POLITECNICO DI TORINO

Master's Degree in Computer Engineering - Embedded systems

Master's Degree Thesis

Post-Quantum Cryptography Acceleration through Near-Memory Computing Architectures

Supervisors

Prof. Massimo PONCINO

Dott. Emanuele VALEA

Candidate

Kristina ROGACHEVA

Academic year 2022-2023

Summary

FrodoKEM is a robust quantum-resistant key encapsulation mechanism that is based on the Learning with Errors problem. Unfortunately, one of the significant computational bottlenecks in FrodoKEM is the product between matrices of integers. However, recent advances in computing architectures such as Near-Memory Computing (NMC) offer a promising approach to accelerate and improve the efficiency of matrix multiplication. NMC drastically reduces the transfer of data between the CPU and memory, thereby increasing the overall computational efficiency. The primary goal of this thesis is to develop a novel acceleration strategy for FrodoKEM based on NMC, implemented on traditional SRAM memories. Through the use of a Computational SRAM and an innovative architecture, the proposed solution reduces the time required for matrix multiplication, resulting in a fourfold improvement in performance with respect to the reference software implementation. Although the current application does not provide all the features offered by the FrodoKEM scheme, it is possible to extend the optimizations to all the cryptographic functions. As the need for post-quantum cryptography continues to grow, the development of novel acceleration strategies such as this one will be crucial to ensure the security of sensitive information in the future.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis is the result of the collaboration with CEA - Alternative Energies and Atomic Energy Commission, a French research organization. I worked for six months in their research center in Grenoble, where I had the honor of collaborating with two teams: Cybersecurity and Memory Design.

This experience allowed me to make use of the knowledge acquired during my university studies, learn new concepts in Cybersecurity, and improve my skills in Digital Design and program development. Collaboration with both teams offered me the opportunity to meet specialists and engineers from different fields, who inspired me and helped me to conclude the work presented in the following document.

The Cybersecurity team is specialized in Post-Quantum Cryptography, in other words, their main field of study is the protection of digital systems against quantum computers. The Memory Design team, on the other hand, works on the development of advanced custom memories. The purpose of this thesis is to exploit the computational memories to improve the performance of Post-Quantum Cryptography algorithms.

The objective of this thesis is to enhance the performance of FrodoKEM, a Post-quantum Cryptography algorithm, by utilizing advanced memory, C-SRAM, based on a Near Memory Computing approach. Whereas the software and hardware used in this work were developed by cybersecurity and hardware design experts, this thesis aims to exploit the computational memory to improve the performance of a computationally heavy algorithm like FrodoKEM. The idea is to modify the existing algorithms to perform challenging operations, such as matrix multiplication, inside the C-SRAM. Although FrodoKEM is composed of three primary functions, Key

Generation, Encryption, and Decryption, this work focuses solely on the first function leaving the other two for potential future improvements (detailed in Chapter 4).

1.1 Thesis outline

- Chapter 2 is an introduction to the hardware and software background on which this work is based, and describes the state of the art;
- Chapter 3 introduces the approach and the different implementations of the algorithms;
- Chapter 4 analyses the obtained results, draws the conclusions and presents possible ways to continue this work.

Chapter 2

Background and State of The Art

2.1 Cybersecurity

Cybersecurity refers to the set of techniques, technologies, and processes aimed at protecting computer systems, networks, and digital data from theft, damage, or unauthorized access. This approach employs advanced technology, well-defined processes, and robust policies to protect digital assets and prevent cyber attacks, which can include malicious activities such as hacking, malware, phishing, and other forms of exploitation. The primary objective of cybersecurity is to maintain the confidentiality, integrity, and availability of digital information and systems. This includes protecting sensitive data, ensuring the reliability and functionality of critical systems, and mitigating risks posed by cyber threats. The use of cryptography plays a crucial role in achieving these goals, as it provides secure communication channels and protects data privacy. Cryptography enables the encryption of sensitive data, thereby ensuring that only authorized parties can access and utilize it. As cybersecurity is a constantly evolving field, cryptography requires continuous vigilance and proactive measures to stay ahead of potential attackers.

Cryptography is a field of study concerned with the techniques used to ensure secure communication. The term *Cryptography* originates from the Greek words `kryptós`, meaning *secret*, and `graphein`, meaning *towrite*. The objective of cryptography is to encrypt and decrypt messages in such a way that their content remains confidential to everyone except the sender and the intended recipient.

Cryptography has a long history, dating back to ancient civilizations, where the

first ciphers, such as transposition and substitution ciphers, were invented. These ciphers involve rearranging the order of letters in a message or substituting letters with others. Cryptographers and mathematicians have been studying methods and strategies for centuries to improve the security of their transmissions and decipher the enemy's intercepted messages.

With the advent of electronic communication in the computer era, cryptography has become increasingly complex, relying on mathematical algorithms and computational complexity to provide security. Modern cryptographic algorithms must provide confidentiality, authentication, integrity, and non-repudiation.

- **Confidentiality**: ensures that data and information are protected from unauthorized readings.
- **Authentication**: allows verification of the identity and origin of data and information.
- **Integrity**: ensures that data and information are protected against manipulations performed by unauthorized parties.
- **Non-repudiation**: ensures that a sender cannot deny being the author of the message.

A plaintext message is a message in a readable form. The process of concealing a message to hide its content is called encryption, and the resulting message is called ciphertext. The reverse process, which takes ciphertext as input and restores the original plaintext, is called decryption. An encryption function E operates on plaintext P to produce ciphertext C, while the decryption function D operates on ciphertext C to produce plaintext P.

$$\mathrm{E}(\mathrm{P}) = \mathrm{C} \tag{1}$$

$$\mathrm{D}(\mathrm{C}) = \mathrm{P} \tag{2}$$

A cipher is a mathematical function used for encryption and decryption of information. All modern encryption algorithms use a key denoted by K, which affects the encryption and decryption functions.

$$\mathrm{E}(\mathrm{K}, \mathrm{P}) = \mathrm{C} \tag{3}$$

$$\mathrm{D}(\mathrm{K}, \mathrm{C}) = \mathrm{P} \tag{4}$$

Symmetric-key encryption uses just one key that must be kept secret. The sender and the receiver use the same algorithm to encode and decode with the same key. This type of encryption is fast and efficient, and it is nearly impossible to decipher the message at reasonable times. Examples of symmetric key algorithms include Advanced Encryption Standard (AES) and Data Encryption Standard (DES). The only downside is that the sender and receiver need a secure way to exchange the secret key.

Asymmetric-key encryption, on the other hand, uses two different keys, one for encryption and another for decryption. The exchange is performed in three stages: key generation, encryption, and decryption. In the first phase, the receiver generates two keys: one kept secret and one shared with the sender. The sender uses the public key to encrypt their message, generating a ciphertext. The ciphertext is then sent to the receiver, who can recover the original message using the private key. Asymmetric encryption is slower and more complex, as it requires three stages and two different keys to complete. The security of asymmetric cryptography depends on the mathematical relationship between the public and private keys. In addition to confidentiality, asymmetric cryptography also provides authenticity and non-repudiation. Examples of asymmetric key algorithms include RSA, Elliptic Curve Cryptography (ECC), and Diffie-Hellman key exchange.

Both encryption techniques have different advantages and disadvantages, so hybrid encryption schemes are often used, combining both symmetric and asymmetric cryptography to provide the benefits of both approaches.

2.2 Post-Quantum Cryptography

2.2.1 What are quantum computers?

In the early 20th century, physicists made groundbreaking discoveries about the behavior of matter at the quantum level. This branch of physics defied classical mechanics and revealed strange, counterintuitive properties of particles and waves at the subatomic scale. Physicists like Werner Heisenberg, Erwin Schrödinger, and Paul Dirac developed a mathematical framework called quantum mechanics that can describe these peculiar behaviors and interactions.

It wasn't until 1982 that physicist Richard Feynman proposed the idea of a quantum computer [1]. Feynman suggested that a quantum computer could simulate quantum systems more efficiently than classical computers. However, this idea remained mostly theoretical at the time due to a lack of practical methods

for building a quantum computer. Nonetheless, physicists continued to propose quantum algorithms that could solve problems that classical computers could not.

The first quantum computers were relatively simple and could only perform small-scale computations. However, they demonstrated the feasibility of building a quantum computer and laid the groundwork for future research and development. Today, the scientific community is especially interested in the potential of quantum computing to solve complex problems that classical computers cannot solve within a reasonable amount of time. Quantum algorithms like the Quantum Approximate Optimization Algorithm (QAOA) can provide faster and more accurate solutions to optimization problems than classical algorithms, which could lead to better decision-making, resource optimization, and increased productivity.

