EFFICIENT PARALLEL IMPLEMENTATION OF MCELIECE SIGNATURE SCHEME ON GPU PLATFORMS

BY

DESMOND HO JIA SHEN

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)

JUNE 2025

COPYRIGHT STATEMENT

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Lee Wai Kong, for his invaluable guidance, support, and encouragement throughout the course of this project. His expertise, insightful feedback, and patience have been instrumental in helping me overcome challenges and complete this work successfully.

My heartfelt thanks go to my parents, Mr. Ho Kok Seng and Mrs. Tan Mei Lan, for their unwavering love, understanding, and moral support throughout my education journey. Without their sacrifices and constant encouragement, this achievement would not have been possible.

Lastly, I am grateful to my friends and peers who provided encouragement, advice, and companionship throughout this process.

Thank you all.

ABSTRACT

This project focuses on speeding up the McEliece post-quantum cryptosystem, a strong candidate for securing data from quantum attacks. Even though McEliece offers strong security, the computational cost is a significant hindrance for practical usage. To meet this challenge, a GPU accelerated version of the McEliece Key Encapsulation Mechanism (KEM) was created on NVIDIA's CUDA platform. The project parallelized important operations, such as encryption and decryption while ensuring correctness by comparing with an CPU implementation. Computational performance was tested with an NVIDIA GTX 1650, which showed notable improvements for encryption ≈6.7x faster and decryption ≈7.4x faster than running on the CPU configurations. The results show that GPU acceleration improves the practical efficiency and scalability of McEliece cryptography for real world practice. Future work includes asynchronous memory operations, improvements to the theoretical speed of the algorithm, and the investigation of multi-GPU deployments for potential even faster speeds.

Area of Study (Minimum 1 and Maximum 2): Cryptography, GPU Computing

Keywords (Minimum 5 and Maximum 10): McEliece Cryptosystem, GPU Acceleration, Parallel Computing, CUDA Programming, Performance Evaluation

TABLE OF CONTENTS

TITLE PAGE i

COPYRIGHT STATEMENT ii

ACKNOWLEDGEMENTS iii

ABSTRACT iv

TABLE OF CONTENTS v

LIST OF FIGURES viii

LIST OF TABLES x

LIST OF SYMBOLS xi

LIST OF ABBREVIATIONS xii

CHAPTER 1 INTRODUCTION 1

1.1 Problem Statement and Motivation 2

1.2 Objectives 3

1.3 Project Scope and Direction 3

1.4 Contributions 4

1.5 Report Organization 4

CHAPTER 2 LITERATURE REVIEW 6

2.1 Review of the Technologies 6

2.1.1 Hardware Platform 6

2.1.2 Firmware / OS 6

2.1.3 Programming Language 7

2.1.4 Algorithm 7

2.1.5 Summary of the Technologies Review 8

2.2 Review of Efficient Implementation of McEliece Cryptosystem on GPU 9

2.2.1 Summary 9

2.2.2 Strength and Weaknesses 9

2.2.3 Proposed Improvement 11

2.3 Review of Fast Hardware Architecture with Efficient Matrix Computations for Key Generation of Classic McEliece 12
2.3.1 Summary 12
2.3.2 Strength and Weaknesses 12
2.3.3 Proposed Improvement 14
2.4 Review of Optimized and Scalable Co-Processor for McEliece with Binary Gopp 15
2.4.1 Summary 15
2.4.2 Strength and Weaknesses 15
2.4.3 Proposed Improvement 17
2.5 Review of Evaluation of Gaussian Elimination Using HLS for Fast Public Key Generation 18
2.5.1 Summary 18
2.5.2 Strength and Weaknesses 18
2.5.3 Proposed Improvement 19
2.6 Review of Symphony of Speeds: Harmonizing Classic McEliece Cryptography with GPU Innovation 20
2.6.1 Summary 20
2.6.2 Strength and Weaknesses 20
2.6.3 Proposed Improvement 21
2.7 Summary for Literature Review 22

CHAPTER 3 SYSTEM METHODOLOGY/APPROACH (FOR DEVELOPMENT-BASED PROJECT) 23
3.1 System Design Diagram/Equation 23
3.1.1 System Architecture Diagram 24
3.1.2 Use Case Diagram and Description 25
3.1.3 Activity Diagram 26

CHAPTER 4 SYSTEM DESIGN 28
4.1 System Block Diagram 28
4.2 System Components Specifications 30
4.2.1 Utility Functions (Device Functions) 30

4.2.2 Kernel Functions 45
4.2.3 Host Functions 62

CHAPTER 5 SYSTEM IMPLEMENTATION (FOR DEVELOPMENT-BASED PROJECT) 71

5.1 Hardware Setup 71
5.2 Software Setup 71
5.3 Setting and Configuration 72
5.4 System Operation (with Screenshot) 72
5.5 Implementation Issues and Challenges 74
5.6 Concluding Remark 75

CHAPTER 6 SYSTEM EVALUATION AND DISCUSSION 76

6.1 System Testing and Performance Metrics 76
6.2 Testing Setup and Result 76
6.3 Project Challenges 82
6.4 Objectives Evaluation 83
6.5 Concluding Remark 84

CHAPTER 7 CONCLUSION AND RECOMMENDATION 85

7.1 Conclusion 85
7.2 Recommendation 87

REFERENCES 88
APPENDIX A: Benchmark CSV Results A-1
APPENDIX B: Additional Graphs B-1
APPENDIX C: Poster C-1

LIST OF FIGURES

Figure Number	Title	Page
Figure 3.1	System Design Diagram	23
Figure 3.1.1	System Architecture Diagram	24
Figure 3.1.2	Use Case Diagram	25
Figure 3.1.3	Activity Diagram	26
Figure 4.1	System Block Diagram	28
Figure 4.2	load_gf	30
Figure 4.3	crypto_uint16_smaller_mask	31
Figure 4.4	uint16_is_smaller_declassify	32
Figure 4.5	crypto_uint32_equal_mask	32
Figure 4.6	uint32_is_equal_declassify	33
Figure 4.7	store8	33
Figure 4.8	ld64_unaligned	34
Figure 4.9	ld32_unaligned	35
Figure 4.10	atomic_xor_bit	35
Figure 4.11	d_vec_add	37
Figure 4.12	d_transpose_64x64	38
Figure 4.13	d_vec_copy	39
Figure 4.14	d_vec_mull	40
Figure 4.15	d_vec_sq	41
Figure 4.16	d_vec_inv	42
Figure 4.17	d_layer_parallel	43
Figure 4.18	d_broadcast_and_beta	43
Figure 4.19	load_gf_kernel	45
Figure 4.20	filter_valid_indices	45
Figure 4.21	check_duplicates	46
Figure 4.22	compute_val	47
Figure 4.23	set_bits	47
Figure 4.24	bitonic_sort_kernel	48

Figure 4.25	warmup_kernel	49
Figure 4.26	syndrome_kernel_vecstyle	50
Figure 4.27	k_preprocess	51
Figure 4.28	k_postprocess	51
Figure 4.29	k_form_error	52
Figure 4.30	k_and_scale	53
Figure 4.31	k_fft	54
Figure 4.32	k_fft_tr (i)	55
Figure 4.33	k_fft_tr (ii)	55
Figure 4.34	k_fft_tr (iii)	56
Figure 4.35	k_sq_rows	57
Figure 4.36	k_build_inv	58
Figure 4.37	k_benes	59
Figure 4.38	k_synd_cmp	60
Figure 4.39	k_weight_check	61
Figure 4.40	crypto_kem_enc_new	62
Figure 4.41	crypto_kem_dec	63
Figure 4.42	decrypt (i)	64
Figure 4.43	decrypt (ii)	65
Figure 4.44	decrypt (iii)	66
Figure 4.45	decrypt (iv)	67
Figure 4.46	decrypt (v)	68
Figure 4.47	main.cpp	70
Figure 5.1	Encrypt Results	73
Figure 5.2	Decrypt Results & SS comparison	73
Figure 5.3	Sample CSV results	74
Figure 6.1	CPU Encrypt Time vs Throughput	79
Figure 6.2	CPU Decrypt Time vs Throughput	79
Figure 6.3	GPU Encrypt Time	80
Figure 6.4	GPU Encrypt Throughput	80
Figure 6.5	GPU Decrypt Time	81
Figure 6.6	GPU Decrypt Throughput	81

LIST OF TABLES

Table Number	Title	Page
Table 5.1	Specifications of laptop	71
Table 6.1	Testing Setup	76
Table 6.2	CPU encrypt results	77
Table 6.3	CPU decrypt results	77
Table 6.4	GPU encrypt results	78
Table 6.5	GPU decrypt results	78

LIST OF SYMBOLS

β	beta

LIST OF ABBREVIATIONS

RSA	Rivest-Sharmir-Adleman algorithm
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
ECC	Elliptic Curve Crytography
PQC	Post-Quantum Cryptography
GPU	Graphics Processing Units
CUDA	Compute Unified Device Architecture
IoT	Internet of Things
CPU	Central Processing Units
NP	Nondeterministic polynomial time
FPGA	Field-Programmable Gate Array
ASIC	Application-Specific Integrated Circuits
NIST	National Institute of Standards and Technology
HLS	High Level Synthesis
RTL	Registry-Transfer Level
CCA	Canonical Correlation Analysis
TLS	Transport Layer Security
BM	Berlekamp-Massey

Chapter 1
Introduction

The development of exponentially powerful quantum computers poses a direct threat to the cryptographic protocols that underpin secure communication today. Cryptographic algorithms such as RSA[1], DSA[2], and ECC[3], which form the basis of internet security, are vulnerable to Shor's algorithm, which can efficiently solve the integer factorisation and discrete logarithm problems that these algorithms rely upon [4]. This vulnerability has led researchers to search for post-quantum cryptography (PQC), cryptographic schemes designed to be secure against both classical and quantum adversaries.

One of the most promising candidates in the PQC competition is the **McEliece cryptosystem**, proposed by Robert McEliece in 1978 [5]. McEliece's scheme is based on the NP-hard problem of decoding general linear codes, making it secure even against quantum computers. Unlike number-theoretic schemes, which are rendered insecure by quantum attacks, McEliece remains robust. However, the cryptosystem has historically faced two challenges: very large public keys and relatively slower operations compared to classical systems [6].

Classic McEliece has since been standardised as a Key Encapsulation Mechanism (KEM) by NIST, involving three key steps: encapsulation (generation of ciphertext and shared secret), decapsulation (recovery of the shared secret), and an underlying decryption/decoding stage that performs syndrome computation, Patterson decoding, error location, and re-encapsulation for CCA security [7]. Among these steps, decapsulation and decoding are typically the most computationally expensive, which may limit performance in practical deployments.

General-purpose GPU computing offers an opportunity to overcome such performance barriers. GPUs are massively parallel devices capable of running thousands of threads simultaneously. NVIDIA's CUDA platform allows developers to exploit this parallelism for high-throughput computation [8]. GPU acceleration is especially well-suited for parts of the McEliece scheme such as syndrome computation, error vector generation, and finite-field arithmetic.

This project focuses on leveraging CUDA to accelerate not only encryption, as done in FYP1, but also encapsulation, decryption verification, and decapsulation, completing the KEM workflow on GPU. By comparing GPU performance against CPU baselines and analysing correctness at multiple levels (error positions, syndromes, shared secrets), this work aims to demonstrate both speedup and reliability, thereby contributing towards making McEliece a more practical choice for post-quantum secure communication.

1.1 Problem Statement and Motivation

Despite its proven security, Classic McEliece's relatively high computational cost remains a barrier to adoption. The decryption and decapsulation processes are especially heavy, involving syndrome computation, root solving over GF(2^m), and error vector reconstruction, which dominate runtime. This is a problem for scenarios where many sessions must be handled per second, such as TLS handshakes in servers or high-volume VPNs.

Parallelising these operations on GPU offers a promising solution. By distributing syndrome computation and error processing across thousands of threads, GPUs can reduce latency and improve throughput. However, designing a correct and efficient GPU implementation is non-trivial: issues such as thread divergence, memory layout efficiency, and synchronization must be addressed carefully. Moreover, correctness must be strictly maintained: even a single bit error can result in an incorrect shared secret, breaking communication. This project is motivated by the need to explore GPU acceleration for the entire McEliece KEM path, not just encryption, to make post-quantum secure systems more practical and efficient.

This FYP2 project builds on the existing GPU-accelerated encryption work from FYP1. Its aim is to complete the KEM workflow on GPU—that is, to implement encapsulation, decryption verification, and decapsulation—using the vec implementation as reference. It focuses on *mceliece348864*, with the goal of measuring performance under different GPU block configurations, comparing to CPU baseline runs, and ensuring correctness at multiple levels of output.

1.2 Objectives

The goals of this project are:

- To implement CUDA versions of the Encapsulation and Decapsulation operations of Classic McEliece KEM for *mceliece348864*, as well as a Decrypt path to reproduce error positions and syndromes as produced by the vec CPU reference.
- To verify correctness by comparing, for each tested input, the GPU outputs to the vec CPU outputs in terms of error vector positions, syndrome values, and final shared secret. Any discrepancy must be identified and corrected.
- To benchmark performance by varying the number of GPU blocks (for example: 1, 2, 4, 8, 16, 32), measuring average execution time (in milliseconds) and throughput (items per second), and to compare the best GPU configuration to the CPU baseline (which uses serial looping sized to match GPU block count for fairness).
- To document implementation challenges, trade-offs, and practical decisions made in the process, so as to provide recommendations for future improvements (e.g., full batching, optimizing memory usage, exploring other decoding algorithms, or other parameter sets).

1.3 Project Scope and Direction

The scope of this work is as follows:

- **Cryptographic setting:** Classic McEliece KEM with parameter set *mceliece348864*. The vec implementation will serve as ground truth for CPU reference.
- **Implementation targets:** GPU implementation in CUDA for Encapsulation, Decryption verification, and Decapsulation. CPU runs (vec) for comparison and correctness checks.
- **Performance metrics:** Average time per operation, throughput in items/sec, variation across numbers of blocks, and comparison of GPU vs CPU.
- **Correctness metrics:** Matching error positions and syndrome computed via GPU vs vec CPU; matching final shared secret; noting any discrepancies.
- **Full Completion Focus**: For the second phase (FYP2), the encapsulation, decryption, decapsulation modules is implemented and tested to complete the full KEM pipeline for the McEliece cryptosystem.

1.4 Contributions

This project makes several concrete contributions over prior work:

First, it extends GPU acceleration beyond encryption (as done in FYP1) to include Encapsulation and Decapsulation, along with a Decrypt verification path, all for *mceliece348864*. This means that the full KEM cycle is implemented and evaluated on GPU.

Second, it ensures rigorous correctness validation. Several levels of output (error vector positions, syndromes, shared secrets) are compared between GPU and vec CPU reference implementations. This gives confidence that the GPU implementation is functionally equivalent to the established CPU reference.

Third, it provides a detailed performance study. By measuring time and throughput across multiple GPU block configurations, and comparing against CPU baseline runs, it identifies the best GPU configurations and quantifies speedups under realistic settings.

Fourth, it surfaces and documents engineering challenges, especially in the decoding and transpose kernels (for example, memory usage of k_fft_tr, synchronization and divergence issues in syndrome calculation), and explains the practical trade-offs made in implementation under time and hardware constraints.

Thus, this work bridges the gap between encryption-only GPU proofs of concept and a full KEM implementation ready for more rigorous deployment or further extension.

1.5 Report Organization

The report is organized to guide the reader from background theory through design, implementation, to evaluation and conclusions. Chapter 2 reviews relevant background in coding theory, the Classic McEliece KEM spec, and related work in hardware/software acceleration. Chapter 3 describes the methodology and system approach: how the KEM operations are mapped onto GPU, how correctness is validated, and which files and functions implement key parts of the system. Chapter 4 provides detailed design: block diagrams, component specifications, memory layout, important kernels (like syndrome computation,

error vector handling, transpose k_fft_tr), and dataflow between host and GPU. Chapter 5 covers the implementation environment, setup, configuration, screenshots, debugging and issues encountered. Chapter 6 presents evaluation results: time vs blocks, throughput vs blocks, comparison with CPU baseline, analysis of where performance gains or bottlenecks occur, and discussion of how well the objectives are met. Finally, Chapter 7 concludes by summarizing findings, noting limitations, and recommending future work (such as better GPU batching, exploring other parameter sets, or improving memory- and synchronization optimization).

Chapter 2
Literature Review

2.1 Review of the technologies

2.1.1 Hardware Platform

This project utilizes an NVIDIA GeForce GTX 1650 GPU with 4 GB of device memory as the main hardware platform. NVIDIA GPUs are among the best for general-purpose parallel computing owing to their massive number of CUDA cores, high memory bandwidth, and capability for efficient context switching between many threads. Naturally, this is complemented by the CPU host, a multi-core AMD Ryzen 5 3550H, which handles the control flow, data transfers, and non-parallelizable portions. In GPU-accelerated workloads, mainly cryptographic algorithms that involve many small data transfers or irregular memory access, critical considerations are the GPU memory (global, shared), and, of course, the PCIe link established between host and device.

2.1.2 Operating System / Firmware

Ubuntu Linux is used as the operating system for development. Linux distributions are commonly chosen for GPU programming because they offer better driver support, more predictable kernel scheduling, extensive toolchain compatibility (gcc, nvcc), and better performance consistency than some alternative OSes. Installer scripts for CUDA toolkit, driver compatibility, kernel module support, and version-coherence (driver vs toolkit vs GCC version) are typically smoother in Ubuntu. According to NVIDIA's CUDA Installation Guide for Linux, a supported GCC compiler toolchain and supported GPU drivers are required prerequisites [9].

2.1.3 Programming Language

The implementation is carried out in C with CUDA extensions ("CUDA C/C++"). CUDA is NVIDIA's parallel computing model that provides APIs and language extensions allowing developers to write kernels that run on the GPU. Key features include:

- Kernel launches with many threads organized in grids and blocks.
- Different memory spaces: global, shared, constant, registers. Efficient use of shared memory and avoidance of global memory bottlenecks are essential.
- Synchronization primitives: __syncthreads() inside blocks; global synchronisation only via host control (kernel boundaries).

The official CUDA C++ Programming Guide emphasizes performance practices such as maximizing parallel execution, minimizing data transfers between host and device, and configuring memory access patterns to reduce latency [10].

2.1.4 Algorithm: McEliece KEM

McEliece cryptosystem is a public-key encryption scheme which has been adapted as a KEM. It is based on binary Goppa codes: error-correcting codes defined over finite fields GF(2^m) with certain properties that make them hard to decode when parameters are chosen appropriately. The main algorithmic components include:

- **Key Generation**: Generate a random irreducible binary Goppa polynomial g(*z*) of degree *t* over GF(2^m), select *n* support elements in GF(2^m), compute the Goppa code and associated public key (parity or generator matrix), and keep the private key (which includes the Goppa polynomial, support set, possibly some transform matrices).
- **Encapsulation (Encrypt)**: A random error vector *e* of weight *t* is sampled, then encoded using the public matrix to produce a syndrome or ciphertext; a shared secret is derived by hashing *e* and/or other elements.

- **Decapsulation (Decrypt)**: Given a ciphertext, the private key is used to compute the syndrome, then the decoding algorithm locates the error vector, recovers *e*, and the shared secret is re-derived. In a KEM setting, there is also a re-encapsulation check (or an equivalent mechanism) to ensure ciphertext integrity under chosen-ciphertext attacks.

Decoding using Patterson's algorithm involves operations in GF(2^m): computing error locator polynomials, solving for roots, etc. The security of Classic McEliece derives from the difficulty of decoding a general linear code when only the public key is known [11]. Recent work such as *Understanding binary-Goppa decoding* provides updated analyses of the algorithm and its complexity [12].

2.1.5 Summary of the Technologies Review

To summarize, this FYP2 project has several foundations. First, the hardware platform, GTX 1650 GPU and a capable CPU host, provides an example of a parallelism and memory architecture that should prove useful in achieving the acceleration of arithmetic and bitwise operations. Second, the software environment, Ubuntu Linux + CUDA C/C++, is an example of a high-throughput computing-ready software stack where stable drivers, toolchains, and programming abstraction coexist. Third, while it is well understood in order to act as a benchmarking reference, particularly the decapsulation part, which is computationally intense and capable of GPU acceleration, there is still a lot to do with the Classic McEliece KEM algorithm concerning *mceliece348864*, error vector sampling, etc.

2.2 Review of Efficient Implementation of McEliece Cryptosystem on GPU

2.2.1 Summary

Elsobky, Ibrahim, and Abd El-Latif [13] describe the acceleration of McEliece public key cryptography through General Purpose GPU (GPGPU) processing techniques. In their results, it is quoted that "utilizing vector data-type with GPU local memory achieves the most efficient implementation of 331x faster than CPU implementation when encrypting 216 messages" [13]. Recognizing that although McEliece provides decent post-quantum security, it is plagued by gigantic key sizes and computationally expensive operations, the authors attempt to enhance its practical viability by taking advantage of the parallelism provided by CUDA-enabled GPUs. Specifically, they attempt to parallelize matrix-vector multiplications, one of the significant bottlenecks in McEliece decryption and encryption.

Through CUDA programming on NVIDIA GPUs, the work effectively offloads the cryptographic operations onto tens of thousands of lightweight GPU threads. Optimizing memory access patterns, for example, by promoting coalesced global memory access, and reducing branch divergence to achieve maximal thread efficiency are also stressed by the authors. Their measurement results indicate that offloading McEliece encryption and decryption onto the GPU provides large speedups relative to a CPU-only solution, validating the applicability of parallelizing conventionally irregular and memory-intensive cryptographic workloads.

This initial effort to port McEliece to GPU platforms delivered a significant proof-of-concept that large-scale code-based cryptographic systems could be effectively run on massively parallel architectures [13].

2.2.2 Strengths and Weaknesses

One of the strongest points of the paper is the clear and practical proof that GPUs are not restricted to floating-point-intensive scientific simulations but can be utilized for discrete, memory-bound applications such as public-key cryptography as well. The authors nicely employ CUDA concepts like memory coalescing, reducing shared memory bank conflicts, and thread divergence reduction, all of which are essential in order to achieve high throughput on GPU architectures. Their empirical results, with measurable speedups over their CPU

equivalents, illustrate the practical importance of their work for accelerating post-quantum secure systems.

In addition, the paper presents valuable insights into common issues encountered in the offloading of cryptographic functions to GPUs, such as performance losses due to non-coalesced memory access patterns and excessive synchronization. This appreciation encourages existing literature for future researchers seeking to undertake similar implementations. Furthermore, the paper highlights the fact that significant acceleration is possible even without cutting-edge GPUs, thus widening the practicable reach to older or typical hardware setups.

In spite of these advantages, there are several limitations that restrict the effectiveness and usefulness of the study. Firstly, the researchers concentrate solely on the encryption and decryption procedure of McEliece and neglect to include the important elements of key generation and key encapsulation, which are essential considerations for full cryptographic application. The lack of GPU acceleration for key generation is a significant omission, as in real-world implementations of McEliece, key generation, which typically entails large-scale matrix operations and Gaussian elimination, is as crucial, if not more crucial, for performance.

Also, the experimental arrangement in the research only uses comparitively older GPU hardware. Given the rapid evolution of GPU hardware with new features such as Tensor Cores, Warp Shuffle Instructions, and Asynchronous Copy introduced in later releases such as Volta and Ampere, the findings may not apply to modern devices. Modern optimisation techniques such as warp-level parallelism, kernel fusion, and the use of CUDA streams to render data transfers concurrent with computation were also not explored. These omissions somewhat diminish the paper's relevance to modern high-performance cryptographic implementations.

Finally, a significant omission is the complete lack of consideration of the security implications of GPU acceleration. Although GPUs can bring massive computational benefits, they are also susceptible to side-channel attacks due to shared resource contention and timing variability. Not considering these threats at all calls into question the suitability of GPU-accelerated McEliece systems for high-assurance environments [14].

2.2.3 Proposed Improvement

Although Elsobky et al. [13] did considerable work in accelerating McEliece through GPU, various enhancements can be proposed keeping pace with cutting-edge technologies and trends in research work. For starters, modern GPU architectures have evolved complex features that could be harnessed in further accelerating matrix operations core to McEliece. Techniques such as Tensor Core utilization for fast matrix multiplication, warp-level intrinsics to execute reductions faster, and async memory prefetching could greatly improve performance over the technique in the paper [14].

Second, a complete system implementation needs to include key generation, key encapsulation, and signature generation modules, not only encryption and decryption. Accelerating the entire lifecycle of McEliece will make the cryptosystem more feasible for real application, particularly in post-quantum secure communication.

Another area that needs to be improved is reducing synchronization overhead and concurrency optimization using CUDA streams. By enabling several encryption operations or memory transfers to be overlapped, considerable latency savings can be realized, particularly for batch processing applications.

Lastly, security cannot be an afterthought. Next-generation GPU-accelerated McEliece implementations must have a side-channel risk analysis, such as vulnerability to timing attack, memory access pattern leakage, and GPU resource contention exploit threats [15]. Implementations must adopt constant-time programming practices, randomized memory access patterns, and potentially even make use of isolation features available on modern GPUs.

