_all_eqn[sig_id] (Only once)
```

Figure 4.5: Algorithm 3: Shared Memory Tiling for Batched LU Decomposition

To further illustrate this transformation, Figure 4.6 visually contrasted the data flow between the naive and optimized implementations of the batched LU decomposition.

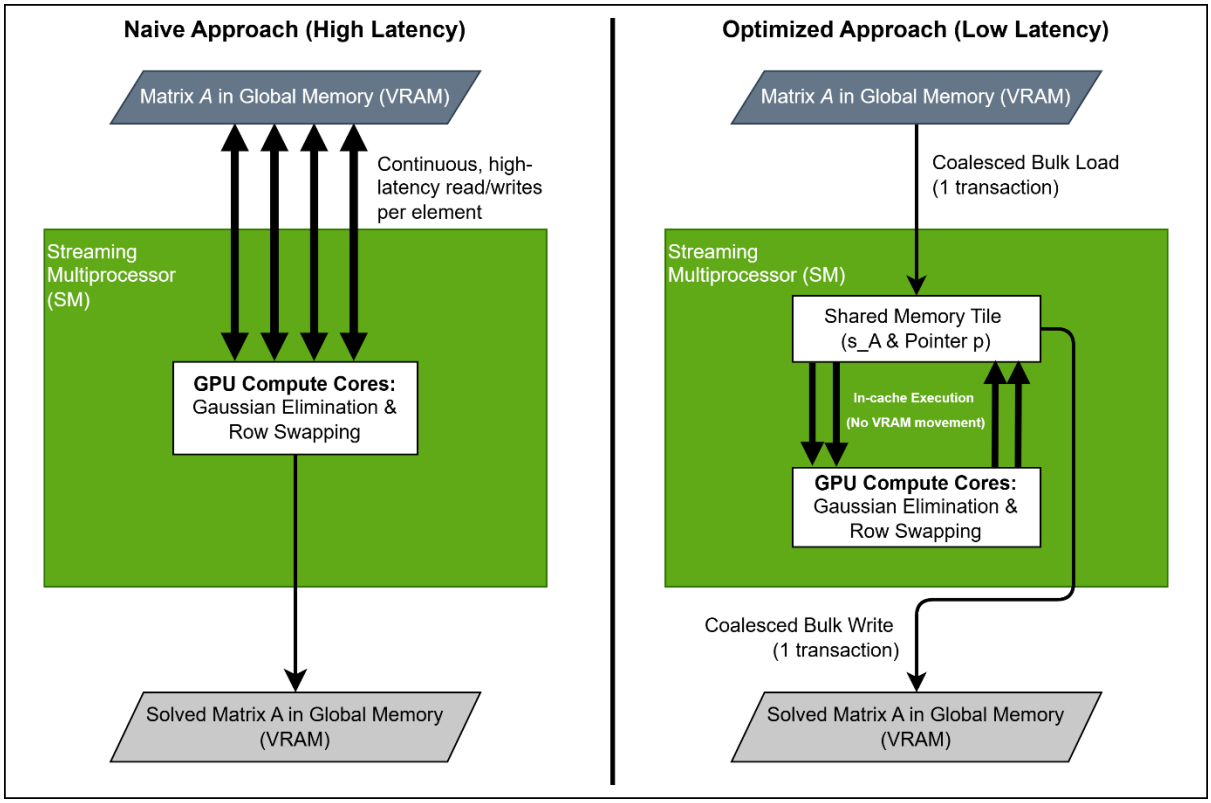


Figure 4.6: Data Flow Comparison of Naive vs Shared Memory Optimized LU Decomposition

As shown in Figure 4.6, in the naive execution model, the data resided exclusively in Global Memory (VRAM). During the arithmetic computation phase, the Streaming Multiprocessors (SMs) were forced to execute continuous, high-latency read and write operations for every individual finite-field scaling and subtraction. Furthermore, physical row swapping required moving entire rows of data within VRAM, which severely degraded memory bandwidth.

Conversely, the optimized data flow utilized a load-compute-store paradigm. The input matrix was retrieved from VRAM via a single coalesced bulk load and staged inside the fast Shared Memory (SRAM) tile (s_A). Once localized, the arithmetic computations (Gaussian elimination) and logical row permutations (via the p tracking array) circulated entirely between the SM registers and SRAM, completely bypassing the VRAM bottleneck. Finally, the solved matrix was dispatched to the output VRAM in a single bulk write.

Thus, by migrating the finite-field matrix operations into shared memory, the implementation eliminated thousands of high-latency global memory read/write cycles per signature. This optimization transformed the workload from being memory-bound to compute-bound. Furthermore, implementing logical row-swapping via the shared pointer array

eliminated unnecessary data movements entirely. Consequently, the execution time for resolving the dense linear systems dropped massively, allowing the GPU to maintain maximal throughput during the heaviest computational phase of the signature generation process.

4.3.4 Hardware-Accelerated Tensor Core Integration (WMMA)

Modern NVIDIA GPUs are equipped with Tensor Cores, which are highly specialized execution units designed to accelerate dense matrix multiplications. While standard CUDA cores compute one multiply-accumulate operation per thread per clock cycle, Tensor Cores can execute matrix math natively at the hardware level, offering drastically higher theoretical throughput. However, applying Tensor Cores to post-quantum cryptography presents a fundamental architectural challenge which is Tensor Cores are optimized for Half-Precision Floating-Point (FP16) arithmetic, whereas the QR-UOV scheme requires exact integer arithmetic over a finite field (Fql). Furthermore, Tensor Cores require strict memory alignments and matrix dimensions that are multiples of 16 for example 16 × 16 × 16 which conflicts with QR-UOV's native parameters of $V = 52$ and $M = 18$.

To harness this hardware without sacrificing cryptographic exactness, a specialized Warp Matrix Multiply Accumulate (WMMA) wrapper was developed. Firstly, to solve the data type mismatch, the 24-bit Fql integer polynomials are manually unpacked into three distinct 8-bit coefficient planes. Each 8-bit integer is safely cast to an FP16 half format and stored in separated shared memory arrays (smem_A and smem_B). This is because the values are strictly bounded so floating-point precision loss is avoided entirely.

Second, to satisfy the strict WMMA dimensional constraints, the cryptographic matrices are padded with zeros. The vinegar variable dimension $V = 52$ is padded to $V\_PAD = 64$, and the oil variable dimension $M = 18$ is padded to $M\_PAD = 32$. Furthermore, to convert the signature verification and generation steps into pure Matrix-Matrix multiplications, the execution groups multiple independent signatures into blocks of 16 (TILE_B=16).

Once the data is staged in shared memory, the algorithm utilizes wmma::mma_sync to perform a 9-step cross-multiplication of the three FP16 coefficient planes which effectively simulating a full polynomial ring multiplication. The results are accumulated in FP32 (smem_C), reduced via modulo arithmetic, and repacked into 24-bit integers. This hardware-mapped optimization is detailed in Figure 4.7.

Algorithm 4: Tensor Core (WMMA) Integration for Finite-Field Matrices
Input: Integers Matrices A (16 × 52) and B (52 × 18) in Global VRAM
Output: Solved Matrix C (16 × 18)

```
// Standard CUDA Core Approach
for each element C[i][j] do
    Compute standard dot product using integer ALUs (d_Fql_mul)
end for

// Proposed Tensor Core (WMMA) Approach
Allocate FP16 SRAM planes: smem_A[3], smem_B[3]
Pad dimensions: V_PAD ← 64, M_PAD ← 32
Unpack 24-bit integers from A and B into three 8-bit planes
Cast 8-bit planes to FP16 (half) and store in smem_A and smem_B
__syncthreads()
for each 16×16 tile do
    Load fragments: wmma::load_matrix_sync(a0, a1, a2)
    Load fragments: wmma::load_matrix_sync(b0, b1, b2)
    Execute 9 hardware MACs:
        wmma::mma_sync(c0, a0, b0, c0)     ▷ 0th degree
        wmma::mma_sync(c1, a0, b1, c1)     ▷ 1th degree
        ... (Compute up to 4th degree)
    Store accumulated FP32 fragments to smem_C
end for
__syncthreads()
Modulo reduce FP32 values, repack to 24-bit integers, write to VRAM
```

Figure 4.7: Algorithm 4: Tensor Core (WMMA) Integration for Finite-Field Matrices

Therefore, integrating the WMMA API unlocks the specialized AI-focused hardware inside modern GPUs for post-quantum cryptographic workloads. By transforming the matrix-vector operations into batched matrix-matrix multiplications, the GPU can leverage the massive theoretical throughput of Tensor Cores. However, this approach introduces a measurable trade-off including the computational overhead required to unpack integers, cast to FP16, pad dimensions, and dynamically allocate large shared memory pools (over 50 KB per block) adds latency. As a result, while Tensor Cores present a highly scalable architectural solution for massive batch sizes or larger cryptographic parameter sets, standard CUDA cores often yield lower latency for smaller workloads where the casting overhead outweighs the raw multiply-accumulate throughput.

CHAPTER 5 SYSTEM IMPLEMENTATION

5.1 Set-up and Key Expansion

The implementation of the batched QR-UOV scheme required a collaborative initialization phase between the host CPU and the GPU device. The execution began on the host, where the system prepared the cryptographic seeds, target vectors, and input messages for the defined batch size *N*.

To ensure cryptographic integrity, the implementation utilized the OpenSSL library for deterministic random bit generation and hashing. The *Expand_mu* function was invoked to hash the original message along with the public key seed (*seed_pk*). This was implemented using the EVP_shake256() function to produce the cryptographic digest mu. Following this, the *randombytes()* function, driven by an AES-CTR Deterministic Random Bit Generator (DRBG), was utilized to generate unique, secure seeds for each signature in the batch, specifically *seed_y, seed_r,* and *seed_sol*.

As generating pseudo-random sequences involves strictly sequential block cipher operations and rejection sampling (*RejSampPRG*), the generation of the raw pseudo-random byte arrays was retained on the CPU. The CPU populated large 1D contiguous arrays (*all_r1* and *all_r2*) with these compact 8-bit raw bytes. Transferring these densely packed byte arrays across the PCIe bus rather than the fully expanded 64-bit matrices significantly minimized the data transfer overhead between the host and the device.

Once the raw byte arrays were securely transferred to the device's global memory via *cudaMemcpy*, the GPU took over to perform the computationally wide key expansion. Dedicated CUDA kernels, specifically *ExpandSymmetricMatrixVxV_batched_kernel* (for the public key matrix *Pi1*) and *ExpandMatrixVxM_batched_kernel* (for matrices *Pi2* and the secret key *Sd*) were launched.

Within these kernels, GPU threads cooperatively unpacked the dense 24-bit finite field sequences. Each thread read three consecutive 8-bit bytes and executed bitwise shifting operations to correctly align the coefficients into 64-bit multi-precision integers. For symmetric matrices like *Pi1*, the kernel dynamically mirrored the unpacked elements across the diagonal to reconstruct the full matrix shape in VRAM. This hybrid initialization strategy effectively

leveraged the CPU's superiority in sequential AES-CTR processing while harnessing the GPU's massive memory bandwidth to format and expand the evaluation matrices.

5.2 Batched Signature Generation

5.2.1 Fused Kernel Operations

In the initial iteration of the parallelized signature generation in separated kernel, the mathematical sequence was divided into multiple isolated CUDA kernels. These included discrete kernels for vector-matrix multiplications (*kernel_VxV*, *kernel_VxM_Pi1Sd*), matrix subtractions (*kernel_M_SUB*), and equation generation (*kernel_EQN_GEN*). While functionally correct, this separated approach suffered from immense global memory access overhead. Each kernel was forced to write its intermediate variables to VRAM which the subsequent kernel immediately had to fetch, bottlenecking the Streaming Multiprocessors (SMs) with high-latency memory transactions and repetitive kernel launch overheads.

To resolve this, the optimized implementation merged these dependent steps into a single continuous execution unit. By utilizing this fused architecture, the overhead of launching multiple kernels was eliminated and the reliance on global memory was drastically reduced.

Within the fused kernel, intermediate cryptographic variables were retained on-chip. The threads dynamically allocated low-latency Shared Memory (SRAM) arrays, such as *__shared__ uint64_t s_yT_Pi1[QRUOV_V]* and *s_yT_Fi2[QRUOV_M]*. The threads cooperatively computed the finite-field dot products (*dot_product_fql*) for the matrix-vector multiplications and stored the results directly into these shared tiles. Thread execution was synchronized using the *__syncthreads()* barrier, ensuring data consistency before the threads immediately fed the intermediate values into the next algebraic step. Consequently, the heavy permutations and polynomial additions required to generate the final linear equations bypassed the VRAM entirely.

A critical enhancement within this fused kernel was the optimization of the *C_GEN* sum reduction phase. After multiplying the intermediate vectors, the threads needed to aggregate their partial dot products into a single scalar value. A naive approach would involve writing these partial sums into shared memory and utilizing *atomicAdd* which forces hardware serialization and stalls thread execution.

Algorithm 5: Warp-Level Parallel Reduction via *__shfl_down_sync()*
Input: Partial sum *local_sum* per thread, Thread ID *tid*
Output: Final accumulated scalar value c_i

```
1: // Phase 1: Block-level partial reduction (Shared Memory)
2: s_cgen[tid] = local_sum
3: __syncthreads()
4: if tid < 32 then
5:     s_cgen[tid] += s_cgen[tid + 32]      ▷ Fold 64 elements to 32
6: end if
7: __syncthreads()

8: // Phase 2: Warp-level reduction (Hardware Registers)
9: if tid < 32 then
10:    val = s_cgen[tid]      ▷ Load into ultra-fast hardware register
11:    val += __shfl_down_sync(0xFFFFFFFF, val, 16)      ▷ Fold 32 to 16
12:    val += __shfl_down_sync(0xFFFFFFFF, val, 8)       ▷ Fold 16 to 8
13:    val += __shfl_down_sync(0xFFFFFFFF, val, 4)       ▷ Fold 8 to 4
14:    val += __shfl_down_sync(0xFFFFFFFF, val, 2)       ▷ Fold 4 to 2
15:    val += __shfl_down_sync(0xFFFFFFFF, val, 1)       ▷ Fold 2 to 1
16:    if tid == 0 then
17:        c[i] ← (val (mod QRUOV_q))      ▷ Final finite-field reduction
18:    end if
19: end if
```

Figure 5.1: Algorithm 5: Warp-Level Parallel Reduction via __shfl_down_sync()

Instead, the implementation utilized warp-level parallel reduction via the *__shfl_down_sync()* hardware primitive shown in Figure 5.1. This intrinsic function allowed the 32 threads within a single warp to securely pass their partial sums directly between their local hardware registers. By systematically folding the values in half (passing values from offset 16, then 8, then 4, 2, and 1), the final accumulated sum was routed securely to Thread 0 without ever touching shared or global memory. Once the reduction was complete, the kernel wrote the final computed *d_all_eqn* matrix and *d_all_c* vector to global memory exactly once, perfectly staging the data for the subsequent LU decomposition phase.

5.2.2 Batched LU Decomposition

Solving the generated linear equations required performing LU decomposition with partial pivoting on the matrices. Because Gaussian elimination is an $O(n^3)$ algorithmic process, performing it sequentially on the CPU for thousands of matrices would require massive computational power and longer completion time. To resolve this, the system used the *LU_decompose_batched_kernel* which was designed to process multiple independent matrices simultaneously across the GPU.

The kernel utilized a strict "one block equals one signature" mapping strategy. The CUDA block index (*blockIdx.x)* was mapped to the signature ID *(sig_id),* ensuring that an entire batch of *N* matrices was distributed evenly across the Streaming Multiprocessors (SMs). This allowed the GPU to decompose thousands of linear systems concurrently without requiring any inter-block synchronization.

Within a single block, the dense matrix operations were highly optimized using Shared Memory (SRAM). As Gaussian elimination involves repeatedly reading, scaling, and overwriting the same matrix elements, executing these operations directly in VRAM would have caused severe memory stalling. To prevent this, the threads cooperatively fetched the entire *QRUOV_m* × *QRUOV_m* matrix from global memory into a low-latency shared memory tile (*__shared__ Fq s_A[QRUOV_m][QRUOV_m]*). Once the data resided in SRAM, the threads synchronized and performed the entire elimination process on-chip.

Furthermore, the implementation aggressively optimized the partial pivoting phase. In standard Gaussian elimination, swapping rows to secure a valid pivot requires physically moving large amounts of data which degrades performance. The proposed architecture bypassed physical row swapping entirely by allocating a shared memory tracking array (*__shared__ int p[QRUOV_m]*). When a row swap was required, a designated thread utilized the *atomicMin* function to locate the smallest row index containing a valid non-zero element. Instead of moving the finite-field elements in *s_A*, the threads simply updated the logical row pointers within the array p.

During the elimination phase, the threads within the block divided the workload. Utilizing a pre-computed inverse lookup table (*d_Fq_inv_table*) stored in constant memory, the threads concurrently scaled the pivot row and subtracted it from all subsequent rows. Once the decomposition process concluded, the threads cooperatively wrote the logically sorted *s_A*

matrix, the matrix's computed rank, and the permutation indices back to global memory in a single coalesced transaction. This data was then utilized by either the GPU sampling kernel or the hybrid CPU fallback controller.

5.2.3 Tensor Core Implementation

To further explore the limits of hardware acceleration, an alternative implementation was developed to make use of NVIDIA's Tensor Cores. Tensor Cores provide immense theoretical throughput for dense matrix multiplications but are primarily engineered for half-precision floating-point (FP16) arithmetic. As the QR-UOV scheme relies on exact integer arithmetic over a finite field (*Fql*), a specialized Warp Matrix Multiply Accumulate (WMMA) wrapper (*wmma_fql_gemm*) was designed to bridge this hardware-software mismatch without sacrificing cryptographic correctness.

The implementation first resolved the data type incompatibility by manually unpacking the 24-bit integer polynomials (*Fql*) into three distinct 8-bit coefficient planes. Each 8-bit integer was safely cast to an FP16 format utilizing the *__uint2half_rn()* intrinsic and staged in dynamically allocated shared memory arrays (*smem_A* and *smem_B*). As the raw numerical values were strictly bounded within 8 bits, floating-point precision loss was completely avoided during the hardware computation.

Furthermore, Tensor Cores enforce strict alignment and dimensional constraints, requiring matrix dimensions to be multiples of 16 (*16×16×16*). Consequently, the original cryptographic parameters were padded with zeros; the vinegar variable dimension $V = 52$ was padded to $V\_PAD = 64$ and the oil variable dimension $M = 18$ was padded to $M\_PAD = 32$. Additionally, due to that Tensor Cores cannot efficiently execute matrix-vector math, the implementation grouped independent signatures into sub-batches of 16 ($TILE\_B = 16$). This successfully transformed the matrix-vector operations into the dense matrix-matrix operations required by the WMMA API.

Once the FP16 data was properly padded and staged in shared memory, the kernel utilized *wmma::mma_sync* to perform a 9-step cross-multiplication of the three coefficient planes, effectively simulating a full polynomial ring multiplication at the hardware level. The intermediate results were accumulated safely in 32-bit floating-point (FP32) fragments (*smem_C*). Finally, the threads synchronized, performed modulo reduction on the FP32 accumulators to return the values to the finite field, and repacked them into standard 24-bit integers.

While this implementation successfully unlocked specialized AI hardware for post-quantum cryptography, it required massive dynamic shared memory allocations (over 73 KB

per block). The computational overhead required to constantly pack, unpack, cast, and pad the data introduced specific latency trade-offs compared to the standard CUDA cores.

5.2.4 Hybrid CPU-GPU Fallback (Rare Case)

A mathematical reality of the QR-UOV scheme is that the matrices generated during the signing process occasionally lack full rank. During the LU decomposition phase, these rank-deficient (singular) matrices cannot be solved directly and require the target vector b to mathematically align with the matrix's null space to produce a valid signature.

If this edge case were handled natively on the device, it would necessitate an indeterminate do-while loop within the CUDA kernel to continuously re-sample the cryptographic salt until a consistent target vector was found. However, modern GPUs operate on Single Instruction, Multiple Threads (SIMT) architecture. If even one thread within a warp entered a prolonged while loop, all other 31 threads in that warp would be forced to wait, causing severe warp divergence. This phenomenon would stall the Streaming Multiprocessors (SMs) and completely collapse the parallel efficiency of the batch execution.

To overcome this hardware limitation, a Hybrid CPU-GPU Fallback architecture was used. Following the batched LU decomposition on the device, the kernel recorded the matrix ranks and transferred the resulting integer rank array back to the host CPU. The CPU then acted as a sequential fallback controller, iterating through the array to assess the rank status of each signature in the batch.

For optimal instances where rank equal to $QRUOV\_m$ (full-rank matrices), the CPU simply executed a single, deterministic pseudo-random generation to yield the salt, hashed the message using SHAKE-256, and computed the target vector b.

Algorithm 6: Hybrid CPU-GPU Fallback Controller for Singular Matrices
Input: Matrix rank array *h_all_rank*, Batch Size *N*
Output: Mathematically consistent target vectors *b* for all signatures

```
1: for n = 0 to N -1 do
2:      Load target vector b_vec and generated vector c_vec
3:      if h_all_rank[n] == QRUOV_m then          ▷ (Optimal Case: Full Rank)
4:          salt ←PRG2_yield(ctx)
5:          msg_hash ←Hash(mu, salt)
6:          b_vec ←Fq_sub(msg_hash, c_vec)
7:      else    ▷ (Rare Case: Rank-Deficient Matrix)
8:          Load decomposed echelon_form matrix from GPU
9:          do
10:             salt ←PRG2_yield(ctx)        ▷ Re-sample cryptographic salt
11:             msg_hash ←Hash(mu, salt)  ▷ Re-hash message
12:             b_vec ←Fq_sub(msg_hash ,c_vec)
13:         while not consistent(echelon_form, b_vec)   ▷ Check null space
14:     end if
15: end for
16: cudaMemcpy(HostToDevice, b_vec_batch)        ▷ Return resolved vectors to GPU
```

Figure 5.2: Algorithm 6: Hybrid CPU-GPU Fallback Controller for Singular Matrices

For the rare occurrences where rank < QRUOV_m that occurred on rank-deficient matrices, the CPU took over the complex workload. The host code triggered a dedicated fallback do-while loop in which the CPU iteratively re-samples the salt via AES-CTR, hashes the message, and verified mathematical consistency by invoking the host-side consistent() and L_inverse() functions. The structure is shown in Figure 5.2. CPUs utilized sophisticated branch predictors and independent thread execution to process this iterative resampling highly efficiently, ensuring the remainder of the batch is not idle. This hybrid approach prevented severe warp divergence and penalty stalls within the GPU's Streaming Multiprocessors that would otherwise occur if the indeterminate loop were executed natively on the device.

Once the CPU successfully identified a mathematically consistent target vector for every singular matrix in the batch, the resolved vectors were transferred back to the GPU. The device then resumed parallel execution for the final signature sampling and packaging kernels. This hybrid methodology ensured strict cryptographic correctness while strategically offloading control-heavy and divergent edge cases to the CPU, thereby preserving maximal parallel throughput on the GPU.

5.3 Batched Signature Verification

5.3.1 CUDA Core Verification Variant

Algorithm 7: Batched Signature Verification Execution
Input: Public Key matrices (*Pi1, Pi2, Pi3*), Signature elements (*oil, vinegar*)
Output: Global Boolean array *d_all_result* containing 1 or 0 (Pass or Fail)

```
1: Map equation_index ← blockIdx.x
2: Map tid ← threadIdx.x
3: for sig_id in batch do        ▷ Grid-stride loop to handle massive batches
4:       Allocate __shared__ tmp_v[V] and tmp_o[M]
5:       if tid < V then
6:             tmp_v[tid] ← fql_mac(Pi2, oil) + fql_mac(Pi1, vinegar)
7:       end if
8:       if tid < M then
9:             tmp_o[tid] ← fql_mac(Pi3, oil)
10:      end if
11:      __syncthreads()