Quantum computers are based on the laws of quantum mechanics, a branch of theoretical physics that replaces classical mechanics at the atomic and subatomic levels. Information in a quantum computer is encoded in qubits, single atoms in different physical states based on the spin of an electron. The two possible states of a qubit are the ground state |0> with the spin up and the excited state |1> with the spin down. Unlike classical bits, which can only exist in one of two states (0 or 1), qubits can exist in multiple states simultaneously. This property is known as superposition and is one of the fundamental principles of quantum mechanics that makes quantum computers so powerful.

Another key property of qubits is entanglement, a phenomenon where two or more qubits become correlated in such a way that their properties are linked, regardless of the distance between them. This enables quantum computers to perform certain computations in parallel and exploit interference between qubits to achieve computational speedup.

Quantum gates are used to manipulate the state of qubits. These gates are analogous to the logic gates used in classical computing, but they operate on quantum bits instead of classical bits. Quantum gates can be combined to create quantum circuits, which perform specific computations on qubits. For example, the Hadamard gate creates a superposition of the 0 and 1 states, and it is used to initialize qubits and create quantum states that can be manipulated by other gates. Another important gate is the CNOT gate, which performs a conditional operation on two qubits.

Despite the potential of quantum computers, they are still in the early stages of development. Building and operating a quantum computer is a challenging task, as qubits are very fragile and sensitive to errors from noise and other environmental

factors. Nonetheless, the potential applications of quantum computing are vast, ranging from cryptography and chemistry to material science and beyond. With further research and development, quantum computers have the potential to revolutionize the way we approach complex problems in various fields.

2.2.2 Post-Quantum Cryptography

The potential of quantum computers to exponentially accelerate certain types of computations poses a substantial threat to modern cryptographic protocols [2]. Traditional cryptographic methods rely on complex mathematical problems that classical computers cannot solve within reasonable time frames. For instance, the RSA algorithm, a widely used public-key cryptographic algorithm, is based on the difficulty of factoring large numbers. In 1994, Peter Shor developed efficient algorithms for factoring integers in polynomial time on a quantum computer [3], jeopardizing asymmetric cryptography . In 1996, Grover's quantum algorithm accelerated the process of breaking cryptographic keys, effectively treating 128-bit keys like 64-bit ones [4], making symmetric cryptography easily breakable. In the case of symmetric cryptography, the security level can be augmented by choosing larger key sizes. It has been estimated that doubling the key size is enough to be resistant to quantum attacks.

Post-Quantum Cryptography is a branch of cybersecurity that seeks to develop new asymmetric cryptography algorithms that remain challenging even for quantum computers. Researchers worldwide are studying and implementing new cryptographic systems based on different kinds of problems, such as lattice-based cryptography, code-based cryptography, hash-based cryptography, and multivariate cryptography, to protect the systems before this threat becomes real. Lattice-based cryptography depends on the computational complexity of solving specific problems within the realm of lattice theory, including the challenge of determining the shortest vector in a given lattice. Code-based cryptography is based on the difficulty of solving specific problems in coding theory, such as decoding random linear codes. Hash-based cryptography relies on the characteristics of hash functions, with a particular emphasis on their ability to resist collisions. Lastly, multivariate cryptography is grounded in the ability to solve systems of multivariate polynomial equations [5].

At this moment, we don't have quantum computers powerful enough to break current cryptographic constructions. However, as soon as they become available, there will be severe consequences for individuals and organizations. All sensitive data, such as personal data, financial data, and government secrets, will be easily

accessed, leading not only to minor crimes, like fraud and theft but also to national security threats.

Although it is not possible to predict precisely when this threat will become a real problem, it is crucial to prepare for it as soon as possible. It takes time not only to develop new algorithms but also to test them to ensure their security, adapt them to work correctly on existing digital systems, develop new hardware to support and optimize the new algorithms, and update all existing infrastructures with new protocols. This transition can take years to complete, emphasizing the importance of working on it now.

In 2019 Google disclosed that a quantum computer solved a problem in 200 seconds when a classical computer would take 10000 years to do the same [6]. At the moment the threat is just theoretical, as the current quantum computers are not powerful enough for this kind of computation, but how soon do we have to worry?

Dr. Michele Mosca, a mathematician at the University of Waterloo in Canada, answers this question with this theorem (see Figure 2.1): given that certain information must be secure for x time, that y is the time needed to convert infrastructure to a quantum-safe one, and that z is the time to build a powerful quantum computer, if $x + y > z$, then we need to worry [7].

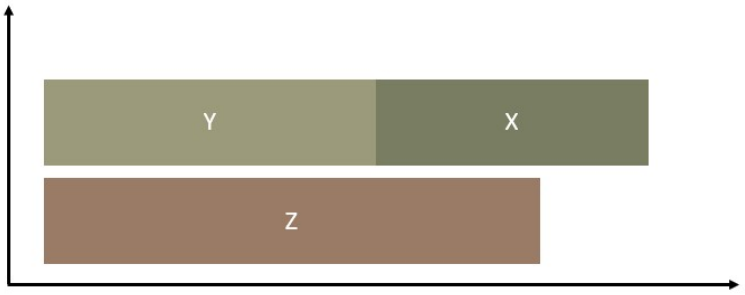

Figure 2.1: Mosca's theorem.

For this reason, in 2016, the National Institute of Standards and Technologies (NIST) of the USA initiated a standardization process that will conclude by 2024. This procedure aims to identify several algorithms that provide security and efficiency.

2.2.3 FrodoKEM

The algorithm we are interested in is FrodoKEM [8], a Key Encapsulation Mechanism (KEM) algorithm that has garnered significant attention due to its ability to ensure high levels of security and performance while also minimizing failure probability. The algorithm was selected by NIST as an *alternate* algorithm in the third round of the standardization process. FrodoKEM is based on the Learning With Errors (LWE) problem and has been developed through a collaboration between leading researchers and engineers at major companies such as Google, NXP Semiconductors, and Microsoft research, as well as prominent universities such as Stanford University and the University of Michigan.

FrodoKEM has three different levels of security described in the table below:

Level	Name	Security Description
I	FrodoKEM-640	At least as hard to break as AES128
III	FrodoKEM-976	At least as hard to break as AES192
V	FrodoKEM-1344	At least as hard to break as AES256

Table 2.1: FrodoKEM different levels of security

While FrodoKEM is simple and secure, it is also computationally heavy. As a result, it was selected as one of the alternates in the third round of the NIST standardization process. However, despite not being selected as the new standard in July 2022, FrodoKEM continues to be recommended by security agencies of several countries, including the Netherlands National Communications Security Agency (NLNCSA), German Bundesamt für Sicherheit in der Informationstechnik (BSI), and French Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI).

FrodoKEM belongs to the Lattice-based cryptography family, which is a relatively new area of cryptography that uses mathematical structures called lattices to create secure cryptographic systems that are believed to be computationally difficult for classical and quantum computers. Lattices are geometric structures that can be used to represent a wide range of mathematical objects, including vectors, matrices, and polynomials, made up of a set of points arranged in a regular pattern. In lattice-based cryptography, encryption keys are generated by selecting a random lattice and a random point within that lattice. The lattice is then made public, while the random point is kept secret. The message to be encrypted is then encoded as a point within the same lattice, and the encryption process involves adding the random point to the message point to produce the ciphertext. Decryption involves subtracting the secret point from the ciphertext

point to recover the original message.

One of the fundamental lattice problems that form the basis for several cryptosystems is the Shortest Vector Problem (SVP), which involves finding the shortest nonzero vector in a given lattice. The SVP can be formally defined as follows:

Theorem 1 *Given a lattice L in n-dimensional space, find the nonzero vector v in L with the smallest Euclidean norm //v//.*

In other words, find the shortest vector in the lattice. The SVP is an important problem in cryptography, as the security of several cryptosystems relies on the assumption that the SVP is hard to solve. An attacker who is able to solve the shortest vector problem would be able to recover the secret point and thus decrypt the ciphertext. Thus, one of the main challenges in implementing lattice-based cryptography is choosing an appropriate lattice.

There are several families of lattices that are commonly used, each with its own strengths and weaknesses. One such family is the NTRU lattice (Nth Degree Truncated Polynomial Ring Units), which is generated from polynomial rings and is particularly well-suited for use in cryptographic systems because it has good properties with respect to the shortest vector problem. Another commonly used family is the Learning With Errors (LWE) lattice, which is generated from a set of random linear equations with small noise terms and is particularly well-suited for use in homomorphic encryption systems, which allow computations to be performed on encrypted data without first decrypting it, making it useful in scenarios where data needs to be processed in a secure manner but cannot be decrypted due to privacy concerns.