2.3 Review of Fast Hardware Architecture with Efficient Matrix Computations for Key Generation of Classic McEliece

2.3.1 Summary

The paper of Zhang et al. [16] presents a novel hardware-friendly approach to speeding up the key generation of the Classic McEliece cryptosystem, a fundamental post-quantum secure candidate. By observing that McEliece key generation is computationally intensive and very reliant on complex matrix operations such as Gaussian elimination and random error generation, the authors propose a bespoke hardware architecture capable of efficiently handling such operations. “Our optimized design for timing and memory access, along with the architecture of two systolic arrays in the systemizer enhance the performance of the Gaussian elimation process.”[16]

Their architecture is directed towards accelerating matrix inversion and multiplication through customized datapaths and memory hierarchies with significantly lower latency compared to software implementations. Pipelined Gaussian elimination modules, optimized memory controllers for fetching data at high speed, and parallelizable structures with low bottlenecks in matrix operations are a few of the salient features of their architecture.

Experimental results on FPGA platforms show remarkable reductions in key generation time with reasonable hardware resource consumption. The study makes a valuable contribution by deflecting attention to hardware acceleration for McEliece, which is a less explored direction compared to encryption and decryption acceleration [16].

2.3.2 Strengths and Weaknesses

The paper's strength is that it focuses on key generation, a process which traditionally has been an overlooked area of McEliece optimization study. Although previous work tended to focus on making encryption and decryption faster, Zhang et al. [16] are correct to assert that key generation is similarly critical as a bottleneck, especially with the increasing need for quick, repeated key refreshing with current security standards.

The suggested architecture design in the paper is highly modular and scalable, providing efficient resource sharing among different matrix computation blocks. Their pipelining of

Gaussian elimination guarantees high performance even for bigger matrices, which is critical in view of the parameter sets proposed for post-quantum security levels.

Furthermore, the experimental implementation of the paper on FPGA platforms demonstrates real-world feasibility over simulation or theoretical modeling alone. This focus on actual hardware deployment adds more validity to their claims and provides concrete benchmarks for future research in the area.

Although these are important strengths, there are important weaknesses. Firstly, the architecture is validated primarily on FPGA rather than ASIC or GPU platforms. Although FPGAs offer the advantages of fast prototyping, they do not necessarily represent the final performance bounds that can be achieved using specialized hardware or multi-processors like GPUs. This limits the external validity of the reported performance results.

Second, the proposed solution is nearly entirely reliant on having a fixed and optimized datapath structure for specific matrix sizes. This may be rigid in accommodating other security parameters (like moving from 2^11 to 2^12 Goppa codes) or other code families that are not binary Goppa structures. GPU-based architectures are more runtime adaptable, with their matrices dynamically scaled without requiring reconfiguration.

Another limitation is little discussion of optimization of memory access in high-loaded scenarios. There is a discussion of effective memory controllers, though little analysis of how memory bandwidth limitations can themselves be bottlenecks in terms of performance as matrix sizes are increased for increased security levels.

2.3.3 Proposed Improvement

Building on the work of Zhang et al. [16], several enhancements may be included to further optimize McEliece key generation performance for better flexibility and scalability.

First, exploring GPU acceleration of key generation would offer a trade-off between high-speed custom hardware and flexible software solutions. Current CUDA-enabled GPUs have native matrix computation libraries, warp shuffle instructions for reduction operations, and large banks of shared memory that would significantly accelerate Gaussian elimination without fixed architecture constraints of FPGA implementations.

Second, a hybrid architecture approach could be employed to further boost performance. For instance, the most performance-critical parts of key generation, such as error generation and initial matrix generation, could be offloaded to special-purpose FPGA or ASIC modules, while the bulk matrix operations (e.g., Gaussian elimination, matrix inversion) are parallelized and run on GPUs. In a hybrid model like this, the best trade-offs between speed, flexibility, and power efficiency could be achieved.

Third, future designs must feature full memory hierarchy optimizations. Memory tiling, double buffering, and bank conflict elimination are methods that must become standard in high-performance matrix computation modules, especially in scaling to large parameters.

So, while the hardware architecture of Zhang et al. [16] is a helpful step towards McEliece acceleration, future systems must be more flexible and scalable to truly meet the demands of next-generation cryptographic applications.

2.4 Review of Optimized and Scalable Co-Processor for McEliece with Binary Goppa Codes

2.4.1 Summary

Massolino et al. [17] proposes a planned, efficient, and scalable co-processor with the sole aim to improve McEliece cryptosystem on binary Goppa codes. Its main concern is to address two major issues: to reduce the extremely expensive computation regarding McEliece cryptography operation and futureproofing according to different sets of parameters so that it would be secure in the scenario of the emerging need of security in the future.

The authors suggest a novel hardware architecture implemented on Field-Programmable Gate Arrays (FPGAs) targeted towards parallelization of matrix operations and Goppa polynomial evaluation, which are performance bottlenecks in McEliece. Among the design innovations of their work is the modularity of the hardware blocks in such a way that they can be extended easily without having to redesign extensively while porting to other sizes of Goppa codes.

Massolino et al. [17] also stated that "with binary Goppa codes, , it is one of the fastest postquantum decryption proposals, which, with encryption, would result in one of the fastest postquantum asymmetric schemes in hardware". Their deployment accomplishes significant throughput gain over traditional CPU-based computation and emphasizes the importance of creating proprietary hardware accelerators for specific cryptographic primitives [17].

2.4.2 Strengths and Weaknesses

One of the paper's strongest points is its focus on scalability, which was ignored by most prior hardware acceleration attempts on McEliece. Rather than locking in a parameter set and optimizing an inflexible accelerator for it, Massolino et al. [17] anticipated the dynamic future of cryptographic standards, particularly with the PQC ecosystem envisioned by NIST. By making it easy for their co-processor to scale to different key sizes and security levels, their design is a model of forward-thinking engineering.

Another key strength is the degree of optimization achieved on the Goppa polynomial evaluation, which is a computationally intensive step in decoding operations. The authors use

pipelined architectures and low-latency finite field arithmetic units, achieving area and speed efficiency. Moreover, their practical FPGA implementation results provide concrete evidence of feasibility in practice, bridging the gap between theory and hardware implementation.

However, despite these concrete contributions, the paper has several limitations. The primary limitation is that the solution is heavily coupled with FPGA platforms. While FPGAs offer a rapid prototyping environment and some deployment possibilities, they are not always the ideal choice for high-throughput production environments where GPUs or ASICs dominate.

A second shortcoming is the absence of discussion on thermal profiles and energy efficiency. As cryptographic systems become increasingly incorporated into mobile devices and IoT nodes, understanding the energy overhead of hardware acceleration becomes essential. Throughput matters, but power-performance trade-offs are equally pertinent to real-world deployment, especially in resource-constrained environments.

Additionally, the work focuses heavily on decryption acceleration, with relatively less on key generation and encryption optimizations. Although decryption is computationally more demanding in McEliece, an optimization of the overall system would require effort in all stages of cryptography, especially for implementation in systems requiring frequent key refresh or ephemeral keys.

2.4.3 Proposed Improvement

Building on the efforts of Massolino et al. [17], it would be feasible to look to future gains through the extension beyond FPGA platforms to heterogeneous computing frameworks that make use of both CPUs, GPUs, and FPGAs together. Existing CUDA-supported GPUs, with Tensor Cores and enormous memory bandwidths, can be made to accelerate the big matrix multiplications and syndrome computations utilized in McEliece key operations without restriction to fixed hardware datapaths [18].

Moreover, an energy-aware design paradigm must be followed. Future co-processor designs must not only take performance into account in terms of operations per second but also energy consumption per key processed or message decrypted. Dynamic voltage and frequency scaling (DVFS) and fine-grained clock gating can accomplish this.

Another area of improvement is security-hardening. With side-channel attacks emerging as a rising threat in hardware cryptography, especially on reconfigurable platforms like FPGAs, countermeasures such as masking, shuffling, and constant-time execution must be integrated into scalable designs of the future [19].

Finally, acceleration of the entire McEliece system from encryption to decryption and key generation needs to be a goal. Optimization of all of McEliece's lifecycle stages, not a single stage, will enable hardware accelerators to efficiently facilitate realistic deployment scenarios where flexibility and adaptability are critical.

Thus, while Massolino et al.'s work [17] offers a solid foundation to achieve accelerated scalability of McEliece, more flexibility, power portability over platforms, energy efficiency, and higher security elements will be imperative for designs becoming future proof.

2.5 Review of Evaluation of Gaussian Elimination Using HLS for Fast Public Key Generation

2.5.1 Summary

The research of Kihara et al. [20] is focused on accelerating the public key generation process of the Classic McEliece cryptosystem by optimizing Gaussian elimination — one of the most intensive parts of the key generation process. The authors, recognizing that key generation involves huge, dense matrix operations in binary fields, therefore "using HLS tools, the Gaussain elimination step in key generation can be significantly sped up while preserving design flexibility." [20]

Their approach is to execute Gaussian elimination in hardware with an HLS toolchain, allowing algorithmic specification in high-level programming language like C/C++ to be synthesised directly into FPGA hardware. By exploiting parallelism inherent in row operations and memory access optimisation, they are able to realise significantly lower overall latency for the key generation process compared to pure software implementations.

Experimental performance figures from FPGA prototypes demonstrate that substantial advances in speeds of key generation are achievable at no loss of flexibility and that hardware-enabling techniques can make McEliece considerably more practicable for everyday post-quantum cryptographic deployment [20].

2.5.2 Strengths and Weaknesses

One major strength of the paper is its focus on accelerating the public key generation phase, which is often overlooked compared to encryption and decryption. By optimizing Gaussian elimination using High-Level Synthesis (HLS) tools, Kihara et al. [20] effectively simplified hardware development while achieving significant reductions in key generation latency. Their use of FPGA prototypes further demonstrates real-world feasibility, providing credible experimental results.

However, there are notable limitations. While HLS accelerates design, it may not produce circuits as efficient as manually optimized RTL implementations. Additionally, the acceleration effort is narrowly focused only on Gaussian elimination, without addressing other

key generation steps such as Goppa polynomial generation or public matrix encoding. This leaves room for hidden bottlenecks. Moreover, “FPGAs are now powerful computing devices in their own right, suitable for use as fine-grained accelerators” [21]. Finally, no energy efficiency or power analysis is provided, an increasingly critical factor for mobile and embedded cryptographic applications.

2.5.3 Proposed Improvement

To improve on Kihara et al.'s work [20], designs following theirs must accelerate the entire key generation process as a whole, and not Gaussian elimination itself, in order to avoid bottleneck displacement. Investigation into GPU acceleration using CUDA for matrix computations can bring improved scalability and adaptability with larger security parameters [21]. Energy profiling should also be incorporated such that great performance comes at no cost of impractical power consumption, especially for mobile post-quantum devices.

2.6 Review of Symphony of Speeds: Harmonizing Classic McEliece Cryptography with GPU Innovation

2.6.1 Summary

According to [22], the first systematic high-performance implementation of the Classic McEliece KEM on NVIDIA GPUs was presented, using a heterogeneous CPU–GPU approach with a kernel fusion strategy to reduce global memory accesses. Their work targets all five NIST parameter sets and optimizes both encapsulation and decapsulation, achieving **28.6 M** ops/s encapsulation and **3.05 M** ops/s decapsulation throughput for *mceliece348864* [22].

A key focus is on optimizing Transpose Additive FFT (TAFFT) and Additive FFT (AFFT) , the two most time-consuming operations in decapsulation — by introducing the Memory Access Stepping Strategy (MASS) and the Layer-Fused MASS (LFMASS). MASS reorganizes memory traversal to improve cache hit rates, while LFMASS reduces code complexity by fusing computation layers [22]. Their experiments show that the "1|2|4" LFMASS scheme yields **15.9 %** improvement in decapsulation throughput for *mceliece8192128,* while the "4|1|1" scheme yields about **9.5 %** improvement for AFFT [22].

Extensive benchmarking on an NVIDIA RTX 4090 demonstrates that their GPU implementation achieves up to **344× faster encapsulation** and **125× faster decapsulation** than the official AVX CPU implementation, outperforming prior ARM Cortex-M4 and FPGA implementations [22].

2.6.2 Strengths and Weaknesses

The work of Wen Wu et al. [22] is significant because it fills a major gap in post-quantum cryptographic research by providing the first end-to-end GPU implementation of Classic McEliece. Their choice of a coarse-grained parallelism model, where each thread processes a complete KEM operation, is well-motivated as it minimizes synchronization overhead and is aligned with maximizing throughput [22]. The reported performance gains, reaching several orders of magnitude over the CPU baseline, demonstrate the feasibility of using GPUs to accelerate McEliece for high-volume applications such as TLS key exchanges. Furthermore, the introduction of MASS and LFMASS offers algorithmic insights that are applicable beyond

GPU platforms, suggesting that similar optimizations could be implemented on CPUs or hardware accelerators.

Nevertheless, several limitations of the paper are noteworthy. The results are obtained using a high-end NVIDIA RTX 4090 GPU with 24 GB of GDDR6X memory, which may not be representative of consumer-grade GPUs or resource-constrained devices. The paper provides limited quantitative discussion on memory footprint per thread or register pressure, which are key factors for occupancy tuning on smaller devices. While the authors focus extensively on maximizing throughput, per-operation latency measurements receive less attention, which could be important in real-time or latency-sensitive applications. Additionally, although they compare against an AVX-optimized CPU baseline, the absence of experiments using multi-threaded batching on CPUs may overstate the relative advantage of the GPU implementation [22].

2.6.3 Proposed Improvement

Future research could extend this work by conducting experiments on mid-range and entry-level GPUs, such as the GTX 1650 or RTX 3060, to better understand how the proposed optimizations scale across different hardware tiers. Profiling of register usage, shared memory allocation, and warp occupancy could provide further insights for tuning kernel launch parameters to maximize efficiency on less powerful GPUs. Another potential improvement would be exploring the use of mixed-precision computation or specialized hardware features such as Tensor Cores to further accelerate AFFT and TAFFT without sacrificing correctness. Moreover, a detailed analysis of latency–throughput trade-offs and batching strategies would allow the implementation to be tailored for different deployment scenarios, from high-volume servers to low-latency client devices. Finally, incorporating constant-time programming techniques and side-channel resistance considerations, such as mitigating timing and memory-coalescing leaks on GPU architectures, would strengthen the security posture of the implementation, which was not explicitly addressed in [22].

2.7 Summary for Literature Review

The literature review shows that while recent works have achieved huge gains in the performance of Classic McEliece operations, most efforts so far have only concerned isolated phases such as key generation or encryption. This narrow view leaves room for further global optimization of the system, which may be hampered by rigid design choices, poor portability across hardware, etc. Recent works have filled some of these truly absent gaps, particularly in offering high-performance implementations exploiting hardware parallelism- especially on GPUs. For example, Wen Wu et al. [22] proved that coarse-grained GPU parallelism, in combination with kernel fusion and access optimization, yields speedup factors of about 344× for encapsulation and 125× for decapsulation relative to CPU baseline implementations. In that respect, full KEM could seemingly run on modern GPUs, but all their results were derived using an up-to-date RTX 4090, leaving some possible questions concerning scaling down to smaller and more accessible GPUs. All this underlined the need for an implementation to be flexible and, at the same time, reproducible across a more extensive range of devices and phases than just encryption and decryption but encapsulation and decapsulation as well, considering the entire system. The present project tries to answer that by implementing and benchmarking encapsulation, decapsulation, and decryption verification on a mid-range GPU, GTX 1650. Hence, performance trends under varying block counts are laid down for a base from which further, more specific GPU optimizations can follow.

Chapter 3

System Methodology/Approach

3.1 System Design Diagram/Equation

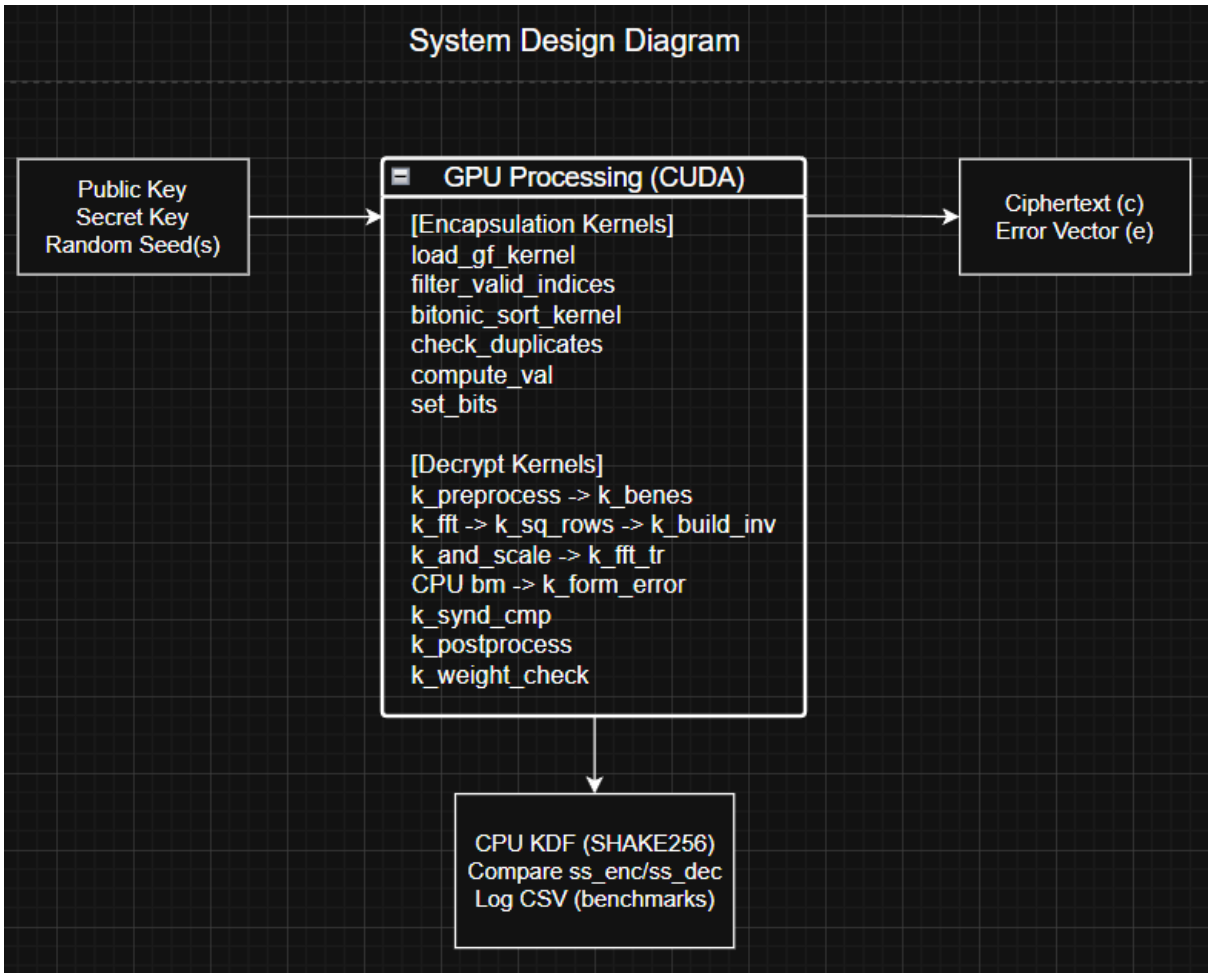


Figure 3.1 System Design Diagram

Figure 3.1 illustrates the overall design of the system, showing the main data flow and processing stages for the Classic McEliece KEM. Public and secret keys, along with random seeds, are loaded on the host and transferred to the GPU. The GPU performs encapsulation by generating an error vector, computing the ciphertext (syndrome), and returning results to the host. The CPU then derives the shared secret using SHAKE256. For decryption and decapsulation, the GPU computes the syndrome, applies the Benes permutation, evaluates the Goppa polynomial, and reconstructs the error vector, with the locator polynomial generated by

the CPU using the Berlekamp–Massey algorithm. The host finally derives the decapsulated shared secret, compares it with the encapsulated result, and records benchmark results such as execution time and throughput.

3.1.1 System Architecture Diagram

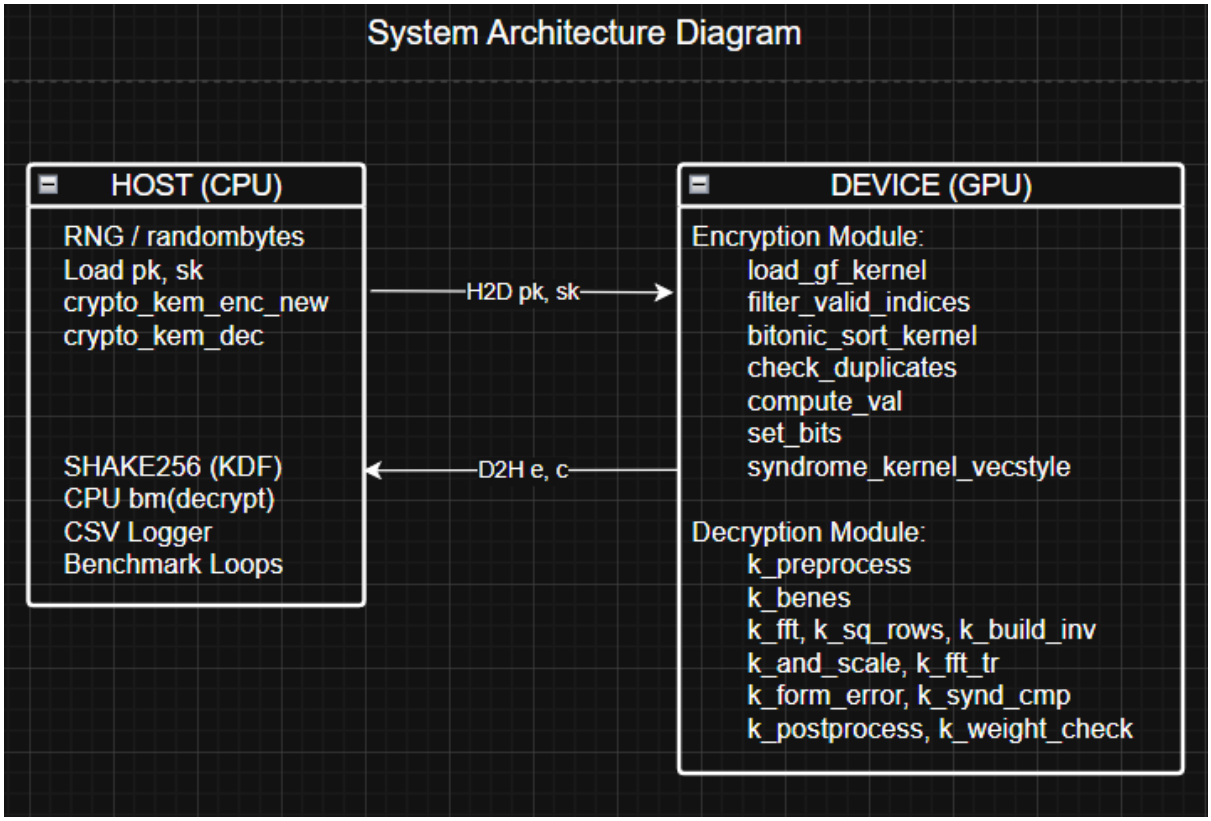


Figure 3.1.1 System Architecture Diagram

Figure 3.1.1 depicts the system architecture and clearly separates host-side and device-side responsibilities. The host manages key loading, random number generation, kernel launches, and result verification. It also performs CPU-based operations such as Berlekamp–Massey decoding and shared secret derivation using SHAKE256. The device executes highly parallel CUDA kernels for encapsulation, including error vector construction and syndrome computation, as well as kernels for decryption tasks such as syndrome preprocessing, FFT operations, error evaluation, and syndrome comparison. Data transfer occurs between the host and device for inputs (keys, ciphertexts) and outputs (error vectors, flags, syndromes). This architecture leverages GPU parallelism for computationally intensive parts while relying on the CPU for control flow and sequential operations.