12:      // Final Evaluation & Atomic Check
13:      if tid == 0 then
14:            acc ← fql_mac(vinegar, tmp_v) + fql_mac(oil, tmp_o)
15:            t0 ← (acc (mod QRUOV_q))          ▷ Extract finite-field result
16:            if d_msg[equation_index] == t0 then
17:                  pass ← 1
18:            else
19:                  pass ← 0
20:            end if
21:            atomicAnd(&d_all_result[sig_id], pass)      ▷ Atomically flag invalid
signatures
22:      end if
23: __syncthreads()
24: end for
```

Figure 5.3: Algorithm 7: Batched Signature Verification Execution

Figure 5.3 illustrates the pseudocode for 5.3 Batched Signature Verification. Unlike signature generation, the verification process in the QR-UOV scheme did not require complex linear system solving or LU decomposition. Instead, it relied purely on the evaluation of public multivariate polynomial equations. This algorithmic simplicity allowed the verification phase

to be implemented as a highly streamlined, single-pass parallel execution pipeline without the need for CPU-bound fallback mechanisms.

In the standard CUDA implementation (*VERIFY_batched_kernel*), the workload was distributed across a 2D grid. The block index was mapped to a specific equation out of the *QRUOV_m* public equations, while the thread index was mapped directly to the vector elements up to *QRUOV_V*. To efficiently process batch sizes that exceeded hardware limits, a grid-stride loop was utilized (*sig_id* += *gridDim*.y), allowing blocks to iteratively process multiple signatures in the batch.

At the core of the verification kernel was the heavily optimized, inline finite-field multiply-accumulate function, *fql_mac*. Because the system operated over the finite field (*Fql*) polynomial ring, standard multiplication could not be used directly. The fql_mac function was designed to unpack the 24-bit *Fql* variables, perform the necessary cross-multiplications for the polynomial coefficients, handle degree folding (reducing an x^3 coefficient down to $x^0 + x^1$), and accumulate the results.

To minimize global memory access, the threads evaluated the partial matrix-vector products concurrently and cached the intermediate vectors into fast shared memory arrays (*__shared__ uint64_t tmp_v[QRUOV_V]* and *tmp_o[QRUOV_M]*). Following a block synchronization barrier, a single designated thread took over to compute the final inner product between the evaluated vectors and the signature variables (vinegar and oil). Thread 0 then applied a final modulo reduction to extract the 8-bit result and compared it against the original hashed message. To record the result securely, the thread utilized the *atomicAnd* hardware primitive on a global boolean array. This ensured that even if a single equation out of the *m* equations failed the consistency check, the entire signature was atomically marked as invalid across the batch.

5.3.2 Tensor Core Verification Variant

To explore hardware acceleration for verification, an alternative VERIFY_tensor_batched_kernel was also developed. A fundamental challenge was that verification primarily relies on matrix-vector multiplications which Tensor Cores cannot execute efficiently. To resolve this, the implementation grouped independent signatures into sub-batches of 16 (*TILE_B = 16*). By evaluating 16 signatures simultaneously against the same public key matrices, the operations were successfully transformed into dense matrix-matrix multiplications compatible with the WMMA API.

While this variant successfully engaged the Tensor Cores, it required massive allocations of dynamic shared memory (over 51 KB per block) to hold the padded FP16 matrices. The continuous overhead of unpacking integers, casting them to half-precision, and repacking the results introduced latency that ultimately made the standard, integer-based Verify kernel the faster and more optimal choice for signature authentication.

CHAPTER 6 SYSTEM EVALUATION AND DISCUSSION

6.1 Hardware and Software Setup

The hardware involved in this project is a computer and the most important NVIDIA GPU. A computer is used to analyze and identify the core component of QR-UOV post quantum cryptography, then apply parallelism techniques in programmable code to those components. A NVIDIA GPU is used for its CUDA framework and Tensor Core which support high acceleration in performance (speed and throughput).

The software stack utilized a 64-bit operating system. The GPU kernels were compiled utilizing the NVIDIA CUDA Toolkit with the nvcc compiler utilizing the -O3 optimization flag to ensure maximum compilation efficiency. For host-side operations, the system integrated the OpenSSL library to handle cryptographic primitives including the AES-CTR Deterministic Random Bit Generator (DRBG) and SHAKE-256 hashing functions.

Description	Specifications
Model	Asus Vivobook Pro 15X OLED (K6501ZM, 12th Gen Intel)
Processor	Intel Core i7-12650H
Operating System	Ubuntu (Mainly) / Windows 11
Graphic	NVIDIA® GeForce® RTX™ 3060 Laptop GPU 6GB GDDR6
Memory	32GB DDR5 RAM
Storage	512GB SAMSUNG SSD

Table 6.1: Specifications of laptop

6.2 Performance Metrics Definition

To objectively quantify the performance gains, scalability, and efficiency of the proposed GPU-accelerated QR-UOV architecture, a comprehensive set of evaluation metrics was defined. These metrics were carefully selected to isolate the parallel computational efficiency of the device (GPU) from the sequential control logic of the host (CPU). By breaking down the performance into smaller components, the evaluation provided a precise analysis of how the heterogeneous system behaved under exponentially increasing workloads.

The primary metric utilized for assessing computational speed was Execution Time, measured in milliseconds (ms). As the architecture employed a hybrid CPU-GPU execution model, the overall latency was subdivided into strictly monitored components. The GPU Time captured the exact duration of the parallel kernel executions including key expansion, fused matrix operations, and batched LU decomposition. This was measured with sub-millisecond precision utilizing hardware-level CUDA Events wrapped directly around the kernel launch boundaries. Conversely, the CPU Time measured the duration required for host-side sequential tasks such as initial pseudo-random data generation, message hashing, and the algorithmic resolution of rank-deficient matrices via the fallback controller. This was tracked using high-resolution POSIX timers (clock_gettime). The Total Time was subsequently calculated as the summation of both the host and device latencies, representing the absolute real-world time required to return a complete batch of valid cryptographic outputs to the application layer.

Beyond raw execution latency, Throughput was used as the most critical metric for evaluating the system's viability in high-demand deployment environments such as cloud servers and Internet of Things (IoT) gateways. Throughput quantified the total volume of cryptographic operations successfully completed within a one-second window. For the signature generation phase, this was measured in signatures per second (sig/s), whereas the authentication phase was measured in verifications per second (ver/s). Throughput was mathematically derived for each benchmark iteration by dividing the total batch size (N) by the total execution time converted into seconds. By plotting throughput against exponentially increasing batch sizes, the evaluation identified the exact workload thresholds where the GPU's Streaming Multiprocessors (SMs) reached maximum hardware saturation.

Finally, the evaluation tracked the Rare Case Frequency to assess the stability and impact of the hybrid fallback architecture. This metric recorded the absolute count of singular,

rank-deficient matrices generated per batch. Monitoring this frequency allowed for a precise correlation between the occurrence of algorithmic edge cases and the corresponding time penalties incurred by the CPU fallback controller.

Metric	Unit	Measurement Method	Significance in Evaluation
GPU Time	Milliseconds (ms)	*cudaEventElapsedTime*	Evaluated the pure parallel processing speed of the device and the efficiency of the optimized CUDA kernels.
CPU Time	Milliseconds (ms)	*clock_gettime*	Measured the sequential overhead of data preparation and the latency penalty of the Hybrid Fallback mechanism.
Total Time	Milliseconds (ms)	CPU Time + GPU Time	Represented the absolute end-to-end execution latency for a complete batch of size *N*.
Throughput	sig/s or ver/s	N/(Total Time in seconds)	Quantified the scalability and maximum volume capacity of the system under saturated workloads.
Rare Case Frequency	Absolute Count	Host-side rank array evaluation	Validated the statistical probability of rank-deficient matrices and their impact on overall CPU fallback time.

Table 6.2: Summary of System Performance Metrics

6.3 Evaluation of Signature Generation Optimizations

The system was tested across exponentially increasing batch sizes, ranging from a single signature up to massive workloads. To thoroughly evaluate and maximize system scalability, the architecture dynamically calculated the required memory footprint per signature. The implementation successfully scaled to the GPU's maximum VRAM capacity, safely allocating up to 75% of available memory before triggering a clean termination to prevent out-of-memory errors. The comprehensive benchmark results for the standard CUDA core implementation without kernel fused, optimized standard CUDA core implementation with kernel fused, the WMMA-accelerated implementation, and the baseline CPU implementation are summarized.

Batch Size	Time (ms)	Throughput (sig/s)
1	2.022	494.6
2	3.691	541.9
4	4.870	821.4
8	11.029	725.4
16	19.184	834.0
32	38.287	835.8
64	75.622	846.3
128	151.154	846.8
256	303.395	843.8
512	604.787	846.6
1024	1215.257	842.6
2048	2429.717	842.9
4096	4961.712	825.5
8192	9786.681	837.1
16384	22603.602	724.8
32768	46048.557	711.6
65536	95826.031	683.9
131072	185907.900	705.0
262144	373414.130	702.0
524288	757792.761	691.9

Table 6.3: Overall Benchmark Results for baseline CPU

Batch Size	GPU Time (ms)	CPU Time (ms)	Total (ms)	Throughput (sig/s)	Rare Case
1	0.274	0.006	0.280	3571.9	0
2	0.265	0.003	0.268	7467.9	0
4	0.328	0.004	0.331	12068.3	0
8	0.447	0.007	0.454	17603.2	0
16	0.669	0.015	0.684	23404.0	0
32	1.112	0.027	1.139	28102.1	0
64	2.293	0.052	2.345	27293.0	0
128	4.724	0.109	4.833	26484.8	0
256	9.542	0.256	9.798	26128.2	1
512	18.815	0.568	19.383	26414.4	4
1024	37.352	2.097	39.450	25957.1	10
2048	66.065	3.626	69.692	29386.5	16
4096	124.808	6.927	131.735	31092.7	30
8192	232.817	17.055	249.872	32784.8	64
16384	467.466	40.284	507.750	32267.8	147
32768	957.054	80.146	1037.200	31592.7	239
65536	1910.242	162.996	2073.238	31610.5	507

Table 6.4: Overall Benchmark Results for GPU Signature Generation without kernel fused

Batch Size	GPU Time (ms)	CPU Time (ms)	Total (ms)	Throughput (sig/s)	Rare Case
1	0.246	0.006	0.252	3965.7	0
2	0.226	0.003	0.229	8717.9	0
4	0.238	0.004	0.242	16536.1	0
8	0.266	0.008	0.274	29218.0	0
16	0.290	0.015	0.305	52533.1	0
32	0.350	0.028	0.378	84575.1	0
64	0.456	0.056	0.511	125160.4	0
128	0.697	0.117	0.815	157126.2	0
256	1.236	0.282	1.518	168649.7	1
512	2.099	0.641	2.740	186873.4	4
1024	3.791	2.414	6.204	165043.8	10
2048	7.175	3.942	11.117	184225.7	16
4096	13.913	6.049	19.962	205189.8	30
8192	27.750	14.748	42.498	192761.3	64
16384	56.152	33.198	89.351	183367.3	147
32768	116.160	57.690	173.851	188483.7	239
65536	205.579	129.207	334.786	195754.8	507
131072	419.913	321.532	741.445	176779.0	1018
262144	876.925	1087.887	1964.812	133419.4	2137
524288	1722.589	1691.596	3414.185	153561.7	4083

Table 6.5: Overall Benchmark Results for Fused GPU Signature Generation

Batch Size	Tensor Time (ms)	CPU Time (ms)	Total (ms)	Throughput (sig/s)	Rare Case
1	0.279	0.004	0.282	3540.5	0
2	0.264	0.002	0.267	7503.5	0
4	0.276	0.004	0.280	14284.3	0
8	0.298	0.007	0.305	26262.6	0
16	0.317	0.012	0.329	48562.8	0
32	0.407	0.023	0.430	74498.0	0
64	0.571	0.046	0.617	103746.7	0
128	0.895	0.091	0.986	129781.6	0
256	1.616	0.220	1.836	139406.2	1
512	2.888	0.529	3.417	149855.3	4
1024	5.441	1.944	7.385	138660.7	10
2048	10.176	3.236	13.412	152696.6	16
4096	20.015	5.846	25.862	158380.8	30
8192	40.823	13.938	54.761	149594.8	64
16384	82.047	28.098	110.145	148749.5	147
32768	142.420	46.625	189.046	173333.8	239
65536	256.822	99.000	355.822	184181.9	507
131072	526.642	287.982	814.624	160898.9	1018
262144	1105.602	745.035	1850.637	141650.7	2137
524288	2137.643	1820.184	3957.827	132468.7	4083

Table 6.6: Overall Benchmark Results for Tensor Core Signature Generation

6.3.1 Performance Gain Moving from CPU to Separated Kernel GPU

The first phase of the system evaluation measured the performance gain achieved by migrating the cryptographic workload from the host CPU to the GPU utilizing an initial and modularized parallel architecture. In the baseline reference implementation, the CPU processed cryptographic batches using a strict sequential for loop. When benchmarked, this approach demonstrated a severe scaling bottleneck. Because the CPU was unable to leverage massive concurrency, the total execution time grew linearly as the batch size increased. The maximum CPU throughput plateaued very early, peaking at merely 846.8 sig/s at *N=128* and actively degraded under heavier workloads, dropping to 691.9 sig/s at *N=524,288*.

By offloading the computation to the device via the separated kernel implementation, the system successfully engaged the GPU's thousands of threads to process the finite-field matrix operations concurrently.

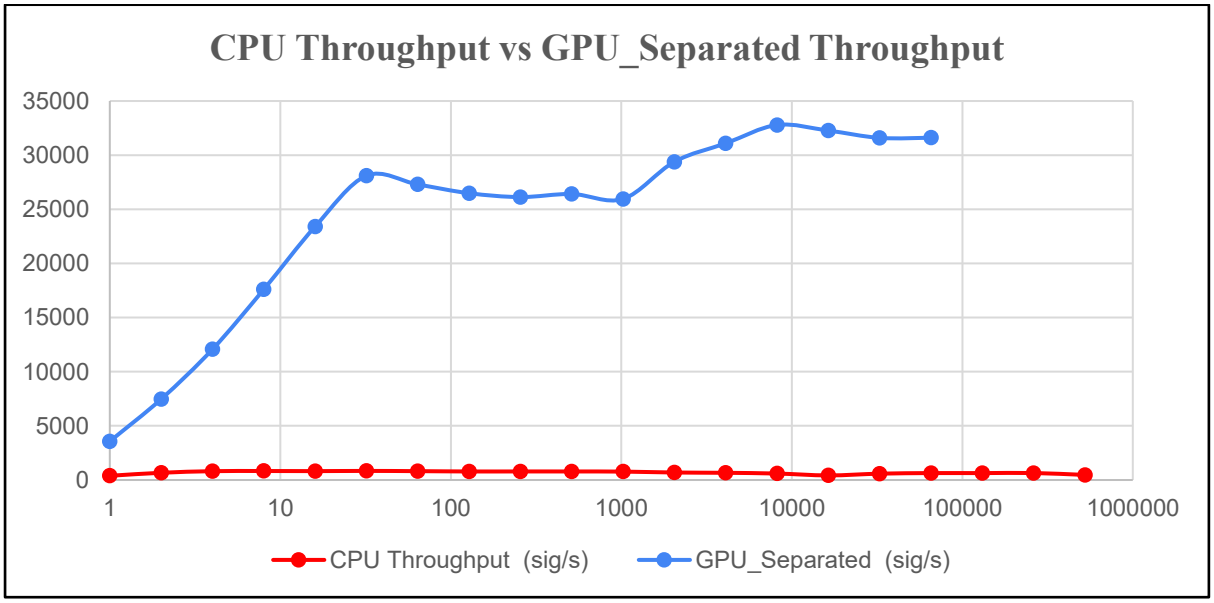


Figure 6.1: Throughput Comparison between Baseline CPU and GPU_Separated Architectures

As illustrated in Figure 6.1, the initial GPU parallelization delivered a substantial performance leap over the host CPU. The separated kernel implementation achieved a peak throughput of 32,784.8 sig/s at a batch size of *N=8192*. This represented a 39x speedup over the CPU's peak performance. However, while the parallel threads successfully accelerated the raw arithmetic, the separated architecture introduced a new bottleneck whereby the it is memory intensive because the algorithm was divided into six distinct and isolated kernels hence intermediate variables had to be continuously written to and fetched from the GPU's

high-latency global memory (VRAM) between each kernel launch. Consequently, the throughput artificially plateaued at around 32,000 sig/s as the Streaming Multiprocessors (SMs) were idled waiting for memory I/O rather than performing actual computations.

6.3.2 Performance Gain from Separated Kernel GPU to Kernel Fusion

To eliminate the severe global memory bottlenecks identified in the separated architecture, the system was aggressively optimized by transitioning to a fused kernel execution model. This architecture merged the distinct algebraic steps into a single, monolithic kernel launch. By doing so, intermediate cryptographic variables were retained entirely on-chip within the ultra-fast Shared Memory (SRAM) and hardware registers, completely bypassing the VRAM read/write cycles that crippled the previous iteration.

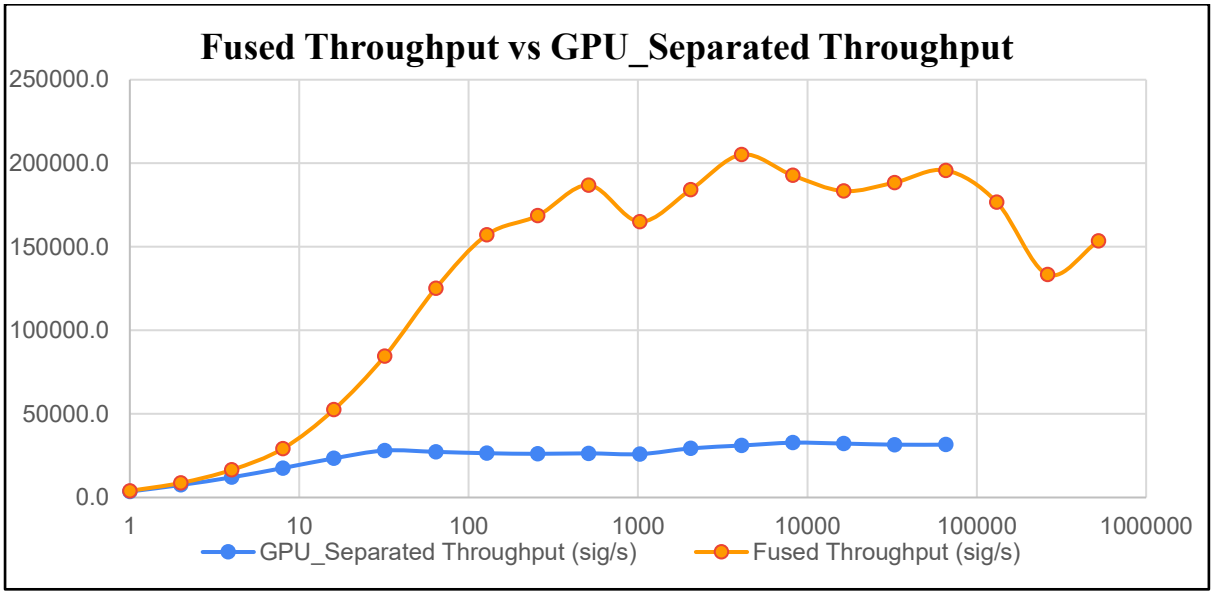


Figure 6.2: Throughput Comparison between Fused GPU and Separated GPU Implementations

As illustrated in Figure 6.2, plotting the throughput of both GPU architectures against exponentially increasing batch sizes reveals a monumental performance gap. The blue line representing the separated architecture, flattened early due to global memory bandwidth saturation. In sharp contrast, the orange line representing the optimized fused kernel architecture, scaled exponentially alongside the batch size. By removing the external synchronization loops and keeping intermediate data in shared memory, the fused implementation reached a massive peak throughput of 205,189.8 sig/s at $N=4096$

This translation from isolated kernels to a fused architecture yielded a 6.2x speedup and an astounding 242x speedup over the original CPU baseline. This data proves that for dense finite-field cryptography and memory transaction latency heavily outweighs raw computational complexity. Combining kernels, utilizing shared memory tiling, and executing warp-level reductions are absolute prerequisites for fully saturating the GPU's hardware and achieving maximal cryptographic throughput.

6.3.3 Tensor Core Acceleration vs. Standard CUDA Cores

To push the hardware to its absolute theoretical limits, an alternative architecture was developed to offload the dense linear algebra of the QR-UOV scheme onto NVIDIA's Tensor Cores. Tensor Cores are highly specialized execution units natively engineered to accelerate dense matrix-matrix multiplications using half-precision floating-point (FP16) arithmetic. The evaluation aimed to determine whether the massive theoretical throughput of this hardware could successfully accelerate integer-based, post-quantum cryptography when compared against the highly optimized fused kernel architecture.

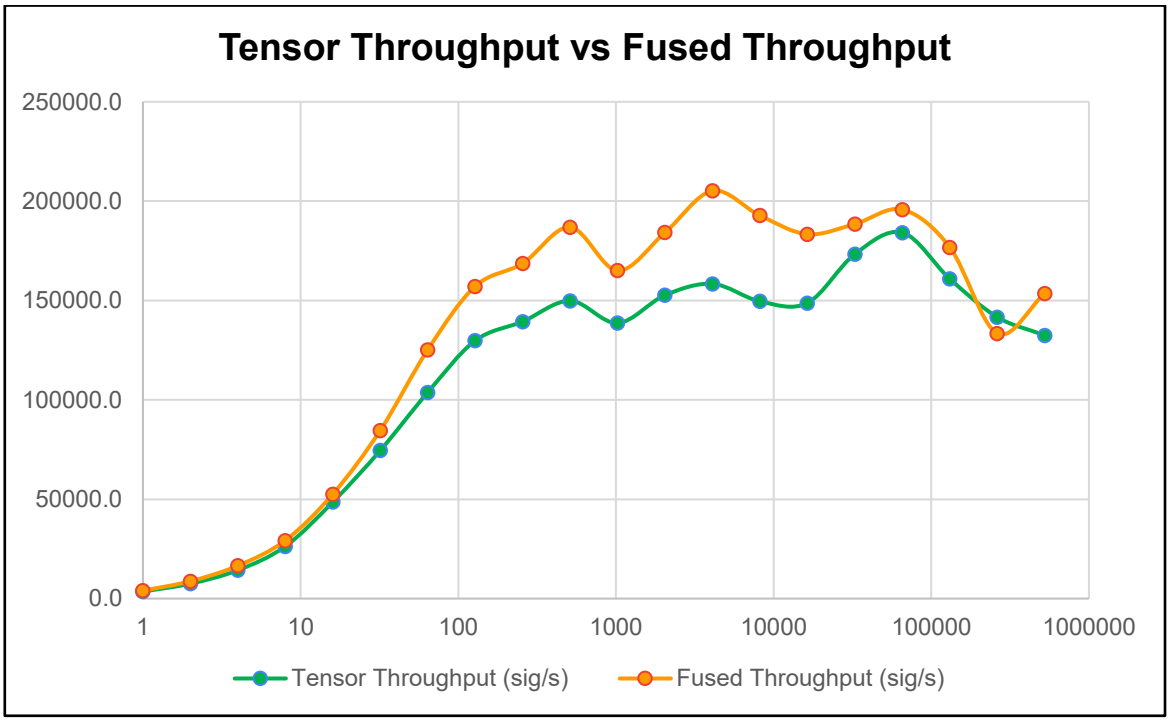


Figure 6.3: Throughput Comparison between Fused CUDA Cores and Tensor Cores (WMMA)

The benchmark data demonstrated that the Tensor Core implementation successfully engaged the specialized hardware, achieving a highly respectable peak throughput of 184,181.9 sig/s at an optimal batch size of *N=65536*. However, as illustrated in Figure 6.3, the standard fused kernel implementation utilizing conventional CUDA cores achieved a superior absolute peak throughput of 205,189.8 sig/s at an optimal batch size of *N=4096*.

While kernel fusion consistently outperformed the Tensor variant across the vast majority of batch workloads, a notable architectural intersection occurred at the extreme scale of *N=262,144*. At this specific workload, the Tensor Core implementation briefly overtook the standard fused architecture, recording 141,650.7 sig/s against the fused kernel's 133,419.4 sig/s.

This indicates that at highly extreme batch scales, the rigid memory staging of the Tensor Cores provided slight resilience against hardware scaling drop-offs, though the fused kernel implementation reclaimed the performance lead at *N=524,288*.

The general performance degradation observed in the Tensor Core variant across the optimal batch sizes was directly attributed to the severe data-formatting overhead required to bridge the hardware-software gap because Tensor Cores strictly operate on FP16 types and require specific memory alignments and the system was forced to execute heavy preprocessing steps within the kernel.

Firstly, the 24-bit finite-field integers had to be unpacked into three distinct 8-bit coefficient planes and cast to FP16 format utilizing the *__uint2half_rn* intrinsic. Secondly, to satisfy the strict 16×16×16 Warp Matrix Multiply Accumulate (WMMA) dimensional constraints, the cryptographic matrices were padded with zeros. The vinegar variable dimension *V=52* was padded to 64 and the oil variable dimension *M=18* was padded to 32. This padding forced the Tensor Cores to expend computational cycles multiplying zeros thus introducing wasted execution time. Finally, the massive dynamic Shared Memory allocations required to stage these FP16 matrices (exceeding 73 KB per block) limited the number of active thread blocks that could reside concurrently on a single Streaming Multiprocessor (SM), thereby lowering the overall hardware occupancy.

The benchmark conclusively verified that while mapping integer-based finite-field cryptography to floating-point Tensor Cores was architecturally viable as the raw multiply-accumulate (MAC) acceleration did not completely offset the heavy casting, padding, and memory alignment latencies. Consequently, highly optimized standard CUDA within a fused architecture proved to be the superior choice for maximizing peak throughput in the QR-UOV signature generation scheme.

6.3.4 Impact of Hybrid CPU-GPU Fallback (Rare Case)

A mathematical reality of the QR-UOV scheme was that the matrices generated during the signing process sometimes lacked of full rank. During the LU decomposition phase, these rank-deficient matrices or singular matrices could not be solved directly. Instead, they required the target vector to mathematically align with the matrix's null space, which necessitated continuously re-sampling the cryptographic salt and re-hashing the message until a consistent target vector was found.

Handling this edge condition primarily on the GPU would have required an indeterminate do-while loop within the CUDA kernel. Due to the Single Instruction, Multiple Threads (SIMT) architecture of modern GPUs, if even a single thread within a warp entered a prolonged loop, the entire warp would suffer from severe divergence. This phenomenon would have stalled the Streaming Multiprocessors (SMs) and completely collapsed the parallel efficiency of the batched execution.

To overcome this, the proposed architecture delegated these specific edge cases to the host CPU via a Hybrid Fallback Controller. The benchmark data provided a definitive analysis of the frequency and performance impact of this architectural decision.

Batch Size	Rare Case Count	Frequency Rate (%)	CPU Resolution Time (ms)	GPU Execution Time (ms)
256	1	0.39	0.282	1.236
1024	10	0.98	2.414	3.791
4096	30	0.73	6.049	13.913
32768	239	0.73	57.69	116.16
524288	4083	0.78	1691.596	1722.589

Table 6.7: Frequency and CPU Resolution Time of Rank-Deficient Matrices

As demonstrated in Table 6.7, the generation of a rank-deficient matrix proved to be a stable statistical anomaly, occurring consistently at a rate of approximately 0.7% to 1.0% across all batch sizes. Due to the low frequency, the host CPU was able to resolve the singular matrices with negligible overhead.

For instance, at the peak optimal batch size of $N=4096$, exactly 30 matrices triggered the fallback condition. The host CPU intercepted and resolved these 30 cases in just 6.049 ms. Even at the massive scale of $N=524,288$, where 4,083 singular matrices occurred, the host CPU

processed the iterative salt re-sampling flawlessly in 1.69 seconds. As modern CPUs possess sophisticated branch predictors and execute threads independently, the iterative do-while loop was processed highly efficiently on the host while the GPU finalized other operations.

In summary, this empirical data proved that the Hybrid CPU-GPU Fallback mechanism successfully guaranteed strict cryptographic correctness without inflicting any measurable penalty or stall on the overall parallel throughput of the system.

6.4 Signature Verification Results

The signature verification phase of the QR-UOV scheme inherently relies on the evaluation of public multivariate polynomial equations. Because this process consists purely of dense matrix-vector multiplications and does not require complex linear system solving or CPU fallback mechanisms, the execution pipeline is algorithmically simpler and highly parallelizable. The benchmark evaluated the scalability of the standard, integer-based CUDA verification implementation against the WMMA-accelerated Tensor Core variant across massive batch sizes, scaling up to *N=4,194,304*. The comprehensive benchmark results comparing both GPU verification architectures are summarized in Table 6.8.

Batch Size	CUDA Throughput (ver/s)	Tensor Throughput (ver/s)	Performance Ratio
1	26803.9	17639.2	1.51
2	136827.0	31123.6	4.39
4	165638.3	48391.0	3.42
8	187450.2	63031.3	2.97
16	227492.5	76120.5	2.98
32	259411.8	77649.2	3.34
64	273980.8	78423.2	3.49
128	286084.7	83777.1	3.41
256	293770.3	86553.8	3.39
512	296558.5	86740.2	3.41
1024	289616.5	87014.1	3.32
2048	292579.8	87107.2	3.35
4096	291986.1	87201.6	3.34
8192	292200.9	87182.0	3.35
16384	290949.7	117838.4	2.46
32768	320082.6	121114.1	2.64
65536	319799.6	121369.6	2.63
131072	313828.6	121450.7	2.58
262144	314534.2	121254.3	2.59
524288	313754.1	120802.6	2.59
1048576	311673.9	120312.8	2.59
2097152	309693.5	119905.3	2.58
4194304	295370.1	119042.0	2.48

Table 6.8: Benchmark Results for Signature Verification Architectures

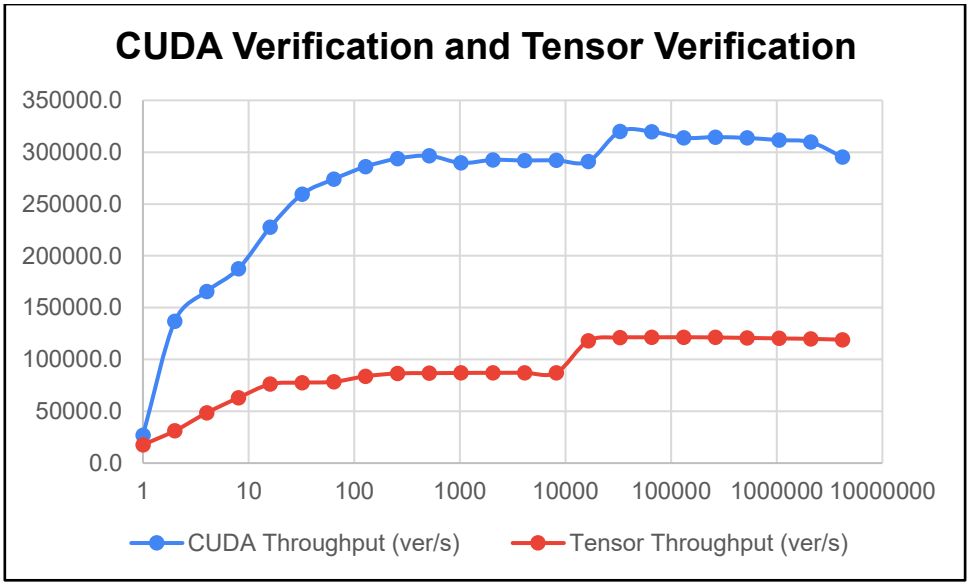


Figure 6.4: Verification Throughput Comparison between Standard CUDA and Tensor Core Architectures

As demonstrated by the empirical data, the standard CUDA verification kernel vastly outperformed the Tensor Core architecture across all evaluated batch sizes. The algorithmic simplicity of the matrix-vector evaluations, combined with the efficient fql_mac (multiply-accumulate) operations and warp-level reduction techniques, allowed the GPU to reach maximal hardware saturation rapidly. The standard CUDA implementation achieved an extraordinary peak throughput of 320,082.6 ver/s at a batch size of $N=32,768$. Furthermore, the architecture exhibited exceptional stability at extreme scales, maintaining a throughput of over 295,000 ver/s even when processing over 4.1 million signatures simultaneously.

In contrast, the Tensor Core implementation peaked at 121,450.7 ver/s at an optimal batch size of $N=131,072$. The substantial performance gap where the standard CUDA implementation delivered approximately 2.6x higher throughput highlighted a fundamental architectural limitation of applying AI-focused hardware to cryptographic verification.

As verification originally relies on matrix-vector multiplications which Tensor Cores cannot execute directly, the workload was forcefully transformed into dense matrix-matrix operations. This was achieved by grouping independent signatures into sub-batches of 16 ($TILE\_B = 16$). While this successfully engaged the Tensor Cores, the overhead of converting, padding (extending dimensions to multiples of 16), and staging the 24-bit finite-field integers into FP16 half-precision formats completely bottlenecked the execution. Additionally, staging these padded matrices required over 51 KB of dynamic Shared Memory per block, which

severely restricted the number of concurrent thread blocks that could reside on a Streaming Multiprocessor (SM).

The verification benchmark conclusively established that for cryptographic operations heavily reliant on matrix-vector mathematics, the latency introduced by data-formatting and memory alignment for the WMMA API negates the theoretical FLOP advantage of Tensor Cores. Consequently, highly streamlined, integer-based CUDA kernels serve as the definitive and optimal choice for real-time high-volume signature authentication.

CHAPTER 7 CONCLUSION AND RECOMMENDATION

7.1 Conclusion

This project successfully achieved its primary objective of accelerating the QR-UOV post-quantum signature scheme utilizing GPU parallelism. By translating the sequential mathematical structures of the algorithm into a highly concurrent execution pipeline, the system dramatically overcame the computational latency inherent to traditional CPU architecture.

The systematic evaluation proved that naively porting cryptographic kernels to the GPU is insufficient due to severe global memory bandwidth limitations. By implementing advanced CUDA optimization paradigms, specifically to Kernel Fusion and Shared Memory Tiling, the architecture successfully bypassed these memory bottlenecks. The fused implementation achieved an extraordinary peak throughput of 205,189.8 signatures per second and 320,082.6 verifications per second, delivering a 242x speedup over the baseline CPU and vastly exceeding the initial project goal of a 5-fold improvement.

Furthermore, the development of the Hybrid CPU-GPU Fallback controller proved highly effective. By strategically delegating rare rank-deficient matrices to the host CPU, the system guaranteed strict mathematical correctness without stalling the GPU's Streaming Multiprocessors, maintaining uninterrupted parallel efficiency. Lastly, the evaluation of the Tensor Core (WMMA) variant provided a valuable architectural analysis. It proved that while mapping integer-based finite-field cryptography to AI-focused FP16 hardware is viable, the data-formatting overhead currently makes standard fused CUDA cores the superior choice for minimizing latency in the QR-UOV scheme. Lastly, this project establishes a strong and highly scalable foundation for deploying post-quantum digital signatures in real-world, high-volume digital infrastructures.

7.2 Recommendations for Future Work

While this project successfully maximized the throughput of the QR-UOV signature scheme on a single Graphics Processing Unit (GPU), there remain several promising avenues for future research to further advance post-quantum cryptographic acceleration. One primary recommendation is the exploration of multi-GPU scaling. The current architecture optimizes resource utilization within the constraints of a single device's Streaming Multiprocessors (SMs) and global memory capacity. For enterprise-level cloud servers and massive Internet of Things (IoT) authentication gateways, future implementations could distribute cryptographic batches across multiple GPUs. By utilizing high-speed interconnects, the workload could be dynamically partitioned, theoretically allowing the cryptographic throughput to scale linearly alongside the addition of hardware.

Furthermore, the integration of specialized AI hardware for cryptography requires continued investigation. The evaluation of the Tensor Core variant demonstrated that the overhead of casting 24-bit integers to half-precision floating-point (FP16) formats bottlenecked the WMMA execution. Future work should explore the capabilities of next-generation GPU architectures such as NVIDIA Hopper, Ada Lovelace, or Blackwell which offer native hardware support for lower-precision data types like FP8 and INT8 Tensor Cores. Adapting the QR-UOV finite-field arithmetic to use these newer integer-focused matrix-multiply-accumulate instructions could potentially eliminate the casting and padding overheads, allowing Tensor Cores to definitively outperform standard CUDA cores in cryptographic operations.

Another critical area for future work is the integration of this GPU-accelerated scheme into live network protocol stacks. The current benchmarking framework evaluates the raw computational throughput in a localized and controlled environment. To validate the real-world impact of the accelerated QR-UOV scheme, future research should integrate the CUDA architecture into live communication protocols such as Transport Layer Security (TLS) 1.3 or MQTT for IoT device communication. Doing so would allow researchers to benchmark the end-to-end handshake latency and assess the impact of network transmission delays when using batched, post-quantum digital signatures.

Finally, the dynamic Hybrid CPU-GPU Fallback mechanism developed in this project provides a strong foundation for handling mathematical edge cases in parallel environments.

Future iterations could refine this architecture by introducing asynchronous execution. By utilizing multiple CUDA streams, the host CPU could resolve the rare rank-deficient matrices concurrently while the GPU immediately begins processing the next batch of signatures without waiting for a hard synchronization barrier. This complete overlapping of CPU and GPU execution would fully mask the latency of the fallback controller, ensuring absolute maximum hardware utilization in continuous real-time cryptographic pipelines.

REFERENCES

[1] Y. Ning, J. Dong, J. Lin, F. Zheng, Y. Fu, and F. Xiao, "GRASP: Accelerating Hash-based PQC Performance on GPU Parallel Architecture," *Cryptology ePrint Archive*, vol. 14, no. 8, Aug. 2021, [Online]. Available: https://eprint.iacr.org/2024/1030.pdf

[2] S. An and S. C. Seo, "Efficient Parallel Implementations of LWE-Based Post Quantum Cryptosystems on Graphics Processing Units," Mathematics, vol. 8, no. 10, p. 1781, Oct. 2020, doi: 10.3390/math8101781.

[3] M. Ivezic and M. Ivezic, "Harvest now, decrypt later (HNDL) risk," *PostQuantum - Quantum Computing, Quantum Security, PQC*, Jun. 07, 2024. https://postquantum.com/post-quantum/harvest-now-decrypt-later-hndl/

[4] B. H. M. E. A. Africa, "Harvest now, decrypt later," Black Hat Middle East and Africa, Jan. 01, 2025. https://insights.blackhatmea.com/harvest-now-decrypt-later/

[5] Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, U.S. Department of Commerce, "Post-Quantum Cryptography | CSRC | CSRC." https://csrc.nist.gov/projects/post-quantum-cryptography

[6] W. Li, H. Wei, S. Shen, H. Yang, W. Dai, and Y. Zhao, "cuFalcon: An Adaptive Parallel GPU Implementation for High-Performance Falcon Acceleration," Cryptology ePrint Archive, vol. 14, no. 8, Feb. 2025, [Online]. Available: https://eprint.iacr.org/2025/249.pdf

[7] W.-K. Lee, R. K. Zhao, R. Steinfeld, A. Sakzad, and S. O. Hwang, "High Throughput Lattice-Based Signatures on GPUs: Comparing Falcon and Mitaka," IEEE Trans. Parallel Distrib. Syst., vol. 35, no. 4, pp. 675–692, Apr. 2024, doi: 10.1109/TPDS.2024.3367319

[8] W.-K. Lee, H. Seo, Z. Zhang, and S. O. Hwang, "TensorCrypto: High Throughput Acceleration of Lattice-Based Cryptography Using Tensor Core on GPU," *IEEE Access*, vol. 10, pp. 20616–20632, 2022, doi: 10.1109/ACCESS.2022.3152217.

[9] F. Hoshino, H. Furue, Y. Ikematsu, T. Saito, Y. Kiyomura, and T. Takagi, "Efficient Software Implementation of Signature Scheme QR-UOV.". [Online]. Available: http://crypto.mist.i.u-tokyo.ac.jp/publications/1A1-5.pdfd

APPENDIX A

QRUOV_Sign_Separated