The main concern of the Learning With Errors problem is finding a random vector that is close to a set of linear equations modulo some value q. To comprehend the LWE problem, we can consider a set of m linear equations with n variables that take the form:

$$\begin{aligned} a_1x_1 + a_2x_2 + ... + a_nx_n &= b \mod q \\ a'_1x_1 + a'_2x_2 + ... + a'_nx_n &= b' \mod q \\ &\cdots \\ a_mx_1 + a_mx_2 + ... + a_mx_n &= b_m \mod q \end{aligned}$$

Here, a_i and b_i are randomly selected from the set 0, 1, 2, ..., q-1. The main goal of the LWE problem is to determine the values of $x_1, x_2, ..., x_n$ with only the knowledge of a_i, b_i, and q. However, this problem becomes challenging to solve due

to the introduction of noise, which is achieved by adding a small random value, e, to each equation. Consequently, the actual set of equations becomes:

$$\begin{aligned} a_1x_1 + a_2x_2 + ... + a_nx_n &= b + e_1 \mod q \\ a'_1x_1 + a'_2x_2 + ... + a'_nx_n &= b' + e_2 \mod q \\ &\dots \\ a_mx_1 + a_mx_2 + ... + a_mx_n &= b_m + e_m \mod q \end{aligned}$$

The values of e_1, e_2, ..., e_m are chosen from a distribution that closely approximates a Gaussian distribution. This noise makes it difficult to recover the values of x_1, x_2, ..., x_n.

Given a set of LWE equations, an attacker cannot recover the values of x_1, x_2, ..., x_n without first solving the Shortest Vector Problem (SVP). This difficulty implies that LWE-based cryptographic schemes have several advantages over other post-quantum cryptographic schemes since they are efficient, easy to implement and provide a high level of security.

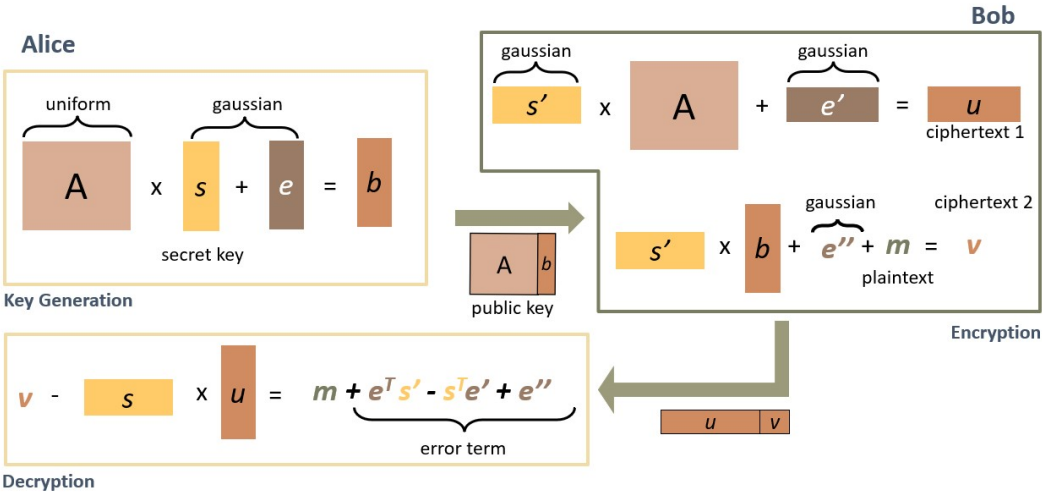

Figure 2.2: Public Key Encryption based on a Learning With Errors problem.

Figure 2.2 provides an illustrative example of a public key encryption algorithm that is built on the Learning With Errors (LWE) problem. The algorithm is composed of three key functions that are executed by Alice and Bob: Key Generation, Encryption, and Decryption. During the first function, Key Generation, Alice generates the public and secret keys by creating a matrix A and two vectors S and E, all existing in the $\mathbb{Z}_q$ domain where their elements are integer numbers

modulo q. The matrix A is uniformly distributed, while the vectors S and E are generated from a Gaussian distribution. The function then computes the product $A * S + E = B$, where the secret key is represented by the array S and the public key is represented by both A and B. Alice sends the public key to Bob to begin the encryption of the secret message.

Bob then uses the second function, Encryption, to create the ciphertext by computing two elements: $U = S' * A + E'$ and $V = S' * B + E" + M$. Here, A and B are parts of the public key, S', E', and $E"$ are vectors generated from a Gaussian distribution, and M is the message Bob wishes to encrypt and send to Alice. These operations have different types of multiplication. The first operation involves matrix multiplication similar to the one in the key generation function, resulting in a vector of the same size as S. The second operation involves scalar multiplication, which results in an integer number added to $E"$ and the message M. Once U and V are computed, they are sent back to Alice.

The final function, Decryption, is used by Alice to recover Bob's message. The operation executed to decrypt the message is $V - S * U = M + error$, where U and V are separate parts of the ciphertext sent by Bob, and S is the secret key computed previously by Alice. The result of this operation is the message M, contaminated by an error term caused by all the noise arrays added in the previous steps. It is crucial to keep the error term as low as possible, or else it may be impossible to retrieve the original message even with the secret key. However, there is always a certain probability of decryption failure with these algorithms, which is guaranteed to be less than 2^{-128}.

This algorithm faces two main bottlenecks: matrix multiplication and sampling of the matrices. The complexity of the first problem is directly proportional to the number of multiplications (scalar or matrix) required for each function, which is dependent on the parameter N that defines the size of the matrix A (N x N) and the length of the various vectors. Regarding the sampling of the matrices, it involves several complex stages. Firstly, a True Random Number Generator (TRNG) generates a random string of bits, the seed. This random string is then sent to the Pseudo-Random Number Generator (PRNG), which is capable of generating uniformly distributed streams of bytes to generate matrix A. However, the other elements of the algorithm are sampled from a Gaussian distribution, which requires an extra step: the Gaussian Sampler. This phase is dependent on the standard deviation of the distribution, which must not be too large as it negatively impacts the algorithm's performance and failure probability, nor too small as it may compromise security.

The LWE problem is at the base of FrodoKEM, but some changes have been made to make this algorithm more efficient. Until this point, the message M was a single number. However, to send larger messages, certain elements' sizes must be adjusted. Figure 2.3 demonstrates how the size of the algorithm components has changed; all the vectors have been transformed into matrices with a width of 8. E, V, and the message M have become square 8x8 matrices. Additionally, the entire A matrix need not be transmitted to Bob. A is a public matrix that can be easily generated by any Pseudo Random Number Generator (PRNG), so sending only the seed is sufficient. The algorithm's functionality remains the same as described above.

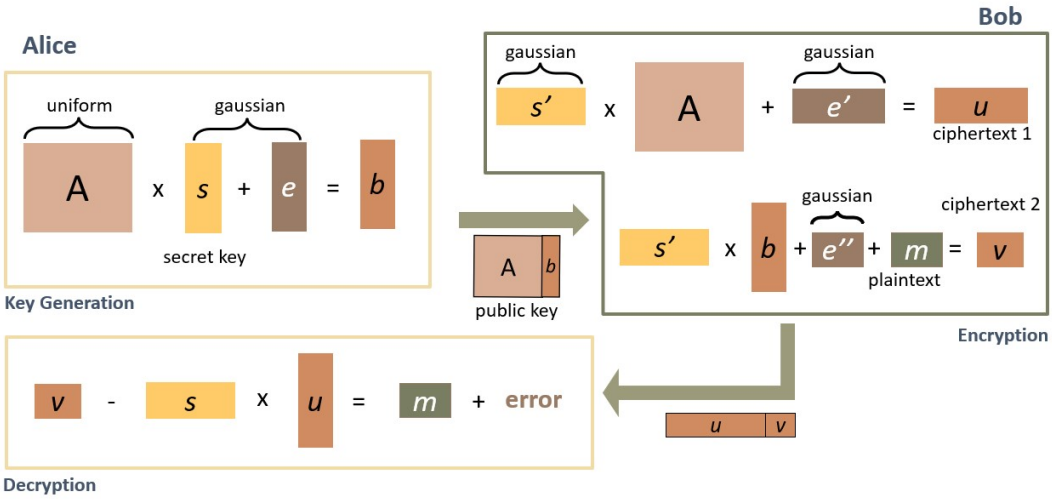

Figure 2.3: FrodoKEM functions

	Frodo-640	Frodo-976	Frodo-1344
n	640	976	1344
B	2	3	4
q	2^{15}	2^{16}	2^{16}

Table 2.2: Main parameters of FrodoKEM

Table 2.2 outlines the other parameters that depend on the algorithm's security level. n is critical to Frodo's security level, as the matrix size directly affects the number of multiplications required. The higher the matrix size, the higher the security. q also increases with Frodo-640 working on 15 bits while Frodo-976 and Frodo-1344 work on 16 bits. The most interesting parameter is B, which represents

the number of bits used to encode each element of the matrices. As the security level rises, the standard deviation used to generate the matrices varies, making them smaller and smaller. However, this doesn't impact the algorithm's security, as the matrix size and PRNG functions maintain their complexity.