3.1.2 Use Case Diagram and Description

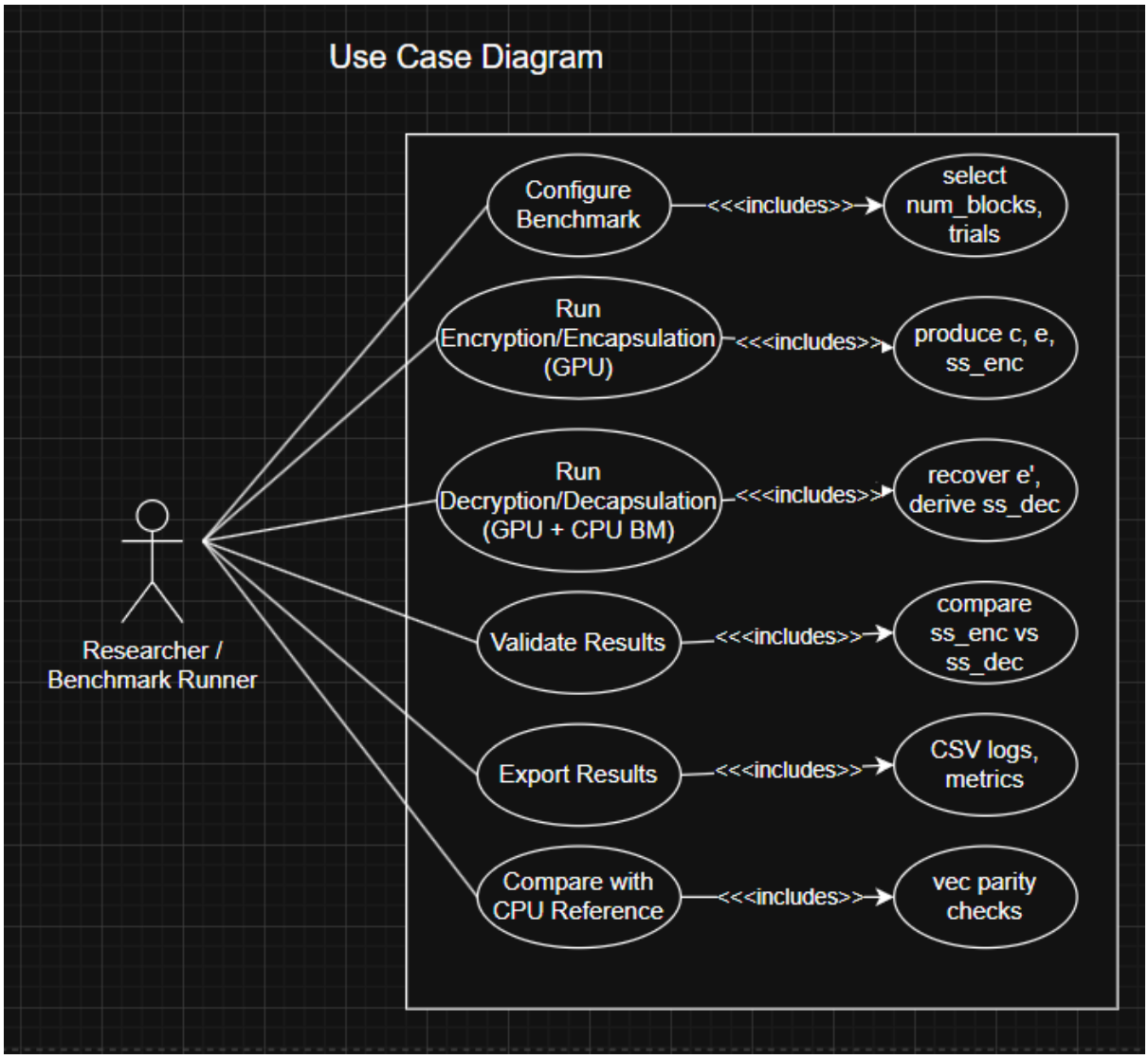


Figure 3.1.2 Use Case Diagram

Figure 3.1.2 presents the use case diagram for the GPU KEM benchmarking system. The primary actor, the researcher or benchmark runner, interacts with the system by configuring benchmark parameters, running encapsulation and decryption/decapsulation, validating the equality of the shared secrets, and exporting results for analysis. An additional use case allows the system to compare GPU outputs against a CPU vec implementation for correctness verification. This diagram highlights the functionality provided by the system from the perspective of its user.

3.1.3 Activity Diagram

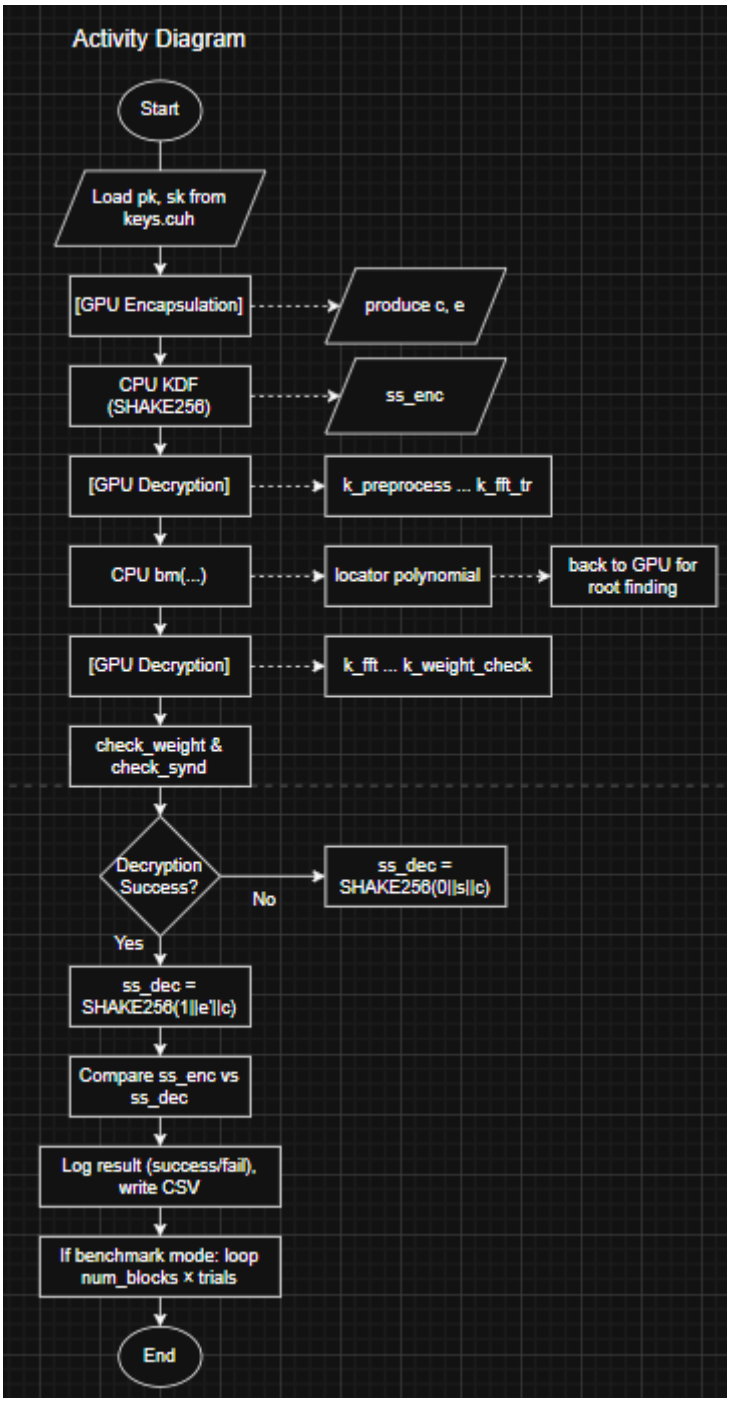


Figure 3.1.3 Activity diagram

Figure 3.1.3 shows the activity diagram outlining the workflow of the project. The process begins with loading keys and preparing inputs, followed by GPU-based encapsulation and CPU derivation of the encapsulated shared secret. The ciphertext is then passed to the GPU for decryption, where kernels compute the syndrome, perform permutation and evaluation steps, and return intermediate results to the CPU for polynomial solving. The GPU then reconstructs

the error vector, which is used to derive the decapsulated shared secret. A decision step determines whether decapsulation was successful, after which the system compares the encapsulated and decapsulated secrets. Results are logged and, if in benchmark mode, the process iterates across all configured numbers of blocks and trials before completion.

Chapter 4

System Design

4.1 System Block Diagram

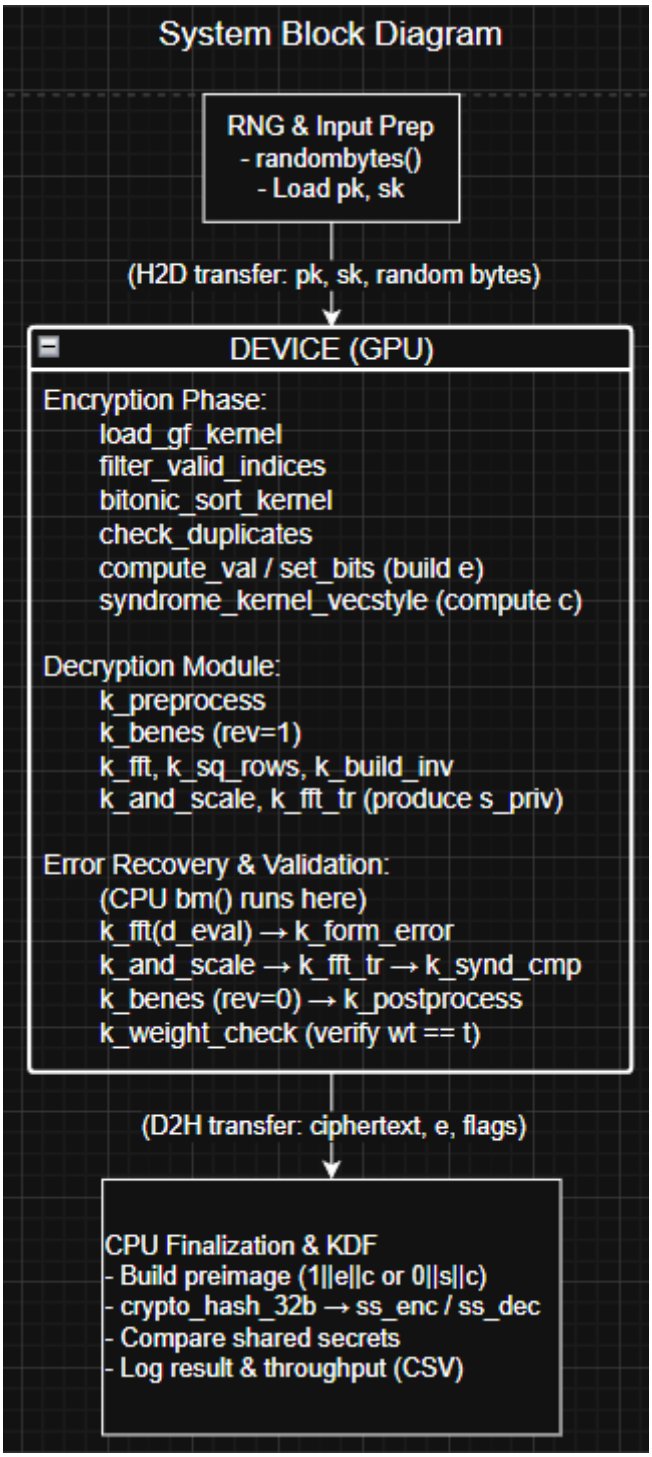


Figure 4.1 System Block Diagram

Figure 4.1 demonstrates the block diagram of the system for implementing a CUDA-accelerated Classic McEliece KEM as discussed in this work. Following a top-down design, the system continues with the preparation of input data on the host side, which involves public and secret keys, random seed material, and candidate error positions. All are prepared before transferring them from the host memory to the device memory in preparation for kernel execution.

The encapsulation phase is entirely performed on the GPU; first, one loads and filters random candidates via load_gf_kernel and filter_valid_indices, respectively, sorts them using bitonic_sort_kernel, computes a valid error vector with compute_val and set_bits, and finally, calculates the ciphertext using the syndrome_kernel_vecstyle kernel by multiplying the parity check matrix with the error vector in parallel. Subsequently, the ciphertext is sent back to the host, where the shared secret ss_enc is deduced through SHAKE256 hashing.

During decryption, the ciphertext is sent to the GPU, where syndrome k_preprocess, permutation through the Benes network k_benes, evaluation of the Goppa polynomial's roots k_fft, k_sq_rows, k_build_inv, and k_and_scale are performed. The resulting intermediate syndrome s_priv feeds back to the CPU, where the Berlekamp–Massey algorithm reconstructs the error locator polynomial. The locator is again transferred to the GPU for root finding; k_fft, k_form_error, rebuiling the syndrome, and checking its validity k_synd_cmp, and k_weight_check.

The reconstructed error vector is returned to the host, where the decapsulation routine performs preimage formation and shared secret re-derivation, ss_dec. Both shared secrets, ss_enc and ss_dec, are compared for equality. Benchmark mode exports execution time and throughput per kernel configuration in CSV format for post-execution analysis.

This gives a full view of the top-down system, showing how tasks are distributed among the cores of the CPU and GPU, reproducible results, and further optimizations in the future.

4.2 System Components and Design Specifications

This section provides a detailed explanation of all functional components used in the CUDA-based Classic McEliece implementation. Components are divided into utility functions, GPU kernel functions, and host functions for clarity. Each description includes the function's purpose, inputs/outputs, launch configuration (if applicable), and its relation to the overall pipeline.

4.2.1 Utility Functions (Device Functions)

This subsection documents the core utility functions used in both the encapsulation and decryption pipelines. These device-side helper functions are invoked inside CUDA kernels to perform small, frequently repeated operations such as field element conversion, bitwise packing, or memory storage. Although they are not launched as kernels themselves, their correctness and efficiency are critical for the overall performance of the system. Each function is explained below with its purpose, input/output specification, and its role within the KEM workflow.

Encrytion

```
__device__ uint16_t load_gf(const unsigned char *src) {
    uint16_t a = src[1];
    a <<= 8;
    a |= src[0];
    return a & GFMASK;
}
```

Figure 4.2 load_gf

Function: load_gf

Purpose: Reads two consecutive bytes from device memory and combines them into a single 16-bit field element within GF(2^m).

Parameters:

- const unsigned char *bytes: Pointer to two consecutive bytes in global memory.

Return Value:

- uint16_t: A 16-bit Galois Field element masked by GFMASK to ensure it lies within the valid finite field range.

Explaination:

This function is used in the *load_gf_kernel* to convert raw random bytes into candidate indices for error vector generation. By applying GFMASK, the function ensures that the resulting value does not exceed the maximum field size (1 << GFBITS). This step is critical because invalid indices could otherwise break syndrome computation. Every GPU thread running *load_gf_kernel* uses *load_gf* to populate the candidate array *d_nums*.

```
__device__ uint16_t crypto_uint16_smaller_mask(uint16_t x, uint16_t y) {
    uint16_t z = x - y;
    z ^= (x ^ y) & (z ^ x ^ (1 << 15));
    return z >> 15;
}
```

Figure 4.3 crypto_uint16_smaller_mask

Function: crypto_uint16_smaller_mask

Purpose: Computes a mask bit indicating whether x < y using arithmetic/bitwise operations suitable for data-oblivious comparisons.

Parameters:

- uint16_t x: first 16-bit operand
- uint16_t y: second 16-bit operand

Return Value:

- uint16_t: returns 1 if x < y, else 0

Explanation:

The function computes z = x - y. For unsigned 16-bit arithmetic, the **MSB of z** (bit 15) is set iff the subtraction underflows, i.e., iff x < y. The line z ^= (x ^ y) & (z ^ x ^ (1 << 15)); adjusts the intermediate so that the sign/borrow information is reliably captured across all input patterns. Finally, z >> 15 extracts the MSB as a **single-bit mask**. This is useful inside kernels where we need a branch-free indicator for bounds checks or ordering (e.g., keeping only indices < SYS_N during candidate filtering, or forming compare keys in a sort network).

```
__device__ uint16_t uint16_is_smaller_declassify(uint16_t t, uint16_t u) {
    return crypto_uint16_smaller_mask(t, u);
}
```

Figure 4.4 uint16_is_smaller_declassify

Function: uint16_is_smaller_declassify

Purpose: Wrapper that exposes the same semantics as *crypto_uint16_smaller_mask* with a clearer call-site name.

Parameters:

- uint16_t t: first 16-bit operand
- uint16_t u: second 16-bit operand

Return Value:

- uint16_t: returns 1 if t < u, else 0

Explaination:

This directly calls *crypto_uint16_smaller_mask*(t, u) and returns its result. The "_declassify" naming indicates it converts a comparison into a plain integer mask that can be used in arithmetic selection (e.g., res = a ^ (mask & (a ^ b))) without control-flow branches.

```
__device__ uint32_t crypto_uint32_equal_mask(uint32_t x, uint32_t y) {
    return ~(x ^ y) ? 0xFFFFFFFF : 0;
}
```

Figure 4.5 crypto_uint32_equal_mask

Function: crypto_uint32_equal_mask

Purpose: Produces a **32-bit all-ones mask** if x == y, and **0** otherwise.

Parameters:

- uint32_t x: first 32-bit operand
- uint32_t y: second 32-bit operand

Return Value:

- uint32_t: 0xFFFFFFFF if x == y, else 0x00000000.

Explaination:

The expression ~(x ^ y) is all ones if x ^ y == 0 (i.e., x == y). The code returns 0xFFFFFFFF when equal, 0 otherwise. This mask form is convenient for **branchless selection** and **bit-**

packing (e.g., turning equality into a mask to zero or keep certain lanes). In CUDA/PTX this typically lowers to predicated instructions, avoiding warp divergence.

```
__device__ uint32_t uint32_is_equal_declassify(uint32_t t, uint32_t u) {
    return crypto_uint32_equal_mask(t, u);
}
```

Figure 4.6 uint32_is_equal_declassify

Function: uint32_is_equal_declassify

Purpose: Wrapper that exposes an equality test as a full-width mask

Parameters:

- uint32_t t: first 32-bit operand.
- uint32_t u: second 32-bit operand.

Return value:

- uint32_t: 0xFFFFFFFF if t == u, else 0x00000000.

Explaination:

Calls crypto_uint32_equal_mask(t, u) and returns the mask. The declassify naming indicates conversion from a boolean condition to an integer value suitable for **mask-and-merge patterns** common in constant-time style code (e.g., dst = (dst & ~mask) | (newval & mask)).

```
static inline void store8(unsigned char *out, uint64_t in)
{
    out[0] = (in >> 0x00) & 0xFF;
    out[1] = (in >> 0x08) & 0xFF;
    out[2] = (in >> 0x10) & 0xFF;
    out[3] = (in >> 0x18) & 0xFF;
    out[4] = (in >> 0x20) & 0xFF;
    out[5] = (in >> 0x28) & 0xFF;
    out[6] = (in >> 0x30) & 0xFF;
    out[7] = (in >> 0x38) & 0xFF;
}
```

Figure 4.7 store8

Function: store8

Purpose: Serialize a 64-bit integer into 8 bytes in **little-endian** order.

Parameters:

- unsigned char *out: destination byte buffer (must have ≥ 8 bytes available).
- uint64_t in: 64-bit value to write

Explaination:

This utility writes in byte-by-byte, least significant first: out[0] gets bits [7:0], …, out[7] gets bits [63:56]. The function is used when the device/host needs a **packed byte representation** of 64-bit lanes, e.g., exporting a packed error-vector chunk or syndrome slice into the contiguous ciphertext buffer. Using an explicit store avoids endianness ambiguity and makes the output layout match the vec reference format exactly.

```
// read 64 bits from possibly unaligned address
__device__ inline uint64_t ld64_unaligned(const uint8_t* p) {
    uint64_t v = 0;
    // little-endian assemble
    #pragma unroll
    for (int k = 0; k < 8; ++k) v |= (uint64_t)p[k] << (8*k);
    return v;
}
```

Figure 4.8 ld64_unaligned

Function: ld64_unaligned

Purpose: Load **64 bits** from a **possibly unaligned** device address and assemble them in **little-endian** order into a uint64_t.

Parameters:

- const uint8_t* p: device pointer to the first of 8 bytes (may be unaligned).

Return value:

- uint64_t: value p[0] | p[1]<<8 | … | p[7]<<56.

Explaination:

Some buffers (e.g., packed ciphertext/syndrome arrays) are byte-packed without 8-byte alignment guarantees per thread. This helper safely reconstructs a 64-bit value by shifting/OR-ing the 8 constituent bytes inside a loop unrolled by #pragma unroll. It is used in preprocessing or inner loops that benefit from **working in 64-bit chunks** even when the start address isn't naturally aligned, improving throughput while avoiding misaligned 64-bit loads.

```cpp
// read 32 bits from possibly unaligned address
__device__ inline uint32_t ld32_unaligned(const uint8_t* p) {
    uint32_t v = 0;
    #pragma unroll
    for (int k = 0; k < 4; ++k) v |= (uint32_t)p[k] << (8*k);
    return v;
}
```

Figure 4.9 ld32_unaligned

Function: ld32_unaligned

Purpose: Load **32 bits** from a **possibly unaligned** device address and assemble them in **little-endian** order into a uint32_t.

Parameters

- const uint8_t* p: device pointer to the first of 4 bytes (may be unaligned).

Return value

- uint32_t: value p[0] | p[1]<<8 | p[2]<<16 | p[3]<<24.

Explaination:

Same rationale as ld64_unaligned, but for 4-byte chunks. This is handy when kernels operate on word-granularity (32-bit) data paths (e.g., parity accumulators, block headers) and the source buffer isn't guaranteed to be 4-byte aligned. The manual assemble keeps behavior well-defined and avoids alignment faults or performance penalties from misaligned native loads.

```cpp
// atomic XOR a single bit in s, but operate on a 32-bit word (CUDA requirement)
__device__ inline void atomic_xor_bit(uint8_t* s, int bit_index) {
    const int byte_index = bit_index >> 3;      // /8
    const int bit_in_byte = bit_index & 7;      // %8
    uint32_t* s32 = reinterpret_cast<uint32_t*>(s);

    const int word_index  = byte_index >> 2;    // /4
    const int byte_in_word = byte_index & 3;    // %4
    const uint32_t mask = ((uint32_t)(1u << bit_in_byte)) << (8 * byte_in_word);

    atomicXor(&s32[word_index], mask);
}
```

Figure 4.10 atomic_xor_bit

Function: atomic_xor_bit

Purpose: Perform an **atomic XOR of a single bit** within a byte-addressed buffer s, by targeting the corresponding **32-bit word** (CUDA provides atomics on 32-bit words, not individual bytes).

Parameters:

- uint8_t* s: base pointer to the destination byte buffer in device memory (e.g., packed syndrome).
- int bit_index: absolute bit position to toggle (bit 0 = s[0] LSB).

Explaination:

CUDA's atomicXor operates on uint32_t*. To flip a single target bit bit_index inside the byte array *s*, the routine:

1. Computes the **byte index**: byte_index = bit_index >> 3 and **bit within that byte**: bit_in_byte = bit_index & 7.
2. Reinterprets s as a uint32_t* (s32) and derives the **32-bit word index** containing that byte: word_index = byte_index >> 2.
3. Builds a 32-bit **mask** that has the desired bit set at the correct byte position inside that word:
   mask = ((uint32_t)(1u << bit_in_byte)) << (8 * (byte_index & 3));
4. Issues atomicXor(&s32[word_index], mask).

Because cudaMalloc returns **at least 4-byte aligned** pointers, &s32[word_index] is word-aligned, keeping the atomic valid. This helper is crucial in **syndrome_kernel_vecstyle** to combine per-thread parity contributions **without races**, while writing into a **packed bitstring** representation of the syndrome. It preserves correctness under parallel updates and avoids warp divergence from per-bit branching.

Decryption

```
// XOR of two bit-sliced vectors
__device__ __forceinline__ void d_vec_add(uint64_t* __restrict__ z,
                                          const uint64_t* __restrict__ x,
                                          const uint64_t* __restrict__ y) {
    #pragma unroll
    for (int b = 0; b < GFBITS; b++) z[b] = x[b] ^ y[b];
}
```

Figure 4.11 d_vec_add

Function: d_vec_add

Purpose: Bitwise XOR of two **bitsliced** vectors: z[b] = x[b] ^ y[b] for b ∈ [0, GFBITS). In GF(2), XOR implements vector addition.

Parameters:

- uint64_t* __restrict__ z: output bitslice array (length GFBITS).
- const uint64_t* __restrict__ x: first input bitslice array.
- const uint64_t* __restrict__ y: second input bitslice array.

Explanation**:**

Each uint64_t lane holds one bit position across 64 elements (bitslicing). This routine performs component-wise XOR across the GFBITS lanes, implementing addition in GF(2^m) with no carries and no branches. Used throughout the decrypt path wherever vector addition is required (e.g., during BM-related transforms or intermediate AFFT/TAFFT steps).

```
// 64x64 bit-matrix transpose (device) — same as transpose.h
__device__ __forceinline__ void d_transpose_64x64(uint64_t* out, const uint64_t* in) {
    static const uint64_t masks[6][2] = {
        {0x5555555555555555ULL, 0xAAAAAAAAAAAAAAAAULL},
        {0x3333333333333333ULL, 0xCCCCCCCCCCCCCCCCULL},
        {0x0F0F0F0F0F0F0F0FULL, 0xF0F0F0F0F0F0F0F0ULL},
        {0x00FF00FF00FF00FFULL, 0xFF00FF00FF00FF00ULL},
        {0x0000FFFF0000FFFFULL, 0xFFFF0000FFFF0000ULL},
        {0x00000000FFFFFFFFULL, 0xFFFFFFFF00000000ULL}
    };

    uint64_t tmp[64];
    #pragma unroll
    for (int i = 0; i < 64; i++) tmp[i] = in[i];

    for (int d = 5; d >= 0; d--) {
        int s = 1 << d;
        for (int i = 0; i < 64; i += s*2) {
            for (int j = i; j < i + s; j++) {
                uint64_t x = (tmp[j]     & masks[d][0]) | ((tmp[j+s] & masks[d][0]) << s);
                uint64_t y = ((tmp[j]    & masks[d][1]) >> s)       | (tmp[j+s] & masks[d][1]);
                tmp[j]   = x;
                tmp[j+s] = y;
            }
        }
    }
    #pragma unroll
    for (int i = 0; i < 64; i++) out[i] = tmp[i];
}
```

Figure 4.12 d_transpose_64x64

Function: d_transpose_64x64

Purpose: Transpose a **64×64 bit matrix** stored as 64 words of 64 bits (row-major) into its bitwise transpose (column-major).

Parameters:

- uint64_t* out: destination array of 64 words (transposed result).
- const uint64_t* in: source array of 64 words.

Explaination:

Implements the classic butterfly/masking transpose in 6 stages using precomputed masks for 1-, 2-, 4-, 8-, 16-, and 32-bit swaps. At each stage d, it swaps bit fields of width s = 1<<d between paired rows using:

- masks[d][0] (keep lower-half fields) and
- masks[d][1] (keep upper-half fields),

  with shifts and ORs to interleave/extract blocks. The routine copies in to a local tmp[64], performs all staged swaps, then writes tmp to out. This transpose is used

where AFFT/TAFFT or Benes/locator steps require changing data orientation between row-wise and column-wise bit layouts, enabling coalesced operations on lanes.

```
__device__ __forceinline__ void d_vec_copy(vec* __restrict__ out,
                                           const vec* __restrict__ in) {
    #pragma unroll
    for (int i = 0; i < GFBITS; i++) out[i] = in[i];
}
```

Figure 4.13 d_vec_copy

Function: d_vec_copy

Purpose: Copy a GFBITS-length **bitsliced** vector.