```
#include "qruov.h"
#include <cstdint>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <openssl/err.h>
#include <openssl/evp.h>
#include <openssl/sha.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define Fql_COPY(A,
C)                                                              \
  {
 \
    C =
A;                                                                      \
  }
#define MATRIX_TRANSPOSE(N, M, A,
C)                                              \
  {
 \
    int _i,
_j;                                                                 \
    _Pragma("omp parallel for private(_i,_j) shared(A, C)") for (_i =
0;       \
                                                                    _i <
N;       \
                                                                    _i++)
{       \
      for (_j = 0; _j < M; _j++)
{                                               \
        Fql_COPY(A[_i][_j],
C[_j][_i]);                                         \
      }
 \
    }
 \
  }

#define EQN(I, J) (eqn[I]->col[J])
```

```
void MATRIX_TRANSPOSE_VxM(const MATRIX_VxM A, MATRIX_MxV C) {
  MATRIX_TRANSPOSE(QRUOV_V, QRUOV_M, A, C);
}

static inline uint64_t h_Fq2Fql(const uint8_t *src) {
  return (uint64_t)src[0] | ((uint64_t)src[1] << Fql_ws) |
         ((uint64_t)src[2] << (Fql_ws * 2));
}

static void Expand_mu(const QRUOV_SEED seed_pk, const uint8_t message[],
                      const size_t message_length, uint8_t mu[QRUOV_MU_LEN]) {
  EVP_MD_CTX *ctx = EVP_MD_CTX_new();
  EVP_DigestInit_ex(ctx, EVP_shake256(), NULL);
  EVP_DigestUpdate(ctx, seed_pk, QRUOV_SEED_LEN);
  EVP_DigestUpdate(ctx, message, message_length);
  EVP_DigestFinalXOF(ctx, mu, QRUOV_MU_LEN);
  EVP_MD_CTX_free(ctx);
}

static void Hash(const uint8_t mu[QRUOV_MU_LEN], const QRUOV_SALT salt,
                 uint8_t dst[QRUOV_m]) {
  uint8_t tmp[QRUOV_tau4];
  EVP_MD_CTX *ctx = EVP_MD_CTX_new();
  EVP_DigestInit_ex(ctx, EVP_shake256(), NULL);
  EVP_DigestUpdate(ctx, mu, QRUOV_MU_LEN);
  EVP_DigestUpdate(ctx, salt, QRUOV_SALT_LEN);
  EVP_DigestFinalXOF(ctx, tmp, QRUOV_tau4);
  RejSamp(QRUOV_m, QRUOV_tau4, tmp);
  memcpy(dst, tmp, QRUOV_m);
  EVP_MD_CTX_free(ctx);
}

static void Expand_sol(const QRUOV_SEED seed_sol, uint8_t dst[QRUOV_m]) {
  const int n4 = QRUOV_L * QRUOV_M;
  uint8_t r4[QRUOV_tau4];
  QRUOV_PRG_CTX *ctx = PRG_init(seed_sol);
  RejSampPRG(ctx, 0, n4, QRUOV_tau4, r4);
  memcpy(dst, r4, n4);
  PRG_final(ctx);
}

Fq Fq_inv_table[QRUOV_q] = {
    0,   1,  64,  85,  32,  51,  106, 109, 16, 113, 89,  104, 53,  88, 118,
17,
    8,   15, 120, 107, 108, 121, 52,  116, 90, 61,  44,  80,  59,  92,
72,  41,
```

```
    4,   77, 71,  98,  60,  103, 117, 114, 54, 31,  124, 65,  26,  48,
58,  100,
    45,  70, 94,  5,   22,  12,  40,  97,  93, 78,  46,  28,  36,  25,
84,  125,
    2,   43, 102, 91,  99,  81,  49,  34,  30, 87,  115, 105, 122, 33,
57,  82,
    27,  69, 79,  101, 62,  3,   96,  73,  13, 10,  24,  67,  29,  56,
50,  123,
    86,  55, 35,  68,  47,  83,  66,  37,  11, 75,  6,   19,  20,  7,  112,
119,
    110, 9,  39,  74,  23,  38,  14,  111, 18, 21,  76,  95,  42,  63, 126};

inline static int Fq_reduction(int Z) {
  Z = (Z & QRUOV_q) + ((Z & ~QRUOV_q) >> QRUOV_ceil_log_2_q);
  int C = ((Z + 1) & ~QRUOV_q);
  Z += (C >> QRUOV_ceil_log_2_q);
  Z -= C;
  return Z;
}
inline static Fq Fq_sub(Fq X, Fq Y) {
  return (Fq)Fq_reduction((int)X - (int)Y + QRUOV_q);
}
inline static Fq Fq_mul(Fq X, Fq Y) {
  return (Fq)Fq_reduction((int)X * (int)Y);
}
inline static Fq Fq_inv(Fq X) { return Fq_inv_table[X]; }

#define EQN(I, J) (eqn[I]->col[J])

static void Fq_mxm_identity(Fq A[QRUOV_m][QRUOV_m]) {
  memset(A, 0, sizeof(Fq) * QRUOV_m * QRUOV_m);
  for (int i = 0; i < QRUOV_m; i++)
    A[i][i] = 1;
}

static void L_inverse(ECHELON_FORM echelon_form, Fq R[QRUOV_m][QRUOV_m]) {
  ROW_s **eqn = echelon_form->eqn;
  int rank = echelon_form->rank;
  Fq_mxm_identity(R);
  for (int i = 0; i < rank; i++) {
    Fq pivot = EQN(i, i);
    Fq inv = Fq_inv(pivot);
    for (int k = 0; k <= i; k++)
      R[i][k] = Fq_mul(R[i][k], inv);
    for (int j = i + 1; j < QRUOV_m; j++)
      for (int k = 0; k <= i; k++)
        R[j][k] = Fq_sub(R[j][k], Fq_mul(EQN(j, i), R[i][k]));
```

```
  }
}

static int consistent(ECHELON_FORM echelon_form, Fq b[QRUOV_m], int *cacheR,
                      Fq R[QRUOV_m][QRUOV_m]) {
  ROW_s **eqn = echelon_form->eqn;
  int rank = echelon_form->rank;
  if (rank == QRUOV_m)
    return 1;
  if (*cacheR == 0) {
    L_inverse(echelon_form, R);
    *cacheR = 1;
  }
  for (int i = rank; i < QRUOV_m; i++) {
    int k;
    uint64_t t = 0;
    for (int j = 0; j < rank; j++) {
      k = eqn[j]->original_row_id;
      t += ((uint64_t)R[i][j]) * ((uint64_t)b[k]);
    }
    k = eqn[i]->original_row_id;
    t += ((uint64_t)R[i][i]) * ((uint64_t)b[k]);
    t %= QRUOV_q;
    if (t)
      return 0;
  }
  return 1;
}

// --- Device ---
#define N_UPPER_TRI (QRUOV_V * (QRUOV_V + 1) / 2)  // 1378
#define N_VXM (QRUOV_V * QRUOV_M)                  // 936
#define N1 (QRUOV_L * QRUOV_V * (QRUOV_V + 1) / 2) // 4134
#define N2 (QRUOV_L * QRUOV_V * QRUOV_M)           // 2808

__global__ void ExpandSymmetricMatrixVxV_batched_kernel(
    const uint8_t *__restrict__ all_src, // [QRUOV_m * N1]
    uint64_t *__restrict__ all_A         // [QRUOV_m * V * V]
) {
  int iter = blockIdx.y;
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k >= N_UPPER_TRI)
    return;

  const uint8_t *src = all_src + iter * N1;
  uint64_t *A = all_A + iter * QRUOV_V * QRUOV_V;
```

```
  float fk = (float)k;
  int i = (int)((2.0f * QRUOV_V + 1.0f -
                 sqrtf((2.0f * QRUOV_V + 1.0f) * (2.0f * QRUOV_V + 1.0f) -
                       8.0f * fk)) /
                2.0f);
  if (i > 0) {
    int start_of_row_i = i * QRUOV_V - i * (i - 1) / 2;
    if (start_of_row_i > k)
      i--;
  }
  int start_of_row = i * QRUOV_V - i * (i - 1) / 2;
  int j = i + (k - start_of_row);

  int byte_offset = k * 3;
  uint64_t z0 = src[byte_offset];
  uint64_t z1 = src[byte_offset + 1];
  uint64_t z2 = src[byte_offset + 2];
  uint64_t fql = z0 | (z1 << 24) | (z2 << 48);

  A[i * QRUOV_V + j] = fql;
  if (i != j) {
    A[j * QRUOV_V + i] = fql;
  }
}

__global__ void ExpandMatrixVxM_batched_kernel(
    const uint8_t *__restrict__ all_src, // [QRUOV_m * N2]
    uint64_t *__restrict__ all_A         // [QRUOV_m * V * M]
) {
  int iter = blockIdx.y;
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k >= N_VXM)
    return;

  const uint8_t *src = all_src + iter * N2;
  uint64_t *A = all_A + iter * QRUOV_V * QRUOV_M;

  int byte_offset = k * 3;
  uint64_t z0 = src[byte_offset];
  uint64_t z1 = src[byte_offset + 1];
  uint64_t z2 = src[byte_offset + 2];
  uint64_t fql = z0 | (z1 << 24) | (z2 << 48);

  A[k] = fql;
}
```

```
__device__ void unpack(uint64_t fql, uint32_t &z0, uint32_t &z1, uint32_t &z2)
{
  z0 = (uint32_t)(fql & 0xFFFFFF);
  z1 = (uint32_t)((fql >> 24) & 0xFFFFFF);
  z2 = (uint32_t)(fql >> 48);
}

__device__ uint64_t repack(uint32_t z0, uint32_t z1, uint32_t z2) {
  return ((uint64_t)(z0 % QRUOV_q)) | ((uint64_t)(z1 % QRUOV_q) << 24) |
         ((uint64_t)(z2 % QRUOV_q) << 48);
}

__device__ uint64_t dot_product_fql(uint64_t *vec, uint64_t *mat_col_base,
                                    int K, int col, int row_stride) {
  uint32_t acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0, acc4 = 0;
  for (int k = 0; k < K; k++) {
    uint32_t a0, a1, a2, b0, b1, b2;
    unpack(vec[k], a0, a1, a2);
    unpack(mat_col_base[k * row_stride + col], b0, b1, b2);
    acc0 += a0 * b0;
    acc1 += a0 * b1 + a1 * b0;
    acc2 += a0 * b2 + a1 * b1 + a2 * b0;
    acc3 += a1 * b2 + a2 * b1;
    acc4 += a2 * b2;
  }
  uint32_t r0 = (acc0 + acc3) % QRUOV_q;
  uint32_t r1 = (acc1 + acc3 + acc4) % QRUOV_q;
  uint32_t r2 = (acc2 + acc4) % QRUOV_q;
  return repack(r0, r1, r2);
}

__device__ uint64_t d_Fql_mul(uint64_t X, uint64_t Y) {
  uint32_t a0, a1, a2, b0, b1, b2;
  unpack(X, a0, a1, a2);
  unpack(Y, b0, b1, b2);

  uint32_t c0 = (uint32_t)((uint64_t)a0 * b0);
  uint32_t c1 = (uint32_t)((uint64_t)a0 * b1 + (uint64_t)a1 * b0);
  uint32_t c2 =
      (uint32_t)((uint64_t)a0 * b2 + (uint64_t)a1 * b1 + (uint64_t)a2 * b0);
  uint32_t c3 = (uint32_t)((uint64_t)a1 * b2 + (uint64_t)a2 * b1);
  uint32_t c4 = (uint32_t)((uint64_t)a2 * b2);

  uint32_t r0 = (c0 + c3) % QRUOV_q;
  uint32_t r1 = (c1 + c3 + c4) % QRUOV_q;
  uint32_t r2 = (c2 + c4) % QRUOV_q;
  return repack(r0, r1, r2);
```

```
}

__global__ void kernel_VxV(uint64_t *d_yT,      // [QRUOV_V]
                           uint64_t *d_all_Pi1, // [QRUOV_m * QRUOV_V *
QRUOV_V]
                           uint64_t *d_yT_Pi1   // [QRUOV_m * QRUOV_V]  output
) {
  int tid = threadIdx.x;
  if (tid >= QRUOV_V)
    return;

  for (int i = blockIdx.x; i < QRUOV_m; i += gridDim.x) {
    uint64_t *my_Pi1 = d_all_Pi1 + i * QRUOV_V * QRUOV_V;
    uint64_t *my_out = d_yT_Pi1 + i * QRUOV_V;
    my_out[tid] = dot_product_fql(d_yT, my_Pi1, QRUOV_V, tid, QRUOV_V);
  }
}

__global__ void
kernel_VxM_Pi1Sd(uint64_t *d_yT_Pi1,   // [QRUOV_m * QRUOV_V]
                 uint64_t *d_Sd,       // [QRUOV_V * QRUOV_M]
                 uint64_t *d_yT_Pi1_Sd // [QRUOV_m * QRUOV_M]  output
) {
  int tid = threadIdx.x;
  if (tid >= QRUOV_M)
    return;

  for (int i = blockIdx.x; i < QRUOV_m; i += gridDim.x) {
    uint64_t *my_vec = d_yT_Pi1 + i * QRUOV_V;
    uint64_t *my_out = d_yT_Pi1_Sd + i * QRUOV_M;
    my_out[tid] = dot_product_fql(my_vec, d_Sd, QRUOV_V, tid, QRUOV_M);
  }
}

__global__ void
kernel_VxM_Pi2(uint64_t *d_yT,      // [QRUOV_V]
               uint64_t *d_all_Pi2, // [QRUOV_m * QRUOV_V * QRUOV_M]
               uint64_t *d_yT_Pi2   // [QRUOV_m * QRUOV_M]  output
) {
  int tid = threadIdx.x;
  if (tid >= QRUOV_M)
    return;

  for (int i = blockIdx.x; i < QRUOV_m; i += gridDim.x) {
    uint64_t *my_Pi2 = d_all_Pi2 + i * QRUOV_V * QRUOV_M;
    uint64_t *my_out = d_yT_Pi2 + i * QRUOV_M;
    my_out[tid] = dot_product_fql(d_yT, my_Pi2, QRUOV_V, tid, QRUOV_M);
```

```
  }
}

__global__ void kernel_M_SUB(uint64_t *d_yT_Pi2,    // [QRUOV_m * QRUOV_M]
                             uint64_t *d_yT_Pi1_Sd, // [QRUOV_m * QRUOV_M]
                             uint64_t *d_yT_Fi2 // [QRUOV_m * QRUOV_M]  output
) {
  int tid = threadIdx.x;
  if (tid >= QRUOV_M)
    return;

  for (int i = blockIdx.x; i < QRUOV_m; i += gridDim.x) {
    uint64_t *my_Pi2 = d_yT_Pi2 + i * QRUOV_M;
    uint64_t *my_Pi1Sd = d_yT_Pi1_Sd + i * QRUOV_M;
    uint64_t *my_out = d_yT_Fi2 + i * QRUOV_M;

    uint32_t a0, a1, a2, b0, b1, b2;
    unpack(my_Pi2[tid], a0, a1, a2);
    unpack(my_Pi1Sd[tid], b0, b1, b2);
    uint32_t r0 = (a0 + QRUOV_q - b0) % QRUOV_q;
    uint32_t r1 = (a1 + QRUOV_q - b1) % QRUOV_q;
    uint32_t r2 = (a2 + QRUOV_q - b2) % QRUOV_q;
    my_out[tid] = repack(r0, r1, r2);
  }
}

#define QRUOV_fe 1
#define
QRUOV_perm(i)                                                          \
  ((i <= (QRUOV_fe - 1)) ? ((QRUOV_fe - 1) -
i)                                  \
                         : (QRUOV_L + (QRUOV_fe - 1) - i))

__device__ uint8_t d_Fql2Fq(uint64_t Z, int i) {
  return ((Z >> ((Fql_ws)*i)) & QRUOV_q);
}

__device__ uint64_t d_Fql_add(uint64_t X, uint64_t Y) {
  uint64_t Z = X + Y;
  uint32_t z0 = ((uint32_t)Z & Fql_wm);
  Z >>= Fql_ws;
  uint32_t z1 = ((uint32_t)Z & Fql_wm);
  Z >>= Fql_ws;
  uint32_t z2 = (uint32_t)Z;
  z0 %= QRUOV_q;
  z1 %= QRUOV_q;
  z2 %= QRUOV_q;
```

```
  return ((uint64_t)z0) | ((uint64_t)z1 << 24) | ((uint64_t)z2 << 48);
}

__global__ void kernel_EQN_GEN(uint64_t *d_yT_Fi2, // [QRUOV_m * QRUOV_M]
                               uint8_t *d_eqn // [QRUOV_m * QRUOV_m]  output
) {
  int j = threadIdx.x;
  if (j >= QRUOV_M)
    return;

  for (int i = blockIdx.x; i < QRUOV_m; i += gridDim.x) {
    uint64_t *my_Fi2 = d_yT_Fi2 + i * QRUOV_M;
    uint8_t *my_eqn = d_eqn + i * QRUOV_m;

    uint64_t u = my_Fi2[j];
    u = d_Fql_add(u, u);

    for (int k = 0; k < QRUOV_L; k++) {
      int perm_k;
      if (k <= (QRUOV_fe - 1))
        perm_k = (QRUOV_fe - 1) - k;
      else
        perm_k = QRUOV_L + (QRUOV_fe - 1) - k;
      my_eqn[QRUOV_L * j + k] = d_Fql2Fq(u, perm_k);
    }
  }
}

__global__ void
kernel_C_GEN(uint64_t *d_yT_Pi1, // [QRUOV_m * QRUOV_V] (= yT_Fi1)
             uint64_t *d_yT,     // [QRUOV_V]
             uint8_t *d_c        // [QRUOV_m]  output
) {
  int tid = threadIdx.x;

  for (int i = blockIdx.x; i < QRUOV_m; i += gridDim.x) {
    uint64_t *my_Fi1 = d_yT_Pi1 + i * QRUOV_V;

    if (tid == 0) {
      uint32_t sum = 0;
      for (int j = 0; j < QRUOV_V; j++) {
        uint64_t product = d_Fql_mul(my_Fi1[j], d_yT[j]);
        sum += (uint32_t)d_Fql2Fq(product, 0);
      }
      d_c[i] = (uint8_t)(sum % QRUOV_q);
    }
  }
```

```
}

__constant__ Fq d_Fq_inv_table[QRUOV_q] = {
    0,   1,  64,  85,  32,  51,  106, 109, 16, 113, 89,  104, 53,  88, 118,
17,
    8,   15, 120, 107, 108, 121, 52,  116, 90, 61,  44,  80,  59,  92,
72,  41,
    4,   77, 71,  98,  60,  103, 117, 114, 54, 31,  124, 65,  26,  48,
58,  100,
    45,  70, 94,  5,   22,  12,  40,  97,  93, 78,  46,  28,  36,  25,
84,  125,
    2,   43, 102, 91,  99,  81,  49,  34,  30, 87,  115, 105, 122, 33,
57,  82,
    27,  69, 79,  101, 62,  3,   96,  73,  13, 10,  24,  67,  29,  56,
50,  123,
    86,  55, 35,  68,  47,  83,  66,  37,  11, 75,  6,   19,  20,  7,  112,
119,
    110, 9,  39,  74,  23,  38,  14,  111, 18, 21,  76,  95,  42,  63, 126};

__device__ __forceinline__ int d_Fq_reduction(int Z) {
  Z = (Z & QRUOV_q) + ((Z & ~QRUOV_q) >> QRUOV_ceil_log_2_q);
  int C = ((Z + 1) & ~QRUOV_q);
  Z += (C >> QRUOV_ceil_log_2_q);
  Z -= C;
  return Z;
}
__device__ __forceinline__ Fq d_Fq_sub(Fq X, Fq Y) {
  return (Fq)d_Fq_reduction((int)X - (int)Y + QRUOV_q);
}
__device__ __forceinline__ Fq d_Fq_mul(Fq X, Fq Y) {
  return (Fq)d_Fq_reduction((int)X * (int)Y);
}

__global__ void LU_decompose_batched_kernel(Fq *d_all_eqn, int *d_all_rank,
                                            int *d_all_indices, int *d_all_p,
                                            int batch_N) {
  int sig_id = blockIdx.x;
  if (sig_id >= batch_N)
    return;
  Fq *global_A = d_all_eqn + sig_id * QRUOV_m * QRUOV_m;
  int *d_rank = d_all_rank + sig_id;
  int *d_indices = d_all_indices + sig_id * QRUOV_m;
  int *d_p = d_all_p + sig_id * QRUOV_m;
  __shared__ Fq s_A[QRUOV_m][QRUOV_m];
  __shared__ int p[QRUOV_m];
  __shared__ int pivot_col_idx[QRUOV_m];
  __shared__ int current_rank;
```

```
__shared__ int s_i, s_c, found_j;
int tid = threadIdx.x;
if (tid < QRUOV_m) {
  p[tid] = tid;
  pivot_col_idx[tid] = -1;
}
for (int idx = tid; idx < QRUOV_m * QRUOV_m; idx += blockDim.x)
  s_A[idx / QRUOV_m][idx % QRUOV_m] = global_A[idx];
if (tid == 0) {
  current_rank = 0;
  s_i = 0;
  s_c = 0;
}
__syncthreads();
while (s_i < QRUOV_m && s_c < QRUOV_m) {
  __shared__ int candidate;
  if (tid == 0)
    candidate = QRUOV_m;
  __syncthreads();
  {
    int my_row = s_i + tid;
    if (my_row < QRUOV_m && s_A[p[my_row]][s_c] != 0)
      atomicMin(&candidate, my_row);
  }
  __syncthreads();
  if (tid == 0)
    found_j = (candidate < QRUOV_m) ? candidate : -1;
  __syncthreads();
  if (found_j == -1) {
    if (tid == 0)
      s_c++;
    __syncthreads();
    continue;
  }
  if (tid == 0) {
    int temp = p[s_i];
    p[s_i] = p[found_j];
    p[found_j] = temp;
    pivot_col_idx[current_rank] = s_c;
    current_rank++;
  }
  __syncthreads();
  int pivot_row = p[s_i];
  Fq pivot_val = s_A[pivot_row][s_c];
  Fq inv = d_Fq_inv_table[pivot_val];
  if (tid == 0)
    s_A[pivot_row][s_i] = pivot_val;
```

```
    for (int k = s_c + 1 + tid; k < QRUOV_m; k += blockDim.x)
      s_A[pivot_row][k] = d_Fq_mul(s_A[pivot_row][k], inv);
    __syncthreads();
    for (int row_idx = s_i + 1 + tid; row_idx < QRUOV_m;
         row_idx += blockDim.x) {
      int tr = p[row_idx];
      s_A[tr][s_i] = s_A[tr][s_c];
    }
    __syncthreads();
    int num_er = QRUOV_m - (s_i + 1), num_ec = QRUOV_m - (s_c + 1);
    for (int w = tid; w < num_er * num_ec; w += blockDim.x) {
      int ri = s_i + 1 + w / num_ec, ci = s_c + 1 + w % num_ec;
      int tr = p[ri];
      s_A[tr][ci] =
          d_Fq_sub(s_A[tr][ci], d_Fq_mul(s_A[tr][s_i], s_A[pivot_row][ci]));
    }
    __syncthreads();
    if (tid == 0) {
      s_i++;
      s_c++;
    }
    __syncthreads();
  }
  if (tid < QRUOV_m) {
    d_indices[tid] = pivot_col_idx[tid];
    d_p[tid] = p[tid];
  }
  if (tid == 0)
    *d_rank = current_rank;
  for (int idx = tid; idx < QRUOV_m * QRUOV_m; idx += blockDim.x) {
    int lr = idx / QRUOV_m, c = idx % QRUOV_m;
    global_A[lr * QRUOV_m + c] = s_A[p[lr]][c];
  }
}

__global__ void sample_a_solution_batched_kernel(
    Fq *d_all_eqn, int *d_all_p, int *d_all_rank, int *d_all_indices,
    Fq *d_all_b, Fq *d_all_random, Fq *d_all_oil_u, Fq *d_all_b2, int batch_N)
{
  int sid = blockIdx.x * blockDim.x + threadIdx.x;
  if (sid >= batch_N)
    return;
  Fq *eqn = d_all_eqn + sid * QRUOV_m * QRUOV_m;
  int *pp = d_all_p + sid * QRUOV_m;
  int rank = d_all_rank[sid];
  int *indices = d_all_indices + sid * QRUOV_m;
  Fq *b = d_all_b + sid * QRUOV_m;
```

```
  Fq *rand_element = d_all_random + sid * QRUOV_m;
  Fq *oil_u = d_all_oil_u + sid * QRUOV_m;
  Fq *b2 = d_all_b2 + sid * QRUOV_m;
  for (int i = 0; i < rank; i++) {
    uint64_t t = 0;
    for (int j = 0; j < i; j++)
      t += ((uint64_t)eqn[i * QRUOV_m + j]) * ((uint64_t)b2[j]);
    Fq tmp = d_Fq_sub(b[pp[i]], (Fq)(t % QRUOV_q));
    b2[i] = d_Fq_mul(tmp, d_Fq_inv_table[eqn[i * QRUOV_m + i]]);
  }
  int i = QRUOV_m - 1;
  for (int j = rank - 1; j >= 0; j--) {
    int k = indices[j];
    for (; i > k; i--)
      oil_u[i] = *rand_element++;
    uint64_t t = 0;
    for (int kk = k + 1; kk < QRUOV_m; kk++)
      t += ((uint64_t)eqn[j * QRUOV_m + kk]) * ((uint64_t)(oil_u[kk]));
    oil_u[i] = d_Fq_sub(b2[j], (Fq)(t % QRUOV_q));
    i--;
  }
  for (; i >= 0; i--)
    oil_u[i] = *rand_element++;
}

__global__ void ExpandMatrixVxM_kernel(const uint8_t *__restrict__ src,
                                       uint64_t *__restrict__ A) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < QRUOV_V * QRUOV_M) {
    int bo = idx * QRUOV_L;
    A[idx] = (uint64_t)src[bo] | ((uint64_t)src[bo + 1] << Fql_ws) |
             ((uint64_t)src[bo + 2] << (Fql_ws * 2));
  }
}

__global__ void MATRIX_TRANSPOSE_VxM_kernel(const uint64_t *__restrict__ A,
                                            uint64_t *__restrict__ C) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < QRUOV_V && j < QRUOV_M)
    C[j * QRUOV_V + i] = A[i * QRUOV_M + j];
}

__global__ void pack0_vinegar_batched_kernel(Fq *d_all_oil_u, uint64_t *d_SdT,
                                             uint64_t *d_all_yT,
                                             uint64_t *d_all_sig_s,
                                             int batch_N) {
```

```
  int gid = blockIdx.x * blockDim.x + threadIdx.x;
  if (gid >= batch_N * QRUOV_V)
    return;
  int sid = gid / QRUOV_V;
  int i = gid % QRUOV_V;
  Fq *oil_u = d_all_oil_u + sid * QRUOV_m;
  uint64_t *yT = d_all_yT + sid * QRUOV_V;
  uint64_t *sig_s = d_all_sig_s + sid * QRUOV_N;

  uint32_t acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0, acc4 = 0;
  for (int j = 0; j < QRUOV_M; j++) {
    uint64_t oj = (uint64_t)oil_u[j * QRUOV_L] |
                  ((uint64_t)oil_u[j * QRUOV_L + 1] << Fql_ws) |
                  ((uint64_t)oil_u[j * QRUOV_L + 2] << (Fql_ws * 2));
    uint64_t sdt = d_SdT[j * QRUOV_V + i];
    uint32_t a0, a1, a2, b0, b1, b2;
    unpack(oj, a0, a1, a2);
    unpack(sdt, b0, b1, b2);
    acc0 += a0 * b0;
    acc1 += a0 * b1 + a1 * b0;
    acc2 += a0 * b2 + a1 * b1 + a2 * b0;
    acc3 += a1 * b2 + a2 * b1;
    acc4 += a2 * b2;
  }
  uint64_t dot = repack((acc0 + acc3) % QRUOV_q, (acc1 + acc3 + acc4) %
QRUOV_q,
                        (acc2 + acc4) % QRUOV_q);
  uint64_t vin = yT[i];
  uint32_t v0, v1, v2, d0, d1, d2;
  unpack(vin, v0, v1, v2);
  unpack(dot, d0, d1, d2);
  sig_s[i] =
      repack((v0 + QRUOV_q - d0) % QRUOV_q, (v1 + QRUOV_q - d1) % QRUOV_q,
             (v2 + QRUOV_q - d2) % QRUOV_q);
}

__global__ void pack0_oil_batched_kernel(Fq *d_all_oil_u, uint64_t
*d_all_sig_s,
                                         int batch_N) {
  int gid = blockIdx.x * blockDim.x + threadIdx.x;
  if (gid >= batch_N * QRUOV_M)
    return;
  int sid = gid / QRUOV_M;
  int oi = gid % QRUOV_M;
  Fq *oil_u = d_all_oil_u + sid * QRUOV_m;
  uint64_t *sig_s = d_all_sig_s + sid * QRUOV_N;
```

```
  sig_s[QRUOV_V + oi] = (uint64_t)oil_u[oi * QRUOV_L] |
                        ((uint64_t)oil_u[oi * QRUOV_L + 1] << Fql_ws) |
                        ((uint64_t)oil_u[oi * QRUOV_L + 2] << (Fql_ws * 2));
}

static void
Expand_pk_batch_on_gpu(QRUOV_PRG_CTX *ctx0,
                       uint64_t *d_all_Pi1, // device ptr [QRUOV_m * V * V]
                       uint64_t *d_all_Pi2  // device ptr [QRUOV_m * V * M]
) {
  uint8_t *all_r1 = (uint8_t *)malloc(QRUOV_m * N1);
  uint8_t *all_r2 = (uint8_t *)malloc(QRUOV_m * N2);

  for (int i = 0; i < QRUOV_m; i++) {
    QRUOV_PRG_CTX *ctx1 = PRG_copy(ctx0);
    uint8_t tmp1[QRUOV_tau1];
    RejSampPRG(ctx1, 2 * i, N1, QRUOV_tau1, tmp1);
    PRG_final(ctx1);
    memcpy(all_r1 + i * N1, tmp1, N1);

    QRUOV_PRG_CTX *ctx2 = PRG_copy(ctx0);
    uint8_t tmp2[QRUOV_tau2];
    RejSampPRG(ctx2, 2 * i + 1, N2, QRUOV_tau2, tmp2);
    PRG_final(ctx2);
    memcpy(all_r2 + i * N2, tmp2, N2);
  }

  uint8_t *d_all_r1, *d_all_r2;
  cudaMalloc(&d_all_r1, QRUOV_m * N1 * sizeof(uint8_t));
  cudaMalloc(&d_all_r2, QRUOV_m * N2 * sizeof(uint8_t));

  cudaMemset(d_all_Pi1, 0, QRUOV_m * QRUOV_V * QRUOV_V * sizeof(uint64_t));
  cudaMemset(d_all_Pi2, 0, QRUOV_m * QRUOV_V * QRUOV_M * sizeof(uint64_t));

  cudaMemcpy(d_all_r1, all_r1, QRUOV_m * N1, cudaMemcpyHostToDevice);
  cudaMemcpy(d_all_r2, all_r2, QRUOV_m * N2, cudaMemcpyHostToDevice);

  int threads = 256;
  dim3 grid_vxv((N_UPPER_TRI + threads - 1) / threads, QRUOV_m);
  ExpandSymmetricMatrixVxV_batched_kernel<<<grid_vxv, threads>>>(d_all_r1,
                                                                 d_all_Pi1);

  dim3 grid_vxm((N_VXM + threads - 1) / threads, QRUOV_m);
  ExpandMatrixVxM_batched_kernel<<<grid_vxm, threads>>>(d_all_r2, d_all_Pi2);

  cudaDeviceSynchronize();
```

```
  cudaFree(d_all_r1);
  cudaFree(d_all_r2);
  free(all_r1);
  free(all_r2);
}

void QRUOV_Sign(const QRUOV_SEED seed_sk, const QRUOV_SEED seed_pk,
                const uint8_t message[], const size_t message_length,
                QRUOV_SIGNATURE sig) {
  cudaDeviceProp prop;
  cudaGetDeviceProperties(&prop, 0);
  size_t free_mem, total_mem;
  cudaMemGetInfo(&free_mem, &total_mem);
  printf("GPU: %s | Memory: %.0f MB free / %.0f MB total\n\n", prop.name,
         free_mem / 1e6, total_mem / 1e6);

  uint8_t mu[QRUOV_MU_LEN];
  Expand_mu(seed_pk, message, message_length, mu);