The only disadvantage of this algorithm is the size of the keys and the ciphertext, as shown in Table 2.3. The public key and ciphertext require significant bandwidth for exchange between Alice and Bob. Meanwhile, the secret key only impacts the memory usage by Alice, who needs to store the key to decrypt Bob's messages.

Scheme	Secret Key	Public Key	Ciphertext
Frodo-640	3.13KB	9.39KB	9.49KB
Frodo-976	4.77KB	15.27KB	15.38KB
Frodo-1344	5.25KB	21.02KB	21.13KB

Table 2.3: Keys and ciphertext sizes of FrodoKEM

2.3 Near Memory Computing

2.3.1 Different architecture types

Currently, most hardware systems are based on the Von-Neumann architecture, which is named after Jon von Neumann, a renowned computer scientist who invented it in the 1940s [9]. This architecture is composed of several essential components, including the Central Processing Unit (CPU), which is responsible for executing instructions and consists of two primary units - the Control Unit (CU) and the Arithmetic-Logic Unit (ALU). In addition to the CPU, this architecture also includes memory, which stores both data and instructions and input/output devices that enable interaction with the external world.

The ability to store and manipulate both data and instructions in the same memory space is a significant advantage of this architecture, as it allows for greater flexibility and efficiency in the execution of programs. However, this scalar computing approach creates a performance bottleneck in data-dependent applications due to the large amount of data that must be transferred between the processing unit and the memories, as it increases the power consumption, slows down the data transfer rate and consequently the execution time of such applications.

One way to improve speed and efficiency is to implement vector computing, for

example, Single Instruction Multiple Data (SIMD). This parallel processing architecture enables the simultaneous processing of multiple data using one instruction. The data is organized into vectors, and each element of the vector is processed in parallel by a vector processing element, but the number of accesses to the memory remains high.

To further improve energy efficiency, the CEA has developed a low-latency memory through Near-Memory Computing (NMC), the Computational SRAM (C-SRAM) [10].

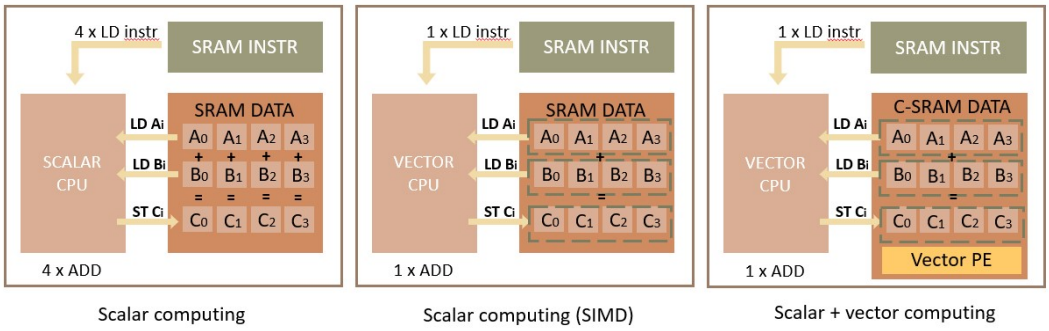

Figure 2.4: Three different types of architecture: scalar, vector and C-SRAM.

Figure 2.4 describes various approaches to perform the same operation: the sum of vectors A and B to obtain the vector C. The first method employs scalar computing, where a single instruction is performed per clock cycle. To compute the sum of two vectors using scalar processors, each element of the vectors must be transferred from memory to the processor, the addition must be executed, and the result must be written back to memory. This process necessitates n instructions and n memory accesses for each array, where n is the number of elements in each array.

The second technique, on the other hand, uses vector computing, which requires a vector processor capable of executing the same operation on multiple data. While this type of processor requires only one instruction to add two vectors, the vectors must still be moved from memory to the processor for computation, resulting in the same number of memory accesses as before.
The NMC approach, as implemented in the C-SRAM, doesn't require special types of processors or data movement to complete the addition of vectors. This method processes the operation using a small local Arithmetic Logic Unit (ALU) inside the memory, enabling the operation to be performed on the entire memory line simultaneously.

2.3.2 Near Memory Computing

The Near Memory Computing approach on which the Computational SRAM is based allows the execution of arithmetic and logic operations without the need to transfer data on the system bus, which is the most energy-consuming step in the computation of an instruction [11].
The aim of this technique is to improve performance and energy efficiency by adding an additional processing element close to the memory. As opposed to the traditional architectures, computations are performed directly on the data stored in memory, reducing the need for data movement. This architecture can be achieved through a variety of techniques, such as using either volatile memories such as Static Random Access Memory (SRAM) and Dynamic Random Access Memory (DRAM) or on Non-Volatile Memories (NVM).

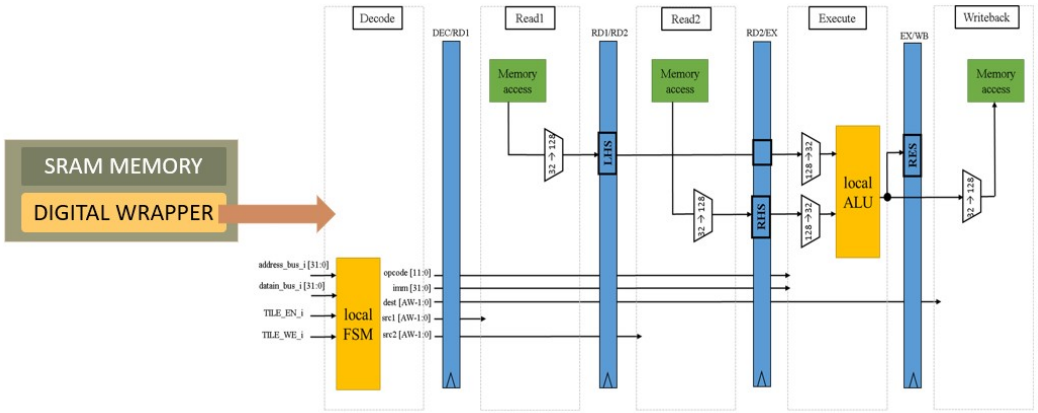

Figure 2.5: C-SRAM instruction pipeline.

2.3.3 C-SRAM and system description

The Computational SRAM is composed of two main parts: the SRAM memory and the Digital Wrapper, as shown in Figure 2.5. The SRAM memory is a regular memory that can be written or read by the CPU through load and store instructions, while the Digital Wrapper is a processing element that contains the vector Arithmetic and Logic Unit. With high parallel computation capabilities, operations can be performed on the entire memory row at once, and a five stage pipeline is used. The C-SRAM system has a single instruction flow for both scalar

and vectorial instructions, making compilation and execution simpler and reducing energy consumption.

Category	Name	Operation	Width(bits)
Logic	not	Logic NOT	Line
Logic	and	Logic AND	Line
Logic	or	Logic OR	Line
Logic	xor	Logic XOR	Line
Logic	nand	Logic NAND	Line
Logic	nor	Logic NOR	Line
Logic	xnor	Logic XNOR	Line
Logic	slli	Logic shift left	8 / 16 / 32 / Line
Memory	copy	Copy entire line	Line
Memory	bcast	Broadcast	8 / 16 / 32
Memory	nop	-	-
Arithmetic	add	Addition	8 / 16 / 32
Arithmetic	sub	Subtraction	8 / 16 / 32
Arithmetic	abs	Absolute value	8 / 16 / 32
Arithmetic	cmp	Compare	8 / 16 / 32
Arithmetic	mulhi	Multiply, keep highest bits	8
Arithmetic	mullo	Finite Field Multiplication	8
Arithmetic	maclo	Finite Field Multiply and Accumulate	8

Table 2.4: Near-Memory-Computing instructions that can be executed on the C-SRAM.

Table 2.4 describes the different Near-Memory-Computing operations that can be executed on the C-SRAM, such as logic, arithmetic, and memory instructions on vectors of data of different length as operands.

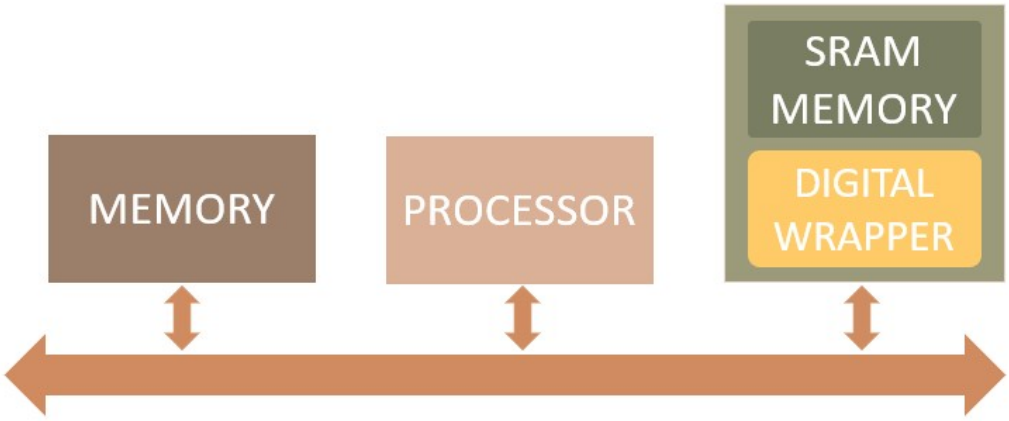

Figure 2.6: Simple system composed of the processor and the C-SRAM.