Parameters:

- vec* __restrict__ out: destination array (GFBITS elements).
- const vec* __restrict__ in: source array (GFBITS elements).

Explaination:

Straight-line copy with #pragma unroll. Keeps register and cache behavior predictable and avoids function call overhead inside tight arithmetic ladders (e.g., inversion ladder below). Used widely in the fixed-sequence field operations.

```
/* bitsliced multiplication in GF(2^12) with modulus x^12 + x^3 + 1 */
__device__ __forceinline__ void d_vec_mul(vec* __restrict__ h,
                                          const vec* __restrict__ f,
                                          const vec* __restrict__ g) {
    vec buf[2*GFBITS - 1];

    #pragma unroll
    for (int i = 0; i < 2*GFBITS - 1; i++) buf[i] = 0;

    #pragma unroll
    for (int i = 0; i < GFBITS; i++) {
        #pragma unroll
        for (int j = 0; j < GFBITS; j++) {
            buf[i + j] ^= (f[i] & g[j]);
        }
    }

    // fold high terms per x^12 = x^3 + 1
    for (int i = 2*GFBITS - 2; i >= GFBITS; i--) {
        buf[i - GFBITS + 3] ^= buf[i];
        buf[i - GFBITS + 0] ^= buf[i];
    }

    #pragma unroll
    for (int i = 0; i < GFBITS; i++) h[i] = buf[i];
}
```

Figure 4.14 d_vec_mull

Function: d_vec_mull

Purpose: **Bitsliced multiplication** in GF(2^{12}) with irreducible modulus $x^{12} + x^3 + 1$.

Parameters:

- vec* __restrict__ h: output bitsliced product (GFBITS lanes).
- const vec* __restrict__ f: multiplicand bitsliced vector.
- const vec* __restrict__ g: multiplier bitsliced vector.

Explaination:

Forms the schoolbook convolution into a temporary array buf[0..2*GFBITS-2] via lane-wise AND/XOR: buf[i+j] ^= f[i] & g[j]. Then reduces high terms using the modulus relation $x^{12} + x^3 + 1$: for indices i ≥ GFBITS, fold as buf[i - GFBITS + 3] ^= buf[i] and buf[i - GFBITS + 0] ^= buf[i].

Finally, the low GFBITS lanes are the product *h*. This routine is a core primitive used in evaluating Goppa polynomials, computing inverses (via ladder), and scaling steps during the decrypt/decaps path.

```
/* bitsliced squaring: same wiring as vec_sq() */
__device__ __forceinline__ void d_vec_sq(vec* __restrict__ out,
                                         const vec* __restrict__ in) {
    vec r[GFBITS];
    r[0]  = in[0] ^ in[6];
    r[1]  = in[11];
    r[2]  = in[1] ^ in[7];
    r[3]  = in[6];
    r[4]  = in[2] ^ in[11] ^ in[8];
    r[5]  = in[7];
    r[6]  = in[3] ^ in[9];
    r[7]  = in[8];
    r[8]  = in[4] ^ in[10];
    r[9]  = in[9];
    r[10] = in[5] ^ in[11];
    r[11] = in[10];

    #pragma unroll
    for (int i = 0; i < GFBITS; i++) out[i] = r[i];
}
```

Figure 4.15 d_vec_sq

Function: d_vec_sq

Purpose: **Bitsliced squaring** in GF(2^{12}) using the same wiring/permutation as *vec_sq*().

Parameters:

- vec* __restrict__ out: output bitsliced square.
- const vec* __restrict__ in: input bitsliced element.

Explaination:

In characteristic-2 fields, squaring is a **linear permutation** of coefficient lanes. This function applies the exact wiring pattern for the field basis (12 lanes) and then copies *r* to *out.* This is heavily used inside the inversion ladder and in AFFT/TAFFT-related transforms.

```
/* bitsliced inversion via fixed ladder (same as vec_inv()) */
__device__ __forceinline__ void d_vec_inv(vec* __restrict__ out,
                                          const vec* __restrict__ in) {
    vec tmp_11[GFBITS];
    vec tmp_1111[GFBITS];

    d_vec_copy(out, in);

    d_vec_sq(out, out);
    d_vec_mul(tmp_11, out, in);          // 11

    d_vec_sq(out, tmp_11);
    d_vec_sq(out, out);
    d_vec_mul(tmp_1111, out, tmp_11);    // 1111

    d_vec_sq(out, tmp_1111);
    d_vec_sq(out, out);
    d_vec_sq(out, out);
    d_vec_sq(out, out);
    d_vec_mul(out, out, tmp_1111);       // 11111111

    d_vec_sq(out, out);
    d_vec_sq(out, out);
    d_vec_mul(out, out, tmp_11);         // 1111111111

    d_vec_sq(out, out);
    d_vec_mul(out, out, in);             // 11111111111

    d_vec_sq(out, out);                  // 111111111110
}
```

Figure 4.16 d_vec_inv

Function: d_vec_inv

Purpose: **Bitsliced inversion** in GF(2^{12}) via a fixed sequence (“ladder”) of squarings and multiplications, equivalent to *vec_inv*().

Parameters:

- vec* __restrict__ out: output bitsliced inverse in^{-1}.
- const vec* __restrict__ in: input bitsliced element.

Explaination:

Implements an addition chain using only *d_vec_sq* and *d_vec_mul*. The sequence builds intermediate exponents, then escalates by repeated squarings and selective multiplies. The exact schedule mirrors the reference *vec_inv*() so that instruction mix and data hazards remain predictable on the GPU. This routine is used in *k_build_inv* to compute $1/g(\alpha_i)$ across evaluation points, a critical step prior to scaling and the transpose FFT in the decrypt/decaps pipeline.

```cpp
// Parallel version of layer(): 32 disjoint pairs per layer
__device__ __forceinline__ void d_layer_parallel(uint64_t* __restrict__ data, const uint64_t* __restrict__ bits,
                                                 int lgs, int t, int nt)
{
    const int s     = 1 << lgs;
    const int pairs = 32;               // 64 lanes form 32 (j, j+s) pairs
    for (int idx = t; idx < pairs; idx += nt) {
        const int blk = idx / s;
        const int off = idx % s;
        const int j   = blk * (2*s) + off;

        uint64_t d = (data[j] ^ data[j + s]) & bits[idx];
        data[j]   ^= d;
        data[j+s] ^= d;
    }
}
```

Figure 4.17 d_layer_parallel

Function: d_layer_parallel

Purpose: Apply one **Benes layer** in parallel over a 64-lane bitsliced vector by processing 32 disjoint pairs (j,j+s), where $s=2^{lgs}$. Each pair is conditionally XOR-swapped under a per-pair control bit from bits[idx].

Explaination:

This device function applies one layer of the Benes network in parallel across 64 lanes. Each thread processes a subset of the 32 disjoint pairs (j,j+s), where s=1 << lgss, and conditionally swaps them based on the corresponding mask in bits. The swap is implemented using branch-free XOR operations, ensuring deterministic execution without warp divergence. This function is called repeatedly in *k_benes* to perform both the forward and inverse permutations during decryption.

```cpp
// builds out01[0], out01[1] from 'in' (64 x GFBITS), byte-for-byte like CPU fft_tr.c
static __device__ __forceinline__
void d_broadcast_and_beta(uint64_t out01[2][GFBITS],
                          uint64_t in[64][GFBITS])
{
    uint64_t pre[6][GFBITS];

    // pre / accumulation chain (identical to CPU order)
    d_vec_copy(pre[0], in[32]); d_vec_add(in[33], in[33], in[32]);
    d_vec_copy(pre[1], in[33]); d_vec_add(in[35], in[35], in[33]);
    d_vec_add(pre[0], pre[0], in[35]); d_vec_add(in[34], in[34], in[35]);
    d_vec_copy(pre[2], in[34]); d_vec_add(in[38], in[38], in[34]);
    d_vec_add(pre[0], pre[0], in[38]); d_vec_add(in[39], in[39], in[38]);
    d_vec_add(pre[1], pre[1], in[39]); d_vec_add(in[37], in[37], in[39]);
    d_vec_add(pre[0], pre[0], in[37]); d_vec_add(in[36], in[36], in[37]);
    d_vec_copy(pre[3], in[36]); d_vec_add(in[44], in[44], in[36]);
    d_vec_add(pre[0], pre[0], in[44]); d_vec_add(in[45], in[45], in[44]);
    d_vec_add(pre[1], pre[1], in[45]); d_vec_add(in[47], in[47], in[45]);
    d_vec_add(pre[0], pre[0], in[47]); d_vec_add(in[46], in[46], in[47]);
    d_vec_add(pre[2], pre[2], in[46]); d_vec_add(in[42], in[42], in[46]);
    d_vec_add(pre[0], pre[0], in[42]); d_vec_add(in[43], in[43], in[42]);
    d_vec_add(pre[1], pre[1], in[43]); d_vec_add(in[41], in[41], in[43]);
    d_vec_add(pre[0], pre[0], in[41]); d_vec_add(in[40], in[40], in[41]);
    d_vec_copy(pre[4], in[40]); d_vec_add(in[56], in[56], in[40]);
    d_vec_add(pre[0], pre[0], in[56]); d_vec_add(in[57], in[57], in[56]);
    d_vec_add(pre[1], pre[1], in[57]); d_vec_add(in[59], in[59], in[57]);
    d_vec_add(pre[0], pre[0], in[59]); d_vec_add(in[58], in[58], in[59]);
    d_vec_add(pre[2], pre[2], in[58]); d_vec_add(in[62], in[62], in[58]);
    d_vec_add(pre[0], pre[0], in[62]); d_vec_add(in[63], in[63], in[62]);
    d_vec_add(pre[1], pre[1], in[63]); d_vec_add(in[61], in[61], in[63]);
    d_vec_add(pre[0], pre[0], in[61]); d_vec_add(in[60], in[60], in[61]);
    d_vec_add(pre[3], pre[3], in[60]); d_vec_add(in[52], in[52], in[60]);
    d_vec_add(pre[0], pre[0], in[52]); d_vec_add(in[53], in[53], in[52]);
    d_vec_add(pre[1], pre[1], in[53]); d_vec_add(in[55], in[55], in[53]);
    d_vec_add(pre[0], pre[0], in[55]); d_vec_add(in[54], in[54], in[55]);
    d_vec_add(pre[1], pre[1], in[9]); d_vec_add(in[11], in[11], in[9]);
    d_vec_add(pre[0], pre[0], in[11]); d_vec_add(in[10], in[10], in[11]);
    d_vec_add(pre[2], pre[2], in[10]); d_vec_add(in[14], in[14], in[10]);
    d_vec_add(pre[0], pre[0], in[14]); d_vec_add(in[15], in[15], in[14]);
    d_vec_add(pre[1], pre[1], in[15]); d_vec_add(in[13], in[13], in[15]);
    d_vec_add(pre[0], pre[0], in[13]); d_vec_add(in[12], in[12], in[13]);
    d_vec_add(pre[3], pre[3], in[12]); d_vec_add(in[4], in[4], in[12]);
    d_vec_add(pre[0], pre[0], in[4]); d_vec_add(in[5], in[5], in[4]);
    d_vec_add(pre[1], pre[1], in[5]); d_vec_add(in[7], in[7], in[5]);
    d_vec_add(pre[0], pre[0], in[7]); d_vec_add(in[6], in[6], in[7]);
    d_vec_add(pre[2], pre[2], in[6]); d_vec_add(in[2], in[2], in[6]);
    d_vec_add(pre[0], pre[0], in[2]); d_vec_add(in[3], in[3], in[2]);
    d_vec_add(pre[1], pre[1], in[3]); d_vec_add(in[1], in[1], in[3]);

    // out0 = in[0] + in[1], and pre[0] += in[1]
    d_vec_add(out01[0], in[0], in[1]);
    d_vec_add(pre[0], pre[0], in[1]);

    // out1 = sum_i pre[i] * mask(beta[i])
    // make masks from c_beta
    uint64_t maskv[GFBITS];
    #pragma unroll
    for (int j = 0; j < GFBITS; j++) maskv[j] = ((c_beta[0] >> j) & 1) ? ~0ULL : 0ULL;
    d_vec_mul(out01[1], pre[0], maskv);

    for (int i = 1; i < 6; i++) {
        #pragma unroll
        for (int j = 0; j < GFBITS; j++) maskv[j] = ((c_beta[i] >> j) & 1) ? ~0ULL : 0ULL;
        uint64_t prod[GFBITS];
        d_vec_mul(prod, pre[i], maskv);
        #pragma unroll
        for (int j = 0; j < GFBITS; j++) out01[1][j] ^= prod[j];
    }
}
```

Figure 4.18 d_broadcast_and_beta

Function: d_broadcast_and_beta

Purpose: Compute the two outputs required by the **transpose additive FFT (TAFFT) front end** exactly like the CPU *fft_tr.c*:

- out01[0] as a specific sum of the first two inputs
- out01[1] as a linear combination of accumulated intermediates weighted by the β constants.

Parameters:

- uint64_t out01[2][GFBITS] — destination arrays for the two GFBITS-lane outputs (bitsliced).
- uint64_t in[64][GFBITS] — working set of 64 input rows (each row is a GFBITS-lane bitsliced vector).

Explaination:

This function reproduces the CPU *fft_tr.c* accumulation chain to build two outputs: out01[0], which is the XOR of the first two input rows, and out01[1], which is a β-weighted sum of pre-accumulated vectors. It uses *d_vec_copy, d_vec_add, and d_vec_mul* to exactly match the CPU's byte-level output, ensuring that the GPU and CPU generate identical intermediate states for subsequent TAFFT processing.

4.2.2 Kernel Functions

Encrypt

```
__global__ void load_gf_kernel(unsigned char *src, uint16_t *nums) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < SYS_T * 2) {
        nums[i] = load_gf(src + i * 2);
    }
}
```

Figure 4.19 load_gf_kernel

Function: load_gf_kernel

Purpose: Convert raw random bytes into candidate field elements for error-position selection

Parameters:

- unsigned char *src: device pointer to the random byte pool
- uint16_t *nums: device output array for candidates (length SYS_T*2)

Return value:

- void (writes nums[i])

Explaination:

Each thread i = blockIdx.x*blockDim.x + threadIdx.x (for i < SYS_T*2) reads two bytes at src + 2*i and calls load_gf to assemble a 16-bit value masked to GF(2^m). This produces 2*SYS_T candidate indices for later filtering.

```
__global__ void filter_valid_indices(uint16_t *nums, uint16_t *ind, int *count) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < SYS_T * 2) {
        if (uint16_is_smaller_declassify(nums[i], SYS_N)) {
            int pos = atomicAdd(count, 1);
            if (pos < SYS_T) {
                ind[pos] = nums[i];
            }
        }
    }
}
```

Figure 4.20 filter_valid_indices

Function: filter_valid_indices

Purpose: Select error indices strictly less than SYS_N and collect up to SYS_T of them.

Parameters:

- uint16_t *nums: input candidates (SYS_T*2).
- uint16_t *ind: output array of accepted indices (capacity SYS_T).
- int *count: global counter (initialized to 0) updated via atomicAdd.

Return value:

- void (writes ind[pos], updates *count).

Explaination:

For each i < SYS_T*2, the kernel checks nums[i] < SYS_N using uint16_is_smaller_declassify. Valid entries reserve a slot with pos = atomicAdd(count,1); if pos < SYS_T, the value is stored in ind[pos]. This yields up to SYS_T distinct positions for the error vector.

```
__global__ void check_duplicates(uint16_t *ind, int *has_duplicates) {
    int i = threadIdx.x;
    //int j = threadIdx.y;

    if (i > 0 && i < SYS_T) {
        if (uint32_is_equal_declassify(ind[i-1], ind[i])) {
            atomicExch(has_duplicates, 1);
        }
    }
}
```

Figure 4.21 check_duplicates

Function: check_duplicates

Purpose: Detect adjacent duplicates in a **sorted** index array.

Parameters:

- uint16_t *ind: sorted indices (length SYS_T).
- int *has_duplicates: flag set to 1 if any duplicate found (must be initialized to 0).

Return value:

- void (Possible *has_duplicates = 1).

Explaination:

Threads with i>0 && i<SYS_T compare ind[i-1] and ind[i] using uint32_is_equal_declassify. On equality, a thread calls atomicExch(has_duplicates,1). This guards against repeated error positions before packing.

```
__global__ void compute_val(uint16_t *d_ind, uint64_t *d_val){
    int j = threadIdx.x; // Each thread corresponds to a `val[j]`

    if (j < SYS_T) {
        d_val[j] = 1ULL << (d_ind[j] & 63); // Compute bit shift for each index
        //printf("d_ind[%d] = %u, d_val[%d] = 0x%016llx\n", j, d_ind[j], j, d_val[j]);
    }
}
```

Figure 4.22 compute_val

Function: compute_val

Purpose: Precompute per-index bitmasks for packing the error vector into 64-bit lanes.

Parameters:

- uint16_t *d_ind: accepted indices (length SYS_T).
- uint64_t *d_val: output bitmasks (length SYS_T).

Return value:

- void (writes d_val[j]).

Explaination:

Thread j < SYS_T computes d_val[j] = 1ULL << (d_ind[j] & 63). These masks represent the bit position within a 64-bit lane and are later OR-reduced by set_bits.

```
__global__ void set_bits(uint16_t *d_ind, uint64_t *d_val, uint64_t *d_e_int) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // Each thread works on one `e[i]`
    if (i >= (SYS_N + 63) / 64){
        //printf("Failed to run set_bits");
        return;
    }

    d_e_int[i] = 0; //Initialize 'e[i]'

    //uint64_t result = 0;
    for (int j = 0; j < SYS_T; j++) {
        uint64_t mask = i ^ (d_ind[j] >> 6);
        mask -= 1;
        mask >>= 63;
        mask = -mask;

        d_e_int[i] |= d_val[j] & mask;
    }
}
```

Figure 4.23 set_bits

Function: set_bits

Purpose: Construct the packed error vector e across 64-bit lanes.

Parameters:

- uint16_t *d_ind: error indices (SYS_T).
- uint64_t *d_val: per-index lane masks (SYS_T).
- uint64_t *d_e_int: output lanes (length (SYS_N+63)/64).

Return value:

- void (writes d_e_int[i]).

Explaination:

Each thread handles one lane i < (SYS_N+63)/64, initializes d_e_int[i]=0, then for all j < SYS_T conditionally ORs d_val[j] when (d_ind[j]>>6) == i. Equality is implemented branch-free via a sign-propagating mask so threads follow identical control flow.

```
__global__ void bitonic_sort_kernel(uint16_t* data, int num_elements) {
    extern __shared__ uint16_t shared[];

    int tid = threadIdx.x;
    if (tid >= num_elements) return;  // Only use threads for actual data

    // Load from global to shared memory
    shared[tid] = data[tid];
    __syncthreads();

    // Bitonic sort on shared memory
    for (int k = 2; k <= num_elements; k *= 2) {
        for (int j = k / 2; j > 0; j /= 2) {
            int ixj = tid ^ j;
            if (ixj > tid && ixj < num_elements) {
                bool ascending = ((tid & k) == 0);
                uint16_t a = shared[tid];
                uint16_t b = shared[ixj];
                if ((ascending && a > b) || (!ascending && a < b)) {
                    shared[tid] = b;
                    shared[ixj] = a;
                }
            }
            __syncthreads();
        }
    }

    // Write back to global memory
    data[tid] = shared[tid];
}
```

Figure 4.24 bitonic_sort_kernel

Function: bitonic_sort_kernel

Purpose: Sort a small array of indices in shared memory using the bitonic network.

Parameters:

- uint16_t *data: array to sort (length num_elements).
- int num_elements: number of active elements (≤ block size).

Return value:

- void (writes back sorted data[tid]).

Explaination:

Each thread tid loads one element to shared[tid], then the kernel runs the standard bitonic k/j stages with pairwise compare-and-swap under ascending = (tid & k) == 0. After synchronization at each step, the sorted results are written back to global memory. This produces an ordered ind for duplicate detection and packing.

```
__global__ void warmup_kernel() {
    // Empty kernel, just uses threads for GPU initialization
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    // No computation, just thread utilization
}
```

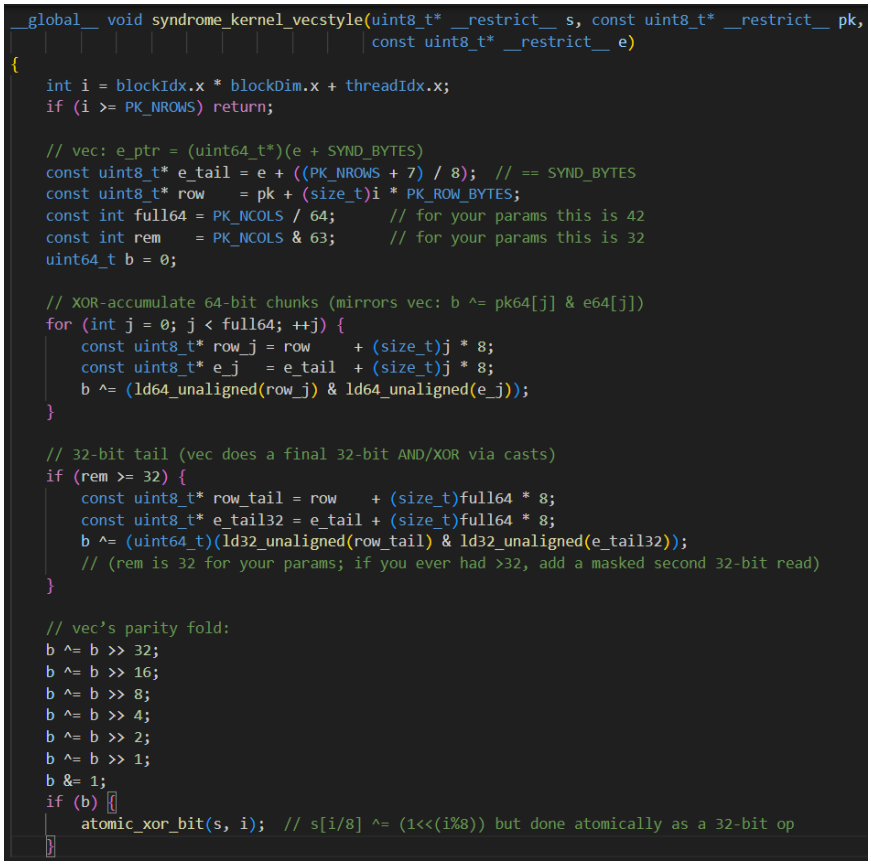
Figure 4.25 warmup_kernel

Function: warmup_kernel

Purpose: Warm up the GPU context and clocks prior to timing measurements.

Explaination:

The kernel performs no work; it simply launches threads to initialize driver/runtime state and stabilize GPU frequency before real kernels are timed.



```
__global__ void syndrome_kernel_vecstyle(uint8_t* __restrict__ s, const uint8_t* __restrict__ pk,
                                         const uint8_t* __restrict__ e)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= PK_NROWS) return;

    // vec: e_ptr = (uint64_t*)(e + SYND_BYTES)
    const uint8_t* e_tail = e + ((PK_NROWS + 7) / 8);  // == SYND_BYTES
    const uint8_t* row    = pk + (size_t)i * PK_ROW_BYTES;
    const int full64 = PK_NCOLS / 64;      // for your params this is 42
    const int rem    = PK_NCOLS & 63;      // for your params this is 32
    uint64_t b = 0;

    // XOR-accumulate 64-bit chunks (mirrors vec: b ^= pk64[j] & e64[j])
    for (int j = 0; j < full64; ++j) {
        const uint8_t* row_j = row    + (size_t)j * 8;
        const uint8_t* e_j   = e_tail + (size_t)j * 8;
        b ^= (ld64_unaligned(row_j) & ld64_unaligned(e_j));
    }

    // 32-bit tail (vec does a final 32-bit AND/XOR via casts)
    if (rem >= 32) {
        const uint8_t* row_tail = row    + (size_t)full64 * 8;
        const uint8_t* e_tail32 = e_tail + (size_t)full64 * 8;
        b ^= (uint64_t)(ld32_unaligned(row_tail) & ld32_unaligned(e_tail32));
        // (rem is 32 for your params; if you ever had >32, add a masked second 32-bit read)
    }

    // vec's parity fold:
    b ^= b >> 32;
    b ^= b >> 16;
    b ^= b >> 8;
    b ^= b >> 4;
    b ^= b >> 2;
    b ^= b >> 1;
    b &= 1;
    if (b) {
        atomic_xor_bit(s, i);  // s[i/8] ^= (1<<(i%8)) but done atomically as a 32-bit op
    }
```

Figure 4.26 syndrome_kernel_vecstyle

Function: syndrome_kernel_vecstyle

Purpose: Compute the Niederreiter **syndrome/ciphertext** bit for each parity-check row, matching the **vec** reference parity fold.

Parameters:

- uint8_t* s: output syndrome buffer (length SYND_BYTES), initialized before launch.
- const uint8_t* pk: public key matrix in row-major, PK_NROWS * PK_ROW_BYTES.
- const uint8_t* e: packed error vector SYS_N/8 bytes (kernel uses e + SYND_BYTES like vec).

Return value:

- void (atomically toggles bits in *s*).