  // Expand_sk (shared) -> d_Sd, d_SdT
  uint64_t *d_Sd, *d_SdT;
  cudaMalloc(&d_Sd, QRUOV_V * QRUOV_M * sizeof(uint64_t));
  cudaMalloc(&d_SdT, QRUOV_M * QRUOV_V * sizeof(uint64_t));
  {
    const int n2 = QRUOV_L * QRUOV_V * QRUOV_M;
    uint8_t r2[QRUOV_tau2];
    QRUOV_PRG_CTX *ctx = PRG_init(seed_sk);
    RejSampPRG(ctx, 0, n2, QRUOV_tau2, r2);
    uint8_t *d_r2;
    cudaMalloc(&d_r2, QRUOV_tau2);
    cudaMemcpy(d_r2, r2, QRUOV_tau2, cudaMemcpyHostToDevice);
    ExpandMatrixVxM_kernel<<<4, 256>>>(d_r2, d_Sd);
    dim3 bd(16, 16), gd((QRUOV_V + 15) / 16, (QRUOV_M + 15) / 16);
    MATRIX_TRANSPOSE_VxM_kernel<<<gd, bd>>>(d_Sd, d_SdT);
    cudaFree(d_r2);
    PRG_final(ctx);
    OPENSSL_cleanse(r2, QRUOV_tau2);
  }

  // Expand_pk (shared)
  uint64_t *d_all_Pi1, *d_all_Pi2;
  cudaMalloc(&d_all_Pi1, QRUOV_m * QRUOV_V * QRUOV_V * sizeof(uint64_t));
  cudaMalloc(&d_all_Pi2, QRUOV_m * QRUOV_V * QRUOV_M * sizeof(uint64_t));
  {
    QRUOV_PRG_CTX *ctx_pk = PRG_init(seed_pk);
    Expand_pk_batch_on_gpu(ctx_pk, d_all_Pi1, d_all_Pi2);
    PRG_final(ctx_pk);
```

```
  }

  int batch_sizes[SIGNATURE_SIZE];
  for (int i = 0; i < SIGNATURE_SIZE; i++)
    batch_sizes[i] = 1 << i;

  for (int bi = 0; bi < SIGNATURE_SIZE; bi++) {
    int N = batch_sizes[bi];
    size_t per_sig =
        (size_t)QRUOV_V * sizeof(uint64_t)       // d_all_yT        [V *
uint64]
        + (size_t)QRUOV_m * QRUOV_m * sizeof(Fq) // d_all_eqn       [m * m *
Fq]
        + (size_t)QRUOV_m * sizeof(Fq)           // d_all_c         [m * Fq]
        + sizeof(int)                            // d_all_rank      [1 * int]
        + (size_t)QRUOV_m * sizeof(int) *
              2 // d_all_indices + d_all_p  [m * int] x2
        +
        (size_t)QRUOV_m * sizeof(Fq) * 4 // d_all_b + d_all_random_buf +
                                         // d_all_oil_u + d_all_b2  [m * Fq]
x4
        + (size_t)QRUOV_N * sizeof(uint64_t) // d_all_sig_s     [N * uint64]
        + (size_t)QRUOV_m * QRUOV_V *
              sizeof(uint64_t) // d_yT_Pi1        [m * V * uint64]
        + (size_t)QRUOV_m * QRUOV_M * sizeof(uint64_t) *
              3; // d_yT_Pi1_Sd + d_yT_Pi2 + d_yT_Fi2  [m * M * uint64] x3
    cudaMemGetInfo(&free_mem, &total_mem);
    if ((size_t)N * per_sig > free_mem * 0.85) {
      printf("%-12d | Skipped — need %.0f MB, only %.0f MB free\n", N,
             (double)N * per_sig / 1e6, free_mem / 1e6);
      break;
    }

    uint64_t *h_all_yT = (uint64_t *)malloc(N * QRUOV_V * sizeof(uint64_t));
    Fq *h_all_b = (Fq *)malloc(N * QRUOV_m);
    Fq *h_all_random = (Fq *)malloc(N * QRUOV_m);
    uint8_t *h_all_salts = (uint8_t *)malloc(N * QRUOV_SALT_LEN);
    QRUOV_PRG2_CTX **all_ctx_r =
        (QRUOV_PRG2_CTX **)malloc(N * sizeof(QRUOV_PRG2_CTX *));

    for (int n = 0; n < N; n++) {
      QRUOV_SEED sy;
      randombytes(sy, QRUOV_SEED_LEN);
      const int n3 = QRUOV_L * QRUOV_V;
      uint8_t r3[QRUOV_tau3];
      QRUOV_PRG_CTX *cy = PRG_init(sy);
      RejSampPRG(cy, 0, n3, QRUOV_tau3, r3);
```

```
    PRG_final(cy);
    for (int v = 0; v < QRUOV_V; v++)
      h_all_yT[n * QRUOV_V + v] = h_Fq2Fql(r3 + v * QRUOV_L);

    QRUOV_SEED sr;
    randombytes(sr, QRUOV_SEED_LEN);
    all_ctx_r[n] = PRG2_init(sr);

    QRUOV_SEED ss;
    randombytes(ss, QRUOV_SEED_LEN);
    Expand_sol(ss, h_all_random + n * QRUOV_m);
  }

  uint64_t *d_all_yT;
  Fq *d_all_eqn, *d_all_c;
  int *d_all_rank, *d_all_indices, *d_all_p;
  Fq *d_all_b, *d_all_random_buf, *d_all_oil_u, *d_all_b2;
  uint64_t *d_all_sig_s;
  uint64_t *d_yT_Pi1, *d_yT_Pi1_Sd, *d_yT_Pi2, *d_yT_Fi2;

  cudaMalloc(&d_all_yT, N * QRUOV_V * sizeof(uint64_t));
  cudaMalloc(&d_all_eqn, N * QRUOV_m * QRUOV_m);
  cudaMalloc(&d_all_c, N * QRUOV_m);
  cudaMalloc(&d_all_rank, N * sizeof(int));
  cudaMalloc(&d_all_indices, N * QRUOV_m * sizeof(int));
  cudaMalloc(&d_all_p, N * QRUOV_m * sizeof(int));
  cudaMalloc(&d_all_b, N * QRUOV_m);
  cudaMalloc(&d_all_random_buf, N * QRUOV_m);
  cudaMalloc(&d_all_oil_u, N * QRUOV_m);
  cudaMalloc(&d_all_b2, N * QRUOV_m);
  cudaMalloc(&d_all_sig_s, N * QRUOV_N * sizeof(uint64_t));
  cudaMalloc(&d_yT_Pi1, N * QRUOV_m * QRUOV_V * sizeof(uint64_t));
  cudaMalloc(&d_yT_Pi1_Sd, N * QRUOV_m * QRUOV_M * sizeof(uint64_t));
  cudaMalloc(&d_yT_Pi2, N * QRUOV_m * QRUOV_M * sizeof(uint64_t));
  cudaMalloc(&d_yT_Fi2, N * QRUOV_m * QRUOV_M * sizeof(uint64_t));

  cudaMemcpy(d_all_yT, h_all_yT, N * QRUOV_V * sizeof(uint64_t),
             cudaMemcpyHostToDevice);
  cudaMemcpy(d_all_random_buf, h_all_random, N * QRUOV_m,
             cudaMemcpyHostToDevice);

  for (int n = 0; n < N; n++) {
    kernel_VxV<<<NUM_BLOCKS, QRUOV_V>>>(d_all_yT + n * QRUOV_V, d_all_Pi1,
                                        d_yT_Pi1 + n * QRUOV_m * QRUOV_V);
  }
  for (int n = 0; n < N; n++) {
    kernel_VxM_Pi1Sd<<<NUM_BLOCKS, QRUOV_M>>>(
```

```
            d_yT_Pi1 + n * QRUOV_m * QRUOV_V, d_Sd,
            d_yT_Pi1_Sd + n * QRUOV_m * QRUOV_M);
    }
    for (int n = 0; n < N; n++) {
      kernel_VxM_Pi2<<<NUM_BLOCKS, QRUOV_M>>>(d_all_yT + n * QRUOV_V,
d_all_Pi2,
                                              d_yT_Pi2 + n * QRUOV_m *
QRUOV_M);
    }
    for (int n = 0; n < N; n++) {
      kernel_M_SUB<<<NUM_BLOCKS, QRUOV_M>>>(d_yT_Pi2 + n * QRUOV_m * QRUOV_M,
                                            d_yT_Pi1_Sd + n * QRUOV_m *
QRUOV_M,
                                            d_yT_Fi2 + n * QRUOV_m * QRUOV_M);
    }
    for (int n = 0; n < N; n++) {
      kernel_EQN_GEN<<<NUM_BLOCKS, QRUOV_M>>>(
          d_yT_Fi2 + n * QRUOV_m * QRUOV_M, d_all_eqn + n * QRUOV_m *
QRUOV_m);
    }
    for (int n = 0; n < N; n++) {
      kernel_C_GEN<<<NUM_BLOCKS, QRUOV_V>>>(d_yT_Pi1 + n * QRUOV_m * QRUOV_V,
                                            d_all_yT + n * QRUOV_V,
                                            d_all_c + n * QRUOV_m);
    }

    LU_decompose_batched_kernel<<<N, 128>>>(d_all_eqn, d_all_rank,
                                            d_all_indices, d_all_p, N);

    Fq msg[QRUOV_m];
    Fq *h_all_eqn = (Fq *)malloc(N * QRUOV_m * QRUOV_m);
    Fq *h_all_c = (Fq *)malloc(N * QRUOV_m);
    int *h_all_rank = (int *)malloc(N * sizeof(int));
    int *h_all_p = (int *)malloc(N * QRUOV_m * sizeof(int));
    int *h_all_indices = (int *)malloc(N * QRUOV_m * sizeof(int));
    cudaMemcpy(h_all_eqn, d_all_eqn, N * QRUOV_m * QRUOV_m,
               cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_p, d_all_p, N * QRUOV_m * sizeof(int),
               cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_indices, d_all_indices, N * QRUOV_m * sizeof(int),
               cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_c, d_all_c, N * QRUOV_m, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_rank, d_all_rank, N * sizeof(int),
cudaMemcpyDeviceToHost);

    int rare_case = 0;
    for (int n = 0; n < N; n++) {
```

```
      QRUOV_PRG2_CTX *ctx_r = all_ctx_r[n];
      Fq *my_b = h_all_b + n * QRUOV_m;
      Fq *my_c = h_all_c + n * QRUOV_m;
      if (h_all_rank[n] == QRUOV_m) {
        PRG2_yield(ctx_r, QRUOV_SALT_LEN, h_all_salts + n * QRUOV_SALT_LEN);
        Hash(mu, h_all_salts + n * QRUOV_SALT_LEN, msg);
        for (int i = 0; i < QRUOV_m; i++)
          my_b[i] = Fq_sub(msg[i], my_c[i]);
      } else {
        rare_case++;
        Fq *eqn_flat = h_all_eqn + n * QRUOV_m * QRUOV_m;
        int *p_vec = h_all_p + n * QRUOV_m;
        int *idx_vec = h_all_indices + n * QRUOV_m;
        ECHELON_FORM echelon_form;
        Fq eqn_storage[QRUOV_m][QRUOV_m];
        for (int r = 0; r < QRUOV_m; r++) {
          for (int c = 0; c < QRUOV_m; c++)
            eqn_storage[r][c] = eqn_flat[r * QRUOV_m + c];
          echelon_form->row[r].col = eqn_storage[r];
          echelon_form->row[r].original_row_id = p_vec[r];
        }
        for (int r = 0; r < QRUOV_m; r++)
          echelon_form->eqn[r] = &echelon_form->row[r];
        echelon_form->rank = h_all_rank[n];
        for (int r = 0; r < QRUOV_m; r++)
          echelon_form->index[r] = idx_vec[r];
        Fq R[QRUOV_m][QRUOV_m];
        int cacheR = 0;
        do {
          PRG2_yield(ctx_r, QRUOV_SALT_LEN, h_all_salts + n * QRUOV_SALT_LEN);
          Hash(mu, h_all_salts + n * QRUOV_SALT_LEN, msg);
          for (int i = 0; i < QRUOV_m; i++)
            my_b[i] = Fq_sub(msg[i], my_c[i]);
        } while (!consistent(echelon_form, my_b, &cacheR, R));
      }
      PRG2_final(ctx_r);
    }
    free(all_ctx_r);
    free(h_all_eqn);
    free(h_all_p);
    free(h_all_indices);

    cudaMemcpy(d_all_b, h_all_b, N * QRUOV_m, cudaMemcpyHostToDevice);
    cudaMemcpy(d_all_rank, h_all_rank, N * sizeof(int),
cudaMemcpyHostToDevice);

    int threads_sa = 256, blocks_sa = (N + threads_sa - 1) / threads_sa;
```

```
    int total_vin = N * QRUOV_V;
    int total_oil = N * QRUOV_M;


    sample_a_solution_batched_kernel<<<blocks_sa, threads_sa>>>(
        d_all_eqn, d_all_p, d_all_rank, d_all_indices, d_all_b,
        d_all_random_buf, d_all_oil_u, d_all_b2, N);
    pack0_vinegar_batched_kernel<<<(total_vin + 255) / 256, 256>>>(
        d_all_oil_u, d_SdT, d_all_yT, d_all_sig_s, N);
    pack0_oil_batched_kernel<<<(total_oil + 255) / 256, 256>>>(d_all_oil_u,
                                                               d_all_sig_s,
N);


    uint64_t *h_all_sig_s = (uint64_t *)malloc(N * QRUOV_N *
sizeof(uint64_t));
    cudaMemcpy(h_all_sig_s, d_all_sig_s, N * QRUOV_N * sizeof(uint64_t),
               cudaMemcpyDeviceToHost);
    memcpy(sig->r, h_all_salts, QRUOV_SALT_LEN);
    memcpy(sig->s, h_all_sig_s, QRUOV_N * sizeof(uint64_t));

    free(h_all_yT);
    free(h_all_b);
    free(h_all_random);
    free(h_all_salts);
    free(h_all_c);
    free(h_all_rank);
    free(h_all_sig_s);
    cudaFree(d_all_yT);
    cudaFree(d_all_eqn);
    cudaFree(d_all_c);
    cudaFree(d_all_rank);
    cudaFree(d_all_indices);
    cudaFree(d_all_p);
    cudaFree(d_all_b);
    cudaFree(d_all_random_buf);
    cudaFree(d_all_oil_u);
    cudaFree(d_all_b2);
    cudaFree(d_all_sig_s);
    cudaFree(d_yT_Pi1);
    cudaFree(d_yT_Pi1_Sd);
    cudaFree(d_yT_Pi2);
    cudaFree(d_yT_Fi2);
  }
  printf("\nDone\n");
  cudaFree(d_all_Pi1);
  cudaFree(d_all_Pi2);
```

```
  cudaFree(d_Sd);
  cudaFree(d_SdT);
}
```

QRUOV_Sign_Tensor

```
#include "qruov.h"
#include <cstdint>
#include <cstdio>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <mma.h>
#include <openssl/err.h>
#include <openssl/evp.h>
#include <openssl/sha.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
using namespace nvcuda;

static void Expand_mu(const QRUOV_SEED seed_pk,    // input
                      const uint8_t message[],     // input
                      const size_t message_length, // input (message)
                      uint8_t mu[QRUOV_MU_LEN]     // output mu[64]
) {
  EVP_MD_CTX *ctx = EVP_MD_CTX_new();
  EVP_DigestInit_ex(ctx, EVP_shake256(), NULL);
  EVP_DigestUpdate(ctx, seed_pk, QRUOV_SEED_LEN);
  EVP_DigestUpdate(ctx, message, message_length);
  EVP_DigestFinalXOF(ctx, mu, QRUOV_MU_LEN);
  EVP_MD_CTX_free(ctx);
}

static void Hash(const uint8_t mu[QRUOV_MU_LEN], // input
                 const QRUOV_SALT salt,          // input
                 uint8_t dst[QRUOV_m]            // output
) {
  uint8_t tmp[QRUOV_tau4];
  EVP_MD_CTX *ctx = EVP_MD_CTX_new();
  EVP_DigestInit_ex(ctx, EVP_shake256(), NULL);
  EVP_DigestUpdate(ctx, mu, QRUOV_MU_LEN);
  EVP_DigestUpdate(ctx, salt, QRUOV_SALT_LEN);
  EVP_DigestFinalXOF(ctx, tmp, QRUOV_tau4);
  RejSamp(QRUOV_m, QRUOV_tau4, tmp);
  memcpy(dst, tmp, QRUOV_m);
  EVP_MD_CTX_free(ctx);
  return;
}
```

```c
/* ----------------------------------------

   ---------------------------------------- */
static void Expand_sol(const QRUOV_SEED seed_sol, // input
                       uint8_t dst[QRUOV_m]       // output
) {
  const int n4 = QRUOV_L * QRUOV_M;
  uint8_t r4[QRUOV_tau4];
  QRUOV_PRG_CTX *ctx = PRG_init(seed_sol);
  RejSampPRG(ctx, 0, n4, QRUOV_tau4, r4);
  memcpy(dst, r4, n4);
  PRG_final(ctx);
}

Fq Fq_inv_table[QRUOV_q] = {
    0,   1,  64,  85,  32,  51,  106, 109, 16, 113, 89,  104, 53,  88, 118,
17,
    8,   15, 120, 107, 108, 121, 52,  116, 90, 61,  44,  80,  59,  92,
72,  41,
    4,   77, 71,  98,  60,  103, 117, 114, 54, 31,  124, 65,  26,  48,
58,  100,
    45,  70, 94,  5,   22,  12,  40,  97,  93, 78,  46,  28,  36,  25,
84,  125,

    2,   43, 102, 91,  99,  81,  49,  34,  30, 87,  115, 105, 122, 33,
57,  82,
    27,  69, 79,  101, 62,  3,   96,  73,  13, 10,  24,  67,  29,  56,
50,  123,
    86,  55, 35,  68,  47,  83,  66,  37,  11, 75,  6,   19,  20,  7,  112,
119,
    110, 9,  39,  74,  23,  38,  14,  111, 18, 21,  76,  95,  42,  63, 126};

inline static int Fq_reduction(int Z) {
  Z = (Z & QRUOV_q) + ((Z & ~QRUOV_q) >> QRUOV_ceil_log_2_q);
  int C = ((Z + 1) & ~QRUOV_q);
  Z += (C >> QRUOV_ceil_log_2_q);
  Z -= C;
  return Z;
}

inline static Fq Fq_sub(Fq X, Fq Y) {
  return (Fq)Fq_reduction((int)X - (int)Y + QRUOV_q);
}
inline static Fq Fq_mul(Fq X, Fq Y) {
  return (Fq)Fq_reduction((int)X * (int)Y);
}
inline static Fq Fq_inv(Fq X) {
```

```
  extern Fq Fq_inv_table[QRUOV_q];
  return Fq_inv_table[X];
}

#define EQN(I, J) (eqn[I]->col[J])

/* ----------------------------------------
    Linear Algebra
   ---------------------------------------- */
static void Fq_mxm_identity(Fq A[QRUOV_m][QRUOV_m]) {
  memset(A, 0, sizeof(Fq) * QRUOV_m * QRUOV_m);
  for (int i = 0; i < QRUOV_m; i++)
    A[i][i] = 1;
}

static void L_inverse(ECHELON_FORM echelon_form, Fq R[QRUOV_m][QRUOV_m]) {
  ROW_s **eqn = echelon_form->eqn;
  int rank = echelon_form->rank;

  Fq_mxm_identity(R);

  for (int i = 0; i < rank; i++) {
    Fq pivot = EQN(i, i);
    Fq inv = Fq_inv(pivot);
    for (int k = 0; k <= i; k++)
      R[i][k] = Fq_mul(R[i][k], inv);
    for (int j = i + 1; j < QRUOV_m; j++) {
      for (int k = 0; k <= i; k++) {
        R[j][k] = Fq_sub(R[j][k], Fq_mul(EQN(j, i), R[i][k]));
      }
    }
  }
}

static int consistent(ECHELON_FORM echelon_form, // input
                      Fq b[QRUOV_m],
                      int *cacheR, // output
                      Fq R[QRUOV_m][QRUOV_m]) {
  ROW_s **eqn = echelon_form->eqn;
  int rank = echelon_form->rank;
  if (rank == QRUOV_m)
    return 1;

  if (*cacheR == 0) {
    L_inverse(echelon_form, R);
    *cacheR = 1;
  }
```

```
  for (int i = rank; i < QRUOV_m; i++) {
    int k;
    uint64_t t = 0;
    for (int j = 0; j < rank; j++) {
      k = eqn[j]->original_row_id;
      t += ((uint64_t)R[i][j]) * ((uint64_t)b[k]);
    }
    k = eqn[i]->original_row_id;
    t += ((uint64_t)R[i][i]) * ((uint64_t)b[k]);
    t %= QRUOV_q;
    if (t)
      return 0;
  }

  return 1;
}

// --- Host helper ---
static inline uint64_t h_Fq2Fql(const uint8_t *src) {
  return (uint64_t)src[0] | ((uint64_t)src[1] << Fql_ws) |
         ((uint64_t)src[2] << (Fql_ws * 2));
}

/* ------------------------------------------
    Kernel
   ------------------------------------------ */
__device__ inline uint64_t d_Fq2Fql(const uint8_t *src) {
  uint64_t z0 = src[0];
  uint64_t z1 = src[1];
  uint64_t z2 = src[2];
  return z0 | (z1 << Fql_ws) | (z2 << (Fql_ws * 2));
}

__global__ void ExpandMatrixVxM_kernel(const uint8_t *__restrict__ src,
                                       uint64_t *__restrict__ A) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int N_VXM_single = QRUOV_V * QRUOV_M;
  if (idx < N_VXM_single) {
    A[idx] = d_Fq2Fql(src + idx * QRUOV_L);
  }
}
__global__ void MATRIX_TRANSPOSE_VxM_kernel(const uint64_t *__restrict__ A,
                                            uint64_t *__restrict__ C) {
  int i = blockIdx.x * blockDim.x + threadIdx.x; // V
  int j = blockIdx.y * blockDim.y + threadIdx.y; // M
```

```
  if (i < QRUOV_V && j < QRUOV_M) {
    C[j * QRUOV_V + i] = A[i * QRUOV_M + j];
  }
}

__global__ void ExpandVectorV_kernel(uint8_t *src, // input src[L*V]
                                     uint64_t *A   // output
) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < QRUOV_V) {
    A[idx] = d_Fq2Fql(src + idx * QRUOV_L);
  }
}

#define N_UPPER_TRI (QRUOV_V * (QRUOV_V + 1) / 2)  // 1378
#define N_VXM (QRUOV_V * QRUOV_M)                  // 936
#define N1 (QRUOV_L * QRUOV_V * (QRUOV_V + 1) / 2) // 4134
#define N2 (QRUOV_L * QRUOV_V * QRUOV_M)           // 2808

__global__ void ExpandSymmetricMatrixVxV_batched_kernel(
    const uint8_t *__restrict__ all_src, // [QRUOV_m * N1]
    uint64_t *__restrict__ all_A         // [QRUOV_m * V * V]
) {
  int iter = blockIdx.y;
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k >= N_UPPER_TRI)
    return;

  const uint8_t *src = all_src + iter * N1;
  uint64_t *A = all_A + iter * QRUOV_V * QRUOV_V;

  float fk = (float)k;
  int i = (int)((2.0f * QRUOV_V + 1.0f -
                 sqrtf((2.0f * QRUOV_V + 1.0f) * (2.0f * QRUOV_V + 1.0f) -
                       8.0f * fk)) /
                2.0f);
  if (i > 0) {
    int start_of_row_i = i * QRUOV_V - i * (i - 1) / 2;
    if (start_of_row_i > k)
      i--;
  }
  int start_of_row = i * QRUOV_V - i * (i - 1) / 2;
  int j = i + (k - start_of_row);

  int byte_offset = k * 3;
  uint64_t z0 = src[byte_offset];
  uint64_t z1 = src[byte_offset + 1];
```

```
  uint64_t z2 = src[byte_offset + 2];
  uint64_t fql = z0 | (z1 << 24) | (z2 << 48);

  A[i * QRUOV_V + j] = fql;
  if (i != j) {
    A[j * QRUOV_V + i] = fql;
  }
}

__global__ void ExpandMatrixVxM_batched_kernel(
    const uint8_t *__restrict__ all_src, // [QRUOV_m * N2]
    uint64_t *__restrict__ all_A         // [QRUOV_m * V * M]
) {
  int iter = blockIdx.y;
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k >= N_VXM)
    return;

  const uint8_t *src = all_src + iter * N2;
  uint64_t *A = all_A + iter * QRUOV_V * QRUOV_M;

  int byte_offset = k * 3;
  uint64_t z0 = src[byte_offset];
  uint64_t z1 = src[byte_offset + 1];
  uint64_t z2 = src[byte_offset + 2];
  uint64_t fql = z0 | (z1 << 24) | (z2 << 48);

  A[k] = fql;
}

__device__ void unpack(uint64_t fql, uint32_t &z0, uint32_t &z1, uint32_t &z2)
{
  z0 = (uint32_t)(fql & 0xFFFFFF);
  z1 = (uint32_t)((fql >> 24) & 0xFFFFFF);
  z2 = (uint32_t)(fql >> 48);
}

__device__ uint64_t repack(uint32_t z0, uint32_t z1, uint32_t z2) {
  return ((uint64_t)(z0 % QRUOV_q)) | ((uint64_t)(z1 % QRUOV_q) << 24) |
         ((uint64_t)(z2 % QRUOV_q) << 48);
}

__device__ uint64_t dot_product_fql(uint64_t *vec, uint64_t *mat_col_base,
                                    int K, int col, int row_stride) {
  uint32_t acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0, acc4 = 0;
  for (int k = 0; k < K; k++) {
    uint32_t a0, a1, a2, b0, b1, b2;
```

```
    unpack(vec[k], a0, a1, a2);
    unpack(mat_col_base[k * row_stride + col], b0, b1, b2);
    acc0 += a0 * b0;
    acc1 += a0 * b1 + a1 * b0;
    acc2 += a0 * b2 + a1 * b1 + a2 * b0;
    acc3 += a1 * b2 + a2 * b1;
    acc4 += a2 * b2;
  }
  uint32_t r0 = (acc0 + acc3) % QRUOV_q;
  uint32_t r1 = (acc1 + acc3 + acc4) % QRUOV_q;
  uint32_t r2 = (acc2 + acc4) % QRUOV_q;
  return repack(r0, r1, r2);
}

__device__ uint8_t d_Fql2Fq(uint64_t Z, int i) {
  return ((Z >> ((Fql_ws)*i)) & QRUOV_q);
}

__device__ uint64_t d_Fql_add(uint64_t X, uint64_t Y) {
  uint64_t Z = X + Y;
  uint32_t z0 = ((uint32_t)Z & Fql_wm);
  Z >>= Fql_ws;
  uint32_t z1 = ((uint32_t)Z & Fql_wm);
  Z >>= Fql_ws;
  uint32_t z2 = (uint32_t)Z;
  z0 %= QRUOV_q;
  z1 %= QRUOV_q;
  z2 %= QRUOV_q;
  return ((uint64_t)z0) | ((uint64_t)z1 << 24) | ((uint64_t)z2 << 48);
}

__device__ uint64_t d_Fql_mul(uint64_t X, uint64_t Y) {
  uint32_t a0, a1, a2, b0, b1, b2;
  unpack(X, a0, a1, a2);
  unpack(Y, b0, b1, b2);

  uint32_t c0 = (uint32_t)((uint64_t)a0 * b0);
  uint32_t c1 = (uint32_t)((uint64_t)a0 * b1 + (uint64_t)a1 * b0);
  uint32_t c2 =
      (uint32_t)((uint64_t)a0 * b2 + (uint64_t)a1 * b1 + (uint64_t)a2 * b0);
  uint32_t c3 = (uint32_t)((uint64_t)a1 * b2 + (uint64_t)a2 * b1);
  uint32_t c4 = (uint32_t)((uint64_t)a2 * b2);

  uint32_t r0 = (c0 + c3) % QRUOV_q;
  uint32_t r1 = (c1 + c3 + c4) % QRUOV_q;
  uint32_t r2 = (c2 + c4) % QRUOV_q;
  return repack(r0, r1, r2);
```

```
}

#define TILE_B 16
#define V_PAD 64
#define M_PAD 32

__device__ void unpack_col_to_half(const uint64_t *src, int src_rows,
                                   int src_cols, int col, half *dst0,
                                   half *dst1, half *dst2, int dst_rows,
                                   int tid, int stride) {
  for (int r = tid; r < dst_rows; r += stride) {
    if (r < src_rows) {
      uint64_t v = src[r * src_cols + col];
      dst0[r] = __uint2half_rn((uint32_t)(v & 0xFFFFFF));
      dst1[r] = __uint2half_rn((uint32_t)((v >> 24) & 0xFFFFFF));
      dst2[r] = __uint2half_rn((uint32_t)(v >> 48));
    } else {
      dst0[r] = __float2half(0.0f);
      dst1[r] = __float2half(0.0f);
      dst2[r] = __float2half(0.0f);
    }
  }
}

__device__ void wmma_fql_gemm(const uint64_t *A_fql, int A_rows, int A_cols,
                              int A_row_stride, const uint64_t *B_fql,
                              int B_rows, int B_cols, int B_row_stride,
                              uint64_t *out_fql, int out_stride, int K_pad,
                              int N_pad,
                              half *smem_A,  // [3][TILE_B][K_pad]
                              half *smem_B,  // [3][K_pad][N_pad]
                              float *smem_C, // [5][TILE_B][N_pad]
                              int tid, int warp_id) {
  const int A_coeff_stride = TILE_B * K_pad;
  const int B_coeff_stride = K_pad * N_pad;
  const int C_coeff_stride = TILE_B * N_pad;

  for (int idx = tid; idx < TILE_B * K_pad; idx += blockDim.x) {
    int r = idx / K_pad, c = idx % K_pad;
    if (r < A_rows && c < A_cols) {
      uint64_t v = A_fql[r * A_row_stride + c];
      smem_A[0 * A_coeff_stride + idx] =
          __uint2half_rn((uint32_t)(v & 0xFFFFFF));
      smem_A[1 * A_coeff_stride + idx] =
          __uint2half_rn((uint32_t)((v >> 24) & 0xFFFFFF));
      smem_A[2 * A_coeff_stride + idx] = __uint2half_rn((uint32_t)(v >> 48));
    } else {
```