A communication protocol has been added to integrate the C-SRAM into a more conventional system, as shown in Figure 2.6, which displays a simple architecture consisting of a scalar RISC-V processor, a memory and the C-SRAM. The processor utilized in the experiments described in this dissertation is the CV32E40P based on the RISC-V architecture, a four-stage, 32-bit pipelined processor. The Open Bus Interface (OBI) protocol is used for the bus interface.

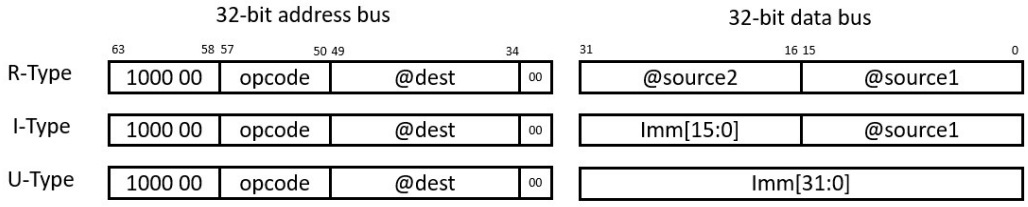

Figure 2.7: C-SRAM instruction set format.

To enable the integration of the C-SRAM instruction set into the existing design of the CPU and the 32-bit bus, a concatenated bus system has been implemented, composed of both the address and data segments. As a result, the instruction set architecture of the C-SRAM is aligned on 64 bits (32-bit data bus + 32-bit address bus) in length. The instruction set is composed of three base formats, namely R/I/U, each corresponding to a distinct set of instructions, as illustrated in Figure 2.7. The R-type format includes instructions that operate on two memory lines for operands, while the I-type format involves instructions that operate on a

16-bit immediate value and one memory line. Lastly, the U-type format includes instructions that operate on a 32-bit immediate value for one memory line. The opcode field specifies the operation to be performed in the C-SRAM, and is encoded over 8 bits, providing the capacity to execute up to 256 operations.

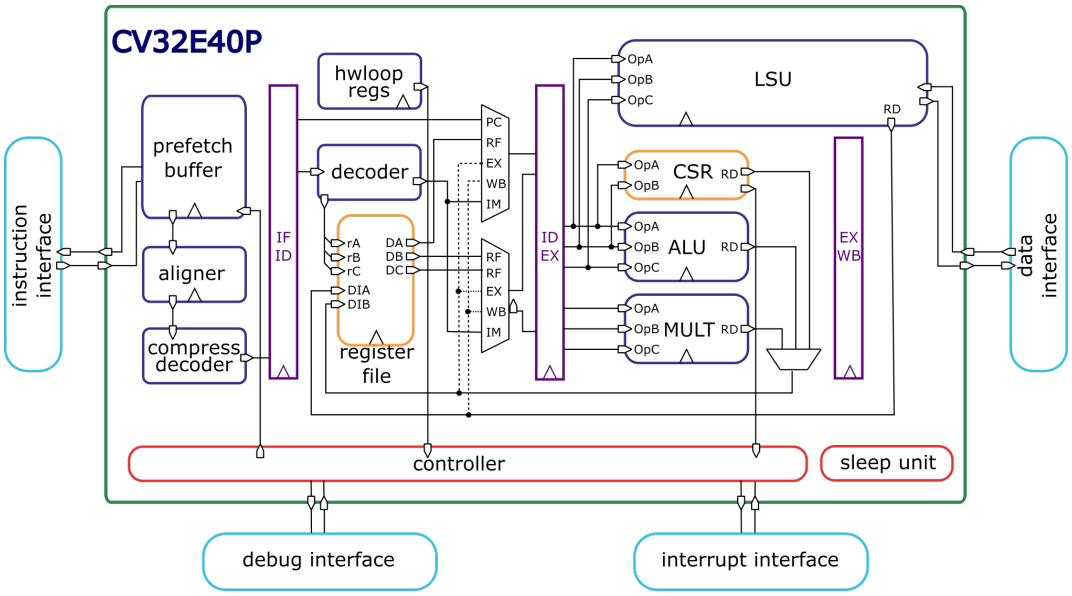

Figure 2.8: Block diagram of CV32E40P RISC-V processor.

2.4 State of The Art

This chapter clarified the importance of post-quantum cryptography and the functioning of the FrodoKEM algorithm, a lattice-based algorithm based on the Learning With Errors problem. The algorithm faces a significant bottleneck in the form of matrix multiplication, which is time-consuming and can be directly observed by profiling the reference software implementation. According to the reference specifications and documentation [8], more than 60% of the execution time is spent on this operation. To address this challenge and enhance the algorithm's performance, we propose to exploit an innovative computational memory that can conduct certain operations within it thanks to the Near Memory Computing approach on which is based.

It is worth noting that our solution is not the first attempt at using hardware to accelerate such algorithms. For instance, in 2016, the first hardware implementation of the LWE scheme in its original form was proposed [12]. The proposal featured

an area-optimized hardware architecture implemented on an FPGA that employed one digital signal processor (DSP) to accelerate matrix multiplication.

Subsequently, in 2018, a low-power implementation was introduced [13]. It was an innovative development because standard lattice-based schemes were deemed impractical on embedded devices, given their large parameters. The architecture is based on a multiplier and an operation scheduling optimized towards low memory usage. Moreover, the matrix generation process is speed up by dedicated hardware accelerators. The FPGA is the target platform for this design.

In 2021, a parallel implementation of FrodoKEM was developed, which relied on multiple digital signal processors (DSPs) to maximize performance [14]. The objective was to concentrate on high throughput by parallelizing the matrix multiplication operations within the cryptographic scheme. Additionally, the sampling of matrices was optimized to feed the multipliers at their maximum capacity. The fastest proposed implementation can feed 16 DSPs in parallel, and the matrix generation process is altered to employ a stream cipher instead of the SHAKE function used in the reference implementation of FrodoKEM [8].

In contrast to the mentioned works, this study proposes a distinct approach to improve matrix multiplication. The parallelism of the computation is intrinsically obtained by processing data directly in the memory. This implies that all matrix elements stored on the same memory line are simultaneously multiplied by the processing unit located adjacent to the memory itself. It is important to note that this solution is not a hardware accelerator. Hardware accelerators are application-dependent, they lack flexibility and versatility, all of which a computational memory possesses.

Chapter 3

Implementation

Chapter 2 provides an overview of cybersecurity, cryptography, and quantum computers, with a particular focus on symmetric and asymmetric cryptography, on which modern cryptosystems are based. We have examined the benefits of the new computing paradigm based on quantum mechanics, while also acknowledging the potential threats that it poses to cryptographic systems. The exponential speedup brought by quantum computers is a major concern that has led to the development of new algorithms capable of running and protecting both classical and quantum computers. In addition, innovative hardware architectures are being explored to improve the performance of these algorithms.

This chapter presents several different implementations: one that computes the key generation function entirely inside the RISC-V, one that delegates the matrix multiplication to the C-SRAM, and finally, one that performs both matrix multiplication and error addition inside the C-SRAM. With the help of C-SRAM, we can boost the performance of FrodoKEM, making it much more efficient in real-world use cases.

3.1 Implementation of the algorithm on RISC-V

Before beginning to work with the C-SRAM, we first had to adapt the implementation of the FrodoKEM key generation function to run on a RISC-V CPU architecture. Our experimental setup utilized a 32-bit RISC-V CPU, with the C-SRAM serving as data memory. Load and store instructions were used to interact with the C-SRAM as if it was a traditional memory. Our starting point for this implementation was the original C code of FrodoKEM developed by Microsoft Research and the FrodoKEM team, which can be found on GitHub [15].

As illustrated in Table 2.1, FrodoKEM has three distinct security levels. In our implementation, we utilized the FrodoKEM-640 version of the Key Generation function, where the variable n is set to 640 and all matrix elements are encoded on 16-bit variables. Both matrices, S and E, are composed of eight columns that each contain 640 values.

The primary objective of the key generation function in FrodoKEM Algorithm 1 is to produce a public-private key pair for the encryption system. The public key is made available to anyone wishing to transmit an encrypted message to the holder of the secret key. The secret key, meanwhile, is kept confidential and used to decode messages that have been encrypted utilizing the associated public key.