Explaination:

Thread i < PK_NROWS processes row i. It AND-accumulates 64-bit chunks of the row with corresponding chunks of e_tail = e + SYND_BYTES using ld64_unaligned, plus a 32-bit tail when rem>=32. The 64-bit accumulator b is then reduced to a single parity bit by successive

XOR shifts (b ^= b>>32; …; b&=1). If the bit is 1, the kernel flips bit i in s via atomic_xor_bit, which performs a 32-bit atomic XOR masked to the correct bit position. This mirrors vec's byte-exact behavior.

Decrypt

```
// preprocess: s(bytes) -> recv[64] (uint64 words), little-endian load8 every 8 bytes
__global__ void k_preprocess(vec* __restrict__ recv, const unsigned char* __restrict__ s) {
    int stride = blockDim.x * gridDim.x;
    for (int i = blockIdx.x*blockDim.x + threadIdx.x; i < 64; i += stride) {
        uint64_t v = 0;
        #pragma unroll
        for (int b = 7; b >= 0; --b) {
            unsigned char byte = 0;
            size_t off = (size_t)i*8 + b;
            if (off < SYND_BYTES) byte = s[off];  // read s[i*8 + b]
            v <<= 8;
            v |= (uint64_t)byte;
        }
        recv[i] = v;
    }
}
```

Figure 4.27 k_preprocess

Function: k_preprocess

Purpose: Pack the byte-array syndrome s into 64 little-endian 64-bit words (recv[0..63]).

Parameters:

- vec* recv: output array of 64 words.
- const unsigned char* s: input syndrome bytes (length SYND_BYTES).

Return value:

- void (writes recv[i]).

Explaination:

Uses a grid-stride loop; each thread assembles one or more rows i by shifting/OR-ing up to 8 bytes (guarded for tail). Produces the bitsliced form consumed by Benes/FFT stages.

```
// postprocess: error[64] lanes -> e[SYS_N/8] (pack first SYS_N bits)
// grid-stride loop + a single 64-bit store per lane
__global__ void k_postprocess(unsigned char* __restrict__ e, const vec* __restrict__ error) {
    int stride = blockDim.x * gridDim.x;
    uint64_t* out64 = reinterpret_cast<uint64_t*>(e);
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < 64; i += stride) {
        out64[i] = error[i];
    }
}
```

Figure 4.28 k_postprocess

Function: k_postprocess

Purpose: Serialize the 64 error lanes into the packed error vector e (first SYS_N bits significant).

Parameters:

- unsigned char* e: output packed error bytes (SYS_N/8 minimum).
- const vec* error: input 64 words (one per lane).

Return value:

- void (writes via uint64_t *out_64).

Explaination:

Grid-stride over i∈[0,64), casting e to uint64_t* and storing error[i] directly. The caller masks excess bits beyond SYS_N when needed.

```
// error formation: error[i] = OR_j eval[i][j] ^ allone
// eval64x: 64 rows, each with GFBITS columns (row-major)
// error64: 64 words; bit j set if evaluation at coordinate j is ZERO in GF
__global__ void k_form_error(vec* __restrict__ error, const vec* __restrict__ eval /*[64][GFBITS]*/)
{
    const int stride = blockDim.x * gridDim.x;
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < 64; i += stride) {
        const vec* row = eval + i * GFBITS;
        vec r = d_vec_or_reduce(row);       // OR of the GFBITS slices
        r ^= d_vec_setbits(1);              // invert: 1s where value==0
        error[i] = r;
    }
}
```

Figure 4.29 k_form_error

Function: k_form_error

Purpose: Form the error mask: set bit j in error[i] iff evaluation at coordinate j is zero.

Parameters:

- vec* error: output 64 words.
- const vec* eval: input matrix eval[64][GFBITS] (row-major).

Return value:

- void (writes error[i]).

Explaination:

For each row i, OR-reduces the GFBITS slices into r, flips bits (r ^= setbits(1)) so zeros become ones, and stores to error[i]. Implements the locator-root test.

```cuda
// scaling_inv / scaling final step:
// grid-stride over rows; each thread processes a whole row's 12 bits in a tiny inner loop
__global__ void k_and_scale(vec* __restrict__ out, const vec* __restrict__ inv, const vec* __restrict__ col) {
    int stride = blockDim.x * gridDim.x;
    for (int i = blockIdx.x*blockDim.x + threadIdx.x; i < 64; i += stride) {
        const uint64_t m = col[i];
        const vec* in_row  = inv    + i*GFBITS;
        vec*       out_row = out    + i*GFBITS;
        #pragma unroll
        for (int b = 0; b < GFBITS; ++b) out_row[b] = in_row[b] & m;
    }
}
```

Figure 4.30 k_and_scale

Function: k_and_scale

Purpose: Apply column-wise scaling: out[i][b] = inv[i][b] & col[i].

Parameters:

- vec* out: output 64×GFBITS.
- const vec* inv: input 64×GFBITS (e.g., 1/g(α_i)).
- const vec* col: column mask per row (col[i] used for all b).

Return value:

- void (writes out[i*GFBITS + b]).

Explaination:

Grid-stride over rows; each thread ANDs the 12 slices of its row with the row mask. Used both after inverse build and during final re-syndrome.

```
__global__ void k_fft(vec* __restrict__ out64x, vec* __restrict__ inGFBITS)
{
    const int tid = threadIdx.x;
    if (blockIdx.x != 0 || tid >= 64) return;  // single-block transform

    // --- shared working sets ---
    __shared__ uint64_t in_radix[GFBITS];        // size 12
    __shared__ uint64_t cur[64][GFBITS];         // 64*12*8 = 6 KB
    __shared__ uint64_t nxt[64][GFBITS];         // another 6 KB

    // ---- load input, radix_conversions(in) on thread 0 ----
    if (tid < GFBITS) in_radix[tid] = inGFBITS[tid];
    __syncthreads();

    if (tid == 0) {
        // same masks as your current k_fft
        const uint64_t mask[5][2] = {
            {0x8888888888888888ULL, 0x4444444444444444ULL},
            {0xC0C0C0C0C0C0C0C0ULL, 0x3030303030303030ULL},
            {0xF000F000F000F000ULL, 0x0F000F000F000F00ULL},
            {0xFF000000FF000000ULL, 0x00FF000000FF0000ULL},
            {0xFFFF000000000000ULL, 0x0000FFFF00000000ULL}
        };

        // inplace radix conversions + scaling by c_scalars[j]
        for (int j = 0; j <= 4; j++) {
            for (int i = 0; i < GFBITS; i++) {
                for (int k = 4; k >= j; k--) {
                    in_radix[i] ^= (in_radix[i] & mask[k][0]) >> (1 << k);
                    in_radix[i] ^= (in_radix[i] & mask[k][1]) >> (1 << k);
                }
            }
            uint64_t tmp[GFBITS];
            d_vec_mul(tmp, in_radix, c_scalars[j]);
            #pragma unroll
            for (int i=0;i<GFBITS;i++) in_radix[i] = tmp[i];
        }
    }
    __syncthreads();

    // ---- broadcast: each thread builds its row from in_radix ----
    #pragma unroll
    for (int b = 0; b < GFBITS; b++) {
        uint64_t bit = (in_radix[b] >> c_reversal[tid]) & 1ULL;
        cur[tid][b] = d_vec_setbits(bit);
    }
    __syncthreads();
    __syncthreads();

    // ---- butterflies with stage constants (parallel per stage) ----
    int consts_ptr = 0;
    for (int st = 0; st <= 5; st++) {
        int s = 1 << st;

        // compute block start for this thread's pair
        int blockStart = (tid / (2*s)) * (2*s);
        bool is_lower = (tid - blockStart) < s;

        if (is_lower) {
            int k  = tid;
            int kp = k + s;
            int off = k - blockStart;

            // tmp = cur[k+s] * consts[consts_ptr + off]
            uint64_t tmp[GFBITS];
            d_vec_mul(tmp, cur[kp], c_consts[consts_ptr + off]);

            // out_low  = cur[k]   ^ tmp
            // out_high = cur[k+s] ^ out_low
            #pragma unroll
            for (int b=0;b<GFBITS;b++) {
                uint64_t low  = cur[k][b]  ^ tmp[b];
                uint64_t high = cur[kp][b] ^ low;
                nxt[k][b]  = low;
                nxt[kp][b] = high;
            }
        }
        __syncthreads();

        // swap: copy nxt -> cur in parallel
        #pragma unroll
        for (int b=0;b<GFBITS;b++) cur[tid][b] = nxt[tid][b];
        __syncthreads();

        consts_ptr += s;
    }

    // ---- add powers contribution & write out ----
    #pragma unroll
    for (int b=0;b<GFBITS;b++) {
        uint64_t v = cur[tid][b] ^ c_powers[tid][b];
        out64x[tid*GFBITS + b] = v;
    }
}
```

Figure 4.31 k_fft

Function: k_fft

Purpose: Compute a single 64×GFBITS additive FFT (AFFT) with radix conversions and stage constants.

Parameters:

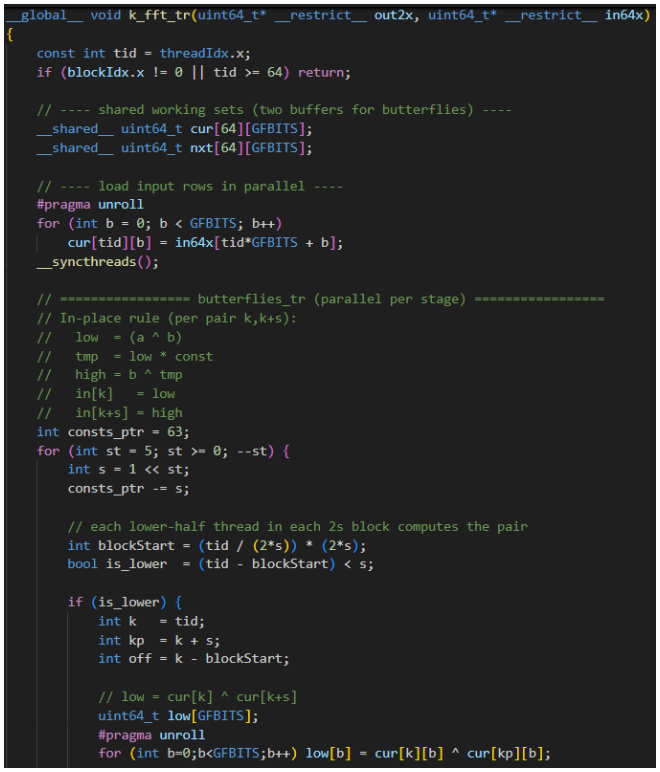
- vec* out64x: output matrix 64×GFBITS.
- vec* inGFBITS: input vector GFBITS (bitsliced).

Return value:

- void (writes out64x[tid*GFBITS + b]).

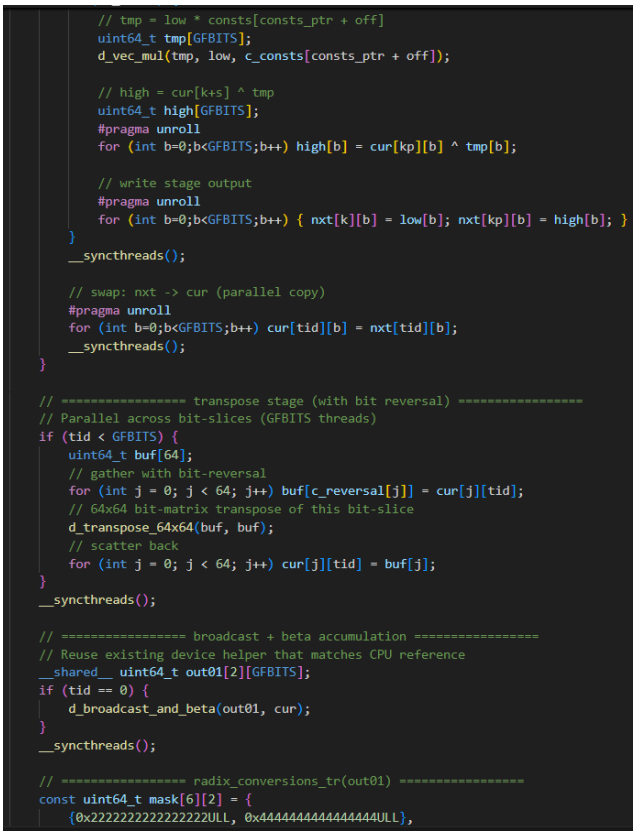
Explaination:

One cooperative block of 64 threads. Thread 0 performs radix conversions and scaling into in_radix. All threads broadcast into cur[64][GFBITS], run six butterfly stages using c_consts, then XOR c_powers and store. Shared memory buffers (cur/nxt) keep accesses coalesced.



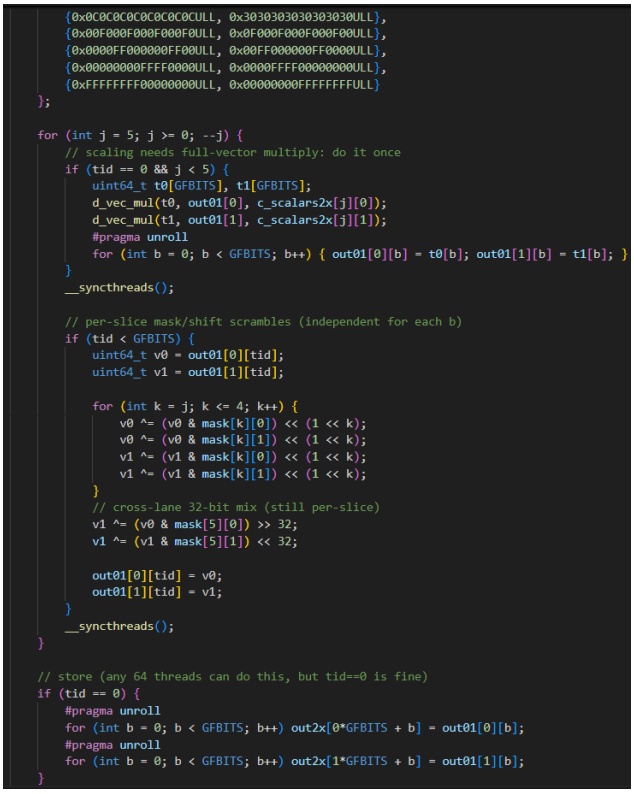
```
__global__ void k_fft_tr(uint64_t* __restrict__ out2x, uint64_t* __restrict__ in64x)
{
    const int tid = threadIdx.x;
    if (blockIdx.x != 0 || tid >= 64) return;

    // ---- shared working sets (two buffers for butterflies) ----
    __shared__ uint64_t cur[64][GFBITS];
    __shared__ uint64_t nxt[64][GFBITS];

    // ---- load input rows in parallel ----
    #pragma unroll
    for (int b = 0; b < GFBITS; b++)
        cur[tid][b] = in64x[tid*GFBITS + b];
    __syncthreads();

    // ================ butterflies_tr (parallel per stage) ================
    // In-place rule (per pair k,k+s):
    //   low  = (a ^ b)
    //   tmp  = low * const
    //   high = b ^ tmp
    //   in[k]   = low
    //   in[k+s] = high
    int consts_ptr = 63;
    for (int st = 5; st >= 0; --st) {
        int s = 1 << st;
        consts_ptr -= s;

        // each lower-half thread in each 2s block computes the pair
        int blockStart = (tid / (2*s)) * (2*s);
        bool is_lower  = (tid - blockStart) < s;

        if (is_lower) {
            int k   = tid;
            int kp  = k + s;
            int off = k - blockStart;

            // low = cur[k] ^ cur[k+s]
            uint64_t low[GFBITS];
            #pragma unroll
            for (int b=0;b<GFBITS;b++) low[b] = cur[k][b] ^ cur[kp][b];
```

Figure 4.32 k_fft_tr (i)



```
            // tmp = low * consts[consts_ptr + off]
            uint64_t tmp[GFBITS];
            d_vec_mul(tmp, low, c_consts[consts_ptr + off]);

            // high = cur[k+s] ^ tmp
            uint64_t high[GFBITS];
            #pragma unroll
            for (int b=0;b<GFBITS;b++) high[b] = cur[kp][b] ^ tmp[b];

            // write stage output
            #pragma unroll
            for (int b=0;b<GFBITS;b++) { nxt[k][b] = low[b]; nxt[kp][b] = high[b]; }
        }
        __syncthreads();

        // swap: nxt -> cur (parallel copy)
        #pragma unroll
        for (int b=0;b<GFBITS;b++) cur[tid][b] = nxt[tid][b];
        __syncthreads();
    }

    // ================ transpose stage (with bit reversal) ================
    // Parallel across bit-slices (GFBITS threads)
    if (tid < GFBITS) {
        uint64_t buf[64];
        // gather with bit-reversal
        for (int j = 0; j < 64; j++) buf[c_reversal[j]] = cur[j][tid];
        // 64x64 bit-matrix transpose of this bit-slice
        d_transpose_64x64(buf, buf);
        // scatter back
        for (int j = 0; j < 64; j++) cur[j][tid] = buf[j];
    }
    __syncthreads();

    // ================ broadcast + beta accumulation ================
    // Reuse existing device helper that matches CPU reference
    __shared__ uint64_t out01[2][GFBITS];
    if (tid == 0) {
        d_broadcast_and_beta(out01, cur);
    }
    __syncthreads();

    // ================ radix_conversions_tr(out01) ================
    const uint64_t mask[6][2] = {
        {0x2222222222222222ULL, 0x4444444444444444ULL},
```

Figure 4.33 k_fft_tr (ii)



```
    {0x0C0C0C0C0C0C0C0CULL, 0x3030303030303030ULL},
    {0x00F000F000F000F0ULL, 0x0F000F000F000F00ULL},
    {0x0000FF000000FF00ULL, 0x00FF000000FF0000ULL},
    {0x00000000FFFF0000ULL, 0x0000FFFF00000000ULL},
    {0xFFFFFFFF00000000ULL, 0x00000000FFFFFFFFULL}
};

for (int j = 5; j >= 0; --j) {
    // scaling needs full-vector multiply: do it once
    if (tid == 0 && j < 5) {
        uint64_t t0[GFBITS], t1[GFBITS];
        d_vec_mul(t0, out01[0], c_scalars2x[j][0]);
        d_vec_mul(t1, out01[1], c_scalars2x[j][1]);
        #pragma unroll
        for (int b = 0; b < GFBITS; b++) { out01[0][b] = t0[b]; out01[1][b] = t1[b]; }
    }
    __syncthreads();

    // per-slice mask/shift scrambles (independent for each b)
    if (tid < GFBITS) {
        uint64_t v0 = out01[0][tid];
        uint64_t v1 = out01[1][tid];

        for (int k = j; k <= 4; k++) {
            v0 ^= (v0 & mask[k][0]) << (1 << k);
            v0 ^= (v0 & mask[k][1]) << (1 << k);
            v1 ^= (v1 & mask[k][0]) << (1 << k);
            v1 ^= (v1 & mask[k][1]) << (1 << k);
        }
        // cross-lane 32-bit mix (still per-slice)
        v1 ^= (v0 & mask[5][0]) >> 32;
        v1 ^= (v1 & mask[5][1]) << 32;

        out01[0][tid] = v0;
        out01[1][tid] = v1;
    }
    __syncthreads();
}

// store (any 64 threads can do this, but tid==0 is fine)
if (tid == 0) {
    #pragma unroll
    for (int b = 0; b < GFBITS; b++) out2x[0*GFBITS + b] = out01[0][b];
    #pragma unroll
    for (int b = 0; b < GFBITS; b++) out2x[1*GFBITS + b] = out01[1][b];
}
```

Figure 4.34 k_fft_tr (iii)

Function: k_fft_tr

Purpose: Transpose FFT (TAFFT): transform in64x (64×GFBITS) into two GFBITS outputs matching CPU fft_tr.

Parameters:

- uint64_t* out2x: output 2×GFBITS.
- uint64_t* in64x: input 64×GFBITS.

Return value:

- void (writes out2x[0..2*GFBITS-1]).

Explaination:

Single block, 64 threads. Runs reverse butterflies with c_consts, performs per-slice 64×64 bit transposes with bit-reversal, executes *d_broadcast_and_beta* to produce the two streams, then applies radix-conversion scrambles (c_scalars2x). Outputs byte-exact halves used for BM/compare.

```
__global__ void k_sq_rows(uint64_t* __restrict__ eval64x) {
    int stride = blockDim.x * gridDim.x;
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < 64; i += stride) {
        uint64_t *row = eval64x + i*GFBITS;
        uint64_t tmp[GFBITS];
        d_vec_sq(tmp, row);
        #pragma unroll
        for (int b=0; b<GFBITS; ++b) row[b] = tmp[b];
    }
}
```

Figure 4.35 k_sq_rows

Function: k_sq_rows

Purpose: Square each row of the 64×GFBITS matrix in GF(2^m).

Parameters:

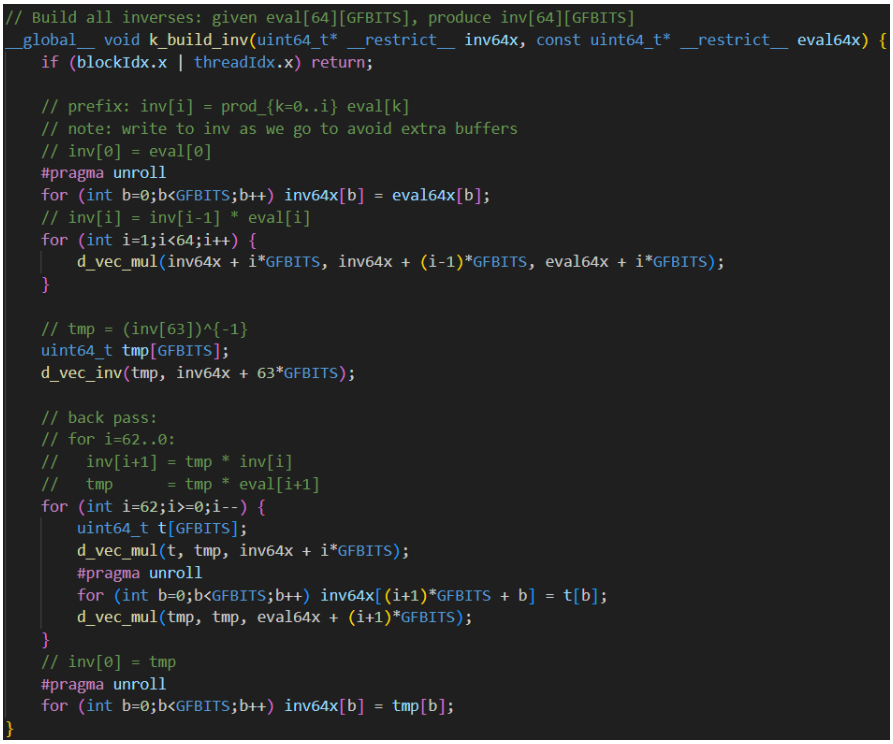
- uint64_t* eval64x: in-place matrix (row = eval64x + i*GFBITS).

Return value:

- void (update rows by squaring)

Explaination:

Grid-stride over rows; computes tmp = sq(row) via d_vec_sq then writes back. Used in the AFFT chain where squaring is a linear lane permutation.



```
// Build all inverses: given eval[64][GFBITS], produce inv[64][GFBITS]
__global__ void k_build_inv(uint64_t* __restrict__ inv64x, const uint64_t* __restrict__ eval64x) {
    if (blockIdx.x | threadIdx.x) return;

    // prefix: inv[i] = prod_{k=0..i} eval[k]
    // note: write to inv as we go to avoid extra buffers
    // inv[0] = eval[0]
    #pragma unroll
    for (int b=0;b<GFBITS;b++) inv64x[b] = eval64x[b];
    // inv[i] = inv[i-1] * eval[i]
    for (int i=1;i<64;i++) {
        d_vec_mul(inv64x + i*GFBITS, inv64x + (i-1)*GFBITS, eval64x + i*GFBITS);
    }

    // tmp = (inv[63])^{-1}
    uint64_t tmp[GFBITS];
    d_vec_inv(tmp, inv64x + 63*GFBITS);

    // back pass:
    // for i=62..0:
    //   inv[i+1] = tmp * inv[i]
    //   tmp      = tmp * eval[i+1]
    for (int i=62;i>=0;i--) {
        uint64_t t[GFBITS];
        d_vec_mul(t, tmp, inv64x + i*GFBITS);
        #pragma unroll
        for (int b=0;b<GFBITS;b++) inv64x[(i+1)*GFBITS + b] = t[b];
        d_vec_mul(tmp, tmp, eval64x + (i+1)*GFBITS);
    }
    // inv[0] = tmp
    #pragma unroll
    for (int b=0;b<GFBITS;b++) inv64x[b] = tmp[b];
}
```

Figure 4.36 k_build_inv

Function: k_build_inv

Purpose: Compute all inverses inv[i] using prefix/suffix trick.