```
      smem_A[0 * A_coeff_stride + idx] = __float2half(0.0f);
      smem_A[1 * A_coeff_stride + idx] = __float2half(0.0f);
      smem_A[2 * A_coeff_stride + idx] = __float2half(0.0f);
    }
  }

  for (int idx = tid; idx < K_pad * N_pad; idx += blockDim.x) {
    int r = idx / N_pad, c = idx % N_pad;
    if (r < B_rows && c < B_cols) {
      uint64_t v = B_fql[r * B_row_stride + c];
      smem_B[0 * B_coeff_stride + idx] =
          __uint2half_rn((uint32_t)(v & 0xFFFFFF));
      smem_B[1 * B_coeff_stride + idx] =
          __uint2half_rn((uint32_t)((v >> 24) & 0xFFFFFF));
      smem_B[2 * B_coeff_stride + idx] = __uint2half_rn((uint32_t)(v >> 48));
    } else {
      smem_B[0 * B_coeff_stride + idx] = __float2half(0.0f);
      smem_B[1 * B_coeff_stride + idx] = __float2half(0.0f);
      smem_B[2 * B_coeff_stride + idx] = __float2half(0.0f);
    }
  }
  __syncthreads();

  int n_tiles = N_pad / 16;
  int k_tiles = K_pad / 16;
  for (int nt = warp_id; nt < n_tiles; nt += 4) {
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c0, c1, c2, c3, c4;
    wmma::fill_fragment(c0, 0.0f);
    wmma::fill_fragment(c1, 0.0f);
    wmma::fill_fragment(c2, 0.0f);
    wmma::fill_fragment(c3, 0.0f);
    wmma::fill_fragment(c4, 0.0f);

    for (int kt = 0; kt < k_tiles; kt++) {
      wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a0,
a1,
          a2;
      wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b0,
b1,
          b2;
      wmma::load_matrix_sync(a0, &smem_A[0 * A_coeff_stride + kt * 16],
K_pad);
      wmma::load_matrix_sync(a1, &smem_A[1 * A_coeff_stride + kt * 16],
K_pad);
      wmma::load_matrix_sync(a2, &smem_A[2 * A_coeff_stride + kt * 16],
K_pad);
      wmma::load_matrix_sync(
```

```
            b0, &smem_B[0 * B_coeff_stride + kt * 16 * N_pad + nt * 16], N_pad);
        wmma::load_matrix_sync(
            b1, &smem_B[1 * B_coeff_stride + kt * 16 * N_pad + nt * 16], N_pad);
        wmma::load_matrix_sync(
            b2, &smem_B[2 * B_coeff_stride + kt * 16 * N_pad + nt * 16], N_pad);

        // 9 MMA operations for Fql polynomial ring product
        wmma::mma_sync(c0, a0, b0, c0); // c0 = a0*b0
        wmma::mma_sync(c1, a0, b1, c1);
        wmma::mma_sync(c1, a1, b0, c1); // c1 = a0*b1 + a1*b0
        wmma::mma_sync(c2, a0, b2, c2);
        wmma::mma_sync(c2, a1, b1, c2);
        wmma::mma_sync(c2, a2, b0, c2); // c2
        wmma::mma_sync(c3, a1, b2, c3);
        wmma::mma_sync(c3, a2, b1, c3); // c3 = a1*b2 + a2*b1
        wmma::mma_sync(c4, a2, b2, c4); // c4 = a2*b2
    }

    wmma::store_matrix_sync(&smem_C[0 * C_coeff_stride + nt * 16], c0, N_pad,
                            wmma::mem_row_major);
    wmma::store_matrix_sync(&smem_C[1 * C_coeff_stride + nt * 16], c1, N_pad,
                            wmma::mem_row_major);
    wmma::store_matrix_sync(&smem_C[2 * C_coeff_stride + nt * 16], c2, N_pad,
                            wmma::mem_row_major);
    wmma::store_matrix_sync(&smem_C[3 * C_coeff_stride + nt * 16], c3, N_pad,
                            wmma::mem_row_major);
    wmma::store_matrix_sync(&smem_C[4 * C_coeff_stride + nt * 16], c4, N_pad,
                            wmma::mem_row_major);
}
__syncthreads();

int out_cols_raw = B_cols; // unpadded output columns
for (int idx = tid; idx < A_rows * out_cols_raw; idx += blockDim.x) {
    int r = idx / out_cols_raw, c = idx % out_cols_raw;
    int flat = r * N_pad + c;
    uint32_t v0 = ((uint32_t)smem_C[0 * C_coeff_stride + flat] +
                   (uint32_t)smem_C[3 * C_coeff_stride + flat]) %
                  QRUOV_q;
    uint32_t v1 = ((uint32_t)smem_C[1 * C_coeff_stride + flat] +
                   (uint32_t)smem_C[3 * C_coeff_stride + flat] +
                   (uint32_t)smem_C[4 * C_coeff_stride + flat]) %
                  QRUOV_q;
    uint32_t v2 = ((uint32_t)smem_C[2 * C_coeff_stride + flat] +
                   (uint32_t)smem_C[4 * C_coeff_stride + flat]) %
                  QRUOV_q;
    out_fql[r * out_stride + c] =
        ((uint64_t)v0) | ((uint64_t)v1 << 24) | ((uint64_t)v2 << 48);
```

```
  }
  __syncthreads();
}

#define SMEM_A_VV (3 * TILE_B * V_PAD) // half count
#define SMEM_B_VV (3 * V_PAD * V_PAD)  // half count
#define SMEM_C_VV (5 * TILE_B * V_PAD) // float count
#define TENSOR_SMEM_BYTES_VV (SMEM_A_VV * 2 + SMEM_B_VV * 2 + SMEM_C_VV * 4)

__global__ void __launch_bounds__(128, 1) kernel_fused_tensor_batched(
    uint64_t *d_all_yT,  // [N * QRUOV_V]
    uint64_t *d_all_Pi1, // [QRUOV_m * V * V]
    uint64_t *d_all_Pi2, // [QRUOV_m * V * M]
    uint64_t *d_Sd,      // [V * M]
    uint8_t *d_all_eqn,  // [N * QRUOV_m * QRUOV_m] output
    uint8_t *d_all_c,    // [N * QRUOV_m] output
    int batch_N) {
  int tid = threadIdx.x;
  int warp_id = tid / 32;
  int sig_base = blockIdx.x * TILE_B;
  int eqn_i = blockIdx.y;
  if (sig_base >= batch_N)
    return;

  int sigs_in_block = min(TILE_B, batch_N - sig_base);

  extern __shared__ char dyn_smem[];
  half *smem_A = (half *)dyn_smem;               // 3*TILE_B*V_PAD
  half *smem_B = smem_A + SMEM_A_VV;             // 3*V_PAD*V_PAD  (largest)
  float *smem_C = (float *)(smem_B + SMEM_B_VV); // 5*TILE_B*V_PAD

  char *after_wmma = (char *)(smem_C + SMEM_C_VV);
  uint64_t *s_yT = (uint64_t *)after_wmma;             // [TILE_B * QRUOV_V]
  uint64_t *s_yT_Pi1 = s_yT + TILE_B * QRUOV_V;        // [TILE_B * QRUOV_V]
  uint64_t *s_yT_Pi2 = s_yT_Pi1 + TILE_B * QRUOV_V;    // [TILE_B * QRUOV_M]
  uint64_t *s_yT_Pi1_Sd = s_yT_Pi2 + TILE_B * QRUOV_M; // [TILE_B * QRUOV_M]
  uint32_t *s_cgen =
      (uint32_t *)(s_yT_Pi1_Sd + TILE_B * QRUOV_M); // [TILE_B * 64]

  for (int idx = tid; idx < TILE_B * QRUOV_V; idx += blockDim.x) {
    int s = idx / QRUOV_V, v = idx % QRUOV_V;
    if (sig_base + s < batch_N)
      s_yT[idx] = d_all_yT[(sig_base + s) * QRUOV_V + v];
    else
      s_yT[idx] = 0;
  }
  __syncthreads();
```

```
uint64_t *Pi1 = d_all_Pi1 + (size_t)eqn_i * QRUOV_V * QRUOV_V;
uint64_t *Pi2 = d_all_Pi2 + (size_t)eqn_i * QRUOV_V * QRUOV_M;

wmma_fql_gemm(s_yT, sigs_in_block, QRUOV_V, QRUOV_V, // A
              Pi1, QRUOV_V, QRUOV_V, QRUOV_V,        // B (global mem)
              s_yT_Pi1, QRUOV_V,                     // output
              V_PAD, V_PAD, smem_A, smem_B, smem_C, tid, warp_id);

wmma_fql_gemm(s_yT, sigs_in_block, QRUOV_V, QRUOV_V, // A
              Pi2, QRUOV_V, QRUOV_M, QRUOV_M,        // B
              s_yT_Pi2, QRUOV_M,                     // output
              V_PAD, M_PAD, smem_A, smem_B, smem_C, tid, warp_id);

wmma_fql_gemm(s_yT_Pi1, sigs_in_block, QRUOV_V,
              QRUOV_V,                          // A = result of Rank 1
              d_Sd, QRUOV_V, QRUOV_M, QRUOV_M, // B
              s_yT_Pi1_Sd, QRUOV_M,             // output
              V_PAD, M_PAD, smem_A, smem_B, smem_C, tid, warp_id);

for (int s = 0; s < sigs_in_block; s++) {
  int sig_id = sig_base + s;
  uint8_t *my_eqn =
      d_all_eqn + (size_t)sig_id * QRUOV_m * QRUOV_m + eqn_i * QRUOV_m;

  if (tid < QRUOV_M) {
    uint32_t a0, a1, a2, b0, b1, b2;
    unpack(s_yT_Pi2[s * QRUOV_M + tid], a0, a1, a2);
    unpack(s_yT_Pi1_Sd[s * QRUOV_M + tid], b0, b1, b2);
    uint64_t fi2 =
        repack((a0 + QRUOV_q - b0) % QRUOV_q, (a1 + QRUOV_q - b1) % QRUOV_q,
               (a2 + QRUOV_q - b2) % QRUOV_q);
    uint64_t u = d_Fql_add(fi2, fi2);
    for (int k = 0; k < QRUOV_L; k++) {
      int perm_k = (k <= (QRUOV_fe - 1)) ? ((QRUOV_fe - 1) - k)
                                         : (QRUOV_L + (QRUOV_fe - 1) - k);
      my_eqn[QRUOV_L * tid + k] = d_Fql2Fq(u, perm_k);
    }
  }
}
__syncthreads();

// ---- C_GEN parallel reduction (per signature) ----
for (int s = 0; s < sigs_in_block; s++) {
  int sig_id = sig_base + s;
  uint64_t *my_yT = s_yT + s * QRUOV_V;
  uint64_t *my_Pi1_v = s_yT_Pi1 + s * QRUOV_V;
```

```
    uint32_t local_sum = 0;
    for (int j = tid; j < QRUOV_V; j += blockDim.x) {
      uint64_t product = d_Fql_mul(my_Pi1_v[j], my_yT[j]);
      local_sum += (uint32_t)d_Fql2Fq(product, 0);
    }
    s_cgen[s * 64 + tid] = local_sum;
    __syncthreads();

    if (tid < 32)
      s_cgen[s * 64 + tid] += s_cgen[s * 64 + tid + 32];
    __syncthreads();
    if (tid < 32) {
      uint32_t val = s_cgen[s * 64 + tid];
      val += __shfl_down_sync(0xFFFFFFFF, val, 16);
      val += __shfl_down_sync(0xFFFFFFFF, val, 8);
      val += __shfl_down_sync(0xFFFFFFFF, val, 4);
      val += __shfl_down_sync(0xFFFFFFFF, val, 2);
      val += __shfl_down_sync(0xFFFFFFFF, val, 1);
      if (tid == 0)
        d_all_c[sig_id * QRUOV_m + eqn_i] = (uint8_t)(val % QRUOV_q);
    }
    __syncthreads();
  }
}

#define S_A_COLS (QRUOV_m + 1)
__constant__ Fq d_Fq_inv_table[QRUOV_q] = {
    0,   1,  64,  85,  32,  51,  106, 109, 16, 113, 89,  104, 53,  88, 118,
17,
    8,   15, 120, 107, 108, 121, 52,  116, 90, 61,  44,  80,  59,  92,
72,  41,
    4,   77, 71,  98,  60,  103, 117, 114, 54, 31,  124, 65,  26,  48,
58,  100,
    45,  70, 94,  5,   22,  12,  40,  97,  93, 78,  46,  28,  36,  25,
84,  125,

    2,   43, 102, 91,  99,  81,  49,  34,  30, 87,  115, 105, 122, 33,
57,  82,
    27,  69, 79,  101, 62,  3,   96,  73,  13, 10,  24,  67,  29,  56,
50,  123,
    86,  55, 35,  68,  47,  83,  66,  37,  11, 75,  6,   19,  20,  7,  112,
119,
    110, 9,  39,  74,  23,  38,  14,  111, 18, 21,  76,  95,  42,  63, 126};

__device__ __forceinline__ int d_Fq_reduction(int Z) {
  Z = (Z & QRUOV_q) + ((Z & ~QRUOV_q) >> QRUOV_ceil_log_2_q);
```

```
  int C = ((Z + 1) & ~QRUOV_q);
  Z += (C >> QRUOV_ceil_log_2_q);
  Z -= C;
  return Z;
}

__device__ __forceinline__ Fq d_Fq_sub(Fq X, Fq Y) {
  return (Fq)d_Fq_reduction((int)X - (int)Y + QRUOV_q);
}

__device__ __forceinline__ Fq d_Fq_mul(Fq X, Fq Y) {
  return (Fq)d_Fq_reduction((int)X * (int)Y);
}

__global__ void LU_decompose_batched_kernel(Fq *d_all_eqn, int *d_all_rank,
                                            int *d_all_indices, int *d_all_p,
                                            int batch_N) {
  int sig_id = blockIdx.x;
  if (sig_id >= batch_N)
    return;

  Fq *global_A = d_all_eqn + sig_id * QRUOV_m * QRUOV_m;
  int *d_rank = d_all_rank + sig_id;
  int *d_indices = d_all_indices + sig_id * QRUOV_m;
  int *d_p = d_all_p + sig_id * QRUOV_m;

  __shared__ Fq s_A[QRUOV_m][QRUOV_m];
  __shared__ int p[QRUOV_m];
  __shared__ int pivot_col_idx[QRUOV_m];
  __shared__ int current_rank;
  __shared__ int s_i, s_c, found_j;

  int tid = threadIdx.x;

  if (tid < QRUOV_m) {
    p[tid] = tid;
    pivot_col_idx[tid] = -1;
  }
  for (int idx = tid; idx < QRUOV_m * QRUOV_m; idx += blockDim.x) {
    s_A[idx / QRUOV_m][idx % QRUOV_m] = global_A[idx];
  }
  if (tid == 0) {
    current_rank = 0;
    s_i = 0;
    s_c = 0;
  }
  __syncthreads();
```

```
while (s_i < QRUOV_m && s_c < QRUOV_m) {
  __shared__ int candidate;
  if (tid == 0)
    candidate = QRUOV_m;
  __syncthreads();
  {
    int my_row = s_i + tid;
    if (my_row < QRUOV_m && s_A[p[my_row]][s_c] != 0)
      atomicMin(&candidate, my_row);
  }
  __syncthreads();
  if (tid == 0)
    found_j = (candidate < QRUOV_m) ? candidate : -1;
  __syncthreads();
  if (found_j == -1) {
    if (tid == 0)
      s_c++;
    __syncthreads();
    continue;
  }

  if (tid == 0) {
    int temp = p[s_i];
    p[s_i] = p[found_j];
    p[found_j] = temp;
    pivot_col_idx[current_rank] = s_c;
    current_rank++;
  }
  __syncthreads();

  int pivot_row = p[s_i];
  Fq pivot_val = s_A[pivot_row][s_c];
  Fq inv = d_Fq_inv_table[pivot_val];
  if (tid == 0)
    s_A[pivot_row][s_i] = pivot_val;
  for (int k = s_c + 1 + tid; k < QRUOV_m; k += blockDim.x)
    s_A[pivot_row][k] = d_Fq_mul(s_A[pivot_row][k], inv);
  __syncthreads();

  for (int row_idx = s_i + 1 + tid; row_idx < QRUOV_m;
       row_idx += blockDim.x) {
    int tr = p[row_idx];
    s_A[tr][s_i] = s_A[tr][s_c];
  }
  __syncthreads();
```

```
    int num_er = QRUOV_m - (s_i + 1), num_ec = QRUOV_m - (s_c + 1);
    for (int w = tid; w < num_er * num_ec; w += blockDim.x) {
      int ri = s_i + 1 + w / num_ec, ci = s_c + 1 + w % num_ec;
      int tr = p[ri];
      s_A[tr][ci] =
          d_Fq_sub(s_A[tr][ci], d_Fq_mul(s_A[tr][s_i], s_A[pivot_row][ci]));
    }
    __syncthreads();
    if (tid == 0) {
      s_i++;
      s_c++;
    }
    __syncthreads();
  }

  if (tid < QRUOV_m) {
    d_indices[tid] = pivot_col_idx[tid];
    d_p[tid] = p[tid];
  }
  if (tid == 0)
    *d_rank = current_rank;
  for (int idx = tid; idx < QRUOV_m * QRUOV_m; idx += blockDim.x) {
    int lr = idx / QRUOV_m, c = idx % QRUOV_m;
    global_A[lr * QRUOV_m + c] = s_A[p[lr]][c];
  }
}

__global__ void sample_a_solution_batched_kernel(
    Fq *d_all_eqn, int *d_all_p, int *d_all_rank, int *d_all_indices,
    Fq *d_all_b, Fq *d_all_random, Fq *d_all_oil_u, Fq *d_all_b2, int batch_N)
{
  int sid = blockIdx.x * blockDim.x + threadIdx.x;
  if (sid >= batch_N)
    return;

  Fq *eqn = d_all_eqn + sid * QRUOV_m * QRUOV_m;
  int *pp = d_all_p + sid * QRUOV_m;
  int rank = d_all_rank[sid];
  int *indices = d_all_indices + sid * QRUOV_m;
  Fq *b = d_all_b + sid * QRUOV_m;
  Fq *rand_element = d_all_random + sid * QRUOV_m;
  Fq *oil_u = d_all_oil_u + sid * QRUOV_m;
  Fq *b2 = d_all_b2 + sid * QRUOV_m;

  for (int i = 0; i < rank; i++) {
    uint64_t t = 0;
    for (int j = 0; j < i; j++)
```

```
      t += ((uint64_t)eqn[i * QRUOV_m + j]) * ((uint64_t)b2[j]);
    Fq tmp = d_Fq_sub(b[pp[i]], (Fq)(t % QRUOV_q));
    b2[i] = d_Fq_mul(tmp, d_Fq_inv_table[eqn[i * QRUOV_m + i]]);
  }

  int i = QRUOV_m - 1;
  for (int j = rank - 1; j >= 0; j--) {
    int k = indices[j];
    for (; i > k; i--)
      oil_u[i] = *rand_element++;
    uint64_t t = 0;
    for (int kk = k + 1; kk < QRUOV_m; kk++)
      t += ((uint64_t)eqn[j * QRUOV_m + kk]) * ((uint64_t)(oil_u[kk]));
    oil_u[i] = d_Fq_sub(b2[j], (Fq)(t % QRUOV_q));
    i--;
  }
  for (; i >= 0; i--)
    oil_u[i] = *rand_element++;
}

__global__ void fused_pack0_SIG_GEN_batched_kernel(Fq *d_all_oil_u,
                                                   uint64_t *d_SdT,
                                                   uint64_t *d_all_yT,
                                                   uint64_t *d_all_sig_s,
                                                   int batch_N) {
  int gid = blockIdx.x * blockDim.x + threadIdx.x;
  if (gid >= batch_N * QRUOV_N)
    return;
  int sid = gid / QRUOV_N;
  int i = gid % QRUOV_N;

  Fq *oil_u = d_all_oil_u + sid * QRUOV_m;
  uint64_t *yT = d_all_yT + sid * QRUOV_V;
  uint64_t *sig_s = d_all_sig_s + sid * QRUOV_N;

  if (i < QRUOV_V) {
    uint32_t acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0, acc4 = 0;
    for (int j = 0; j < QRUOV_M; j++) {
      uint64_t oj = (uint64_t)oil_u[j * QRUOV_L] |
                    ((uint64_t)oil_u[j * QRUOV_L + 1] << Fql_ws) |
                    ((uint64_t)oil_u[j * QRUOV_L + 2] << (Fql_ws * 2));
      uint64_t sdt = d_SdT[j * QRUOV_V + i];
      uint32_t a0, a1, a2, b0, b1, b2;
      unpack(oj, a0, a1, a2);
      unpack(sdt, b0, b1, b2);
      acc0 += a0 * b0;
      acc1 += a0 * b1 + a1 * b0;
```

```
      acc2 += a0 * b2 + a1 * b1 + a2 * b0;
      acc3 += a1 * b2 + a2 * b1;
      acc4 += a2 * b2;
    }
    uint64_t dot =
        repack((acc0 + acc3) % QRUOV_q, (acc1 + acc3 + acc4) % QRUOV_q,
               (acc2 + acc4) % QRUOV_q);
    uint64_t vin = yT[i];
    uint32_t v0, v1, v2, d0, d1, d2;
    unpack(vin, v0, v1, v2);
    unpack(dot, d0, d1, d2);
    sig_s[i] =
        repack((v0 + QRUOV_q - d0) % QRUOV_q, (v1 + QRUOV_q - d1) % QRUOV_q,
               (v2 + QRUOV_q - d2) % QRUOV_q);
  } else {
    int oi = i - QRUOV_V;
    sig_s[i] = (uint64_t)oil_u[oi * QRUOV_L] |
               ((uint64_t)oil_u[oi * QRUOV_L + 1] << Fql_ws) |
               ((uint64_t)oil_u[oi * QRUOV_L + 2] << (Fql_ws * 2));
  }
}

#define QRUOV_PRG2_CTX EVP_CIPHER_CTX
void print_fql(const char *label, Fql val) {
  uint32_t z0 = (uint32_t)(val & Fql_wm);
  uint32_t z1 = (uint32_t)((val >> Fql_ws) & Fql_wm);
  uint32_t z2 = (uint32_t)(val >> (Fql_ws * 2));
  printf("%s: [%u, %u, %u]\n", label, z0, z1, z2);
}

#define CONSISTENT_BATCH_SIZE_B 4
#define SIGNATURE_SIZE 30
int count = 0;
void QRUOV_Sign(const QRUOV_SEED seed_sk,     // input (signing key)
                const QRUOV_SEED seed_pk,     // input (signing key)
                const uint8_t message[],      // input (message)
                const size_t message_length, // input (message)
                QRUOV_SIGNATURE sig           // output
) {
  cudaDeviceProp prop;
  cudaGetDeviceProperties(&prop, 0);
  size_t free_mem, total_mem;
  cudaMemGetInfo(&free_mem, &total_mem);
  printf("GPU: %s | Memory: %.0f MB free / %.0f MB total\n\n", prop.name,
         free_mem / 1e6, total_mem / 1e6);

  // Expand mu
```

```
uint8_t mu[QRUOV_MU_LEN];
Expand_mu(seed_pk, message, message_length, mu);

// Expand_sk (shared)
uint64_t *d_Sd, *d_SdT;
cudaMalloc(&d_Sd, QRUOV_V * QRUOV_M * sizeof(uint64_t));
cudaMalloc(&d_SdT, QRUOV_M * QRUOV_V * sizeof(uint64_t));
{
  const int n2 = QRUOV_L * QRUOV_V * QRUOV_M;
  uint8_t r2[QRUOV_tau2];
  QRUOV_PRG_CTX *ctx = PRG_init(seed_sk);
  RejSampPRG(ctx, 0, n2, QRUOV_tau2, r2);
  uint8_t *d_r2;
  cudaMalloc(&d_r2, QRUOV_tau2);
  cudaMemcpy(d_r2, r2, QRUOV_tau2, cudaMemcpyHostToDevice);
  ExpandMatrixVxM_kernel<<<4, 256>>>(d_r2, d_Sd);
  dim3 bd(16, 16), gd((QRUOV_V + 15) / 16, (QRUOV_M + 15) / 16);
  MATRIX_TRANSPOSE_VxM_kernel<<<gd, bd>>>(d_Sd, d_SdT);
  cudaFree(d_r2);
  PRG_final(ctx);
  OPENSSL_cleanse(r2, QRUOV_tau2);
}

// Expand_pk (shared)
uint64_t *d_all_Pi1, *d_all_Pi2;
cudaMalloc(&d_all_Pi1, QRUOV_m * QRUOV_V * QRUOV_V * sizeof(uint64_t));
cudaMalloc(&d_all_Pi2, QRUOV_m * QRUOV_V * QRUOV_M * sizeof(uint64_t));
{
  QRUOV_PRG_CTX *ctx_pk = PRG_init(seed_pk);
  uint8_t *all_r1 = (uint8_t *)malloc(QRUOV_m * N1);
  uint8_t *all_r2 = (uint8_t *)malloc(QRUOV_m * N2);
  for (int i = 0; i < QRUOV_m; i++) {
    QRUOV_PRG_CTX *ctx_1 = PRG_copy(ctx_pk);
    uint8_t t1[QRUOV_tau1];
    RejSampPRG(ctx_1, 2 * i, N1, QRUOV_tau1, t1);
    PRG_final(ctx_1);
    memcpy(all_r1 + i * N1, t1, N1);

    QRUOV_PRG_CTX *ctx_2 = PRG_copy(ctx_pk);
    uint8_t t2[QRUOV_tau2];
    RejSampPRG(ctx_2, 2 * i + 1, N2, QRUOV_tau2, t2);
    PRG_final(ctx_2);
    memcpy(all_r2 + i * N2, t2, N2);
  }
  uint8_t *d_r1, *d_r2;
  cudaMalloc(&d_r1, QRUOV_m * N1);
  cudaMalloc(&d_r2, QRUOV_m * N2);
```

```
    cudaMemset(d_all_Pi1, 0, QRUOV_m * QRUOV_V * QRUOV_V * sizeof(uint64_t));
    cudaMemset(d_all_Pi2, 0, QRUOV_m * QRUOV_V * QRUOV_M * sizeof(uint64_t));
    cudaMemcpy(d_r1, all_r1, QRUOV_m * N1, cudaMemcpyHostToDevice);
    cudaMemcpy(d_r2, all_r2, QRUOV_m * N2, cudaMemcpyHostToDevice);
    dim3 gvv((N_UPPER_TRI + 255) / 256, QRUOV_m);
    ExpandSymmetricMatrixVxV_batched_kernel<<<gvv, 256>>>(d_r1, d_all_Pi1);
    dim3 gvm((N_VXM + 255) / 256, QRUOV_m);
    ExpandMatrixVxM_batched_kernel<<<gvm, 256>>>(d_r2, d_all_Pi2);
    cudaFree(d_r1);
    cudaFree(d_r2);
    free(all_r1);
    free(all_r2);
    PRG_final(ctx_pk);
  }

  int batch_sizes[SIGNATURE_SIZE];
  for (int i = 0; i < SIGNATURE_SIZE; i++) {
    batch_sizes[i] = 1 << i;
  }

  for (int i = 0; i < SIGNATURE_SIZE; i++) {
    int N = batch_sizes[i];
    size_t per_sig_mem =
        (size_t)QRUOV_V * sizeof(uint64_t) // d_all_yT:        52 * 8 = 416
        + (size_t)QRUOV_m * QRUOV_m *
              sizeof(Fq)                   // d_all_eqn:       54 * 54 =
2916
        + (size_t)QRUOV_m * sizeof(Fq)     // d_all_c:         54
        + (size_t)sizeof(int)              // d_all_rank:      4
        + (size_t)QRUOV_m * sizeof(int)    // d_all_indices:   216
        + (size_t)QRUOV_m * sizeof(int)    // d_all_p:         216
        + (size_t)QRUOV_m * sizeof(Fq)     // d_all_b:         54
        + (size_t)QRUOV_m * sizeof(Fq)     // d_all_random:    54
        + (size_t)QRUOV_m * sizeof(Fq)     // d_all_oil_u:     54
        + (size_t)QRUOV_m * sizeof(Fq)     // d_all_b2:        54
        + (size_t)QRUOV_N * sizeof(uint64_t); // d_all_sig_s:     70 * 8 = 560
    size_t needed = (size_t)N * per_sig_mem;
    cudaMemGetInfo(&free_mem, &total_mem);
    if (needed > free_mem * 0.75) {
      printf("%-12d | Skipped — need %.0f MB, only %.0f MB free\n", N,
             needed / 1e6, free_mem / 1e6);
      printf("Allocate 25%% buffer for GPU runtime/context overhead\n");
      break;
    }

    uint64_t *h_all_yT = (uint64_t *)malloc(N * QRUOV_V * sizeof(uint64_t));
    Fq *h_all_b = (Fq *)malloc(N * QRUOV_m * sizeof(Fq));
```

```
Fq *h_all_random = (Fq *)malloc(N * QRUOV_m * sizeof(Fq));
uint8_t *h_all_salts = (uint8_t *)malloc(N * QRUOV_SALT_LEN);
QRUOV_PRG2_CTX **all_ctx_r =
    (QRUOV_PRG2_CTX **)malloc(N * sizeof(QRUOV_PRG2_CTX *));

for (int n = 0; n < N; n++) {
  QRUOV_SEED seed_y;
  randombytes(seed_y, QRUOV_SEED_LEN);
  const int n3 = QRUOV_L * QRUOV_V;
  uint8_t r3[QRUOV_tau3];
  QRUOV_PRG_CTX *cy = PRG_init(seed_y);
  RejSampPRG(cy, 0, n3, QRUOV_tau3, r3);
  PRG_final(cy);
  for (int v = 0; v < QRUOV_V; v++)
    h_all_yT[n * QRUOV_V + v] = h_Fq2Fql(r3 + v * QRUOV_L);

  QRUOV_SEED seed_r;
  randombytes(seed_r, QRUOV_SEED_LEN);
  all_ctx_r[n] = PRG2_init(seed_r);

  QRUOV_SEED seed_sol;
  randombytes(seed_sol, QRUOV_SEED_LEN);
  Expand_sol(seed_sol, h_all_random + n * QRUOV_m);
}

uint64_t *d_all_yT;
Fq *d_all_eqn, *d_all_c;
int *d_all_rank, *d_all_indices, *d_all_p;
Fq *d_all_b, *d_all_random_buf, *d_all_oil_u, *d_all_b2;
uint64_t *d_all_sig_s;

cudaMalloc(&d_all_yT, N * QRUOV_V * sizeof(uint64_t));
cudaMalloc(&d_all_eqn, N * QRUOV_m * QRUOV_m * sizeof(Fq));
cudaMalloc(&d_all_c, N * QRUOV_m * sizeof(Fq));
cudaMalloc(&d_all_rank, N * sizeof(int));
cudaMalloc(&d_all_indices, N * QRUOV_m * sizeof(int));
cudaMalloc(&d_all_p, N * QRUOV_m * sizeof(int));
cudaMalloc(&d_all_b, N * QRUOV_m * sizeof(Fq));
cudaMalloc(&d_all_random_buf, N * QRUOV_m * sizeof(Fq));
cudaMalloc(&d_all_oil_u, N * QRUOV_m * sizeof(Fq));
cudaMalloc(&d_all_b2, N * QRUOV_m * sizeof(Fq));
cudaMalloc(&d_all_sig_s, N * QRUOV_N * sizeof(uint64_t));

cudaMemcpy(d_all_yT, h_all_yT, N * QRUOV_V * sizeof(uint64_t),
           cudaMemcpyHostToDevice);
cudaMemcpy(d_all_random_buf, h_all_random, N * QRUOV_m,
           cudaMemcpyHostToDevice);
```