Algorithm 1 Reference implementation of the FrodoKEM key generation.

Choose uniformly random seed **seedSE**
Generate a pseudorandom **seedA** with SHAKE
Sample error matrices **S** and **E**
Generate the matrix $\mathbf{A} \in \mathbb{Z}$
Compute $\mathbf{B} \leftarrow \mathbf{A*S + E}$
Return secret key **S** and public key **seedA + B**

To begin the key generation process, it is first essential to generate a random secret value, known as seed, one for matrices S and E, and another one for matrix A. These seeds are subsequently used to create two n by 8 Gaussian random matrices, S and E, where S is the secret key and E is the error. The public key is constructed by concatenating the seed of matrix A and the public matrix B, such that $B = A * S + E$. To perform this matrix multiplication, the function initially generates matrix A using SHAKE128, which is a cryptographic hash function that produces a fixed-length output for an arbitrary-length input. SHAKE128 is a member of the SHA-3 family of hash functions and is designed to provide a high degree of security against all known attacks. The matrix is generated by hashing a sequence of values using the provided seedA as the input to the hash function.

Following the generation of matrix A, the function proceeds to multiply it with S using a standard matrix multiplication algorithm. For each row i of A and each column k of S, the function computes the dot product of the corresponding row of A and the corresponding column of S, and adds the result to the corresponding element of E, the error. Algorithm 2 illustrates the pseudo-code of the matrix multiplication $A * S$ as it is implemented in the reference implementation. The vector B is initialized with values from E, onto which the result of the product is

accumulated in order to derive the final public key.

Algorithm 2 Reference implementation of the FrodoKEM matrix product

```
Generate s, e
b ← e
for i 0 → 640 do
    Generate A[i]
    for j 0 → 8 do
        for k 0 → 640 do
            b[i][j] += A[i][k] × s[k][j]
        end for
    end for
end for
Return b
```

The execution time of this matrix product is largely dominated by the load instructions required to fetch the values of A and S from memory, the MAC instructions, and the storage of the elements of B back into memory.

While the original FrodoKEM library was compatible with numerous platforms running Linux, Windows, or MacOS, our system was bare-metal. This meant that there was no operating system or other abstraction layer between the software application and the hardware. Bare metal programming is commonly used in applications where performance and control are crucial, and where the overhead of an operating system or other abstraction layer would be too high. To adapt the software application to this structure, several changes were necessary.

As the code was running directly on the hardware, we had to rewrite several C functions in a low-level format, simplifying some functions and completely rewriting others. However, one important function, the rand() function in C, could not be updated. This function is a pseudo-random number generator that returns a random integer within a specified range, but the system we were working with lacked any hardware-based random number generation functions. To temporarily overcome this obstacle, we removed the use of the rand() function in the seed generation for the matrices, at the cost of the security of the algorithm. For these experiments, the random seed was hard coded.

In the original code, the matrix A was fully generated before proceeding with the matrix multiplication. However, the memory present in the system was not large enough to contain Frodo-640 A matrix. To work around this issue, we decided

to rewrite the function and generate the A matrix row-wise. Each row was then filled with random values and multiplied by all the columns of the S matrix. As each element of the matrices is 2 bytes, this reduced the memory usage from $2N^2$ to $2N$ bytes.

This initial implementation of the Key Generation function serves as the benchmark against which subsequent implementations utilizing the C-SRAM are evaluated.

3.2 First implementation on C-SRAM: matrix multiplication optimization

After completing the modifications on the Key Generation function, we proceeded with the development of an NMC-based implementation that replaces some computations executed by the CPU with NMC operations performed directly inside the C-SRAM. Our target architecture features a C-SRAM with 256KB and 128-bit memory lines, enabling the multi-lane ALU to operate on 128-bit vectors. Matrix multiplication, one of the primary bottlenecks of the FrodoKEM algorithm, was the operation we aimed to improve.

To clarify, matrix multiplication involves multiplying each element of each row of matrix A by each element of each column of matrix S and subsequently adding the products to obtain a new matrix with dimensions equal to the number of rows of A and the number of columns of S. For example, to compute $B_{i,j}$ with $B = A * S + E$ using matrices A and S, we must sum the following products:

$$B_{1,1} = A_{1,1}S_{1,1} + A_{1,2}S_{2,1} + A_{1,3}S_{3,1} + ...$$

This process is repeated for each element of matrix B.

We can leverage the C-SRAM's Multiply-And-Accumulate (MAC) instruction to perform this operation. However, upon analyzing the code, we discovered that the required instruction for matrix multiplication was the Multiply-and-Accumulate operation on 16 bits, which was absent in the Digital Wrapper's ALU. While it is possible to execute 16-bit operations with 8-bit instructions, this approach creates an overhead that makes pointless the use of a powerful tool such as the C-SRAM. To solve this hardware constraint, we decided to apply some design changes to the ALU of the C-SRAM.

Hardware modification

The Multiply-and-Accumulate (MAC) operation is a fundamental arithmetic operation that involves the multiplication of two numbers and the addition of the result to an accumulator. In hardware, the MAC unit is composed of two components, a multiplier, and an adder. The multiplier takes two input values and generates their product as the output. The adder takes two input values and produces their sum as the output. The MAC unit combines these two operations into a single unit that can multiply two numbers, add the result to an accumulator, and output the updated value of the accumulator in two clock cycles.

To execute a MAC operation, the input values are initially multiplied together using the multiplier. The resulting product is then added to the accumulator using the adder. The updated value of the accumulator is subsequently produced as the updated outcome of the MAC operation. This process can be repeated several times to perform multiple MAC operations sequentially.

Before implementing the Finite Field Multiply-and-Accumulate (FFMAC) operation, a modular multiplication on 16 bits had to be designed. Two different multiplication operators have been added to the instruction set of the C-SRAM as shown in Table 2.4. The first instruction, known as **MULLO**, performs the modular multiplication of two 16-bit operands, and truncates the result to the lowest 16-bits. The resulting operation can be represented as

$$A_i * B_i (mod\ 2^{16})$$

The second instruction, known as **MULHI**, performs the multiplication of two operands on 16 bits, but truncates the outcome to the highest 16 bits. The resulting operation can be represented as

$$(A_i * B_i >> 16)(mod\ 2^{16})$$

The MAC instruction (illustrated in Figure 3.1) is executed in two clock cycles: one cycle is used to multiply the values with **MULLO** and store the 16 least significant bits of the outcome in an internal register. The second clock cycle is used to execute the accumulation. The resulting operation is

$$\sum_{i=1}^{N} (A_i * B_i)(mod\ 2^{16})$$

and requires three sources: A and B for the multiplication and C, which represents the partial sum. The operands and the result are represented in 16 bits, as this is a modular operation.

The new operations can have different addressing modes, depending on the origin of the sources or on the destination of the results: internal registers or the memory. To perform the **FFMAC** operation correctly, two instructions with different addressing modes are required. If we assume that we need to multiply and accumulate `N` times, for `N - 1` times, we require `RMM`, which implies that the sources of the multiplication must be taken from memory, while the partial sum must be stored in an internal register inside the C-SRAM. For the `N-th` MAC operation, however, we need to store the final result back in the memory, and hence the addressing mode will be `MMM`.

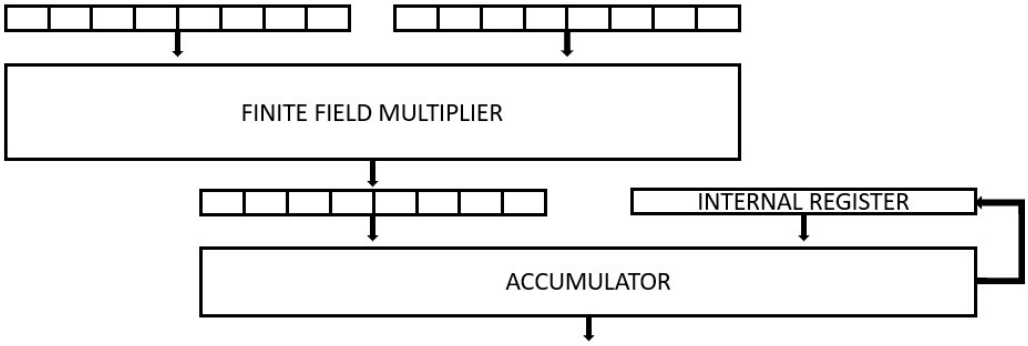

Figure 3.1: Multiply and Accumulate operation scheme

Simulations were conducted to validate the behavior of the new instructions, and a synthesis was performed to analyze the impact of the modifications on the original architecture. The first metric evaluated was the area. Table 3.1 illustrates the difference between the two designs. The impact of the modifications on the ALU shows an increase of the area by 44.59%, but for the entire C-SRAM memory, this change represents just a 0.27% increase in the area, as the Digital Wrapper area is the 5% of the C-SRAM memory.