Parameters:

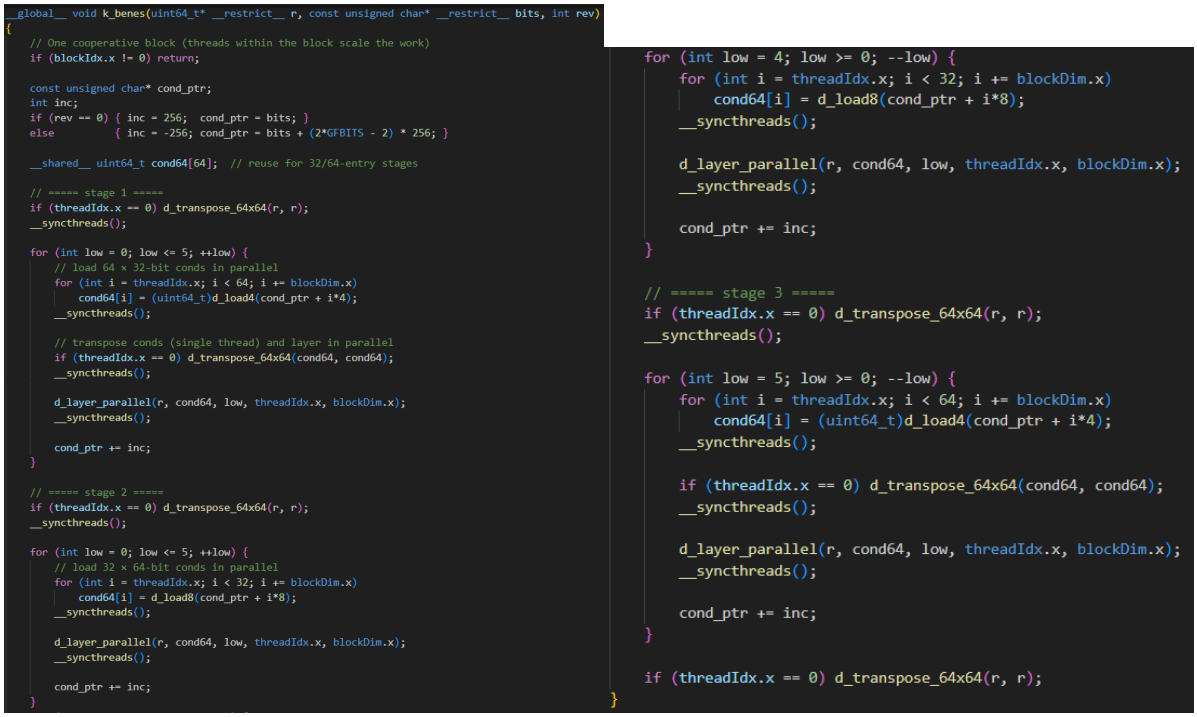
- uint64_t* inv64x: output 64×GFBITS.
- const uint64_t* eval64x: input 64×GFBITS

Return value:

- void (fills inv64x).

Explaination:

Single-thread kernel builds forward products into inv, inverts the last with *d_vec_inv*, then walks backward: writes inv[i+1] = tmp*inv[i], updates tmp *= eval[i+1], and finally sets inv[0]=tmp. Avoids extra buffers and matches reference schedule.



```
__global__ void k_benes(uint64_t* __restrict__ r, const unsigned char* __restrict__ bits, int rev)
{
    // One cooperative block (threads within the block scale the work)
    if (blockIdx.x != 0) return;

    const unsigned char* cond_ptr;
    int inc;
    if (rev == 0) { inc = 256;  cond_ptr = bits; }
    else          { inc = -256; cond_ptr = bits + (2*GFBITS - 2) * 256; }

    __shared__ uint64_t cond64[64];  // reuse for 32/64-entry stages

    // ===== stage 1 =====
    if (threadIdx.x == 0) d_transpose_64x64(r, r);
    __syncthreads();

    for (int low = 0; low <= 5; ++low) {
        // load 64 x 32-bit conds in parallel
        for (int i = threadIdx.x; i < 64; i += blockDim.x)
            cond64[i] = (uint64_t)d_load4(cond_ptr + i*4);
        __syncthreads();

        // transpose conds (single thread) and layer in parallel
        if (threadIdx.x == 0) d_transpose_64x64(cond64, cond64);
        __syncthreads();

        d_layer_parallel(r, cond64, low, threadIdx.x, blockDim.x);
        __syncthreads();

        cond_ptr += inc;
    }

    // ===== stage 2 =====
    if (threadIdx.x == 0) d_transpose_64x64(r, r);
    __syncthreads();

    for (int low = 0; low <= 5; ++low) {
        // load 32 x 64-bit conds in parallel
        for (int i = threadIdx.x; i < 32; i += blockDim.x)
            cond64[i] = d_load8(cond_ptr + i*8);
        __syncthreads();

        d_layer_parallel(r, cond64, low, threadIdx.x, blockDim.x);
        __syncthreads();

        cond_ptr += inc;
    }

    for (int low = 4; low >= 0; --low) {
        for (int i = threadIdx.x; i < 32; i += blockDim.x)
            cond64[i] = d_load8(cond_ptr + i*8);
        __syncthreads();

        d_layer_parallel(r, cond64, low, threadIdx.x, blockDim.x);
        __syncthreads();

        cond_ptr += inc;
    }

    // ===== stage 3 =====
    if (threadIdx.x == 0) d_transpose_64x64(r, r);
    __syncthreads();

    for (int low = 5; low >= 0; --low) {
        for (int i = threadIdx.x; i < 64; i += blockDim.x)
            cond64[i] = (uint64_t)d_load4(cond_ptr + i*4);
        __syncthreads();

        if (threadIdx.x == 0) d_transpose_64x64(cond64, cond64);
        __syncthreads();

        d_layer_parallel(r, cond64, low, threadIdx.x, blockDim.x);
        __syncthreads();

        cond_ptr += inc;
    }

    if (threadIdx.x == 0) d_transpose_64x64(r, r);
}
```

Figure 4.37 k_benes

Function: k_benes

Purpose:

Apply the Benes permutation (forward or inverse) to 64 lanes using the secret bit schedule.

Parameters:

- uint64_t* r: in-place 64-lane vector.
- const unsigned char* bits: packed control bits.
- int rev: 0 for forward, non-zero for reverse.

Return value:

- void (updates r).

Explaination:

One block cooperates through three stages separated by 64×64 transposes. Each stage loads control words *(d_load4/d_load8*), optionally transposes them, then calls *d_layer_parallel* with stride 2^low to do 32 conditional swaps per layer. rev selects pointer direction/stride.

```cuda
__global__ void k_synd_cmp(const uint64_t* __restrict__ s0_2x,
                           const uint64_t* __restrict__ s1_2x,
                           unsigned short* __restrict__ out_ok,
                           unsigned int* __restrict__ d_flag,
                           unsigned int* __restrict__ d_done)
{
    // grid-stride OR over the 2*GFBITS words (e.g., 24)
    uint64_t local = 0;
    int stride = blockDim.x * gridDim.x;
    for (int idx = blockIdx.x*blockDim.x + threadIdx.x; idx < 2*GFBITS; idx += stride) {
        local |= (s0_2x[idx] ^ s1_2x[idx]);
    }

    // If any bit differs in this thread's chunk, set the global flag to 1
    if (local) atomicOr(d_flag, 1u);

    __syncthreads(); // intra-block

    // Last block writes out_ok (no host round-trip needed)
    if (threadIdx.x == 0) {
        unsigned int ticket = atomicAdd(d_done, 1u);  // returns old value
        if (ticket == gridDim.x - 1) {
            *out_ok = (*d_flag == 0u) ? 1 : 0;
        }
    }
}
```

Figure 4.38 k_synd_cmp

Function: k_synd_cmp

Purpose: Check equality of two 2×GFBITS syndrome halves and set a single success flag.

Parameters:

- const uint64_t* s0_2x, const uint64_t* s1_2x: inputs to compare.
- unsigned short* out_ok: output 1 if equal, else 0.
- unsigned int* d_flag: global OR flag (set to 1 if any diff).
- unsigned int* d_done: completion counter for block-local reduction.

Return value:

- void (sets *out_ok once per grid).

Explaination:

Grid-stride OR-reduces (s0^s1) into d_flag. The last finishing block (ticket pattern on d_done) writes *out_ok = (d_flag==0). Avoids host round-trip and provides a single boolean result.

```
// Compute w0 (popcount over all 4096 support bits) and w1 (popcount over first SYS_N bits)
__global__ void k_weight_check(const uint64_t* __restrict__ Err64,   // 64 lanes
                               const unsigned char* __restrict__ E,  // packed bits (4096/8 bytes)
                               unsigned int* __restrict__ out_w0,
                               unsigned int* __restrict__ out_w1)
{
    unsigned int local_w0 = 0;

    // Sum popcounts over 64 x 64-bit words (Err64[0..63])
    int stride = blockDim.x * gridDim.x;
    for (int i = blockIdx.x*blockDim.x + threadIdx.x; i < 64; i += stride) {
        local_w0 += __popcll(Err64[i]);
    }
                                                Loading...
    // Sum popcounts over first SYS_N bits of E (mask final byte)
    int total_bytes = (SYS_N + 7) >> 3;          // ceil(SYS_N / 8)
    int last_bits   =  SYS_N       & 7;          // remaining bits in last byte (0..7)
    unsigned int local_w1 = 0;

    for (int i = blockIdx.x*blockDim.x + threadIdx.x; i < total_bytes; i += stride) {
        unsigned char b = E[i];
        if (last_bits && (i == total_bytes - 1)) {
            b &= (unsigned char)((1u << last_bits) - 1u);
        }
        local_w1 += __popc((unsigned int)b);
    }

    // Atomically accumulate to outputs
    if (local_w0) atomicAdd(out_w0, local_w0);
    if (local_w1) atomicAdd(out_w1, local_w1);
}
```

Figure 4.39 k_weight_check

Function: k_weight_check

Purpose: Compute Hamming weights: w0 over all 4096 support bits, w1 over the first SYS_N bits.

Parameters:

- const uint64_t* Err64: 64 error lanes.
- const unsigned char* E: packed error vector (bytes).
- unsigned int* out_w0, unsigned int* out_w1: accumulators.

Return value:

- void (atomically adds to output)

Explaination:

Grid-stride popcounts: w0 via __popcll over 64 words; w1 via byte popcounts on E[0..ceil(SYS_N/8)-1] with a final-byte mask. Atomically accumulates partial sums to the global outputs.

4.2.3 Host Functions

```
int crypto_kem_enc_new(unsigned char *c, unsigned char *key, const unsigned char *pk)
{
    // e: error vector (SYS_N bits => SYS_N/8 bytes)
    unsigned char e[SYS_N/8];
    // one_ec: 1 || e || c  (first byte set to 1, rest zeroed)
    unsigned char one_ec[1 + SYS_N/8 + SYND_BYTES] = {1};

    // Run your CUDA encrypt to fill c (syndrome) and e (error vector)
    // Note: your encrypt writes: s -> c (SYND_BYTES), e -> error vector
    memset(e, 0, sizeof(e));     // good hygiene; encrypt should set bits anyway
    encrypt(c, pk, e);

    // Concatenate: 1 || e || c
    memcpy(one_ec + 1, e, SYS_N/8);
    memcpy(one_ec + 1 + SYS_N/8,  c, SYND_BYTES);

    // Derive 32-byte shared key via SHAKE256
    crypto_hash_32b(key, one_ec, sizeof(one_ec));

    return 0;
}
```

Figure 4.40 crypto_kem_enc_new

Function: crypto_kem_enc_new

Purpose: Top-level encapsulation API: builds the error vector e on GPU, computes ciphertext c (syndrome), then derives the shared secret key with SHAKE256.

Parameters:

- unsigned char *c: output ciphertext (SYND_BYTES).
- unsigned char *key: output shared secret (32 bytes).
- const unsigned char *pk: input public key

Explaination:

Calls the GPU encrypt pipeline (candidate load → filter/sort → set_bits → syndrome_kernel_vecstyle), copies c back, forms preimage 1||e||c, and hashes via crypto_hash_32b to produce key. Used by main.cpp for correctness and by the benchmark loop for CSV timing.

```c
int crypto_kem_dec(unsigned char *key, const unsigned char *c,const unsigned char *sk)
{
    int i;
    unsigned char ret_decrypt = 0;
    uint16_t m;

    unsigned char e[SYS_N/8];
    unsigned char preimage[1 + SYS_N/8 + SYND_BYTES];
    unsigned char *x = preimage;
    const unsigned char *s = sk + 40 + IRR_BYTES + COND_BYTES;

    /* Niederreiter decrypt: returns 0 on success, 1 on failure */
    ret_decrypt = decrypt(e, sk + 40, c);

    /* m = 0xFF on success, 0x00 on failure */
    m  = ret_decrypt;
    m -= 1;
    m >>= 8;

    *x++ = (unsigned char)(m & 1);

    for (i = 0; i < SYS_N/8; i++)
        *x++ = (unsigned char)(((~m) & s[i]) | (m & e[i]));

    for (i = 0; i < SYND_BYTES; i++)
        *x++ = c[i];

    crypto_hash_32b(key, preimage, sizeof(preimage));

    return 0;
}
```

Figure 4.41 crypto_kem_dec

Function: crypto_kem_dec

Purpose: Top-level decapsulation API: uses decrypt to recover e', then derives the shared secret with CCA-style masking.

Parameters:

- unsigned char *key: output shared secret (32 bytes).
- const unsigned char *c: input ciphertext (SYND_BYTES).
- const unsigned char *sk: secret key.

Explaination:

Calls decrypt to get e' and a success bit (from k_synd_cmp + weights). Builds preimage (1||e'||c) on success, otherwise (0||s||c) (with s embedded in sk), and hashes via crypto_hash_32b to produce key. Ensures CCA-secure behavior and matches the vec reference convention.

```
// ============================ Host decrypt ===================================
extern "C"
int decrypt(unsigned char *e, const unsigned char *sk, const unsigned char *s)
{
    // Load __constant__ FFT tables (once per process)
    fft_tables_cuda_init();

    using vec = uint64_t;

    // ============= HOST WORK BUFFERS ============
    vec        h_irr_int[GFBITS];               // filled by irr_load(...)
    uint64_t   h_s_priv[2][GFBITS];             // filled from d_s_priv
    uint64_t   locator[GFBITS];                 // filled by bm(...)
    unsigned short h_ok = 0;                    // overwritten by cudaMemcpy from d_ok
    unsigned char  e_full[(1<<GFBITS)/8];       // filled by cudaMemcpy from d_e
    unsigned int   h_w0 = 0, h_w1 = 0;          // overwritten by cudaMemcpy from d_w0/d_w1


    // ==================== ALL DEVICE ALLOS UP FRONT ====================
    unsigned char *d_s=nullptr, *d_bits=nullptr, *d_e=nullptr;
    vec *d_recv=nullptr, *d_inv=nullptr, *d_scaled=nullptr, *d_eval=nullptr, *d_error=nullptr,
        *d_s_priv=nullptr, *d_s_priv_cmp=nullptr, *d_irr=nullptr, *d_locator=nullptr;
    unsigned short *d_ok=nullptr;
    unsigned int *d_flag = nullptr, *d_done = nullptr;
    unsigned int *d_w0=nullptr, *d_w1=nullptr;

    // ==================== DEVICE ALLOCATIONS ====================
    cudaMalloc(&d_s,       SYND_BYTES);
    cudaMalloc(&d_bits,    COND_BYTES);
    cudaMalloc(&d_e,       ((1<<GFBITS)/8));

    cudaMalloc(&d_recv,        64*sizeof(vec));
    cudaMalloc(&d_inv,   64*GFBITS*sizeof(vec));
    cudaMalloc(&d_scaled,64*GFBITS*sizeof(vec));
    cudaMalloc(&d_eval,  64*GFBITS*sizeof(vec));
    cudaMalloc(&d_error,       64*sizeof(vec));
    cudaMalloc(&d_s_priv,   2*GFBITS*sizeof(vec));
    cudaMalloc(&d_s_priv_cmp, 2*GFBITS*sizeof(vec));
    cudaMalloc(&d_irr,          GFBITS*sizeof(vec));
    cudaMalloc(&d_locator,      GFBITS*sizeof(vec));

    cudaMalloc(&d_ok,   sizeof(unsigned short));
    cudaMalloc(&d_flag, sizeof(unsigned int));
    cudaMalloc(&d_done, sizeof(unsigned int));
    cudaMalloc(&d_w0,   sizeof(unsigned int));
    cudaMalloc(&d_w1,   sizeof(unsigned int));
```

Figure 4.42 decrypt (i)

```
// ==================== ALL INITIAL H2D COPIES ====================
// s -> device
cudaMemcpy(d_s, s, SYND_BYTES, cudaMemcpyHostToDevice);

// Benes bits from secret key
cudaMemcpy(d_bits, sk + IRR_BYTES, COND_BYTES, cudaMemcpyHostToDevice);

// irr (bitsliced) -> device
irr_load(reinterpret_cast<uint64_t*>(h_irr_int), sk);
cudaMemcpy(d_irr, h_irr_int, GFBITS*sizeof(vec), cudaMemcpyHostToDevice);

// ================== Timing Setup ==============================
time_t now = time(0);
tm *ltm = localtime(&now);
char timestamp[64];
strftime(timestamp, sizeof(timestamp), "%Y-%m-%d_%H:%M:%S", ltm);

std::ostringstream oss;
oss << "results_dec/decrypt_" << timestamp << ".csv";
std::string filename = oss.str();

FILE *log_file = fopen(filename.c_str(), "w");
if (!log_file) {
    fprintf(stderr, "Failed to open result file: %s\n", filename.c_str());
    // still continue the actual decrypt once with num_blocks=1
} else {
    fprintf(log_file, "num_blocks,trial,time_ms,throughput\n");
}

const float total_items = float(SYS_N) / 8.0f; // bytes processed

// ================== Num_Blocks Loop (Batch)==============================
for (int num_blocks = 1; num_blocks <= 32; num_blocks *= 2){
    printf("\n===== DECRYPT: testing with %d blocks =====\n", num_blocks);

    float total_ms = 0.0f, total_throughput = 0.0f;

    int trial;
    // ============ Trials Loop =============
    for (trial = 1; trial <= 10; ++trial){
        //Reset every trials
        cudaMemset(d_ok,   0, sizeof(unsigned short));
        cudaMemset(d_flag, 0, sizeof(unsigned int));
        cudaMemset(d_done, 0, sizeof(unsigned int));
        cudaMemset(d_w0, 0, sizeof(unsigned int));
        cudaMemset(d_w1, 0, sizeof(unsigned int));
```

Figure 4.43 decrypt (ii)

```
// timer
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

// ==================== PIPELINE ====================
// Preprocess: syndrome -> recv
k_preprocess<<<num_blocks,64>>>(d_recv, d_s);

// Benes forward (Only 1 block)
k_benes<<<1,64>>>((uint64_t*)d_recv, d_bits, /*rev=*/1);

// FFT(irr), square rows, build inv (1 block only)
k_fft<<<1,64>>>(d_eval, d_irr);
k_sq_rows<<<num_blocks,64>>>(d_eval);
k_build_inv<<<1,1>>>(d_inv, d_eval);

// scaled = inv & recv
k_and_scale<<<num_blocks, 64>>>(d_scaled, d_inv, d_recv);

// fft_tr(s_priv, scaled) on GPU (1 block)
k_fft_tr<<<1,64>>>(d_s_priv, d_scaled);

// ---------- BM on CPU -> to Device ----------
cudaMemcpy(h_s_priv, d_s_priv, 2*GFBITS*sizeof(uint64_t), cudaMemcpyDeviceToHost);
bm(locator, h_s_priv);
cudaMemcpy(d_locator, locator, GFBITS*sizeof(uint64_t), cudaMemcpyHostToDevice);

// FFT(eval, locator) 1 block only
k_fft<<<1,64>>>(d_eval, d_locator);

// form error
k_form_error<<<num_blocks,64>>>(d_error, d_eval);
cudaDeviceSynchronize();

// scaled = inv & error
k_and_scale<<<num_blocks, 64>>>(d_scaled, d_inv, d_error);

// fft_tr(s_priv_cmp, scaled) + device compare
k_fft_tr<<<1,64>>>(d_s_priv_cmp, d_scaled);
k_synd_cmp<<<num_blocks,64>>>((const uint64_t*)d_s_priv, (const uint64_t*)d_s_priv_cmp, d_ok, d_flag, d_done);

// inverse Benes on error (Only 1 block)
k_benes<<<1,64>>>((uint64_t*)d_error, d_bits, /*rev=*/0);
```

Figure 4.44 decrypt (iii)

```
        // postprocess: pack error to bytes
        k_postprocess<<<num_blocks,64>>>(d_e, d_error);

        // weight check
        k_weight_check<<<num_blocks,64>>>((uint64_t*)d_error, d_e, d_w0, d_w1);

        // stop timer
        cudaEventRecord(stop);
        cudaEventSynchronize(stop);

        float ms = 0.0f;
        cudaEventElapsedTime(&ms, start, stop);
        float seconds = ms / 1000.0f;
        float throughput = (total_items * num_blocks) / (seconds > 0 ? seconds : 1e-9f); //Bytes processed / sec

        printf("Trial %d: Time = %.6f ms | Throughput = %.2f items/s\n", trial, ms, throughput);
        if (log_file) fprintf(log_file, "%d,%d,%.6f,%.2f\n", num_blocks, trial, ms, throughput);

        total_ms += ms;
        total_throughput += throughput;

        cudaEventDestroy(start);
        cudaEventDestroy(stop);

        // bring back flags and e for invariance check
        cudaMemcpy(&h_ok, d_ok, sizeof(unsigned short), cudaMemcpyDeviceToHost);
        cudaMemcpy(&h_w0, d_w0, sizeof(unsigned int),  cudaMemcpyDeviceToHost);
        cudaMemcpy(&h_w1, d_w1, sizeof(unsigned int),  cudaMemcpyDeviceToHost);

        cudaMemcpy(e_full, d_e, (1<<GFBITS)/8, cudaMemcpyDeviceToHost);
    }//trials loop

    float avg_ms = total_ms / float(trial-1);
    float avg_throughput = total_throughput / float(trial-1);
    printf("Average for %d blocks: Time = %.6f ms | Throughput = %.2f items/s\n", num_blocks, avg_ms, avg_throughput);
    if (log_file) fprintf(log_file, "%d,avg,%.6f,%.2f\n", num_blocks, avg_ms, avg_throughput);

}//NumBlocks loop

if (log_file) fclose(log_file);

// set caller's e[] from the first (reference) run
memcpy(e, e_full, SYS_N/8);

// final correctness flags from the last run (should be stable)
```

Figure 4.45 decrypt (iv)

```
    }//NumBlocks loop

    if (log_file) fclose(log_file);

    // set caller's e[] from the first (reference) run
    memcpy(e, e_full, SYS_N/8);

    // final correctness flags from the last run (should be stable)
    uint16_t check = ((h_w0 ^ SYS_T) | (h_w1 ^ SYS_T));
    check -= 1; check >>= 15;
    uint16_t check_weight = check;

    // Bring back the 1-bit result
    cudaMemcpy(&h_ok, d_ok, sizeof(unsigned short), cudaMemcpyDeviceToHost);
    uint16_t check_synd = h_ok; // 1==equal

    // ---------- Results (compare with encrypt) ----------
    {
        int k;
        printf("\nDecrypt e: positions");
        for (k = 0; k < SYS_N; ++k)
            if (e[k/8] & (1u << (k & 7)))
                printf(" %d", k);
        printf("\n");
    }
    {
        printf("\nSyndrome s (Decrypt): ");
        for (int i = 0; i < SYND_BYTES; i++) {
            printf("%02X", s[i]);
            if ((i + 1) % 16 == 0) printf("\n");
            else printf(" ");
        }
        printf("\n");
    }

    unsigned char ret = 1 - (check_synd & check_weight); // 0=success, 1=failure

    // ==================== cudaFree ====================
    cudaFree(d_s); cudaFree(d_bits); cudaFree(d_e);
    cudaFree(d_recv); cudaFree(d_inv); cudaFree(d_scaled); cudaFree(d_eval); cudaFree(d_error);
    cudaFree(d_s_priv); cudaFree(d_s_priv_cmp); cudaFree(d_irr); cudaFree(d_locator);
    cudaFree(d_ok); cudaFree(d_flag); cudaFree(d_done); cudaFree(d_w0); cudaFree(d_w1);

    return ret;
}
```

Figure 4.46 decrypt (v)

Function: decrypt

Purpose: Recover the error vector e from ciphertext s using the secret key sk (GPU + CPU hybrid).

Parameters:

- unsigned char *e: output packed error vector (SYS_N/8).
- const unsigned char *sk: secret key (Benes bits, Goppa data).
- const unsigned char *s: input ciphertext (SYND_BYTES).

Explaination:

Runs GPU decrypt: k_preprocess → k_benes → k_fft/k_sq_rows/k_build_inv → k_and_scale → k_fft_tr, then copies s_priv halves to CPU and calls BM to build the locator. Locator is sent back to GPU for k_form_error, re-syndrome/compare (k_synd_cmp), inverse Benes + pack (k_postprocess), and weight checks (k_weight_check). On success, fills e.

Function: bm(locator, h_s_priv) in CPU

Purpose: CPU Berlekamp–Massey to compute the error locator polynomial from the two syndrome halves.

Parameters:

- uint64_t locator[GFBITS]: variable to output locator polynomial in bitsliced/word form
- uint64_t h_s_priv 2[GFBITS]: variable to s_priv halves copied from GPU

Explaination:

Runs sequential BM over GF(2^m) using the bitsliced representation to produce the locator used by GPU root-finding. Kept on CPU to simplify control flow and avoid GPU divergence.

Function: crypto_hash_32b

Purpose: Derive a 32-byte shared secret via SHAKE256 from a preimage (tag || payload).

Parameters:

- unsigned char *out: 32-byte output.
- const unsigned char *in: preimage buffer (1||e||c or 0||s||c).
- size_t inlen: preimage length.