```
    size_t tensor_smem = TENSOR_SMEM_BYTES_VV // WMMA workspace
                       + (TILE_B * QRUOV_V * 2 + TILE_B * QRUOV_M * 2) *
                             sizeof(uint64_t)            // Fql buffers
                       + TILE_B * 64 * sizeof(uint32_t); // cgen scratch
    cudaFuncSetAttribute(kernel_fused_tensor_batched,
                         cudaFuncAttributeMaxDynamicSharedMemorySize,
                         (int)tensor_smem);

    dim3 grid_fused((N + TILE_B - 1) / TILE_B, 54);


    kernel_fused_tensor_batched<<<grid_fused, 128, tensor_smem>>>(
        d_all_yT, d_all_Pi1, d_all_Pi2, d_Sd, d_all_eqn, d_all_c, N);
    LU_decompose_batched_kernel<<<N, 128>>>(d_all_eqn, d_all_rank,
                                            d_all_indices, d_all_p, N);


    Fq msg[QRUOV_m];
    Fq *h_all_eqn = (Fq *)malloc(N * QRUOV_m * QRUOV_m * sizeof(Fq));
    Fq *h_all_c = (Fq *)malloc(N * QRUOV_m);
    int *h_all_rank = (int *)malloc(N * sizeof(int));
    int *h_all_p = (int *)malloc(N * QRUOV_m * sizeof(int));
    int *h_all_indices = (int *)malloc(N * QRUOV_m * sizeof(int));
    cudaMemcpy(h_all_eqn, d_all_eqn, N * QRUOV_m * QRUOV_m * sizeof(Fq),
               cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_p, d_all_p, N * QRUOV_m * sizeof(int),
               cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_indices, d_all_indices, N * QRUOV_m * sizeof(int),
               cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_c, d_all_c, N * QRUOV_m, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_rank, d_all_rank, N * sizeof(int),
cudaMemcpyDeviceToHost);

    int rare_case = 0;
    for (int n = 0; n < N; n++) {
      QRUOV_PRG2_CTX *ctx_r = all_ctx_r[n];
      Fq *my_b = h_all_b + n * QRUOV_m;
      Fq *my_c = h_all_c + n * QRUOV_m;

      if (h_all_rank[n] == QRUOV_m) {
        // Full rank — single salt attempt always consistent
        PRG2_yield(ctx_r, QRUOV_SALT_LEN, h_all_salts + n * QRUOV_SALT_LEN);
        Hash(mu, h_all_salts + n * QRUOV_SALT_LEN, msg);
        for (int i = 0; i < QRUOV_m; i++)
          my_b[i] = Fq_sub(msg[i], my_c[i]);
      } else {
```

```
        rare_case++;
        // printf("N=%d, n=%d, rank=%d\n", N, n, h_all_rank[n]);
        Fq *eqn_flat = h_all_eqn + n * QRUOV_m * QRUOV_m;
        int *p_vec = h_all_p + n * QRUOV_m;
        int *idx_vec = h_all_indices + n * QRUOV_m;

        ECHELON_FORM echelon_form;
        Fq eqn_storage[QRUOV_m][QRUOV_m];
        for (int r = 0; r < QRUOV_m; r++) {
          for (int c = 0; c < QRUOV_m; c++)
            eqn_storage[r][c] = eqn_flat[r * QRUOV_m + c];
          echelon_form->row[r].col = eqn_storage[r];
          echelon_form->row[r].original_row_id = p_vec[r];
        }
        for (int r = 0; r < QRUOV_m; r++)
          echelon_form->eqn[r] = &echelon_form->row[r];
        echelon_form->rank = h_all_rank[n];
        for (int r = 0; r < QRUOV_m; r++)
          echelon_form->index[r] = idx_vec[r];

        Fq R[QRUOV_m][QRUOV_m];
        int cacheR = 0;

        do {
          PRG2_yield(ctx_r, QRUOV_SALT_LEN, h_all_salts + n * QRUOV_SALT_LEN);
          Hash(mu, h_all_salts + n * QRUOV_SALT_LEN, msg);
          for (int i = 0; i < QRUOV_m; i++)
            my_b[i] = Fq_sub(msg[i], my_c[i]);
        } while (!consistent(echelon_form, my_b, &cacheR, R));
      }
      PRG2_final(ctx_r);
    }
    free(all_ctx_r);
    free(h_all_eqn);
    free(h_all_p);
    free(h_all_indices);

    cudaMemcpy(d_all_b, h_all_b, N * QRUOV_m * sizeof(Fq),
               cudaMemcpyHostToDevice);
    cudaMemcpy(d_all_rank, h_all_rank, N * sizeof(int),
cudaMemcpyHostToDevice);

    int threads_sa = 256;
    int blocks_sa = (N + threads_sa - 1) / threads_sa;
    int total_siggen = N * QRUOV_N;
```

```
    sample_a_solution_batched_kernel<<<blocks_sa, threads_sa>>>(
        d_all_eqn, d_all_p, d_all_rank, d_all_indices, d_all_b,
        d_all_random_buf, d_all_oil_u, d_all_b2, N);
    fused_pack0_SIG_GEN_batched_kernel<<<(total_siggen + 255) / 256, 256>>>(
        d_all_oil_u, d_SdT, d_all_yT, d_all_sig_s, N);


    uint64_t *h_all_sig_s = (uint64_t *)malloc(N * QRUOV_N *
sizeof(uint64_t));
    cudaMemcpy(h_all_sig_s, d_all_sig_s, N * QRUOV_N * sizeof(uint64_t),
               cudaMemcpyDeviceToHost);

    memcpy(sig->r, h_all_salts, QRUOV_SALT_LEN);
    memcpy(sig->s, h_all_sig_s, QRUOV_N * sizeof(uint64_t));

    free(h_all_yT);
    free(h_all_b);
    free(h_all_random);
    free(h_all_salts);
    free(h_all_c);
    free(h_all_rank);
    free(h_all_sig_s);

    cudaFree(d_all_yT);
    cudaFree(d_all_eqn);
    cudaFree(d_all_c);
    cudaFree(d_all_rank);
    cudaFree(d_all_indices);
    cudaFree(d_all_p);
    cudaFree(d_all_b);
    cudaFree(d_all_random_buf);
    cudaFree(d_all_oil_u);
    cudaFree(d_all_b2);
    cudaFree(d_all_sig_s);
  }
  printf("\nDone\n");

  cudaFree(d_all_Pi1);
  cudaFree(d_all_Pi2);
  cudaFree(d_Sd);
  cudaFree(d_SdT);

  return;
}
```

QRUOV_Sign_Fused

```
#include "qruov.h"
#include <cstdint>
#include <cstdio>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <openssl/err.h>
#include <openssl/evp.h>
#include <openssl/sha.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

static void Expand_mu(const QRUOV_SEED seed_pk,    // input
                      const uint8_t message[],     // input
                      const size_t message_length, // input (message)
                      uint8_t mu[QRUOV_MU_LEN]     // output mu[64]
) {
  EVP_MD_CTX *ctx = EVP_MD_CTX_new();
  EVP_DigestInit_ex(ctx, EVP_shake256(), NULL);
  EVP_DigestUpdate(ctx, seed_pk, QRUOV_SEED_LEN);
  EVP_DigestUpdate(ctx, message, message_length);
  EVP_DigestFinalXOF(ctx, mu, QRUOV_MU_LEN);
  EVP_MD_CTX_free(ctx);
}

static void Hash(const uint8_t mu[QRUOV_MU_LEN], // input
                 const QRUOV_SALT salt,          // input
                 uint8_t dst[QRUOV_m]            // output
) {
  uint8_t tmp[QRUOV_tau4];
  EVP_MD_CTX *ctx = EVP_MD_CTX_new();
  EVP_DigestInit_ex(ctx, EVP_shake256(), NULL);
  EVP_DigestUpdate(ctx, mu, QRUOV_MU_LEN);
  EVP_DigestUpdate(ctx, salt, QRUOV_SALT_LEN);
  EVP_DigestFinalXOF(ctx, tmp, QRUOV_tau4);
  RejSamp(QRUOV_m, QRUOV_tau4, tmp);
  memcpy(dst, tmp, QRUOV_m);
  EVP_MD_CTX_free(ctx);
  return;
}

/* ----------------------------------------
```

```
   ------------------------------------------ */
static void Expand_sol(const QRUOV_SEED seed_sol, // input
                       uint8_t dst[QRUOV_m]       // output
) {
  const int n4 = QRUOV_L * QRUOV_M;
  uint8_t r4[QRUOV_tau4];
  QRUOV_PRG_CTX *ctx = PRG_init(seed_sol);
  RejSampPRG(ctx, 0, n4, QRUOV_tau4, r4);
  memcpy(dst, r4, n4);
  PRG_final(ctx);
}

Fq Fq_inv_table[QRUOV_q] = {
    0,   1,  64,  85,  32,  51,  106, 109, 16, 113, 89,  104, 53,  88, 118,
17,
    8,   15, 120, 107, 108, 121, 52,  116, 90, 61,  44,  80,  59,  92,
72,  41,
    4,   77, 71,  98,  60,  103, 117, 114, 54, 31,  124, 65,  26,  48,
58,  100,
    45,  70, 94,  5,   22,  12,  40,  97,  93, 78,  46,  28,  36,  25,
84,  125,

    2,   43, 102, 91,  99,  81,  49,  34,  30, 87,  115, 105, 122, 33,
57,  82,
    27,  69, 79,  101, 62,  3,   96,  73,  13, 10,  24,  67,  29,  56,
50,  123,
    86,  55, 35,  68,  47,  83,  66,  37,  11, 75,  6,   19,  20,  7,  112,
119,
    110, 9,  39,  74,  23,  38,  14,  111, 18, 21,  76,  95,  42,  63, 126};

inline static int Fq_reduction(int Z) {
  Z = (Z & QRUOV_q) + ((Z & ~QRUOV_q) >> QRUOV_ceil_log_2_q);
  int C = ((Z + 1) & ~QRUOV_q);
  Z += (C >> QRUOV_ceil_log_2_q);
  Z -= C;
  return Z;
}

inline static Fq Fq_sub(Fq X, Fq Y) {
  return (Fq)Fq_reduction((int)X - (int)Y + QRUOV_q);
}
inline static Fq Fq_mul(Fq X, Fq Y) {
  return (Fq)Fq_reduction((int)X * (int)Y);
}
inline static Fq Fq_inv(Fq X) {
  extern Fq Fq_inv_table[QRUOV_q];
  return Fq_inv_table[X];
```

```
}

#define EQN(I, J) (eqn[I]->col[J])

/* ----------------------------------------
    Linear Algebra
   ---------------------------------------- */
static void Fq_mxm_identity(Fq A[QRUOV_m][QRUOV_m]) {
  memset(A, 0, sizeof(Fq) * QRUOV_m * QRUOV_m);
  for (int i = 0; i < QRUOV_m; i++)
    A[i][i] = 1;
}

static void L_inverse(ECHELON_FORM echelon_form, Fq R[QRUOV_m][QRUOV_m]) {
  ROW_s **eqn = echelon_form->eqn;
  int rank = echelon_form->rank;

  Fq_mxm_identity(R);

  for (int i = 0; i < rank; i++) {
    Fq pivot = EQN(i, i);
    Fq inv = Fq_inv(pivot);
    for (int k = 0; k <= i; k++)
      R[i][k] = Fq_mul(R[i][k], inv);
    for (int j = i + 1; j < QRUOV_m; j++) {
      for (int k = 0; k <= i; k++) {
        R[j][k] = Fq_sub(R[j][k], Fq_mul(EQN(j, i), R[i][k]));
      }
    }
  }
}

static int consistent(ECHELON_FORM echelon_form, // input
                      Fq b[QRUOV_m],
                      int *cacheR, // output
                      Fq R[QRUOV_m][QRUOV_m]) {
  ROW_s **eqn = echelon_form->eqn;
  int rank = echelon_form->rank;
  if (rank == QRUOV_m)
    return 1;

  if (*cacheR == 0) {
    L_inverse(echelon_form, R);
    *cacheR = 1;
  }

  for (int i = rank; i < QRUOV_m; i++) {
```

```
    int k;
    uint64_t t = 0;
    for (int j = 0; j < rank; j++) {
      k = eqn[j]->original_row_id;
      t += ((uint64_t)R[i][j]) * ((uint64_t)b[k]);
    }
    k = eqn[i]->original_row_id;
    t += ((uint64_t)R[i][i]) * ((uint64_t)b[k]);
    t %= QRUOV_q;
    if (t)
      return 0;
  }

  return 1;
}


// --- Host helper ---
static inline uint64_t h_Fq2Fql(const uint8_t *src) {
  return (uint64_t)src[0] | ((uint64_t)src[1] << Fql_ws) |
         ((uint64_t)src[2] << (Fql_ws * 2));
}

/* ------------------------------------------
    Kernel
   ------------------------------------------ */
__device__ inline uint64_t d_Fq2Fql(const uint8_t *src) {
  uint64_t z0 = src[0];
  uint64_t z1 = src[1];
  uint64_t z2 = src[2];
  return z0 | (z1 << Fql_ws) | (z2 << (Fql_ws * 2));
}

__global__ void ExpandMatrixVxM_kernel(const uint8_t *__restrict__ src,
                                       uint64_t *__restrict__ A) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int N_VXM_single = QRUOV_V * QRUOV_M;
  if (idx < N_VXM_single) {
    A[idx] = d_Fq2Fql(src + idx * QRUOV_L);
  }
}
__global__ void MATRIX_TRANSPOSE_VxM_kernel(const uint64_t *__restrict__ A,
                                            uint64_t *__restrict__ C) {
  int i = blockIdx.x * blockDim.x + threadIdx.x; // V
  int j = blockIdx.y * blockDim.y + threadIdx.y; // M

  if (i < QRUOV_V && j < QRUOV_M) {
```

```
    C[j * QRUOV_V + i] = A[i * QRUOV_M + j];
  }
}

__global__ void ExpandVectorV_kernel(uint8_t *src, // input src[L*V]
                                     uint64_t *A   // output
) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < QRUOV_V) {
    A[idx] = d_Fq2Fql(src + idx * QRUOV_L);
  }
}

#define N_UPPER_TRI (QRUOV_V * (QRUOV_V + 1) / 2)  // 1378
#define N_VXM (QRUOV_V * QRUOV_M)                  // 936
#define N1 (QRUOV_L * QRUOV_V * (QRUOV_V + 1) / 2) // 4134
#define N2 (QRUOV_L * QRUOV_V * QRUOV_M)           // 2808

__global__ void ExpandSymmetricMatrixVxV_batched_kernel(
    const uint8_t *__restrict__ all_src, // [QRUOV_m * N1]
    uint64_t *__restrict__ all_A         // [QRUOV_m * V * V]
) {
  int iter = blockIdx.y;
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k >= N_UPPER_TRI)
    return;

  const uint8_t *src = all_src + iter * N1;
  uint64_t *A = all_A + iter * QRUOV_V * QRUOV_V;

  float fk = (float)k;
  int i = (int)((2.0f * QRUOV_V + 1.0f -
                 sqrtf((2.0f * QRUOV_V + 1.0f) * (2.0f * QRUOV_V + 1.0f) -
                       8.0f * fk)) /
                2.0f);
  if (i > 0) {
    int start_of_row_i = i * QRUOV_V - i * (i - 1) / 2;
    if (start_of_row_i > k)
      i--;
  }
  int start_of_row = i * QRUOV_V - i * (i - 1) / 2;
  int j = i + (k - start_of_row);

  int byte_offset = k * 3;
  uint64_t z0 = src[byte_offset];
  uint64_t z1 = src[byte_offset + 1];
  uint64_t z2 = src[byte_offset + 2];
```

```
  uint64_t fql = z0 | (z1 << 24) | (z2 << 48);

  A[i * QRUOV_V + j] = fql;
  if (i != j) {
    A[j * QRUOV_V + i] = fql;
  }
}

__global__ void ExpandMatrixVxM_batched_kernel(
    const uint8_t *__restrict__ all_src, // [QRUOV_m * N2]
    uint64_t *__restrict__ all_A         // [QRUOV_m * V * M]
) {
  int iter = blockIdx.y;
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k >= N_VXM)
    return;

  const uint8_t *src = all_src + iter * N2;
  uint64_t *A = all_A + iter * QRUOV_V * QRUOV_M;

  int byte_offset = k * 3;
  uint64_t z0 = src[byte_offset];
  uint64_t z1 = src[byte_offset + 1];
  uint64_t z2 = src[byte_offset + 2];
  uint64_t fql = z0 | (z1 << 24) | (z2 << 48);

  A[k] = fql;
}

__device__ void unpack(uint64_t fql, uint32_t &z0, uint32_t &z1, uint32_t &z2)
{
  z0 = (uint32_t)(fql & 0xFFFFFF);
  z1 = (uint32_t)((fql >> 24) & 0xFFFFFF);
  z2 = (uint32_t)(fql >> 48);
}

__device__ uint64_t repack(uint32_t z0, uint32_t z1, uint32_t z2) {
  return ((uint64_t)(z0 % QRUOV_q)) | ((uint64_t)(z1 % QRUOV_q) << 24) |
         ((uint64_t)(z2 % QRUOV_q) << 48);
}

__device__ uint64_t dot_product_fql(uint64_t *vec, uint64_t *mat_col_base,
                                    int K, int col, int row_stride) {
  uint32_t acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0, acc4 = 0;
  for (int k = 0; k < K; k++) {
    uint32_t a0, a1, a2, b0, b1, b2;
    unpack(vec[k], a0, a1, a2);
```

```
    unpack(mat_col_base[k * row_stride + col], b0, b1, b2);
    acc0 += a0 * b0;
    acc1 += a0 * b1 + a1 * b0;
    acc2 += a0 * b2 + a1 * b1 + a2 * b0;
    acc3 += a1 * b2 + a2 * b1;
    acc4 += a2 * b2;
  }
  uint32_t r0 = (acc0 + acc3) % QRUOV_q;
  uint32_t r1 = (acc1 + acc3 + acc4) % QRUOV_q;
  uint32_t r2 = (acc2 + acc4) % QRUOV_q;
  return repack(r0, r1, r2);
}

__device__ uint8_t d_Fql2Fq(uint64_t Z, int i) {
  return ((Z >> ((Fql_ws)*i)) & QRUOV_q);
}

__device__ uint64_t d_Fql_add(uint64_t X, uint64_t Y) {
  uint64_t Z = X + Y;
  uint32_t z0 = ((uint32_t)Z & Fql_wm);
  Z >>= Fql_ws;
  uint32_t z1 = ((uint32_t)Z & Fql_wm);
  Z >>= Fql_ws;
  uint32_t z2 = (uint32_t)Z;
  z0 %= QRUOV_q;
  z1 %= QRUOV_q;
  z2 %= QRUOV_q;
  return ((uint64_t)z0) | ((uint64_t)z1 << 24) | ((uint64_t)z2 << 48);
}

__device__ uint64_t d_Fql_mul(uint64_t X, uint64_t Y) {
  uint32_t a0, a1, a2, b0, b1, b2;
  unpack(X, a0, a1, a2);
  unpack(Y, b0, b1, b2);

  uint32_t c0 = (uint32_t)((uint64_t)a0 * b0);
  uint32_t c1 = (uint32_t)((uint64_t)a0 * b1 + (uint64_t)a1 * b0);
  uint32_t c2 =
      (uint32_t)((uint64_t)a0 * b2 + (uint64_t)a1 * b1 + (uint64_t)a2 * b0);
  uint32_t c3 = (uint32_t)((uint64_t)a1 * b2 + (uint64_t)a2 * b1);
  uint32_t c4 = (uint32_t)((uint64_t)a2 * b2);

  uint32_t r0 = (c0 + c3) % QRUOV_q;
  uint32_t r1 = (c1 + c3 + c4) % QRUOV_q;
  uint32_t r2 = (c2 + c4) % QRUOV_q;
  return repack(r0, r1, r2);
}
```

```
__global__ void
kernel_fused_batched(uint64_t *d_all_yT,  // [N * QRUOV_V] per-signature
                     uint64_t *d_all_Pi1, // [QRUOV_m * V * V] SHARED
                     uint64_t *d_all_Pi2, // [QRUOV_m * V * M] SHARED
                     uint64_t *d_Sd,      // [V * M] SHARED
                     uint8_t *d_all_eqn,  // [N * QRUOV_m * QRUOV_m] output
                     uint8_t *d_all_c,    // [N * QRUOV_m] output
                     int batch_N) {
  int tid = threadIdx.x;
  int sig_id = blockIdx.x;
  if (sig_id >= batch_N)
    return;

  uint64_t *my_yT = d_all_yT + sig_id * QRUOV_V;
  uint8_t *my_eqn_base = d_all_eqn + sig_id * QRUOV_m * QRUOV_m;
  uint8_t *my_c = d_all_c + sig_id * QRUOV_m;

  __shared__ uint64_t s_yT[QRUOV_V];
  __shared__ uint64_t s_yT_Pi1[QRUOV_V];
  __shared__ uint64_t s_yT_Pi1_Sd[QRUOV_M];
  __shared__ uint64_t s_yT_Pi2[QRUOV_M];
  __shared__ uint64_t s_yT_Fi2[QRUOV_M];
  __shared__ uint32_t s_cgen[QRUOV_m];

  if (tid < QRUOV_V)
    s_yT[tid] = my_yT[tid];
  __syncthreads();

  for (int i = blockIdx.y; i < QRUOV_m; i += gridDim.y) {
    uint64_t *Pi1 = d_all_Pi1 + i * QRUOV_V * QRUOV_V;
    if (tid < QRUOV_V)
      s_yT_Pi1[tid] = dot_product_fql(s_yT, Pi1, QRUOV_V, tid, QRUOV_V);
    __syncthreads();

    uint64_t *Pi2 = d_all_Pi2 + i * QRUOV_V * QRUOV_M;
    if (tid < QRUOV_M)
      s_yT_Pi2[tid] = dot_product_fql(s_yT, Pi2, QRUOV_V, tid, QRUOV_M);
    __syncthreads();

    if (tid < QRUOV_M)
      s_yT_Pi1_Sd[tid] = dot_product_fql(s_yT_Pi1, d_Sd, QRUOV_V, tid,
QRUOV_M);
    __syncthreads();

    if (tid < QRUOV_M) {
      uint32_t a0, a1, a2, b0, b1, b2;
```

```
      unpack(s_yT_Pi2[tid], a0, a1, a2);
      unpack(s_yT_Pi1_Sd[tid], b0, b1, b2);
      s_yT_Fi2[tid] =
          repack((a0 + QRUOV_q - b0) % QRUOV_q, (a1 + QRUOV_q - b1) % QRUOV_q,
                 (a2 + QRUOV_q - b2) % QRUOV_q);
    }
    __syncthreads();

    if (tid < QRUOV_M) {
      uint8_t *my_eqn = my_eqn_base + i * QRUOV_m;
      uint64_t u = d_Fql_add(s_yT_Fi2[tid], s_yT_Fi2[tid]);
      for (int k = 0; k < QRUOV_L; k++) {
        int perm_k = (k <= (QRUOV_fe - 1)) ? ((QRUOV_fe - 1) - k)
                                           : (QRUOV_L + (QRUOV_fe - 1) - k);
        my_eqn[QRUOV_L * tid + k] = d_Fql2Fq(u, perm_k);
      }
    }

    // C_GEN parallel reduction
    {
      uint32_t local_sum = 0;
      for (int j = tid; j < QRUOV_V; j += blockDim.x) {
        uint64_t product = d_Fql_mul(s_yT_Pi1[j], s_yT[j]);
        local_sum += (uint32_t)d_Fql2Fq(product, 0);
      }
      s_cgen[tid] = local_sum;
    }
    __syncthreads();
    if (tid < 32)
      s_cgen[tid] += s_cgen[tid + 32];
    __syncthreads();
    if (tid < 32) {
      uint32_t val = s_cgen[tid];
      val += __shfl_down_sync(0xFFFFFFFF, val, 16);
      val += __shfl_down_sync(0xFFFFFFFF, val, 8);
      val += __shfl_down_sync(0xFFFFFFFF, val, 4);
      val += __shfl_down_sync(0xFFFFFFFF, val, 2);
      val += __shfl_down_sync(0xFFFFFFFF, val, 1);
      if (tid == 0)
        my_c[i] = (uint8_t)(val % QRUOV_q);
    }
    __syncthreads();
  }
}

#define S_A_COLS (QRUOV_m + 1)
__constant__ Fq d_Fq_inv_table[QRUOV_q] = {
```

```
    0,   1,  64,  85,  32,  51,  106, 109, 16, 113, 89,  104, 53,  88, 118,
17,
    8,   15, 120, 107, 108, 121, 52,  116, 90, 61,  44,  80,  59,  92,
72,  41,
    4,   77, 71,  98,  60,  103, 117, 114, 54, 31,  124, 65,  26,  48,
58,  100,
    45,  70, 94,  5,   22,  12,  40,  97,  93, 78,  46,  28,  36,  25,
84,  125,

    2,   43, 102, 91,  99,  81,  49,  34,  30, 87,  115, 105, 122, 33,
57,  82,
    27,  69, 79,  101, 62,  3,   96,  73,  13, 10,  24,  67,  29,  56,
50,  123,
    86,  55, 35,  68,  47,  83,  66,  37,  11, 75,  6,   19,  20,  7,  112,
119,
    110, 9,  39,  74,  23,  38,  14,  111, 18, 21,  76,  95,  42,  63, 126};

__device__ __forceinline__ int d_Fq_reduction(int Z) {
  Z = (Z & QRUOV_q) + ((Z & ~QRUOV_q) >> QRUOV_ceil_log_2_q);
  int C = ((Z + 1) & ~QRUOV_q);
  Z += (C >> QRUOV_ceil_log_2_q);
  Z -= C;
  return Z;
}

__device__ __forceinline__ Fq d_Fq_sub(Fq X, Fq Y) {
  return (Fq)d_Fq_reduction((int)X - (int)Y + QRUOV_q);
}

__device__ __forceinline__ Fq d_Fq_mul(Fq X, Fq Y) {
  return (Fq)d_Fq_reduction((int)X * (int)Y);
}

__global__ void LU_decompose_batched_kernel(Fq *d_all_eqn, int *d_all_rank,
                                            int *d_all_indices, int *d_all_p,
                                            int batch_N) {
  int sig_id = blockIdx.x;
  if (sig_id >= batch_N)
    return;

  Fq *global_A = d_all_eqn + sig_id * QRUOV_m * QRUOV_m;
  int *d_rank = d_all_rank + sig_id;
  int *d_indices = d_all_indices + sig_id * QRUOV_m;
  int *d_p = d_all_p + sig_id * QRUOV_m;

  __shared__ Fq s_A[QRUOV_m][QRUOV_m];
  __shared__ int p[QRUOV_m];
```

```
__shared__ int pivot_col_idx[QRUOV_m];
__shared__ int current_rank;
__shared__ int s_i, s_c, found_j;

int tid = threadIdx.x;

if (tid < QRUOV_m) {
  p[tid] = tid;
  pivot_col_idx[tid] = -1;
}
for (int idx = tid; idx < QRUOV_m * QRUOV_m; idx += blockDim.x) {
  s_A[idx / QRUOV_m][idx % QRUOV_m] = global_A[idx];
}
if (tid == 0) {
  current_rank = 0;
  s_i = 0;
  s_c = 0;
}
__syncthreads();

while (s_i < QRUOV_m && s_c < QRUOV_m) {
  __shared__ int candidate;
  if (tid == 0)
    candidate = QRUOV_m;
  __syncthreads();
  {
    int my_row = s_i + tid;
    if (my_row < QRUOV_m && s_A[p[my_row]][s_c] != 0)
      atomicMin(&candidate, my_row);
  }
  __syncthreads();
  if (tid == 0)
    found_j = (candidate < QRUOV_m) ? candidate : -1;
  __syncthreads();
  if (found_j == -1) {
    if (tid == 0)
      s_c++;
    __syncthreads();
    continue;
  }

  if (tid == 0) {
    int temp = p[s_i];
    p[s_i] = p[found_j];
    p[found_j] = temp;
    pivot_col_idx[current_rank] = s_c;
    current_rank++;
```

```
    }
    __syncthreads();

    int pivot_row = p[s_i];
    Fq pivot_val = s_A[pivot_row][s_c];
    Fq inv = d_Fq_inv_table[pivot_val];
    if (tid == 0)
      s_A[pivot_row][s_i] = pivot_val;
    for (int k = s_c + 1 + tid; k < QRUOV_m; k += blockDim.x)
      s_A[pivot_row][k] = d_Fq_mul(s_A[pivot_row][k], inv);
    __syncthreads();

    for (int row_idx = s_i + 1 + tid; row_idx < QRUOV_m;
         row_idx += blockDim.x) {
      int tr = p[row_idx];
      s_A[tr][s_i] = s_A[tr][s_c];
    }
    __syncthreads();

    int num_er = QRUOV_m - (s_i + 1), num_ec = QRUOV_m - (s_c + 1);
    for (int w = tid; w < num_er * num_ec; w += blockDim.x) {
      int ri = s_i + 1 + w / num_ec, ci = s_c + 1 + w % num_ec;
      int tr = p[ri];
      s_A[tr][ci] =
          d_Fq_sub(s_A[tr][ci], d_Fq_mul(s_A[tr][s_i], s_A[pivot_row][ci]));
    }
    __syncthreads();
    if (tid == 0) {
      s_i++;
      s_c++;
    }
    __syncthreads();
  }

  if (tid < QRUOV_m) {
    d_indices[tid] = pivot_col_idx[tid];
    d_p[tid] = p[tid];
  }
  if (tid == 0)
    *d_rank = current_rank;
  for (int idx = tid; idx < QRUOV_m * QRUOV_m; idx += blockDim.x) {
    int lr = idx / QRUOV_m, c = idx % QRUOV_m;
    global_A[lr * QRUOV_m + c] = s_A[p[lr]][c];
  }
}

__global__ void sample_a_solution_batched_kernel(
```

```
    Fq *d_all_eqn, int *d_all_p, int *d_all_rank, int *d_all_indices,
    Fq *d_all_b, Fq *d_all_random, Fq *d_all_oil_u, Fq *d_all_b2, int batch_N)
{
  int sid = blockIdx.x * blockDim.x + threadIdx.x;
  if (sid >= batch_N)
    return;

  Fq *eqn = d_all_eqn + sid * QRUOV_m * QRUOV_m;
  int *pp = d_all_p + sid * QRUOV_m;
  int rank = d_all_rank[sid];
  int *indices = d_all_indices + sid * QRUOV_m;
  Fq *b = d_all_b + sid * QRUOV_m;
  Fq *rand_element = d_all_random + sid * QRUOV_m;
  Fq *oil_u = d_all_oil_u + sid * QRUOV_m;
  Fq *b2 = d_all_b2 + sid * QRUOV_m;

  for (int i = 0; i < rank; i++) {
    uint64_t t = 0;
    for (int j = 0; j < i; j++)
      t += ((uint64_t)eqn[i * QRUOV_m + j]) * ((uint64_t)b2[j]);
    Fq tmp = d_Fq_sub(b[pp[i]], (Fq)(t % QRUOV_q));
    b2[i] = d_Fq_mul(tmp, d_Fq_inv_table[eqn[i * QRUOV_m + i]]);
  }

  int i = QRUOV_m - 1;
  for (int j = rank - 1; j >= 0; j--) {
    int k = indices[j];
    for (; i > k; i--)
      oil_u[i] = *rand_element++;
    uint64_t t = 0;
    for (int kk = k + 1; kk < QRUOV_m; kk++)
      t += ((uint64_t)eqn[j * QRUOV_m + kk]) * ((uint64_t)(oil_u[kk]));
    oil_u[i] = d_Fq_sub(b2[j], (Fq)(t % QRUOV_q));
    i--;
  }
  for (; i >= 0; i--)
    oil_u[i] = *rand_element++;
}

__global__ void fused_pack0_SIG_GEN_batched_kernel(Fq *d_all_oil_u,
                                                   uint64_t *d_SdT,
                                                   uint64_t *d_all_yT,
                                                   uint64_t *d_all_sig_s,
                                                   int batch_N) {
  int gid = blockIdx.x * blockDim.x + threadIdx.x;
  if (gid >= batch_N * QRUOV_N)
    return;
```