Design	abs. area before	abs. area after	% area increase
C-SRAM controller	595808.96	597440.28	0,27%
alu	5699.00	8240.13	44,59%

Table 3.1: Hierarchical design area before and after the modifications.

The second metric that has been analyzed is power consumption. The two designs have been evaluated separately, and a summary of the results can be observed in Table 3.2. The power consumption was assessed based on the Internal Power, Switching Power, Leakage Power, and Total Power Consumption. Internal power denotes the power that a device consumes due to the flow of current through its internal circuitry. Switching power, on the other hand, is the power that a device consumes when its transistors switch between states. Leakage power refers to the power that a device consumes when its transistors are inactive but still allow a small amount of current to flow. It is worth noting that some slight variations between the two designs were observed during the analysis. Specifically, there was a minor decrease of the Internal Power by 0.20%, a decrease of the Switching Power by 26.98%, and an increase of the Leakage Power by 0.32%.

Power	Internal P.[mW]	Switch. P.[mW]	Leak. P.[mW]	Tot. P.[uW]
before	4.5945	0.3058	3.727e+03	8.6282
after	4.5852	0.2233	3.7388e+03	8.5473

Table 3.2: Power report results before and after the modifications.

Moreover, we conducted an analysis of the differences in the timing between the two implementations. Timing is a crucial aspect of hardware design that determines how fast a device can operate and how quickly it can respond to input signals. Table 3.3 provides a summary of the results of the timing report. The data required time and data arrival time both increased by approximately 3%. However, this slight change had a drastic impact on the slack, which decreased by 93.65%.

Numerous factors can contribute to timing differences between the two architectures, such as differences in clock frequency, clock distribution, and clock skew. In this case, timing was affected by the addition of new operators, which modified the critical path of the memory. The critical path refers to the path through a circuit that has the longest delay time, namely the time it takes for a signal to propagate through the path, and it is influenced by various factors, such as the capacitance and resistance of the wires and the gate delay of the transistors.

Timing	Before	After
Data required time	4829.15	4984.53
Data arrival time	4821.59	4984.05
slack	7.56	0.48

Table 3.3: Timing report before and after the modifications

Software modification

Before modifying the matrix multiplication function, it is essential to address a crucial matter, which is the arrangement of data within the C-SRAM. To achieve optimal results and minimize the number of transitions to and from the memory, careful consideration is required. Fortunately, an ideal solution has been developed, which is illustrated in detail in Figure 3.2. This strategy makes sure that the way data is arranged in the C-SRAM is improved to make the matrix multiplication function work better and faster.

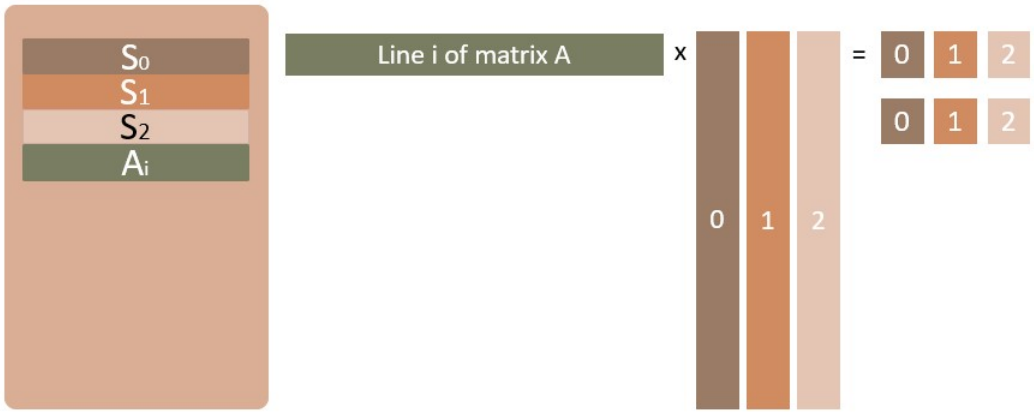

Figure 3.2: Data organization for the first implementation on the C-SRAM

To achieve efficient matrix multiplication, the entire matrix S is loaded into the C-SRAM since every line of matrix A needs to be multiplied by the entire matrix S. As with the previous implementation, matrix A is generated one row at a time and loaded into memory for the matrix multiplication. However, to perform the multiplication, we needed to consider the size of the C-SRAM and compare it with the size of the rows of matrix A (which is the same as the size of each column of matrix S). The current size of the C-SRAM has a width of 128 bits, which means that one line of matrix A for FrodoKEM-640 occupies 80 lines inside the memory. Therefore, we added $N_{ELEMENTS}$ and N_{LINES} parameters to

the existing parameters to make it easier to adapt the code to different memory sizes.

The $N_{ELEMENTS}$ and N_{LINES} parameters play important roles in determining the performance of the matrix multiplication algorithm. $N_{ELEMENTS}$ refers to the degree of parallelism or the number of elements that can be accommodated on a single line of the C-SRAM, which, for our implementation, is always 8 since the width of the memory remains fixed at 128 bits, and all elements are 16 bits in size. On the other hand, N_{LINES} is a critical parameter that determines the number of lines in memory occupied by each line of matrix A. This parameter varies with each implementation of Frodo, depending on the size of the matrices being used, as summarized in Table 3.4. Careful adjustment of N_{LINES} is essential to ensure that it is optimized for the specific size of the matrices being used.

	FrodoKEM-640	FrodoKEM-976	FrodoKEM-1344
N_{LINES}	80	122	168
$N_{ELEMENTS}$	8	8	8

Table 3.4: N_{LINES} parameter with respect to the sizes of FrodoKEM

Figure 3.2 illustrates the modifications made to the code. The matrix multiplication operation was replaced with a series of calls to a function that executes the C-SRAM 16-bit Finite Field Multiply and Accumulate operation, i.e., `nmc_mac16`. Initially, the matrix S is loaded into memory, after which the multiplication process is initiated. A new line of A is generated and saved in memory, while the internal register of the C-SRAM is set to zero. The Multiply-And-Accumulate operation is executed $N\_LINES - 1$ times, with each iteration processing 8 elements of A and 8 elements of the first column of S. Eight MAC operations can be executed in parallel with a single NMC instruction. Since many consecutive accumulations must be done, the results of the MAC instructions are stored in the internal register. For the final MAC operation, the result is saved in the C-SRAM, occupying a single memory line and containing 8 values that are yet to be summed. To calculate the final result, the values are read from memory and added. Subsequently, the error is added to the multiplication, and the process is repeated for each column of S, followed by each line of A.

Algorithm 3 presents the pseudo-code of the NMC-based FrodoKEM implementation. It is worth noting that the number of iterations of the inner loop is reduced by a factor of eight, corresponding to the number of variables processed simultaneously with the `nmc_mac16` instruction. Moreover, the `nmc_mac16` operation requires only three CPU instructions, namely one move immediate instruction

for writing the address field of the `nmc_mac16` instruction on a CPU register, one move immediate instruction for writing the data field of the `nmc_mac16` instruction on another CPU register, and one store instruction that uses the two previously set registers as address and data for sending the command corresponding to the `nmc_mac16` instruction to the C-SRAM.

Algorithm 3 C-SRAM implementation of the FrodoKEM matrix product

```
Generate S, E
Load S in the C-SRAM
b ← E
for i 0 → 640 do
    Generate A[i]
    for j 0 → 8 do
        for k 0 → 640/8 do
            nmc_mac16(reg, A[i][k/8], s[k/8][j])
        end for
        for l 0 → 8 do
            b[i][j] += reg[l]
        end for
    end for
end for
Return b
```

In the original reference implementation, the CPU executed 640 loops, each containing two load, one MAC, and one store instruction. The proposed NMC-based implementation reduces the number of loops to $640/8 = 80$, each containing only one store instruction and two move instructions for initializing CPU registers.