Explaination:

Used in both encapsulation and decapsulation to KDF the final shared secret. Ensures identical secrets when decrypt succeeds (ss_enc == ss_dec).

```c
int main() {

    //Act as seed for random number generator
    for (int i=0; i<48; i++)
        entropy_input[i] = i;
    randombytes_init(entropy_input, NULL, 256);

    for (int i=0; i<KATNUM; i++){
        randombytes(seed[i], 48);
        randombytes_init(seed[i], NULL, 256);
    }

    //We dont run crypto_kem_keypair so have to push the randombytes stream
    randombytes(dummy, 32);

    // Local buffers for outputs (don't overwrite any expected vectors)
    unsigned char ct_local[crypto_kem_CIPHERTEXTBYTES];
    unsigned char ss_enc  [crypto_kem_BYTES];
    unsigned char ss_dec  [crypto_kem_BYTES];

    //Encapsulate with public key -> ct, ss
    if (crypto_kem_enc_new(ct_local, ss_enc, pk) != 0) {
        printf("crypto_kem_enc_new failed\n");
        return 1;
    }

    //Decapsulate with secret key -> ss1
    if (crypto_kem_dec(ss_dec, ct_local, sk) != 0) {
        printf("crypto_kem_dec failed (nonzero return)\n");
        return 1;
    }

    //Compare shared secrets
    if (std::memcmp(ss_enc, ss_dec, crypto_kem_BYTES) != 0) {
        printf("Mismatch: ss (enc) != ss1 (dec)\n");
        return 1;
    }

    printf("Success: ss (enc) == ss1 (dec)\n");
    return 0;
}
```

Figure 4.47 main.cpp

Function: main

Purpose: Correctness harness and demo.

Explaination:

Loads embedded pk/sk, calls crypto_kem_enc_new to get (c, ss_enc), then crypto_kem_dec to get ss_dec, compares secrets, and prints pass/fail. Optional code paths invoke the benchmark sweep to produce CSVs.

Chapter 5

System Implementation

5.1 Hardware Setup

The project uses a laptop as the primary hardware. The laptop's GPU will handle the parallelized implementation of the McEliece signature scheme, focusing on encryption, encapsulation, decryption and decapsulation through CUDA programming. This setup ensures efficient performance and scalability on resource-limited hardware. This configuration is also sufficient to host both the CPU vec reference code and the CUDA kernels for parallelized KEM operations.

Description	Specifications
Model	ASUS TUF Gaming FX505DT
Processor	AMD Ryzen 5 3550H
Operating System	Windows 10 Home – 64-bit
Graphic	NVIDIA GeForce GTX 1650 (4GB)
Memory	12GB DDR4 2400MHz
Storage	512GB Intel SSDPEKNW512G8 SSD

Table 5.1 Specifications of laptop

5.2 Software Setup

All development and testing were conducted on **Ubuntu 24.04.2 LTS**, using:

- **CUDA Toolkit 12.4** (driver version 550.144.03) for GPU compilation.
- **GCC** for compiling host C/C++ code.
- **nvcc** for building .cu files into device-executable kernels.
- **Makefile** build scripts to compile both CPU and GPU versions.
- **Jupyter Notebook (Python)** using Matplotlib for plotting performance graphs.

The environment ensured deterministic builds and allowed direct comparison between CPU (vec) and GPU outputs.

5.3 Setting and Configuration

Key runtime parameters include:

- SYS_T, SYS_N, PK_NROWS, and GFBITS constants set for *mceliece348864* parameter set.
- Kernel launch configurations tuned for performance, e.g. num_blocks ∈ [1, 32], threads_per_block = 64 or 128 depending on kernel.
- CPU reference implementation compiled and run serially for baseline timing.
- GPU kernels executed with warmup launches before timing to stabilize clock frequencies.
- Output verification enabled: each GPU run compared syndrome, error vector, and shared secret against CPU vec reference for correctness.

5.4 System Operation (With Screenshot)

The implemented CUDA-based McEliece KEM system was tested to ensure that both encapsulation and decapsulation procedures function correctly on the target hardware. A complete execution was performed involving the GPU kernels and host functions to generate the ciphertext and recover the shared secret key.

In the encapsulation step, the GPU first produces and filters random candidate error positions, sorts and eliminates duplicates, and builds the packed error vector. The syndrome is calculated via *syndrome_kernel_vecstyle*, and the resulting ciphertext *c*. The shared secret *ss_enc* is obtained from the hash of 1 || e || c preimage using SHAKE256 on the host.

Whereas in decapsulation, the ciphertext *c* is first loaded into bitsliced format by *k_preprocess*, followed by Benes network permutation, additive FFT, and inverse computations. After that, the CPU computes the locator polynomial using the Berlekamp–Massey algorithm, and then the GPU will mark error locations, rebuild the error vector, and recompute the syndrome. If it is verified, the shared secret *ss_dec* will be derived from the preimage 1 || e' || c.

Figure 5.1 and 5.2 shows the terminal output confirming that the encapsulation and decapsulation paths produce identical error vector, syndrome and shared secrets, proving correctness of the full pipeline.

```
Encrypt e: positions 0 81 92 105 216 378 389 468 504 586 617 738 928 1030 1061 1
070 1072 1177 1294 1316 1366 1389 1510 1670 1721 1728 1746 1783 1888 1893 1951 1
970 1995 2003 2053 2068 2082 2113 2202 2363 2418 2508 2541 2553 2603 2641 2704 2
731 2818 2951 2974 3091 3132 3157 3158 3174 3182 3215 3330 3348 3377 3382 3392 3
468

Syndrome s (Encrypt): DE F6 19 08 A7 0A 30 99 E4 5B 4D 5D 91 95 7A DE
70 F5 71 D2 10 D5 25 D6 55 DB 72 94 51 5F 91 D9
77 95 F2 35 36 15 BC 7C DF 13 50 21 81 E5 BC C8
C9 AB FE F3 18 19 D6 6D D2 76 03 63 69 4F 78 96
02 26 4A 3E 24 44 56 81 A0 18 3C E3 43 A2 26 4F
DF F9 6C 82 AB 31 8A E8 88 D1 05 D5 2D 59 BC 1B
```

Figure 5.1 Encrypt Results

```
Decrypt e: positions 0 81 92 105 216 378 389 468 504 586 617 738 928 1030 1061 1
070 1072 1177 1294 1316 1366 1389 1510 1670 1721 1728 1746 1783 1888 1893 1951 1
970 1995 2003 2053 2068 2082 2113 2202 2363 2418 2508 2541 2553 2603 2641 2704 2
731 2818 2951 2974 3091 3132 3157 3158 3174 3182 3215 3330 3348 3377 3382 3392 3
468

Syndrome s (Decrypt): DE F6 19 08 A7 0A 30 99 E4 5B 4D 5D 91 95 7A DE
70 F5 71 D2 10 D5 25 D6 55 DB 72 94 51 5F 91 D9
77 95 F2 35 36 15 BC 7C DF 13 50 21 81 E5 BC C8
C9 AB FE F3 18 19 D6 6D D2 76 03 63 69 4F 78 96
02 26 4A 3E 24 44 56 81 A0 18 3C E3 43 A2 26 4F
DF F9 6C 82 AB 31 8A E8 88 D1 05 D5 2D 59 BC 1B

Success: ss (enc) == ss1 (dec)
```

Figure 5.2 Decrypt Results & SS comparison

Benchmark data is collected automatically besides the correctness verification and are saved in CSV files for later analysis. Figure 5.3 shows a glimpse of the produced CSV file with results for different kernel launch configurations. The data discussed in this section are the basis for the upcoming performance evaluation in chapter 6.

num_block	trial	time_ms	throughput
1	1	1.115808	390748.2
1	2	0.811008	537602.6
1	3	0.761856	572286.6
1	4	0.739168	589852.4
1	5	0.765728	569392.8
1	6	0.758784	574603.6
1	7	0.755136	577379.4
1	8	0.746912	583736.8
1	9	0.739264	589775.8
1	10	0.73728	591362.9
1	avg	0.793094	557674.1
2	1	0.738976	1180011
2	2	0.750176	1162394
2	3	0.73728	1182726
2	4	0.73856	1180676
2	5	0.732704	1190112
2	6	0.741376	1176191
2	7	0.74752	1166524
2	8	0.736832	1183445
2	9	0.740864	1177004
2	10	0.739904	1178531
2	avg	0.740419	1177761
4	1	0.749536	2326773
4	2	0.864736	2016800
4	3	0.812256	2147107

Figure 5.3 Sample CSV results

This confirms system-level testing that functionality works as it should. Shared secret equality tests pass, and benchmark logs are generated for in-depth analysis. In-depth performance graphs and discussion are contained in Section 6.2.

5.5 Implementation Issues and Challenges

During development and testing, there are several issues that came up and need to be addressed with constant debugging and optimization. One of the earliest problems was wrong results from the syndrome computation kernel. After debugging, it was found that the issues lies in misaligned memory access, which makes the kernel's initial reading was from the wrong offset of the error vector. This was solved by offsetting the pointer to e_tail = e + SYND_BYTES, making sure that the GPU computed parity bits which is similar to the CPU vec reference.

Furthermore, the kernel's bit toggle operation was modified to use atomic_xor_bit to avoid race conditions when multiple threads attempted to update the syndrome byte concurrently.

Another issue is with the memory usage in k_fft_tr kernel, where the shared memory buffers were over the SM limit, causing failed launches on certain block sizes. The buffers were rebuilt and aligned to reduce shared memory pressure and avoid bank conflicts. This change makes it more stable and allowed consistent kernel launches across all the tested configurations.

Besides that, early versions of error vector produced duplicate or unmatching results. This problem is solved by integrating the *bitonic_sort_kernel* to sort possible indices and implementing *check_duplicates* kernel to flag and notify any sets that contain duplicates.

These challenges not only improved system stability but also deepened understanding of GPU memory behavior, synchronization, and deterministic execution which are all crucial for a cryptographic implementation where correctness and reproducibility are paramount.

5.6 Concluding Remark

This chapter provides a coherent picture of how the CUDA-accelerated McEliece KEM system fits together, from the hardware/software setup through runtime configuration to system operation. The resolution of each of the individual kernel correctness, memory optimization, and performance tuning issues finally culminated in a solid and righteously functioning solution. Thus, this system was able to generate valid ciphertexts, recover the shared secret, and provide benchmark results with which the performances could be compared.

Chapter 6 will summarize the formal testing and performance evaluation of the implemented system: comparison between CPU and GPU results, analysis of throughput trends over kernel launch configurations, and an overall assessment of how well the project objectives have been achieved.

Chapter 6

System Evaluation and Discussion

6.1 System Testing and Performance Metrics

The project was evaluated using **average execution time per operation** and **average throughput** as the key performance indicators. Execution time directly measures the computational efficiency of encryption and decryption, while throughput reflects how many items can be processed per second. Throughput was calculated using:

$$\text{Throughput} = \frac{\text{num\_blocks} \times \frac{\text{SYS\_N}}{8}}{\text{time\_ms} / 1000}$$

Where num_blocks represents the number of serial iterations on CPU or parallel blocks on GPU. For each num_block configuration, 10 trials were conducted, and the average results were used for consistency.

Correctness was validated by:

- Matching GPU shared secret (ss_enc) with (ss_dec).
- Matching GPU error vector and syndrome from encryption and decryption.
- Verifying that these GPU results are matched with the CPU vec reference.

6.2 Testing Setup and Results

Components	Details
CPU	AMD Ryzen 5 3550H
GPU	NVIDIA GeForce GTX 1650, 4 GB (laptop)
OS	Ubuntu 24.04.2 LTS
Driver	NVIDIA 550.144.03
CUDA	Version 12.4
Implementation	C/CUDA (nvcc compiled)

Table 6.1 Testing Setup

The CPU implementation was executed in serial loops equal to num_blocks, while the GPU implementation launched kernels with the specified number of thread blocks. This ensures fair comparison of equivalent workloads.

Results

The following tables will show the average results of the 10 trials run by each num_blocks configuration for both CPU and GPU.

CPU encrypt average results

Serial Loops	Time_ms	Throughput (items/sec)
1	0.044921	10917018.96
2	0.073283	13532555.18
4	0.139582	12782441.67
8	0.275987	12806684.73
16	0.560969	12542706.99
32	1.148872	12219651.07

Table 6.2 CPU encrypt results

CPU decrypt average results

Serial Loops	Time_ms	Throughput (items/sec)
1	0.173462	2515560.08
2	0.343000	2542354.51
4	0.692773	2517718.61
8	1.403712	2488658.68
16	2.894678	2418709.33
32	5.565257	2508264.32

Table 6.3 CPU decrypt results

GPU encrypt average results

Num_blocks	Time_ms	Throughput (items/sec)
1	0.192954	2314525.80
2	0.182693	4791434.80
4	0.177197	9871298.00
8	0.172177	20305654.40
16	0.168165	41614452.00
32	0.171911	81403555.20

Table 6.4 GPU encrypt results

GPU decrypt average results

Num_blocks	Time_ms	Throughput (items/sec)
1	0.799638	553932.70
2	0.748664	1165266.35
4	0.758231	2303712.95
8	0.749318	4657335.60
16	0.754025	9264162.40
32	0.754278	18515560.00

Table 6.5 GPU decrypt results

Graphical Representation

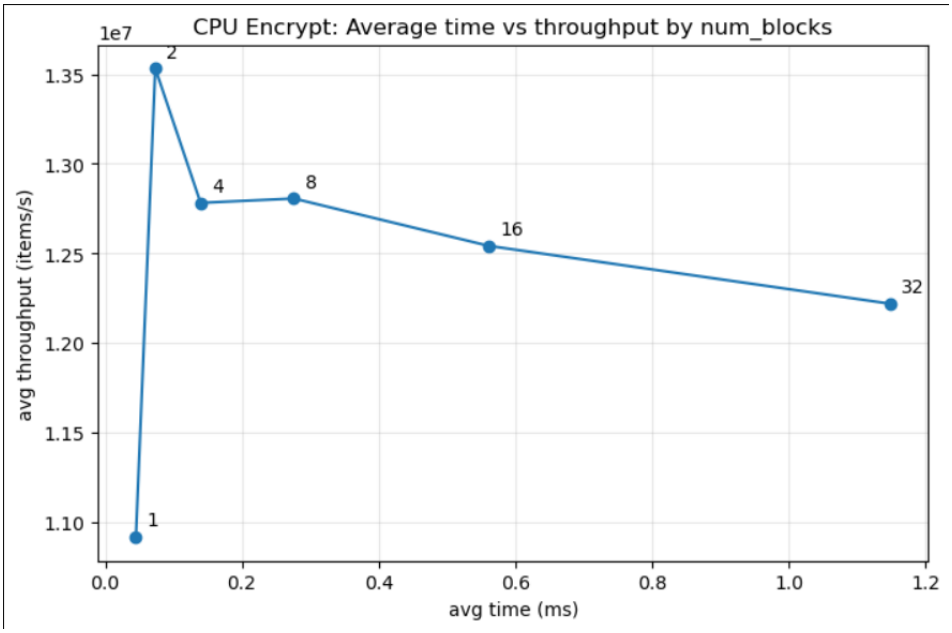


Figure 6.1 CPU Encrypt Time vs Throughput

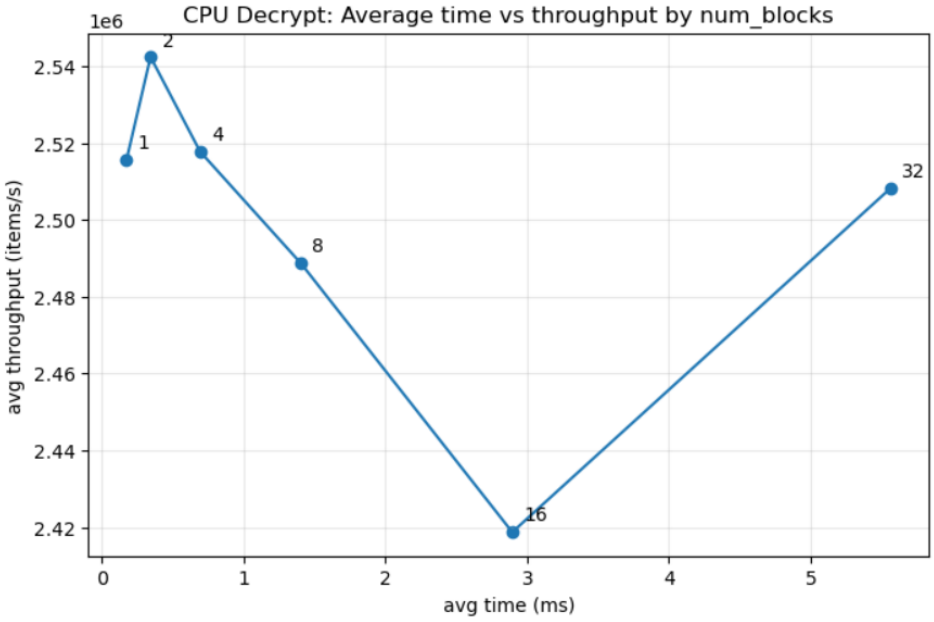


Figure 6.2 GPU Decrypt Time vs Throughput

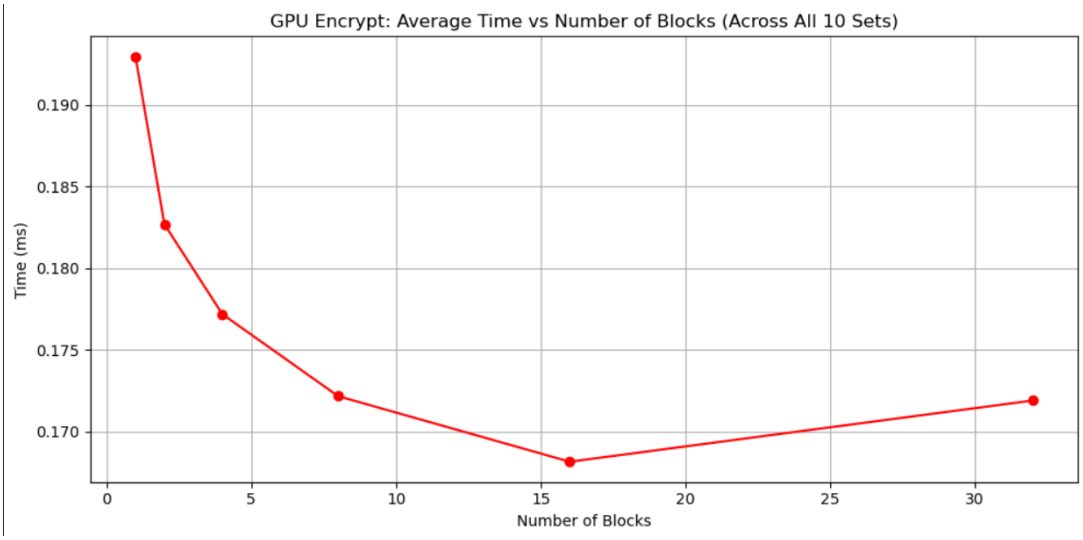


Figure 6.3 GPU Encrypt Time

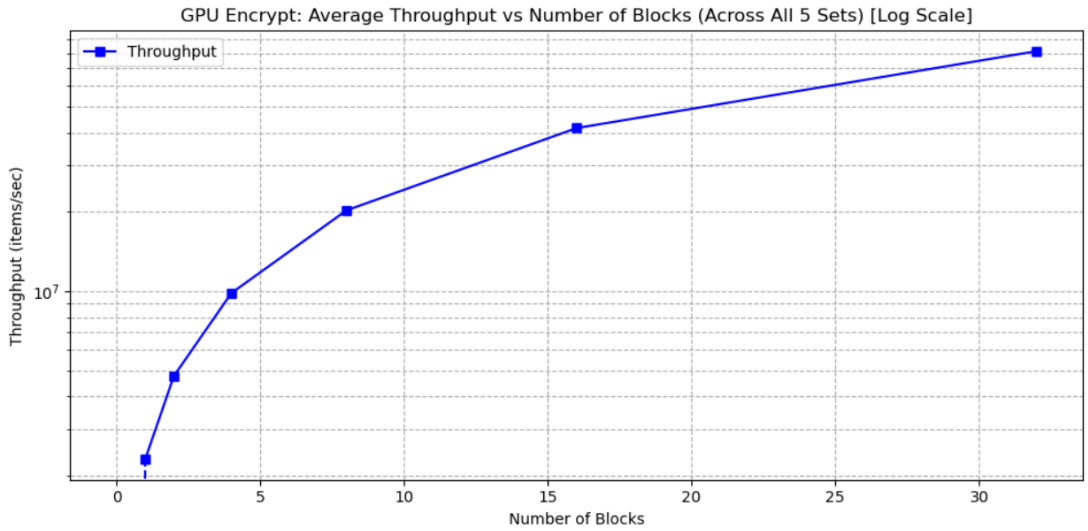


Figure 6.4 GPU Encrypt Throughput

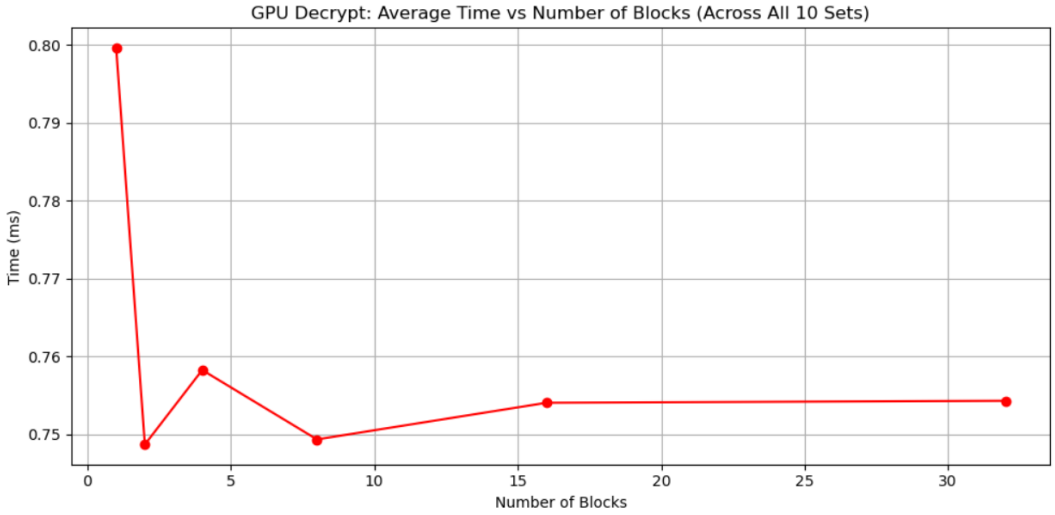


Figure 6.5 GPU Decrypt Time

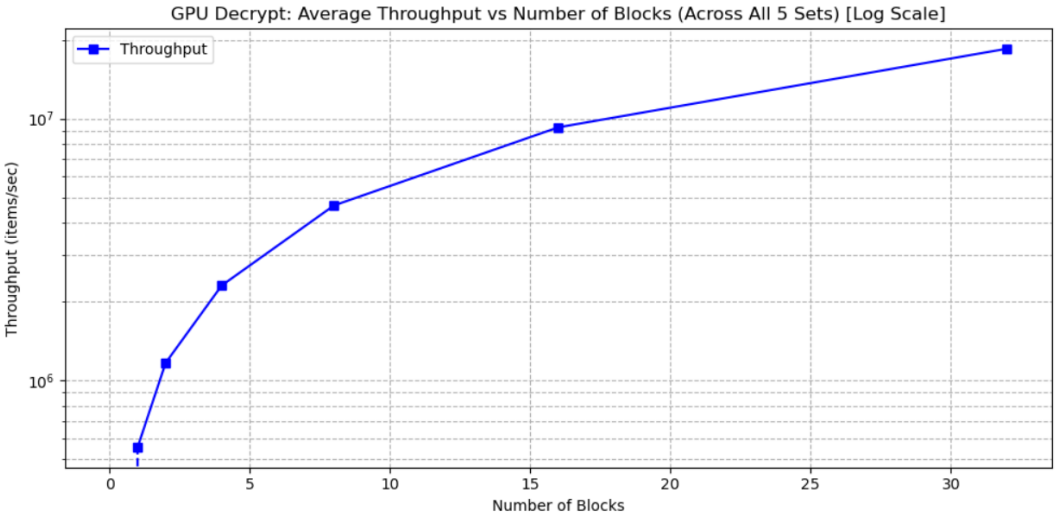


Figure 6.6 GPU Decrypt Throughput

Discussion of Results

The results stated clearly to the difference between sequential CPU execution and massively parallel GPU execution. As can be seen in Table 6.2-6.3 and Figures 6.1-6.2, CPU encryption and decryption throughput hardly changes with different num_blocks configurations. This behavior is because the CPU implementation serializes each loop: when increasing the number of blocks, the quantity of work and execution time almost proportionally increases. Thus, the ratio (work/ time) and subsequently, the throughput remains nearly flat. Minor throughput variation is due to cache effects and operating system scheduling noise, which do not imply any significant meaningful speedup.

In stark contrast to this, the GPU results in Table 6.4-6.5 and Figures 6.3-6.6 show the expected boost in throughput with increasing blocks. While the work does increase with num_blocks, the increase in execution time is minimal, thanks to CUDA's capability of scheduling thousands of threads simultaneously. For encryption, the throughput goes from 2.31×10^{6} items/s at one block to 8.14×10^{7} items/s at 32 blocks which represents a ~35× improvement. Decryption scales similarly, reaching 1.85×10^{7} items/s at 32 blocks. The flattening of GPU time curves beyond num_blocks = 16 reflects that the GPU saturates its compute resources, after which additional blocks do not yield large time reductions but continue to improve total throughput by amortizing work.