```
  int sid = gid / QRUOV_N;
  int i = gid % QRUOV_N;

  Fq *oil_u = d_all_oil_u + sid * QRUOV_m;
  uint64_t *yT = d_all_yT + sid * QRUOV_V;
  uint64_t *sig_s = d_all_sig_s + sid * QRUOV_N;

  if (i < QRUOV_V) {
    uint32_t acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0, acc4 = 0;
    for (int j = 0; j < QRUOV_M; j++) {
      uint64_t oj = (uint64_t)oil_u[j * QRUOV_L] |
                    ((uint64_t)oil_u[j * QRUOV_L + 1] << Fql_ws) |
                    ((uint64_t)oil_u[j * QRUOV_L + 2] << (Fql_ws * 2));
      uint64_t sdt = d_SdT[j * QRUOV_V + i];
      uint32_t a0, a1, a2, b0, b1, b2;
      unpack(oj, a0, a1, a2);
      unpack(sdt, b0, b1, b2);
      acc0 += a0 * b0;
      acc1 += a0 * b1 + a1 * b0;
      acc2 += a0 * b2 + a1 * b1 + a2 * b0;
      acc3 += a1 * b2 + a2 * b1;
      acc4 += a2 * b2;
    }
    uint64_t dot =
        repack((acc0 + acc3) % QRUOV_q, (acc1 + acc3 + acc4) % QRUOV_q,
               (acc2 + acc4) % QRUOV_q);
    uint64_t vin = yT[i];
    uint32_t v0, v1, v2, d0, d1, d2;
    unpack(vin, v0, v1, v2);
    unpack(dot, d0, d1, d2);
    sig_s[i] =
        repack((v0 + QRUOV_q - d0) % QRUOV_q, (v1 + QRUOV_q - d1) % QRUOV_q,
               (v2 + QRUOV_q - d2) % QRUOV_q);
  } else {
    int oi = i - QRUOV_V;
    sig_s[i] = (uint64_t)oil_u[oi * QRUOV_L] |
               ((uint64_t)oil_u[oi * QRUOV_L + 1] << Fql_ws) |
               ((uint64_t)oil_u[oi * QRUOV_L + 2] << (Fql_ws * 2));
  }
}

#define QRUOV_PRG2_CTX EVP_CIPHER_CTX
void print_fql(const char *label, Fql val) {
  uint32_t z0 = (uint32_t)(val & Fql_wm);
  uint32_t z1 = (uint32_t)((val >> Fql_ws) & Fql_wm);
  uint32_t z2 = (uint32_t)(val >> (Fql_ws * 2));
  printf("%s: [%u, %u, %u]\n", label, z0, z1, z2);
```

```
}

#define CONSISTENT_BATCH_SIZE_B 4
#define SIGNATURE_SIZE 30
int count = 0;
void QRUOV_Sign(
    const QRUOV_SEED seed_sk, // input (signing key)
    const QRUOV_SEED seed_pk, // input (signing key)
    const uint8_t message[],     // input (message)
    const size_t message_length, // input (message)
    QRUOV_SIGNATURE sig // output
) {
  // GPU info
  cudaDeviceProp prop;
  cudaGetDeviceProperties(&prop, 0);
  size_t free_mem, total_mem;
  cudaMemGetInfo(&free_mem, &total_mem);
  printf("GPU: %s | Memory: %.0f MB free / %.0f MB total\n\n", prop.name,
         free_mem / 1e6, total_mem / 1e6);

  // Expand mu
  uint8_t mu[QRUOV_MU_LEN];
  Expand_mu(seed_pk, message, message_length, mu);

  // Expand_sk (shared)
  uint64_t *d_Sd, *d_SdT;
  cudaMalloc(&d_Sd, QRUOV_V * QRUOV_M * sizeof(uint64_t));
  cudaMalloc(&d_SdT, QRUOV_M * QRUOV_V * sizeof(uint64_t));
  {
    const int n2 = QRUOV_L * QRUOV_V * QRUOV_M;
    uint8_t r2[QRUOV_tau2];
    QRUOV_PRG_CTX *ctx = PRG_init(seed_sk);
    RejSampPRG(ctx, 0, n2, QRUOV_tau2, r2);
    uint8_t *d_r2;
    cudaMalloc(&d_r2, QRUOV_tau2);
    cudaMemcpy(d_r2, r2, QRUOV_tau2, cudaMemcpyHostToDevice);
    ExpandMatrixVxM_kernel<<<4, 256>>>(d_r2, d_Sd);
    dim3 bd(16, 16), gd((QRUOV_V + 15) / 16, (QRUOV_M + 15) / 16);
    MATRIX_TRANSPOSE_VxM_kernel<<<gd, bd>>>(d_Sd, d_SdT);
    cudaFree(d_r2);
    PRG_final(ctx);
    OPENSSL_cleanse(r2, QRUOV_tau2);
  }

  // Expand_pk (shared)
  uint64_t *d_all_Pi1, *d_all_Pi2;
  cudaMalloc(&d_all_Pi1, QRUOV_m * QRUOV_V * QRUOV_V * sizeof(uint64_t));
```

```
cudaMalloc(&d_all_Pi2, QRUOV_m * QRUOV_V * QRUOV_M * sizeof(uint64_t));
{
  QRUOV_PRG_CTX *ctx_pk = PRG_init(seed_pk);
  uint8_t *all_r1 = (uint8_t *)malloc(QRUOV_m * N1);
  uint8_t *all_r2 = (uint8_t *)malloc(QRUOV_m * N2);
  for (int i = 0; i < QRUOV_m; i++) {
    QRUOV_PRG_CTX *ctx_1 = PRG_copy(ctx_pk);
    uint8_t t1[QRUOV_tau1];
    RejSampPRG(ctx_1, 2 * i, N1, QRUOV_tau1, t1);
    PRG_final(ctx_1);
    memcpy(all_r1 + i * N1, t1, N1);

    QRUOV_PRG_CTX *ctx_2 = PRG_copy(ctx_pk);
    uint8_t t2[QRUOV_tau2];
    RejSampPRG(ctx_2, 2 * i + 1, N2, QRUOV_tau2, t2);
    PRG_final(ctx_2);
    memcpy(all_r2 + i * N2, t2, N2);
  }
  uint8_t *d_r1, *d_r2;
  cudaMalloc(&d_r1, QRUOV_m * N1);
  cudaMalloc(&d_r2, QRUOV_m * N2);
  cudaMemset(d_all_Pi1, 0, QRUOV_m * QRUOV_V * QRUOV_V * sizeof(uint64_t));
  cudaMemset(d_all_Pi2, 0, QRUOV_m * QRUOV_V * QRUOV_M * sizeof(uint64_t));
  cudaMemcpy(d_r1, all_r1, QRUOV_m * N1, cudaMemcpyHostToDevice);
  cudaMemcpy(d_r2, all_r2, QRUOV_m * N2, cudaMemcpyHostToDevice);
  dim3 gvv((N_UPPER_TRI + 255) / 256, QRUOV_m);
  ExpandSymmetricMatrixVxV_batched_kernel<<<gvv, 256>>>(d_r1, d_all_Pi1);
  dim3 gvm((N_VXM + 255) / 256, QRUOV_m);
  ExpandMatrixVxM_batched_kernel<<<gvm, 256>>>(d_r2, d_all_Pi2);
  cudaFree(d_r1);
  cudaFree(d_r2);
  free(all_r1);
  free(all_r2);
  PRG_final(ctx_pk);
}

// Initialize batch sizes to test
int batch_sizes[SIGNATURE_SIZE];
for (int i = 0; i < SIGNATURE_SIZE; i++) {
  batch_sizes[i] = 1 << i;
}

for (int i = 0; i < SIGNATURE_SIZE; i++) {
  int N = batch_sizes[i];

  // Estimate per-signature for GPU memory:
  size_t per_sig_mem =
```

```
        (size_t)QRUOV_V * sizeof(uint64_t)           // d_all_yT:        52 *
8 = 416
        + (size_t)QRUOV_m * QRUOV_m * sizeof(Fq)     // d_all_eqn:       54 *
54 = 2916
        + (size_t)QRUOV_m * sizeof(Fq)               // d_all_c:         54
        + (size_t)sizeof(int)                        // d_all_rank:      4
        + (size_t)QRUOV_m * sizeof(int)              // d_all_indices:   216
        + (size_t)QRUOV_m * sizeof(int)              // d_all_p:         216
        + (size_t)QRUOV_m * sizeof(Fq)               // d_all_b:         54
        + (size_t)QRUOV_m * sizeof(Fq)               // d_all_random:    54
        + (size_t)QRUOV_m * sizeof(Fq)               // d_all_oil_u:     54
        + (size_t)QRUOV_m * sizeof(Fq)               // d_all_b2:        54
        + (size_t)QRUOV_N * sizeof(uint64_t);        // d_all_sig_s:     70 *
8 = 560
    size_t needed = (size_t)N * per_sig_mem;
    cudaMemGetInfo(&free_mem, &total_mem);
    if (needed > free_mem * 0.75) {
      printf("%-12d | Skipped — need %.0f MB, only %.0f MB free\n", N,
             needed / 1e6, free_mem / 1e6);
      printf("Allocate 25%% buffer for GPU runtime/context overhead\n");
      break;
    }

    // CPU PREP: generate per-signature data
    uint64_t *h_all_yT = (uint64_t *)malloc(N * QRUOV_V * sizeof(uint64_t));
    Fq *h_all_b = (Fq *)malloc(N * QRUOV_m * sizeof(Fq));
    Fq *h_all_random = (Fq *)malloc(N * QRUOV_m * sizeof(Fq));
    uint8_t *h_all_salts = (uint8_t *)malloc(N * QRUOV_SALT_LEN);
    QRUOV_PRG2_CTX **all_ctx_r = (QRUOV_PRG2_CTX **)malloc(N *
sizeof(QRUOV_PRG2_CTX *));

    for (int n = 0; n < N; n++) {
      // Generate unique seed_y per signature
      QRUOV_SEED seed_y;
      randombytes(seed_y, QRUOV_SEED_LEN);
      const int n3 = QRUOV_L * QRUOV_V;
      uint8_t r3[QRUOV_tau3];
      QRUOV_PRG_CTX *cy = PRG_init(seed_y);
      RejSampPRG(cy, 0, n3, QRUOV_tau3, r3);
      PRG_final(cy);
      for (int v = 0; v < QRUOV_V; v++)
        h_all_yT[n * QRUOV_V + v] = h_Fq2Fql(r3 + v * QRUOV_L);

      // Generate unique seed_r per signature
      QRUOV_SEED seed_r;
      randombytes(seed_r, QRUOV_SEED_LEN);
      all_ctx_r[n] = PRG2_init(seed_r);
```

```
      // Generate unique seed_sol per signature
      QRUOV_SEED seed_sol;
      randombytes(seed_sol, QRUOV_SEED_LEN);
      Expand_sol(seed_sol, h_all_random + n * QRUOV_m);
    }

    uint64_t *d_all_yT;
    Fq *d_all_eqn, *d_all_c;
    int *d_all_rank, *d_all_indices, *d_all_p;
    Fq *d_all_b, *d_all_random_buf, *d_all_oil_u, *d_all_b2;
    uint64_t *d_all_sig_s;

    cudaMalloc(&d_all_yT, N * QRUOV_V * sizeof(uint64_t));
    cudaMalloc(&d_all_eqn, N * QRUOV_m * QRUOV_m * sizeof(Fq));
    cudaMalloc(&d_all_c, N * QRUOV_m * sizeof(Fq));
    cudaMalloc(&d_all_rank, N * sizeof(int));
    cudaMalloc(&d_all_indices, N * QRUOV_m * sizeof(int));
    cudaMalloc(&d_all_p, N * QRUOV_m * sizeof(int));
    cudaMalloc(&d_all_b, N * QRUOV_m * sizeof(Fq));
    cudaMalloc(&d_all_random_buf, N * QRUOV_m * sizeof(Fq));
    cudaMalloc(&d_all_oil_u, N * QRUOV_m * sizeof(Fq));
    cudaMalloc(&d_all_b2, N * QRUOV_m * sizeof(Fq));
    cudaMalloc(&d_all_sig_s, N * QRUOV_N * sizeof(uint64_t));

    cudaMemcpy(d_all_yT, h_all_yT, N * QRUOV_V * sizeof(uint64_t),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_all_random_buf, h_all_random, N * QRUOV_m,
cudaMemcpyHostToDevice);

    dim3 grid_fused(N, 54);


    kernel_fused_batched<<<grid_fused, 64>>>(d_all_yT, d_all_Pi1, d_all_Pi2,
d_Sd, d_all_eqn, d_all_c, N);
    LU_decompose_batched_kernel<<<N, 128>>>(d_all_eqn, d_all_rank,
d_all_indices, d_all_p, N);


    Fq msg[QRUOV_m];
    Fq *h_all_eqn = (Fq *)malloc(N * QRUOV_m * QRUOV_m * sizeof(Fq));
    Fq *h_all_c = (Fq *)malloc(N * QRUOV_m);
    int *h_all_rank = (int *)malloc(N * sizeof(int));
    int *h_all_p = (int *)malloc(N * QRUOV_m * sizeof(int));
    int *h_all_indices = (int *)malloc(N * QRUOV_m * sizeof(int));
    cudaMemcpy(h_all_eqn, d_all_eqn, N * QRUOV_m * QRUOV_m * sizeof(Fq),
cudaMemcpyDeviceToHost);
```

```
    cudaMemcpy(h_all_p, d_all_p, N * QRUOV_m * sizeof(int),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_indices, d_all_indices, N * QRUOV_m * sizeof(int),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_c, d_all_c, N * QRUOV_m, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_rank, d_all_rank, N * sizeof(int),
cudaMemcpyDeviceToHost);

    int rare_case = 0;
    for (int n = 0; n < N; n++) {
      QRUOV_PRG2_CTX *ctx_r = all_ctx_r[n];
      Fq *my_b = h_all_b + n * QRUOV_m;
      Fq *my_c = h_all_c + n * QRUOV_m;

      if (h_all_rank[n] == QRUOV_m) {
        // Full rank — single salt attempt always consistent
        PRG2_yield(ctx_r, QRUOV_SALT_LEN, h_all_salts + n * QRUOV_SALT_LEN);
        Hash(mu, h_all_salts + n * QRUOV_SALT_LEN, msg);
        for (int i = 0; i < QRUOV_m; i++)
          my_b[i] = Fq_sub(msg[i], my_c[i]);
      } else {
        rare_case++;
        Fq *eqn_flat = h_all_eqn + n * QRUOV_m * QRUOV_m;
        int *p_vec = h_all_p + n * QRUOV_m;
        int *idx_vec = h_all_indices + n * QRUOV_m;

        ECHELON_FORM echelon_form;
        Fq eqn_storage[QRUOV_m][QRUOV_m];
        for (int r = 0; r < QRUOV_m; r++) {
          for (int c = 0; c < QRUOV_m; c++)
            eqn_storage[r][c] = eqn_flat[r * QRUOV_m + c];
          echelon_form->row[r].col = eqn_storage[r];
          echelon_form->row[r].original_row_id = p_vec[r];
        }
        for (int r = 0; r < QRUOV_m; r++)
          echelon_form->eqn[r] = &echelon_form->row[r];
        echelon_form->rank = h_all_rank[n];
        for (int r = 0; r < QRUOV_m; r++)
          echelon_form->index[r] = idx_vec[r];

        Fq R[QRUOV_m][QRUOV_m];
        int cacheR = 0;

        do {
          PRG2_yield(ctx_r, QRUOV_SALT_LEN, h_all_salts + n * QRUOV_SALT_LEN);
          Hash(mu, h_all_salts + n * QRUOV_SALT_LEN, msg);
          for (int i = 0; i < QRUOV_m; i++)
```

```
            my_b[i] = Fq_sub(msg[i], my_c[i]);
        } while (!consistent(echelon_form, my_b, &cacheR, R));
      }
      PRG2_final(ctx_r);
    }
    free(all_ctx_r);
    free(h_all_eqn);
    free(h_all_p);
    free(h_all_indices);

    cudaMemcpy(d_all_b, h_all_b, N * QRUOV_m * sizeof(Fq),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_all_rank, h_all_rank, N * sizeof(int),
cudaMemcpyHostToDevice);

    int threads_sa = 256;
    int blocks_sa = (N + threads_sa - 1) / threads_sa;
    int total_siggen = N * QRUOV_N;


    sample_a_solution_batched_kernel<<<blocks_sa, threads_sa>>>(
        d_all_eqn, d_all_p, d_all_rank, d_all_indices, d_all_b,
d_all_random_buf, d_all_oil_u, d_all_b2, N);
    fused_pack0_SIG_GEN_batched_kernel<<<(total_siggen + 255) / 256, 256>>>(
        d_all_oil_u, d_SdT, d_all_yT, d_all_sig_s, N);


    uint64_t *h_all_sig_s = (uint64_t *)malloc(N * QRUOV_N *
sizeof(uint64_t));
    cudaMemcpy(h_all_sig_s, d_all_sig_s, N * QRUOV_N *
sizeof(uint64_t),cudaMemcpyDeviceToHost);

    memcpy(sig->r, h_all_salts, QRUOV_SALT_LEN);
    memcpy(sig->s, h_all_sig_s, QRUOV_N * sizeof(uint64_t));

    free(h_all_yT);
    free(h_all_b);
    free(h_all_random);
    free(h_all_salts);
    free(h_all_c);
    free(h_all_rank);
    free(h_all_sig_s);

    cudaFree(d_all_yT);
    cudaFree(d_all_eqn);
    cudaFree(d_all_c);
    cudaFree(d_all_rank);
```

```
    cudaFree(d_all_indices);
    cudaFree(d_all_p);
    cudaFree(d_all_b);
    cudaFree(d_all_random_buf);
    cudaFree(d_all_oil_u);
    cudaFree(d_all_b2);
    cudaFree(d_all_sig_s);  }
  printf("\nDone\n");

  cudaFree(d_all_Pi1);
  cudaFree(d_all_Pi2);
  cudaFree(d_Sd);
  cudaFree(d_SdT);

  return;
}
```

QRUOV_Verify

```
#include "qruov.h"
#include <cuda_runtime.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

static void Expand_mu(const QRUOV_SEED seed_pk, const uint8_t message[],
                      const size_t message_length, uint8_t mu[QRUOV_MU_LEN]) {
  EVP_MD_CTX *ctx = EVP_MD_CTX_new();
  EVP_DigestInit_ex(ctx, EVP_shake256(), NULL);
  EVP_DigestUpdate(ctx, seed_pk, QRUOV_SEED_LEN);
  EVP_DigestUpdate(ctx, message, message_length);
  EVP_DigestFinalXOF(ctx, mu, QRUOV_MU_LEN);
  EVP_MD_CTX_free(ctx);
}

static void Hash(const uint8_t mu[QRUOV_MU_LEN], const QRUOV_SALT salt,
                 uint8_t dst[QRUOV_m]) {
  uint8_t tmp[QRUOV_tau4];
  EVP_MD_CTX *ctx = EVP_MD_CTX_new();
  EVP_DigestInit_ex(ctx, EVP_shake256(), NULL);
  EVP_DigestUpdate(ctx, mu, QRUOV_MU_LEN);
  EVP_DigestUpdate(ctx, salt, QRUOV_SALT_LEN);
  EVP_DigestFinalXOF(ctx, tmp, QRUOV_tau4);
  RejSamp(QRUOV_m, QRUOV_tau4, tmp);
  memcpy(dst, tmp, QRUOV_m);
  EVP_MD_CTX_free(ctx);
}

__global__ void ExpandSymmetricMatrixVxV_batched_kernel(
    const uint8_t *__restrict__ all_src, // [QRUOV_m * N1]
    uint64_t *__restrict__ all_A         // [QRUOV_m * V * V]
) {
  int iter = blockIdx.y;
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k >= N_UPPER_TRI)
    return;

  const uint8_t *src = all_src + iter * N1;
  uint64_t *A = all_A + iter * QRUOV_V * QRUOV_V;

  float fk = (float)k;
  int i = (int)((2.0f * QRUOV_V + 1.0f -
                sqrtf((2.0f * QRUOV_V + 1.0f) * (2.0f * QRUOV_V + 1.0f) -
```

```
                    8.0f * fk)) /
               2.0f);
  if (i > 0) {
    int start_of_row_i = i * QRUOV_V - i * (i - 1) / 2;
    if (start_of_row_i > k)
      i--;
  }
  int start_of_row = i * QRUOV_V - i * (i - 1) / 2;
  int j = i + (k - start_of_row);

  int byte_offset = k * 3;
  uint64_t z0 = src[byte_offset];
  uint64_t z1 = src[byte_offset + 1];
  uint64_t z2 = src[byte_offset + 2];
  uint64_t fql = z0 | (z1 << 24) | (z2 << 48);

  A[i * QRUOV_V + j] = fql;
  if (i != j)
    A[j * QRUOV_V + i] = fql;
}

__global__ void ExpandMatrixVxM_batched_kernel(
    const uint8_t *__restrict__ all_src, // [QRUOV_m * N2]
    uint64_t *__restrict__ all_A         // [QRUOV_m * V * M]
) {
  int iter = blockIdx.y;
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k >= N_VXM)
    return;

  const uint8_t *src = all_src + iter * N2;
  uint64_t *A = all_A + iter * QRUOV_V * QRUOV_M;

  int byte_offset = k * 3;
  uint64_t z0 = src[byte_offset];
  uint64_t z1 = src[byte_offset + 1];
  uint64_t z2 = src[byte_offset + 2];

  A[k] = z0 | (z1 << 24) | (z2 << 48);
}

__device__ __forceinline__ void fql_mac(uint32_t &acc0, uint32_t &acc1,
                                        uint32_t &acc2, uint64_t A,
                                        uint64_t B) {
  uint32_t a0 = (uint32_t)(A & Fql_wm);
  uint32_t a1 = (uint32_t)((A >> 24) & Fql_wm);
  uint32_t a2 = (uint32_t)(A >> 48);
```

```
  uint32_t b0 = (uint32_t)(B & Fql_wm);
  uint32_t b1 = (uint32_t)((B >> 24) & Fql_wm);
  uint32_t b2 = (uint32_t)(B >> 48);

  uint32_t c3 = a1 * b2 + a2 * b1; // x^3 coefficient: folds to x^0 + x^1
  uint32_t c4 = a2 * b2;           // x^4 coefficient: folds to x^1 + x^2

  acc0 += a0 * b0 + c3;
  acc1 += a0 * b1 + a1 * b0 + c3 + c4;
  acc2 += a0 * b2 + a1 * b1 + a2 * b0 + c4;
}

__device__ __forceinline__ uint64_t fql_pack(uint32_t z0, uint32_t z1,
                                             uint32_t z2) {
  return (uint64_t)(z0 % QRUOV_q) | ((uint64_t)(z1 % QRUOV_q) << 24) |
         ((uint64_t)(z2 % QRUOV_q) << 48);
}

__global__ void VERIFY_batched_kernel(
    const uint64_t *__restrict__ d_Pi1, // [m * V * V]
    const uint64_t
        *__restrict__ d_Pi2,           // [m * V * M]  (Pi2T[k][j] =
Pi2[j*M+k])
    const uint64_t *__restrict__ d_P3, // [m * M * M]
    const uint64_t *__restrict__ d_all_oil,     // [batch_N * M]
    const uint64_t *__restrict__ d_all_vinegar, // [batch_N * V]
    const uint8_t *__restrict__ d_all_msg,      // [batch_N * m]
    int *d_all_result,                          // [batch_N]
    int batch_N) {
  const int i = blockIdx.x;    // equation index
  const int tid = threadIdx.x; // 0 .. QRUOV_V-1

  const uint64_t *Pi1 = d_Pi1 + (size_t)i * QRUOV_V * QRUOV_V;
  const uint64_t *Pi2 = d_Pi2 + (size_t)i * QRUOV_V * QRUOV_M;
  const uint64_t *Pi3 = d_P3 + (size_t)i * QRUOV_M * QRUOV_M;

  for (int sig_id = blockIdx.y; sig_id < batch_N; sig_id += gridDim.y) {
    __shared__ uint64_t tmp_v[QRUOV_V];
    __shared__ uint64_t tmp_o[QRUOV_M];

    const uint64_t *d_oil = d_all_oil + (size_t)sig_id * QRUOV_M;
    const uint64_t *d_vinegar = d_all_vinegar + (size_t)sig_id * QRUOV_V;
    const uint8_t *d_msg = d_all_msg + (size_t)sig_id * QRUOV_m;

    if (tid < QRUOV_V) {
      uint32_t acc0 = 0, acc1 = 0, acc2 = 0;
```

```
      for (int k = 0; k < QRUOV_M; k++)
        fql_mac(acc0, acc1, acc2, Pi2[tid * QRUOV_M + k], d_oil[k]);

      acc0 <<= 1;
      acc1 <<= 1;
      acc2 <<= 1;

      for (int k = 0; k < QRUOV_V; k++)
        fql_mac(acc0, acc1, acc2, Pi1[tid * QRUOV_V + k], d_vinegar[k]);

      tmp_v[tid] = fql_pack(acc0, acc1, acc2);
    }

    if (tid < QRUOV_M) {
      uint32_t acc0 = 0, acc1 = 0, acc2 = 0;

      for (int k = 0; k < QRUOV_M; k++)
        fql_mac(acc0, acc1, acc2, Pi3[tid * QRUOV_M + k], d_oil[k]);

      tmp_o[tid] = fql_pack(acc0, acc1, acc2);
    }

    __syncthreads();

    if (tid == 0) {
      uint32_t acc0 = 0, acc1 = 0, acc2 = 0;

      for (int j = 0; j < QRUOV_V; j++)
        fql_mac(acc0, acc1, acc2, d_vinegar[j], tmp_v[j]);

      for (int j = 0; j < QRUOV_M; j++)
        fql_mac(acc0, acc1, acc2, d_oil[j], tmp_o[j]);

      uint8_t t0 = (uint8_t)(acc0 % QRUOV_q);
      int pass = (d_msg[i] == t0) ? 1 : 0;
      atomicAnd(&d_all_result[sig_id], pass);
    }

    __syncthreads();
  }
}

#define BENCHMARK_SIZE 30

void QRUOV_Verify_Kernel_Batch(const QRUOV_SEED seed_pk, const QRUOV_P3 P3,
                               const uint8_t message[],
                               const size_t message_length,
```

```
                                const QRUOV_SIGNATURE sig) {
  cudaDeviceProp prop;
  cudaGetDeviceProperties(&prop, 0);
  size_t free_mem, total_mem;
  cudaMemGetInfo(&free_mem, &total_mem);
  printf("GPU: %s | Memory: %.0f MB free / %.0f MB total\n\n", prop.name,
         free_mem / 1e6, total_mem / 1e6);

  uint8_t mu[QRUOV_MU_LEN];
  Expand_mu(seed_pk, message, message_length, mu);

  Fq msg[QRUOV_m];
  Hash(mu, sig->r, msg);

  const Fql *vinegar = sig->s;
  const Fql *oil = sig->s + QRUOV_V;

  // Expand_pk (shared)
  uint64_t *d_Pi1, *d_Pi2;
  cudaMalloc(&d_Pi1, (size_t)QRUOV_m * QRUOV_V * QRUOV_V * sizeof(uint64_t));
  cudaMalloc(&d_Pi2, (size_t)QRUOV_m * QRUOV_V * QRUOV_M * sizeof(uint64_t));
  cudaMemset(d_Pi1, 0, (size_t)QRUOV_m * QRUOV_V * QRUOV_V *
sizeof(uint64_t));
  cudaMemset(d_Pi2, 0, (size_t)QRUOV_m * QRUOV_V * QRUOV_M *
sizeof(uint64_t));

  {
    uint8_t *all_r1 = (uint8_t *)malloc((size_t)QRUOV_m * N1);
    uint8_t *all_r2 = (uint8_t *)malloc((size_t)QRUOV_m * N2);

    QRUOV_PRG_CTX *ctx_pk = PRG_init(seed_pk);
    for (int i = 0; i < QRUOV_m; i++) {
      QRUOV_PRG_CTX *ctx1 = PRG_copy(ctx_pk);
      uint8_t t1[QRUOV_tau1];
      RejSampPRG(ctx1, 2 * i, N1, QRUOV_tau1, t1);
      PRG_final(ctx1);
      memcpy(all_r1 + i * N1, t1, N1);

      QRUOV_PRG_CTX *ctx2 = PRG_copy(ctx_pk);
      uint8_t t2[QRUOV_tau2];
      RejSampPRG(ctx2, 2 * i + 1, N2, QRUOV_tau2, t2);
      PRG_final(ctx2);
      memcpy(all_r2 + i * N2, t2, N2);
    }
    PRG_final(ctx_pk);

    uint8_t *d_r1, *d_r2;
```

```
  cudaMalloc(&d_r1, (size_t)QRUOV_m * N1);
  cudaMalloc(&d_r2, (size_t)QRUOV_m * N2);
  cudaMemcpy(d_r1, all_r1, (size_t)QRUOV_m * N1, cudaMemcpyHostToDevice);
  cudaMemcpy(d_r2, all_r2, (size_t)QRUOV_m * N2, cudaMemcpyHostToDevice);

  dim3 gvv((N_UPPER_TRI + 255) / 256, QRUOV_m);
  ExpandSymmetricMatrixVxV_batched_kernel<<<gvv, 256>>>(d_r1, d_Pi1);

  dim3 gvm((N_VXM + 255) / 256, QRUOV_m);
  ExpandMatrixVxM_batched_kernel<<<gvm, 256>>>(d_r2, d_Pi2);

  cudaFree(d_r1);
  cudaFree(d_r2);
  free(all_r1);
  free(all_r2);
}

uint64_t *d_P3;
cudaMalloc(&d_P3, (size_t)QRUOV_m * QRUOV_M * QRUOV_M * sizeof(uint64_t));
cudaMemcpy(d_P3, P3, (size_t)QRUOV_m * QRUOV_M * QRUOV_M * sizeof(uint64_t),
           cudaMemcpyHostToDevice);

for (int bi = 0; bi < BENCHMARK_SIZE; bi++) {
  int batch_N = 1 << bi;

  // Allow 25% GPU VRAM for buffer
  size_t per_sig_mem = (size_t)QRUOV_V * sizeof(uint64_t)   // d_all_vinegar
                     + (size_t)QRUOV_M * sizeof(uint64_t) // d_all_oil
                     + (size_t)QRUOV_m * sizeof(uint8_t)  // d_all_msg
                     + (size_t)sizeof(int);               // d_all_result
  size_t needed = (size_t)batch_N * per_sig_mem;
  cudaMemGetInfo(&free_mem, &total_mem);
  if (needed > free_mem * 0.75) {
    printf("%-12d | Skipped — need %.0f MB, only %.0f MB free\n", batch_N,
           needed / 1e6, free_mem / 1e6);
    break;
  }

  uint64_t *h_all_vinegar =
      (uint64_t *)malloc((size_t)batch_N * QRUOV_V * sizeof(uint64_t));
  uint64_t *h_all_oil =
      (uint64_t *)malloc((size_t)batch_N * QRUOV_M * sizeof(uint64_t));
  uint8_t *h_all_msg =
      (uint8_t *)malloc((size_t)batch_N * QRUOV_m * sizeof(uint8_t));
  int *h_all_result = (int *)malloc((size_t)batch_N * sizeof(int));

  for (int n = 0; n < batch_N; n++) {
```

```
      memcpy(h_all_vinegar + n * QRUOV_V, vinegar, QRUOV_V *
sizeof(uint64_t));
      memcpy(h_all_oil + n * QRUOV_M, oil, QRUOV_M * sizeof(uint64_t));
      memcpy(h_all_msg + n * QRUOV_m, msg, QRUOV_m * sizeof(uint8_t));
      h_all_result[n] = 1;
    }

    uint64_t *d_all_vinegar, *d_all_oil;
    uint8_t *d_all_msg;
    int *d_all_result;

    cudaMalloc(&d_all_vinegar, (size_t)batch_N * QRUOV_V * sizeof(uint64_t));
    cudaMalloc(&d_all_oil, (size_t)batch_N * QRUOV_M * sizeof(uint64_t));
    cudaMalloc(&d_all_msg, (size_t)batch_N * QRUOV_m * sizeof(uint8_t));
    cudaMalloc(&d_all_result, (size_t)batch_N * sizeof(int));

    cudaMemcpy(d_all_vinegar, h_all_vinegar,
               (size_t)batch_N * QRUOV_V * sizeof(uint64_t),
               cudaMemcpyHostToDevice);
    cudaMemcpy(d_all_oil, h_all_oil,
               (size_t)batch_N * QRUOV_M * sizeof(uint64_t),
               cudaMemcpyHostToDevice);
    cudaMemcpy(d_all_msg, h_all_msg,
               (size_t)batch_N * QRUOV_m * sizeof(uint8_t),
               cudaMemcpyHostToDevice);
    cudaMemcpy(d_all_result, h_all_result, (size_t)batch_N * sizeof(int),
               cudaMemcpyHostToDevice);

    int grid_y;
    if (batch_N < 65535) {
      grid_y = batch_N;
    } else {
      grid_y = 65535;
    }
    dim3 grid(QRUOV_m, grid_y);
    VERIFY_batched_kernel<<<grid, QRUOV_V>>>(d_Pi1, d_Pi2, d_P3, d_all_oil,
                                             d_all_vinegar, d_all_msg,
                                             d_all_result, batch_N);

    cudaMemcpy(h_all_result, d_all_result, (size_t)batch_N * sizeof(int),
               cudaMemcpyDeviceToHost);

    int all_pass = 1;
    for (int n = 0; n < batch_N; n++) {
      if (!h_all_result[n]) {
        all_pass = 0;
        break;
```

```
      }
    }

    free(h_all_vinegar);
    free(h_all_oil);
    free(h_all_msg);
    free(h_all_result);
    cudaFree(d_all_vinegar);
    cudaFree(d_all_oil);
    cudaFree(d_all_msg);
    cudaFree(d_all_result);
  }

  printf("\nDone\n");

  cudaFree(d_Pi1);
  cudaFree(d_Pi2);
  cudaFree(d_P3);
}
```

QRUOV_Tensor

```
#include "qruov.h"
#include <cuda_runtime.h>
#include <mma.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
using namespace nvcuda;

static void Expand_mu(const QRUOV_SEED seed_pk, const uint8_t message[],
                      const size_t message_length, uint8_t mu[QRUOV_MU_LEN]) {
  EVP_MD_CTX *ctx = EVP_MD_CTX_new();
  EVP_DigestInit_ex(ctx, EVP_shake256(), NULL);
  EVP_DigestUpdate(ctx, seed_pk, QRUOV_SEED_LEN);
  EVP_DigestUpdate(ctx, message, message_length);
  EVP_DigestFinalXOF(ctx, mu, QRUOV_MU_LEN);
  EVP_MD_CTX_free(ctx);
}

static void Hash(const uint8_t mu[QRUOV_MU_LEN], const QRUOV_SALT salt,
                 uint8_t dst[QRUOV_m]) {
  uint8_t tmp[QRUOV_tau4];
  EVP_MD_CTX *ctx = EVP_MD_CTX_new();
  EVP_DigestInit_ex(ctx, EVP_shake256(), NULL);
  EVP_DigestUpdate(ctx, mu, QRUOV_MU_LEN);
  EVP_DigestUpdate(ctx, salt, QRUOV_SALT_LEN);
  EVP_DigestFinalXOF(ctx, tmp, QRUOV_tau4);
  RejSamp(QRUOV_m, QRUOV_tau4, tmp);
  memcpy(dst, tmp, QRUOV_m);
  EVP_MD_CTX_free(ctx);
}

__global__ void ExpandSymmetricMatrixVxV_batched_kernel(
    const uint8_t *__restrict__ all_src, // [QRUOV_m * N1]
    uint64_t *__restrict__ all_A         // [QRUOV_m * V * V]
) {
  int iter = blockIdx.y;
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k >= N_UPPER_TRI)
    return;

  const uint8_t *src = all_src + iter * N1;
  uint64_t *A = all_A + iter * QRUOV_V * QRUOV_V;

  float fk = (float)k;
```

```
  int i = (int)((2.0f * QRUOV_V + 1.0f -
                 sqrtf((2.0f * QRUOV_V + 1.0f) * (2.0f * QRUOV_V + 1.0f) -
                       8.0f * fk)) /
                2.0f);
  if (i > 0) {
    int start_of_row_i = i * QRUOV_V - i * (i - 1) / 2;
    if (start_of_row_i > k)
      i--;
  }
  int start_of_row = i * QRUOV_V - i * (i - 1) / 2;
  int j = i + (k - start_of_row);

  int byte_offset = k * 3;
  uint64_t z0 = src[byte_offset];
  uint64_t z1 = src[byte_offset + 1];
  uint64_t z2 = src[byte_offset + 2];
  uint64_t fql = z0 | (z1 << 24) | (z2 << 48);

  A[i * QRUOV_V + j] = fql;
  if (i != j)
    A[j * QRUOV_V + i] = fql;
}

__global__ void ExpandMatrixVxM_batched_kernel(
    const uint8_t *__restrict__ all_src, // [QRUOV_m * N2]
    uint64_t *__restrict__ all_A         // [QRUOV_m * V * M]
) {
  int iter = blockIdx.y;
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k >= N_VXM)
    return;

  const uint8_t *src = all_src + iter * N2;
  uint64_t *A = all_A + iter * QRUOV_V * QRUOV_M;

  int byte_offset = k * 3;
  uint64_t z0 = src[byte_offset];
  uint64_t z1 = src[byte_offset + 1];
  uint64_t z2 = src[byte_offset + 2];

  A[k] = z0 | (z1 << 24) | (z2 << 48);
}

__device__ __forceinline__ void fql_mac(uint32_t &acc0, uint32_t &acc1,
                                        uint32_t &acc2, uint64_t A,
                                        uint64_t B) {
  uint32_t a0 = (uint32_t)(A & Fql_wm);
```

```
  uint32_t a1 = (uint32_t)((A >> 24) & Fql_wm);
  uint32_t a2 = (uint32_t)(A >> 48);
  uint32_t b0 = (uint32_t)(B & Fql_wm);
  uint32_t b1 = (uint32_t)((B >> 24) & Fql_wm);
  uint32_t b2 = (uint32_t)(B >> 48);

  uint32_t c3 = a1 * b2 + a2 * b1; // x^3 coefficient: folds to x^0 + x^1
  uint32_t c4 = a2 * b2;           // x^4 coefficient: folds to x^1 + x^2

  acc0 += a0 * b0 + c3;
  acc1 += a0 * b1 + a1 * b0 + c3 + c4;
  acc2 += a0 * b2 + a1 * b1 + a2 * b0 + c4;
}

__device__ __forceinline__ uint64_t fql_pack(uint32_t z0, uint32_t z1,
                                             uint32_t z2) {
  return (uint64_t)(z0 % QRUOV_q) | ((uint64_t)(z1 % QRUOV_q) << 24) |
         ((uint64_t)(z2 % QRUOV_q) << 48);
}

#define TILE_B 16 // signatures per block (matches WMMA N-dim)
#define V_PAD 64  // QRUOV_V=52 padded to mult of 16
#define M_PAD 32  // QRUOV_M=18 padded to mult of 16

#define V_SMEM_A_HALF (3 * V_PAD * V_PAD)
#define V_SMEM_B_HALF (3 * V_PAD * TILE_B)
#define V_SMEM_C_FLOAT (5 * V_PAD * TILE_B)
#define
V_TENSOR_SMEM_BYTES                                                    \
  (V_SMEM_A_HALF * 2 + V_SMEM_B_HALF * 2 + V_SMEM_C_FLOAT * 4)

__device__ void wmma_fql_gemm_verify(const uint64_t *A_fql, int A_rows,
                                     int A_cols, int A_row_stride,
                                     const uint64_t *B_fql, int B_rows,
                                     int B_cols, int B_row_stride,
                                     uint64_t *out_fql, int out_stride,
                                     int M_dim, int K_pad, int N_dim,
                                     half *smem_A,  // [3][M_dim][K_pad]
                                     half *smem_B,  // [3][K_pad][N_dim]
                                     float *smem_C, // [5][M_dim][N_dim]
                                     int tid, int warp_id) {
  const int A_coeff_stride = M_dim * K_pad;
  const int B_coeff_stride = K_pad * N_dim;
  const int C_coeff_stride = M_dim * N_dim;

  for (int idx = tid; idx < M_dim * K_pad; idx += blockDim.x) {
    int r = idx / K_pad, c = idx % K_pad;
```

```
    if (r < A_rows && c < A_cols) {
      uint64_t v = A_fql[r * A_row_stride + c];
      smem_A[0 * A_coeff_stride + idx] =
          __uint2half_rn((uint32_t)(v & 0xFFFFFF));
      smem_A[1 * A_coeff_stride + idx] =
          __uint2half_rn((uint32_t)((v >> 24) & 0xFFFFFF));
      smem_A[2 * A_coeff_stride + idx] = __uint2half_rn((uint32_t)(v >> 48));
    } else {
      smem_A[0 * A_coeff_stride + idx] = __float2half(0.0f);
      smem_A[1 * A_coeff_stride + idx] = __float2half(0.0f);
      smem_A[2 * A_coeff_stride + idx] = __float2half(0.0f);
    }
  }

  for (int idx = tid; idx < K_pad * N_dim; idx += blockDim.x) {
    int r = idx / N_dim, c = idx % N_dim;
    if (r < B_rows && c < B_cols) {
      uint64_t v = B_fql[r * B_row_stride + c];
      smem_B[0 * B_coeff_stride + idx] =
          __uint2half_rn((uint32_t)(v & 0xFFFFFF));
      smem_B[1 * B_coeff_stride + idx] =
          __uint2half_rn((uint32_t)((v >> 24) & 0xFFFFFF));
      smem_B[2 * B_coeff_stride + idx] = __uint2half_rn((uint32_t)(v >> 48));
    } else {
      smem_B[0 * B_coeff_stride + idx] = __float2half(0.0f);
      smem_B[1 * B_coeff_stride + idx] = __float2half(0.0f);
      smem_B[2 * B_coeff_stride + idx] = __float2half(0.0f);
    }
  }
  __syncthreads();

  int m_tiles = M_dim / 16;
  int n_tiles = N_dim / 16;
  int k_tiles = K_pad / 16;
  int total_tiles = m_tiles * n_tiles;

  for (int tile = warp_id; tile < total_tiles; tile += 4) {
    int mt = tile / n_tiles;
    int nt = tile % n_tiles;

    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c0, c1, c2, c3, c4;
    wmma::fill_fragment(c0, 0.0f);
    wmma::fill_fragment(c1, 0.0f);
    wmma::fill_fragment(c2, 0.0f);
    wmma::fill_fragment(c3, 0.0f);
    wmma::fill_fragment(c4, 0.0f);
```

```
    for (int kt = 0; kt < k_tiles; kt++) {
      wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a0,
a1,
          a2;
      wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b0,
b1,
          b2;

      wmma::load_matrix_sync(
          a0, &smem_A[0 * A_coeff_stride + mt * 16 * K_pad + kt * 16], K_pad);
      wmma::load_matrix_sync(
          a1, &smem_A[1 * A_coeff_stride + mt * 16 * K_pad + kt * 16], K_pad);
      wmma::load_matrix_sync(
          a2, &smem_A[2 * A_coeff_stride + mt * 16 * K_pad + kt * 16], K_pad);

      wmma::load_matrix_sync(
          b0, &smem_B[0 * B_coeff_stride + kt * 16 * N_dim + nt * 16], N_dim);
      wmma::load_matrix_sync(
          b1, &smem_B[1 * B_coeff_stride + kt * 16 * N_dim + nt * 16], N_dim);
      wmma::load_matrix_sync(
          b2, &smem_B[2 * B_coeff_stride + kt * 16 * N_dim + nt * 16], N_dim);

      wmma::mma_sync(c0, a0, b0, c0);
      wmma::mma_sync(c1, a0, b1, c1);
      wmma::mma_sync(c1, a1, b0, c1);
      wmma::mma_sync(c2, a0, b2, c2);
      wmma::mma_sync(c2, a1, b1, c2);
      wmma::mma_sync(c2, a2, b0, c2);
      wmma::mma_sync(c3, a1, b2, c3);
      wmma::mma_sync(c3, a2, b1, c3);
      wmma::mma_sync(c4, a2, b2, c4);
    }

    wmma::store_matrix_sync(
        &smem_C[0 * C_coeff_stride + mt * 16 * N_dim + nt * 16], c0, N_dim,
        wmma::mem_row_major);
    wmma::store_matrix_sync(
        &smem_C[1 * C_coeff_stride + mt * 16 * N_dim + nt * 16], c1, N_dim,
        wmma::mem_row_major);
    wmma::store_matrix_sync(
        &smem_C[2 * C_coeff_stride + mt * 16 * N_dim + nt * 16], c2, N_dim,
        wmma::mem_row_major);
    wmma::store_matrix_sync(
        &smem_C[3 * C_coeff_stride + mt * 16 * N_dim + nt * 16], c3, N_dim,
        wmma::mem_row_major);
    wmma::store_matrix_sync(
        &smem_C[4 * C_coeff_stride + mt * 16 * N_dim + nt * 16], c4, N_dim,
```

```
        wmma::mem_row_major);
  }
  __syncthreads();

  int out_rows = A_rows, out_cols = B_cols;
  for (int idx = tid; idx < out_rows * out_cols; idx += blockDim.x) {
    int r = idx / out_cols, c = idx % out_cols;
    int flat = r * N_dim + c;
    uint32_t v0 = ((uint32_t)smem_C[0 * C_coeff_stride + flat] +
                   (uint32_t)smem_C[3 * C_coeff_stride + flat]) %
                  QRUOV_q;
    uint32_t v1 = ((uint32_t)smem_C[1 * C_coeff_stride + flat] +
                   (uint32_t)smem_C[3 * C_coeff_stride + flat] +
                   (uint32_t)smem_C[4 * C_coeff_stride + flat]) %
                  QRUOV_q;
    uint32_t v2 = ((uint32_t)smem_C[2 * C_coeff_stride + flat] +
                   (uint32_t)smem_C[4 * C_coeff_stride + flat]) %
                  QRUOV_q;
    out_fql[r * out_stride + c] =
        ((uint64_t)v0) | ((uint64_t)v1 << 24) | ((uint64_t)v2 << 48);
  }
  __syncthreads();
}

__global__ void __launch_bounds__(128, 1) VERIFY_tensor_batched_kernel(
    const uint64_t *__restrict__ d_Pi1,         // [m * V * V]
    const uint64_t *__restrict__ d_Pi2,         // [m * V * M]
    const uint64_t *__restrict__ d_P3,          // [m * M * M]
    const uint64_t *__restrict__ d_all_oil,     // [batch_N * M]
    const uint64_t *__restrict__ d_all_vinegar, // [batch_N * V]
    const uint8_t *__restrict__ d_all_msg,      // [batch_N * m]
    int *d_all_result,                          // [batch_N]
    int batch_N) {
  const int eqn_i = blockIdx.x; // equation index
  const int tid = threadIdx.x;
  const int warp_id = tid / 32;

  for (int sig_base = blockIdx.y * TILE_B; sig_base < batch_N;
       sig_base += gridDim.y * TILE_B) {
    int sigs_in_block = min(TILE_B, batch_N - sig_base);

    const uint64_t *Pi1 = d_Pi1 + (size_t)eqn_i * QRUOV_V * QRUOV_V;
    const uint64_t *Pi2 = d_Pi2 + (size_t)eqn_i * QRUOV_V * QRUOV_M;
    const uint64_t *Pi3 = d_P3 + (size_t)eqn_i * QRUOV_M * QRUOV_M;

    extern __shared__ char dyn_smem[];
    half *smem_A = (half *)dyn_smem;
```

```
    half *smem_B = smem_A + V_SMEM_A_HALF;
    float *smem_C = (float *)(smem_B + V_SMEM_B_HALF);

    char *after = (char *)(smem_C + V_SMEM_C_FLOAT);

    uint64_t *s_vinegar_batch = (uint64_t *)after; // [QRUOV_V * TILE_B]
    uint64_t *s_oil_batch =
        s_vinegar_batch + QRUOV_V * TILE_B;              // [QRUOV_M *
TILE_B]
    uint64_t *s_Pi1_vin = s_oil_batch + QRUOV_M * TILE_B; // [QRUOV_V *
TILE_B]
    uint64_t *s_Pi2_oil = s_Pi1_vin + QRUOV_V * TILE_B;   // [QRUOV_V *
TILE_B]
    uint64_t *s_Pi3_oil = s_Pi2_oil + QRUOV_V * TILE_B;   // [QRUOV_M *
TILE_B]

    for (int idx = tid; idx < QRUOV_V * TILE_B; idx += blockDim.x) {
      int v = idx / TILE_B, s = idx % TILE_B;
      if (s < sigs_in_block)
        s_vinegar_batch[idx] = d_all_vinegar[(sig_base + s) * QRUOV_V + v];
      else
        s_vinegar_batch[idx] = 0;
    }

    for (int idx = tid; idx < QRUOV_M * TILE_B; idx += blockDim.x) {
      int m = idx / TILE_B, s = idx % TILE_B;
      if (s < sigs_in_block)
        s_oil_batch[idx] = d_all_oil[(sig_base + s) * QRUOV_M + m];
      else
        s_oil_batch[idx] = 0;
    }
    __syncthreads();

    wmma_fql_gemm_verify(Pi1, QRUOV_V, QRUOV_V, QRUOV_V, s_vinegar_batch,
                         QRUOV_V, sigs_in_block, TILE_B, s_Pi1_vin, TILE_B,
                         V_PAD, V_PAD, TILE_B, smem_A, smem_B, smem_C, tid,
                         warp_id);

    wmma_fql_gemm_verify(Pi2, QRUOV_V, QRUOV_M, QRUOV_M, s_oil_batch, QRUOV_M,
                         sigs_in_block, TILE_B, s_Pi2_oil, TILE_B, V_PAD,
M_PAD,
                         TILE_B, smem_A, smem_B, smem_C, tid, warp_id);

    for (int s = 0; s < sigs_in_block; s++) {
      if (tid < QRUOV_M) {
        uint32_t acc0 = 0, acc1 = 0, acc2 = 0;
        for (int k = 0; k < QRUOV_M; k++)
```

```
          fql_mac(acc0, acc1, acc2, Pi3[tid * QRUOV_M + k],
                  s_oil_batch[k * TILE_B + s]);
        s_Pi3_oil[tid * TILE_B + s] = fql_pack(acc0, acc1, acc2);
      }
    }
    __syncthreads();

    for (int s = 0; s < sigs_in_block; s++) {
      int sig_id = sig_base + s;

      if (tid == 0) {
        uint32_t acc0 = 0, acc1 = 0, acc2 = 0;

        for (int j = 0; j < QRUOV_V; j++) {
          uint64_t pi2_val = s_Pi2_oil[j * TILE_B + s];
          uint32_t p2_0 = (uint32_t)(pi2_val & Fql_wm);
          uint32_t p2_1 = (uint32_t)((pi2_val >> 24) & Fql_wm);
          uint32_t p2_2 = (uint32_t)(pi2_val >> 48);

          uint32_t dp0 = (p2_0 * 2) % QRUOV_q;
          uint32_t dp1 = (p2_1 * 2) % QRUOV_q;
          uint32_t dp2 = (p2_2 * 2) % QRUOV_q;

          uint64_t p1v = s_Pi1_vin[j * TILE_B + s];
          uint32_t pv0 = (uint32_t)(p1v & Fql_wm);
          uint32_t pv1 = (uint32_t)((p1v >> 24) & Fql_wm);
          uint32_t pv2 = (uint32_t)(p1v >> 48);
          uint64_t tmp_v = fql_pack(dp0 + pv0, dp1 + pv1, dp2 + pv2);

          fql_mac(acc0, acc1, acc2, s_vinegar_batch[j * TILE_B + s], tmp_v);
        }

        for (int j = 0; j < QRUOV_M; j++) {
          fql_mac(acc0, acc1, acc2, s_oil_batch[j * TILE_B + s],
                  s_Pi3_oil[j * TILE_B + s]);
        }

        uint8_t t0 = (uint8_t)(acc0 % QRUOV_q);
        const uint8_t *d_msg = d_all_msg + (size_t)sig_id * QRUOV_m;
        int pass = (d_msg[eqn_i] == t0) ? 1 : 0;
        atomicAnd(&d_all_result[sig_id], pass);
      }
      __syncthreads();
    }
  }
}
```

```
#define BENCHMARK_SIZE 30

void QRUOV_Verify_Kernel_Batch(const QRUOV_SEED seed_pk, const QRUOV_P3 P3,
                               const uint8_t message[],
                               const size_t message_length,
                               const QRUOV_SIGNATURE sig) {
  cudaDeviceProp prop;
  cudaGetDeviceProperties(&prop, 0);
  size_t free_mem, total_mem;
  cudaMemGetInfo(&free_mem, &total_mem);
  printf("GPU: %s | Memory: %.0f MB free / %.0f MB total\n\n", prop.name,
         free_mem / 1e6, total_mem / 1e6);

  uint8_t mu[QRUOV_MU_LEN];
  Expand_mu(seed_pk, message, message_length, mu);

  Fq msg[QRUOV_m];
  Hash(mu, sig->r, msg);

  const Fql *vinegar = sig->s;
  const Fql *oil = sig->s + QRUOV_V;

  // Expand_pk (shared)
  uint64_t *d_Pi1, *d_Pi2;
  cudaMalloc(&d_Pi1, (size_t)QRUOV_m * QRUOV_V * QRUOV_V * sizeof(uint64_t));
  cudaMalloc(&d_Pi2, (size_t)QRUOV_m * QRUOV_V * QRUOV_M * sizeof(uint64_t));
  cudaMemset(d_Pi1, 0, (size_t)QRUOV_m * QRUOV_V * QRUOV_V *
sizeof(uint64_t));
  cudaMemset(d_Pi2, 0, (size_t)QRUOV_m * QRUOV_V * QRUOV_M *
sizeof(uint64_t));

  {
    uint8_t *all_r1 = (uint8_t *)malloc((size_t)QRUOV_m * N1);
    uint8_t *all_r2 = (uint8_t *)malloc((size_t)QRUOV_m * N2);

    QRUOV_PRG_CTX *ctx_pk = PRG_init(seed_pk);
    for (int i = 0; i < QRUOV_m; i++) {
      QRUOV_PRG_CTX *ctx1 = PRG_copy(ctx_pk);
      uint8_t t1[QRUOV_tau1];
      RejSampPRG(ctx1, 2 * i, N1, QRUOV_tau1, t1);
      PRG_final(ctx1);
      memcpy(all_r1 + i * N1, t1, N1);

      QRUOV_PRG_CTX *ctx2 = PRG_copy(ctx_pk);
      uint8_t t2[QRUOV_tau2];
      RejSampPRG(ctx2, 2 * i + 1, N2, QRUOV_tau2, t2);
      PRG_final(ctx2);
```

```
    memcpy(all_r2 + i * N2, t2, N2);
  }
  PRG_final(ctx_pk);

  uint8_t *d_r1, *d_r2;
  cudaMalloc(&d_r1, (size_t)QRUOV_m * N1);
  cudaMalloc(&d_r2, (size_t)QRUOV_m * N2);
  cudaMemcpy(d_r1, all_r1, (size_t)QRUOV_m * N1, cudaMemcpyHostToDevice);
  cudaMemcpy(d_r2, all_r2, (size_t)QRUOV_m * N2, cudaMemcpyHostToDevice);

  dim3 gvv((N_UPPER_TRI + 255) / 256, QRUOV_m);
  ExpandSymmetricMatrixVxV_batched_kernel<<<gvv, 256>>>(d_r1, d_Pi1);

  dim3 gvm((N_VXM + 255) / 256, QRUOV_m);
  ExpandMatrixVxM_batched_kernel<<<gvm, 256>>>(d_r2, d_Pi2);

  cudaFree(d_r1);
  cudaFree(d_r2);
  free(all_r1);
  free(all_r2);
}

uint64_t *d_P3;
cudaMalloc(&d_P3, (size_t)QRUOV_m * QRUOV_M * QRUOV_M * sizeof(uint64_t));
cudaMemcpy(d_P3, P3, (size_t)QRUOV_m * QRUOV_M * QRUOV_M * sizeof(uint64_t),
           cudaMemcpyHostToDevice);

for (int bi = 0; bi < BENCHMARK_SIZE; bi++) {
  int batch_N = 1 << bi;

  // Allow 25% GPU VRAM for buffer
  size_t per_sig_mem = (size_t)QRUOV_V * sizeof(uint64_t)   // d_all_vinegar
                     + (size_t)QRUOV_M * sizeof(uint64_t) // d_all_oil
                     + (size_t)QRUOV_m * sizeof(uint8_t)  // d_all_msg
                     + (size_t)sizeof(int);               // d_all_result
  size_t needed = (size_t)batch_N * per_sig_mem;
  cudaMemGetInfo(&free_mem, &total_mem);
  if (needed > free_mem * 0.75) {
    printf("%-12d | Skipped — need %.0f MB, only %.0f MB free\n", batch_N,
           needed / 1e6, free_mem / 1e6);
    break;
  }

  uint64_t *h_all_vinegar =
      (uint64_t *)malloc((size_t)batch_N * QRUOV_V * sizeof(uint64_t));
  uint64_t *h_all_oil =
      (uint64_t *)malloc((size_t)batch_N * QRUOV_M * sizeof(uint64_t));
```

```
    uint8_t *h_all_msg =
        (uint8_t *)malloc((size_t)batch_N * QRUOV_m * sizeof(uint8_t));
    int *h_all_result = (int *)malloc((size_t)batch_N * sizeof(int));

    for (int n = 0; n < batch_N; n++) {
      memcpy(h_all_vinegar + n * QRUOV_V, vinegar, QRUOV_V *
sizeof(uint64_t));
      memcpy(h_all_oil + n * QRUOV_M, oil, QRUOV_M * sizeof(uint64_t));
      memcpy(h_all_msg + n * QRUOV_m, msg, QRUOV_m * sizeof(uint8_t));
      h_all_result[n] = 1;
    }

    uint64_t *d_all_vinegar, *d_all_oil;
    uint8_t *d_all_msg;
    int *d_all_result;

    cudaMalloc(&d_all_vinegar, (size_t)batch_N * QRUOV_V * sizeof(uint64_t));
    cudaMalloc(&d_all_oil, (size_t)batch_N * QRUOV_M * sizeof(uint64_t));
    cudaMalloc(&d_all_msg, (size_t)batch_N * QRUOV_m * sizeof(uint8_t));
    cudaMalloc(&d_all_result, (size_t)batch_N * sizeof(int));

    cudaMemcpy(d_all_vinegar, h_all_vinegar,
               (size_t)batch_N * QRUOV_V * sizeof(uint64_t),
               cudaMemcpyHostToDevice);
    cudaMemcpy(d_all_oil, h_all_oil,
               (size_t)batch_N * QRUOV_M * sizeof(uint64_t),
               cudaMemcpyHostToDevice);
    cudaMemcpy(d_all_msg, h_all_msg,
               (size_t)batch_N * QRUOV_m * sizeof(uint8_t),
               cudaMemcpyHostToDevice);
    cudaMemcpy(d_all_result, h_all_result, (size_t)batch_N * sizeof(int),
               cudaMemcpyHostToDevice);

    size_t verify_tensor_smem =
        V_TENSOR_SMEM_BYTES +
        (QRUOV_V * TILE_B * 3 + QRUOV_M * TILE_B * 2) * sizeof(uint64_t);
    cudaFuncSetAttribute(VERIFY_tensor_batched_kernel,
                         cudaFuncAttributeMaxDynamicSharedMemorySize,
                         (int)verify_tensor_smem);

    int grid_y = ((batch_N + TILE_B - 1) / TILE_B);
    if (grid_y > 65535)
      grid_y = 65535;
    dim3 grid(QRUOV_m, grid_y);
    VERIFY_tensor_batched_kernel<<<grid, 128, verify_tensor_smem>>>(
        d_Pi1, d_Pi2, d_P3, d_all_oil, d_all_vinegar, d_all_msg, d_all_result,
        batch_N);
```

```
    cudaMemcpy(h_all_result, d_all_result, (size_t)batch_N * sizeof(int),
               cudaMemcpyDeviceToHost);

    int all_pass = 1;
    for (int n = 0; n < batch_N; n++) {
      if (!h_all_result[n]) {
        all_pass = 0;
        break;
      }
    }

    double total_ms = (double)gpu_ms + cpu_ms;
    double throughput = (total_ms > 0) ? (batch_N / (total_ms / 1000.0)) : 0;
    printf("%-12d | %14.3f | %14.3f | %14.3f | %18.1f | %10s\n", batch_N,
           (double)gpu_ms, cpu_ms, total_ms, throughput,
           all_pass ? "YES" : "NO");

    free(h_all_vinegar);
    free(h_all_oil);
    free(h_all_msg);
    free(h_all_result);
    cudaFree(d_all_vinegar);
    cudaFree(d_all_oil);
    cudaFree(d_all_msg);
    cudaFree(d_all_result);
  }

  printf("\nDone\n");

  cudaFree(d_Pi1);
  cudaFree(d_Pi2);
  cudaFree(d_P3);
}
```

POSTER

Universiti Tunku Abdul Rahman

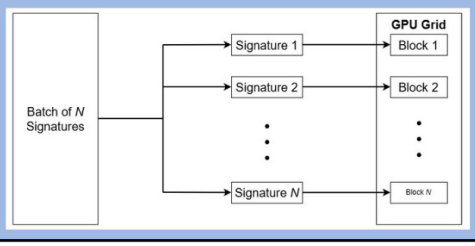
EFFICIENT PARALLEL IMPLEMENTATION OF QRUOV POST-QUANTUM SIGNATURE SCHEME ON A GPU

INTRODUCTION

- **Problem:** QR-UOV offers strong quantum security, but dense matrix operations create severe CPU bottlenecks.
- **Objective:** Achieve ≥ 5x speedup using GPU parallelism while preserving 100% cryptographic exactness.

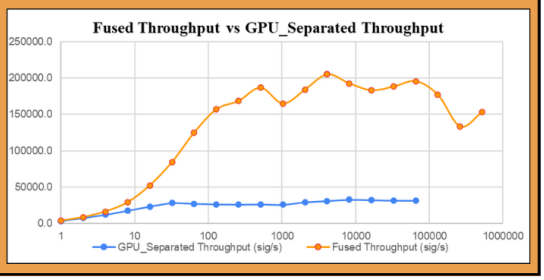
METHODOLOGY

- **Coarse-Grained Batching** (1 Block = 1 Signature)
- **Kernel Fusion & SRAM Tiling** (Bypasses high-latency Global VRAM)
- **Hybrid CPU-GPU Fallback** (resolves rare rank-deficient matrices)



RESULTS

- **Massive Speedup:** Fused GPU architecture reached a peak throughput of **205,189 sig/s**, delivering astounding **242x speedup** over the CPU baseline.



DISCUSSION

- **Memory Wall Defeated:** Kernel fusion yielded a 6.2x speedup over separated GPU kernels by eliminating VRAM read/write cycles.
- **Tensor Cores vs. CUDA:** Standard Fused CUDA cores outperformed WMMA Tensor Cores. FP16 casting and padding overhead negated raw MAC acceleration.
- **Stable Edge-Cases:** Hybrid Fallback resolved singular matrices (~0.7% frequency) on the CPU in <7ms without stalling the GPU pipeline

Final Year Project

Supervisor: Dr Lee Wai Kong
Student : Chai Yau Yuen