When comparing CPU and GPU execution time at num_blocks = 32, GPU acceleration achieves **~6.66× faster encryption** and **~7.38× faster decryption** relative to CPU. These results are consistent with expectations, as decryption involves matrix transformations and bit-sliced operations that map efficiently to SIMT execution on CUDA cores.

In short, the results affirm the project's main objective which was to improve the encapsulation and decapsulation processes on McEliece using GPU parallelism. The advantages of the GPU execution platform are reducing latencies and almost linear scaling throughput with respect to workload until saturation — whereas the CPU is sequentially bounded. Thus, the proposed method will serve as a feasible approach to real-world, high throughput deployments of PQC.

6.3 Project Challenges

The major challenge for evaluation was to adjust for the number of thread blocks (num_blocks) for a fair and meaningful comparison between CPU and GPU implementations referring to the serial nature of CPU implementation as the loops had to be aligned based on GPU's num_blocks for fair work done. Another complication found during evaluation was how to achieve stabilized timing results, which was managed by adding warm-up kernels to ensure the GPU was initialized and trialing 10 repetitions then averaging to smooth runtime variability per configuration. These considerations guaranteed the results reported in 6.2 accurately represented any performance benefit of using GPU acceleration.

Another challenge is the imperfectness of the decrypt function. As time constraint and payoff are taken into consideration, the parallelization of the BM function remains untouched and it remains using the CPU version of the BM function in the CUDA version. This caused time wasted when copying memory from device to host to perform BM and finally copying it back from host to device to continue with the decrypt function.

6.4 Objectives Evaluation

The project successfully met its objectives defined in Chapter 1:

- **Parallelization of McEliece Operations:** Achieved GPU-based implementation of encapsulation, decryption, and decapsulation. All key kernels (e.g., syndrome_kernel_vecstyle, k_fft, k_fft_tr, k_benes) except bm were successfully parallelized and verified against CPU outputs.
- **Correctness and Verification:** The GPU decrypt system consistently produced shared secrets (ss_dec) identical to the GPU encrypt (ss_enc), with no syndrome mismatches or weight check failures across all test cases. Error vector and syndrome are also tested with the CPU version and proved to be correct.
- **Performance Improvement:** The GPU implementation achieved **≈6.66× faster encryption** and **≈7.38× faster decryption** at num_blocks = 32 compared to CPU execution, demonstrating substantial acceleration. Throughput scaled almost linearly with num_blocks until kernel saturation, confirming effective exploitation of GPU parallelism.

Overall, the project met both functional and performance expectations, showing that CUDA-based acceleration can make McEliece operations viable for real-world high-throughput post-quantum applications.

6.5 Concluding Remark

This chapter provided a detailed assessment of the McEliece KEM scheme that utilizes GPU acceleration through CUDA. The results confirmed that GPU acceleration is not only able to preserve correctness, but it is also able to deliver exceptional gains over CPU-based execution, especially at heightened workloads where parallelism can be maximized. The results are encouraging due to the low latency, nearly linear scaling in throughput rates, and ability to replicate results. All of these confirm that the system is robust and efficient. These results bode well for the future of GPU acceleration for PQC schemes, paving the way for additional future work related to optimization, true batching, and scaling up clusters of GPUs.

Chapter 7
Conclusion and Recommendation

7.1 Conclusion

This project was able to carry out and evaluate a McEliece KEM that utilized CUDA for acceleration, in regard to the layered encryption, encapsulation, decryption, and decapsulation stages of the implementation. The key functions (such as error vector generation, syndrome computation, Benes network permutation, FFT/FFT-transpose, and weight-checking) were all ported to kernels that were executed on the GPU. These kernels were debugged and tested to ensure they operated correctly by comparing them with the CPU vec reference implementation of code for the aforementioned functions.

Much testing showed that the GPU code produced matching shared secrets at all times (ss_enc == ss_dec), thus there was confirmed functional correctness of the system. Furthermore, benchmark results showed performance improvements over the CPU code implementation: for example, at num_blocks = 32, the GPU encryption was approximately $\approx 6.66\times$ times faster, and the GPU decryption was approximately $\approx 7.38\times$ times faster. Throughput met expectations of almost scaling linearly with increasing num_blocks until saturation, and execution time was nearly constant beyond 16 number of blocks, so the application exhibited efficient capabilities for taking advantage of the additional parallel resources.

These results confirm the project's main goal which is to speed up the operations of the McEliece cryptosystem with GPU parallelism and suggest that deploying a CUDA-enabled device can render PQC more achievable in a real-world throughput scenario. Additional work towards a production implementation or sensation in a multi-GPU setting would be left to future works, as we provide a repeatable and scalable option within this implementation.

7.2 Recommendation

While the current implementation has met its goals, some ideas could further improve performance, scalability, and usability:

1. Berlekamp–Massey (BM) Algorithm Parallelization:
   The current implementation executes (CPU-based) Berlekamp–Massey (BM) algorithm (that calculates the error locator polynomial) on the CPU; this is a sequential computation that is a performance bottleneck for decryption. Future work could explore parallelizing the BM algorithm implementation on the GPU or a hybrid CPU–GPU implementation to provide further speed up and eliminate decryption latency.

2. True GPU Batching:
   The current implementation uses num_blocks to represent a batch but does not provide true parallel batch encapsulations or decapsulation. True GPU batching would allow multiple key operations in a single batch to occur within a GPU kernel, maximizing throughput and minimizing kernel launch overhead.

3. Asynchronous Memory Operations:
   Future work should utilize CUDA streams and asynchronous memory transfers to overlap host-device data movement and kernel execution. This would further reduce end-to-end latency and improve GPU utilization.

4. Multi-GPU Scaling:
   Scaling the design across multiple GPUs would allow the system to handle enterprise-level workloads such as bulk key generation or bulk key distribution in secure communication systems.

5. Optimization of Algorithm:
   There are opportunities to optimize computations of FFT, scheduling of Benes network, and arithmetic operations in Galois Field. For instance, precomputed tables stored in either constant or shared memory could reduce the per-thread calculation and maximize throughput.

6. Support for Multiple sets of parameters:
   The present system has been targeted for mceliece348864. An extension of support to other NIST-recommended parameter sets such as mceliece460896 and mceliece6688128 would increase the generality and applicability of the solution.

7. Robust Framework for Benchmarking:
   Future versions could include an automated benchmarking framework that tests correctness and performance and reproducibility across multiple hardware platforms. The benchmarking framework would facilitate better performance comparison, as well as enable performance tuning.

Following recommendations would help bring the proposed solution one step closer to being a production ready, high-performance, quantum-resistant cryptographic solution that meets the requirements of large-scale and real-time applications.

REFERENCES

[1] E. Milanov, “The RSA Algorithm,” 2009. Available: http://susanka.org/MathPhysics2/RSA_Algorithm_Yevgeny.pdf

[2] Samya Al Busafi and B. Kumar, “Review and Analysis of Cryptography Techniques,” Dec. 2020, doi: https://doi.org/10.1109/smart50582.2020.9336792.

[3] Y. Yan, “The Overview of Elliptic Curve Cryptography (ECC),” *Journal of Physics: Conference Series*, vol. 2386, no. 1, p. 012019, Dec. 2022, doi: https://doi.org/10.1088/1742-6596/2386/1/012019.

[4] P. W. Shor, “Algorithms for quantum computation: discrete logarithms and factoring,” *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, 1994, doi: https://doi.org/10.1109/sfcs.1994.365700.

[5] R. J. McEliece, "A Public-Key Cryptosystem Based On Algebraic Coding Theory," *DSN Progress Report*, vol. 42-44, pp. 114-116, 1978. https://home.cs.colorado.edu/~jrblack/class/csci7000/f03/papers/mceliece.pdf

[6] D. J. Bernstein, T. Lange, and C. Peters, “Attacking and Defending the McEliece Cryptosystem,” *Post-Quantum Cryptography*, pp. 31–46, 2008, doi: https://doi.org/10.1007/978-3-540-88403-3_3.

[7] D. J. Bernstein *et al.*, “Classic McEliece: conservative code-based cryptography,” 2024. Accessed: Apr. 26, 2025. [Online]. Available: https://cr.yp.to/talks/2024.09.17/slides-djb-20240917-mceliece-16x9.pdf

[8] NVIDIA, "CUDA C Programming Guide," *NVIDIA Developer Documentation*, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[9] NVIDIA, “CUDA Installation Guide for Linux,” *Nvidia.com*, 2023. https://docs.nvidia.com/cuda/cuda-installation-guide-linux/

[10] NVIDIA, “CUDA C++ PROGRAMMING GUIDE Design Guide,” 2020. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[11] M. Repka and P. Zajac, “Overview of the Mceliece Cryptosystem and its Security,” *Tatra Mountains Mathematical Publications*, vol. 60, no. 1, pp. 57–83, Sep. 2014, doi: https://doi.org/10.2478/tmmp-2014-0025.

[12] D. J. Bernstein, “Understanding Binary-Goppa Decoding*” IACR Communications in Cryptology, 2024*. [Online]. Available: https://cr.yp.to/papers/goppadecoding-20240412.pdf

[13] A. M. Elsobky, A. K. Farag, and A. Keshk, “Efficient Implementation of McEliece Cryptosystem on Graphic Processing Unit,” *Proceedings of the 10th International Conference on Informatics and Systems*, pp. 247–253, May 2016, doi: https://doi.org/10.1145/2908446.2908491.

[14] S. Cook, “CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, Morgan Kaufmann, 2012. [Online]. Available: https://books.google.com.my/books?hl=en&lr=&id=EX2LNkSqViUC&oi=fnd&pg=PP1&dq=CUDA+Programming:+A+Developer%27s+Guide+to+Parallel+Computing+with+GPUs&ots=XBhiwnMSaM&sig=dkNTgNSwF5Y6eg_zBGJe5WK1fqg&redir_esc=y#v=onepage&q=CUDA%20Programming%3A%20A%20Developer's%20Guide%20to%20Parallel%20Computing%20with%20GPUs&f=false

[15] H. Naghibijouybari, A. Neupane, Z. Qian and N. Abu-Ghazaleh, "Beyond the CPU: Side–Channel Attacks on GPUs," in *IEEE Design & Test*, vol. 38, no. 3, pp. 15-21, June 2021, doi: https://doi.org/10.1109/MDAT.2021.3063359

[16] H. Zhang, X. Qiao, J. Tian, S. Song and Z. Wang, "Fast Hardware Architecture With Efficient Matrix Computations for the Key Generation of Classic McEliece," in *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 72, no. 3, pp. 1321-1331, March 2025, doi: https://doi.org/10.1109/TCSI.2025.3528119

[17] P.M.C. Massolino, Paulo, and W. V. Ruggiero, “Optimized and Scalable Co-Processor for McEliece with Binary Goppa Codes,” *ACM transactions on embedded computing systems*, vol. 14, no. 3, pp. 1–32, Apr. 2015, doi: https://doi.org/10.1145/2736284.

[18] N. Corporation, "NVIDIA Ampere Architecture In-Depth," *NVIDIA Developer Blog*, 2020. [Online]. Available: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

[19] M. Kasper, W. Schindler and M. Stöttinger, "A stochastic method for security evaluation of cryptographic FPGA implementations," *2010 International Conference on Field-Programmable Technology*, Beijing, China, 2010, pp. 146-153, doi: https://doi.org/10.1109/FPT.2010.5681772

[20] M. Kihara, K. Iwai, T. Matsubara, and T. Kurokawa, “Evaluation of Gaussian elimination using HLS for fast public key generation in the Classic McEliece,” *Bulletin of Networking, Computing, Systems, and Software*, vol. 14, no. 1, pp. 22–26, 2025, Accessed: Apr. 28, 2025. [Online]. Available: http://ww.bncss.org/index.php/bncss/article/view/186

[21] A. Putnam *et al.*, “A reconfigurable fabric for accelerating large-scale datacenter services,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 13–24, Oct. 2014, doi: https://doi.org/10.1145/2678373.2665678.

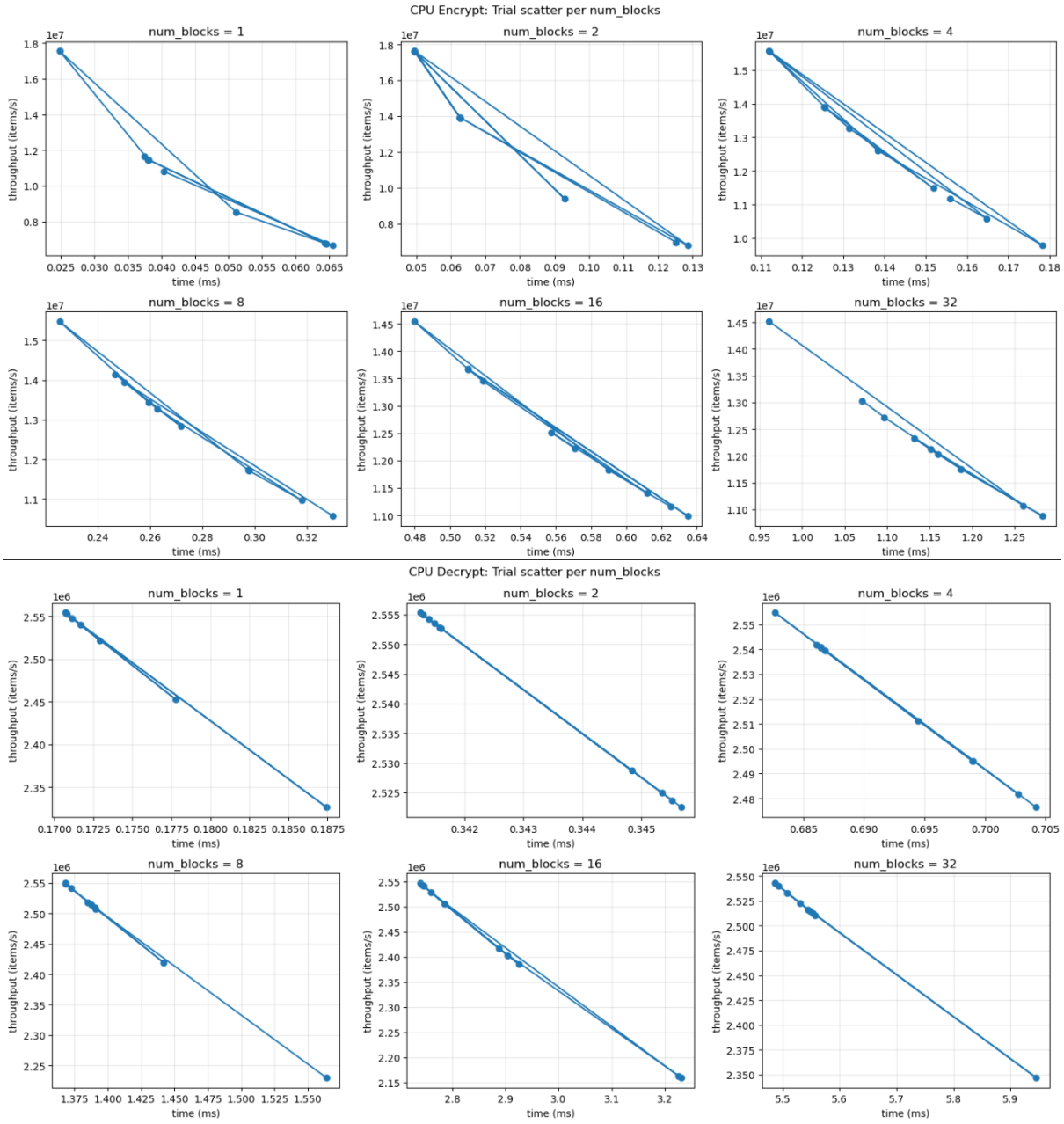
[22] W. Wu *et al.*, “Symphony of Speeds: Harmonizing Classic McEliece Cryptography with GPU Innovation,” *Cryptology ePrint Archive*, 2025. https://eprint.iacr.org/2025/748 (accessed Sep. 19, 2025).

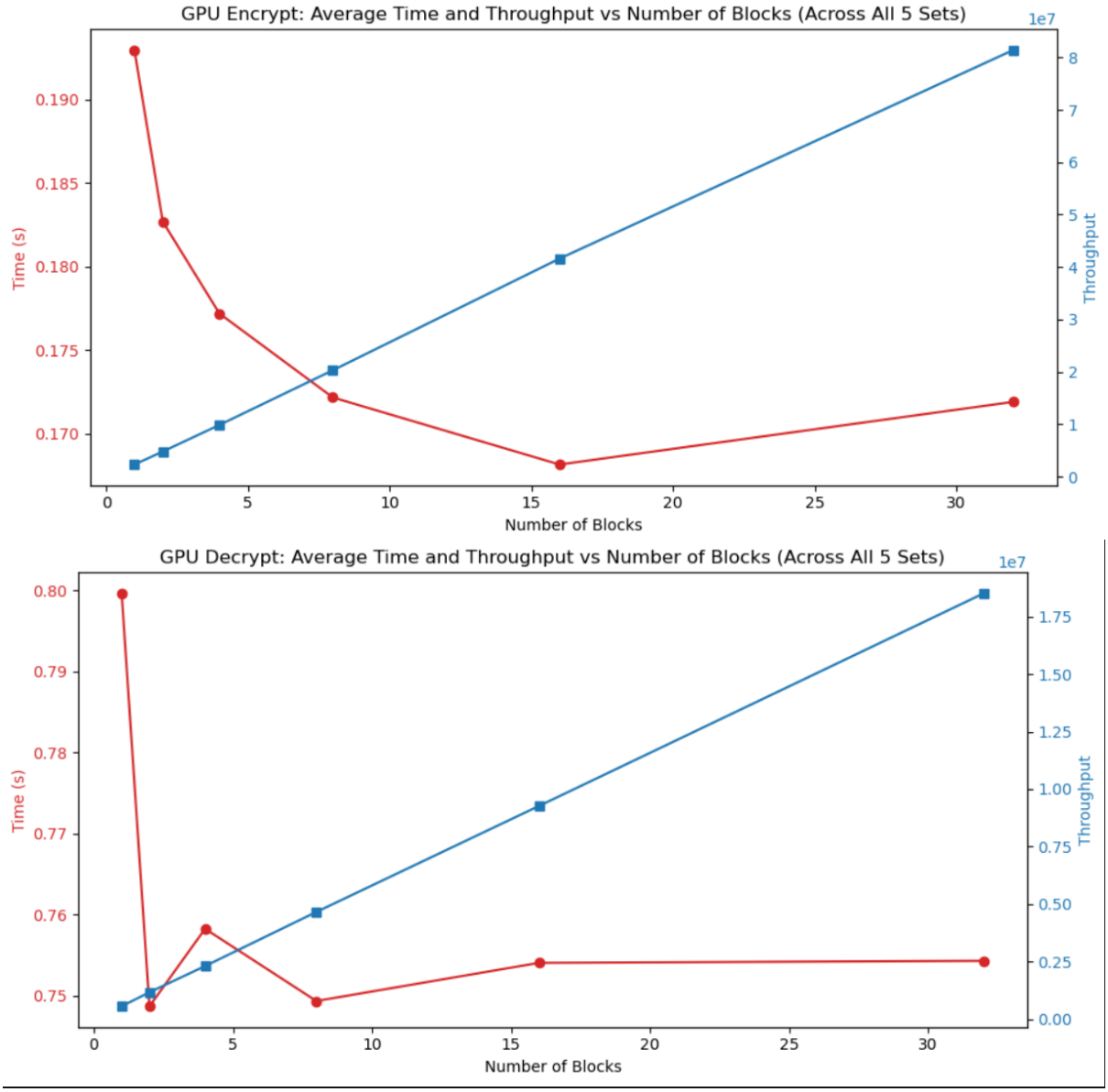
Appendix A: Benchmark CSV Sample

Timing for CPU encrypt
num_blocks,trial,time_ms,throughput
1, 1, 0.040347,10806255.73
1, 2, 0.065574,6648976.73
1, 3, 0.064532,6756337.94
1, 4, 0.038022,11467045.39
1, 5, 0.064371,6773236.40
1, 6, 0.051127,8527783.75
1, 7, 0.024847,17547390.03
1, 8, 0.024827,17561525.76
1, 9, 0.038092,11445972.91
1, 10, 0.037471,11635664.91
1, avg, 0.044921, 10917018.96
2, 1, 0.125156,6967304.80
2, 2, 0.062618,13925708.26
2, 3, 0.128643,6778448.89
2, 4, 0.049444,17636113.58
…….
32, 10, 1.070702,13030703.22
32, avg, 1.148872, 12219651.07

Timing for CPU decrypt
num_blocks,trial,time_ms,throughput
1, 1, 0.172907, 2521586.75
1, 2, 0.171674, 2539697.33
1, 3, 0.171123, 2547874.92
1, 4, 0.170792, 2552812.78
1, 5, 0.177766, 2452662.49
1, 6, 0.170702, 2554158.71
1, 7, 0.170712, 2554009.09
1, 8, 0.170762, 2553261.26
1, 9, 0.187424, 2326276.25
1, 10, 0.170762, 2553261.26
1, avg, 0.173462, 2515560.08
2, 1, 0.345683, 2522542.33
2, 2, 0.341295, 2554974.44
2, 3, 0.341605, 2552655.85
2, 4, 0.344842, 2528694.30
…….
32, 9, 5.493124, 2539902.61
32, 10, 5.486401, 2543014.99
32, avg, 5.565257, 2508264.32

Appendix B: Additional Graphs




GPU Encrypt: Average Time and Throughput vs Number of Blocks (Across All 5 Sets)
1e7
Time (s)
Throughput
Number of Blocks
GPU Decrypt: Average Time and Throughput vs Number of Blocks (Across All 5 Sets)
1e7
Time (s)
Throughput
Number of Blocks

Appendix C: POSTER

BY: DESMOND HO
SUPERVISOR: DR LEE WAI KONG

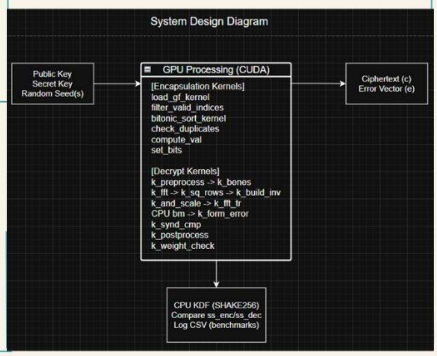
EFFICIENT PARALLEL IMPLEMENTATION OF MCELIECE SIGNATURE SCHEME ON GPU PLATFORMS

BACHELOR OF COMPUTER SCIENCE (HONS)

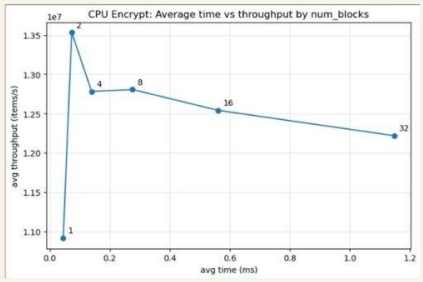
1 INTRODUCTION

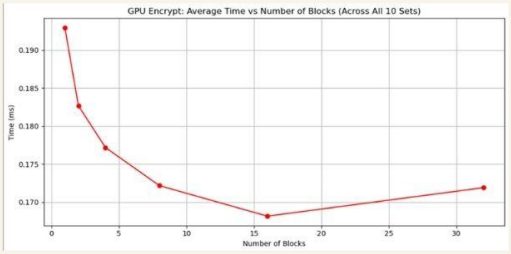
Quantum computing poses a serious threat to traditional cryptosystems such as RSA and ECC, as algorithms like Shor's can efficiently break them. The McEliece cryptosystem is a strong post-quantum candidate due to its security based on the hard problem of decoding random linear codes. However, its high computational cost, particularly in encryption and decryption, limits practical deployment. This project implements a CUDA-accelerated McEliece KEM to parallelize key encapsulation, decryption, and decapsulation, aiming to achieve significant speedup over CPU execution.

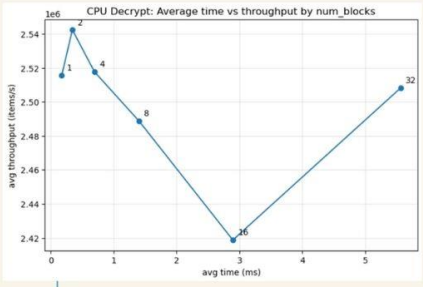
2 METHODOLOGY

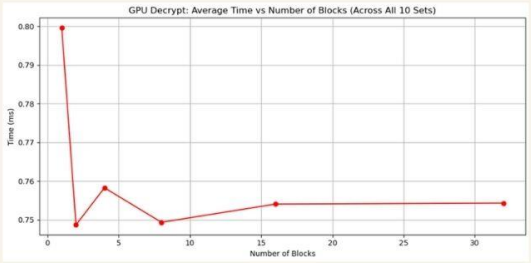


3 RESULTS









4 DISCUSSION

- GPU achieved near-linear throughput scaling until saturation.
- CPU throughput stayed almost flat — no parallel speedup.
- Encryption ran ≈6.7× faster, decryption ≈7.4× faster than CPU at 32 blocks.
- Correctness verified by matching shared secrets and syndromes.

5 CONCLUSION

The CUDA-based implementation successfully accelerated McEliece KEM operations, with GPU achieving over 6x improvement in encryption and 7x in decryption performance. These results show that GPU parallelization can effectively overcome McEliece's computational bottlenecks, making it more viable for high-throughput, real-world cryptographic applications.