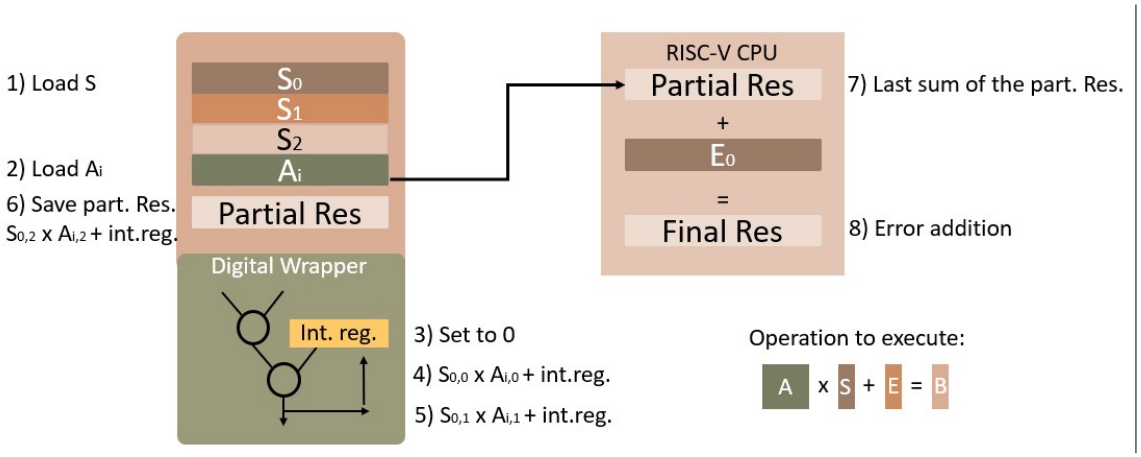

Figure 3.3: Example of the matrix multiplication optimization with on the C-SRAM

Figure 3.3 illustrates a simple example of the operations inside the C-SRAM. Suppose we seek to perform the matrix multiplication of two matrices: A, an n by n matrix, and S, an 8 by n matrix, where n is equal to 24. Following Algorithm 3, we first generate the matrices S and E, and proceed to load S into the C-SRAM. Subsequently, we generate the first row of A, and also load it into the C-SRAM.

Each memory line of the C-SRAM is 128 bits wide and capable of storing 8 elements, where each element occupies 16 bits. Hence, each column of the S matrix and each row of the A matrix occupies 3 memory lines. Before performing the matrix multiplication, the internal register of the C-SRAM is set to zero. The first multiply-and-accumulate (MAC) operation is then executed, wherein the first 8 elements of A and S are multiplied, and the result is stored in the internal register. Next, we extract the second set of 8 elements from both matrices and execute another finite field multiplication. The resulting value is then added to the previous value stored in the internal register. These steps are repeated once more for the last set of 8 elements, after which the resulting value is written back into a new memory line in the C-SRAM. This memory line contains only a partial result of 128 bits, as it is composed of 8 elements.

Finally, this memory line is transmitted to the processor's CPU to perform the last sum of these 8 values and obtain the first number of the resulting matrix. The entire process is then repeated for each column of S and for each row of A. Once the new matrix is computed it is possible to add the error and complete the operation.

3.3 Second implementation on C-SRAM: matrix multiplication and addition optimizations

The previous section described all the modifications made to implement the FrodoKEM Key Generation algorithm onto a Near Memory Computing architecture. Although we took advantage of the C-SRAM memory to speed up the matrix multiplication, this memory is not limited to this operation alone. To improve the results even more we decided to perform also the error addition inside the C-SRAM.

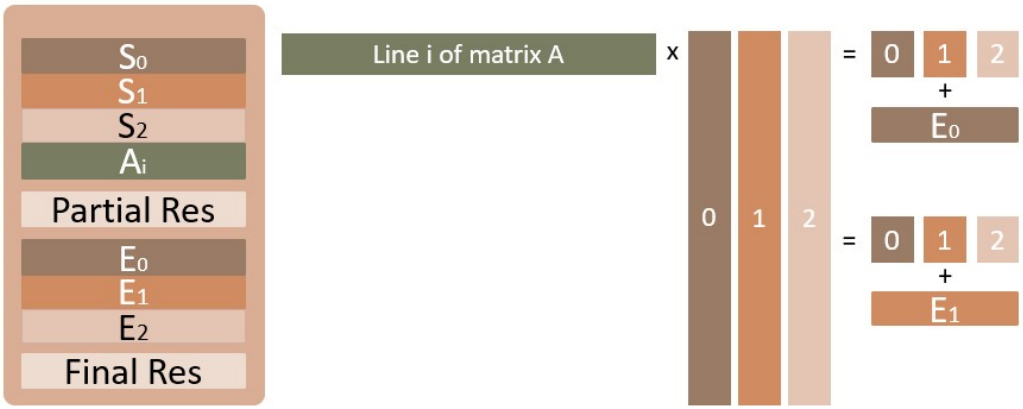

Figure 3.4: Data organization for the second implementation on the C-SRAM

The pseudo-code of this implementation is presented in Algorithm 4. Notably, the central portion of the algorithm remained identical to Algorithm 3, up to the last stage of adding the partial result of each MAC within the RISC-V. In this stage, the values are read from the memory and added together to obtain the ultimate result. This sequence of instructions is subsequently repeated for each column of S, generating a vector that contains the first 8 values of the new matrix. This vector is saved in the C-SRAM memory so we are able to perform an NMC addition between 2 lines and add the first row of the new matrix to the first row of the E matrix, previously stored in the C-SRAM memory after the S matrix.

Algorithm 4 C-SRAM implementation of the FrodoKEM matrix product with the error addition inside the C-SRAM

Generate **S**, **E**
Load **S**, **E** in the C-SRAM
for i $0 \rightarrow 640$ **do**
    Generate $\mathbf{A}[i]$
    for j $0 \rightarrow 8$ **do**
        for k $0 \rightarrow 640/8$ **do**
            `nmc_mac16(reg, A`$[i][k/8]$`, S`$[k/8][j]$`)`
        end for
        for l $0 \rightarrow 8$ **do**
            tmp[l] $\leftarrow$ `reg`[l]
        end for
    end for
    Load **tmp** in the C-SRAM
    `nmc_add16(reg, tmp, E`$[i]$`)`
    b[i] $\leftarrow$ `reg`
end for
Return **b**

Chapter 4

Discussion of the results and conclusion

Results

Chapter 4 provides a detailed exposition of the distinct implementations and alterations made to both hardware and software components, with the aim of improving the performance of the FrodoKEM Key Generation algorithm. To evaluate the resultant outcomes, it was necessary to establish specific metrics, serving as quantitative measures to evaluate the efficacy or efficiency of the developed system. This facilitated an objective and uniform approach to measuring the performance of the system and comparing different versions of the same.

The initial metric that was considered was the execution time, referring to the total time taken to run a given program or algorithm. The quicker the execution time, the better the performance. In order to make a comparison between the outcomes of various implementations, we measured the execution time of the reference matrix product implementation of FrodoKEM, adapted for the target CPU. Table 4.1 provides details of the execution time of the $\mathbf{B} = \mathbf{A} \times \mathbf{S} + \mathbf{E}$ operation, in terms of clock cycles and the number of CPU instructions.

Table 4.1: FrodoKEM-640 Reference Implementation

	Clock Cycles	CPU instructions
Matrix multiplication	39,383,040	29,545,600

Upon obtaining the golden model or benchmark, we proceeded to alter the reference implementation of FrodoKEM, in order to make use of the Computational SRAM (as demonstrated in Algorithm 3). C macros were utilized to automatically construct NMC instructions, while at the same time, permitting the programmer to use an assembly-like syntax. This method simplified software support for the C-SRAM, without necessitating any modifications to the compilation toolchain. Table 4.2 illustrates the execution time performance of the $\mathbf{B} = \mathbf{A} \times \mathbf{S} + \mathbf{E}$ operation, in terms of clock cycles and the number of CPU instructions, when utilizing the NMC approach.

	Clock Cycles	CPU instructions
Matrix multiplication	9,597,440	5,485,440
Performance Gain	76%	81%

Table 4.2: FrodoKEM-640 NMC based Implementation

The present study has determined that the NMC architecture offers a notable improvement in computational efficiency, with a 76% reduction in clock cycles required for computation when compared to the reference implementation. This acceleration of computational speed results in a reduction of the execution time required for computation by a factor of approximately four. Furthermore, the study found that the NMC architecture resulted in a significant improvement in performance, with the number of executed instructions being reduced by a factor of approximately five. It is notable, however, that while the theoretical expectation would be a factor of eight improvements in computational speed, the practical implementation is impacted by certain overheads. In particular, when eight Multiply-and-Accumulate (MAC) operations are performed in parallel on the same C-SRAM line, eight partial results are obtained on the destination line. These intermediate values must then be summed together by the CPU to obtain the final result. This post-processing of the MAC results adds additional computation overhead, which partly explains the difference between the theoretical and practical gains in computational efficiency observed in this study.

	Execution time (CC)
SW ARM M4 [13]	85,585,315
SW RISC-V [**this work**]	82,500,000
SW NMC [**this work**]	72,610,000
HW (1 DSP) [13]	3,276,800
HW (8 DSPs) [14]	408,988

Table 4.3: State-of-the-art implementations of the FrodoKEM-640 key generation

The NMC-based implementation outperforms the implementations on both RISC-V and ARM microprocessors. Specifically, a 15% improvement in execution time is observed compared to the ARM M4 software implementation in [12], and a 12% improvement compared to the RISC-V software implementation used as the reference in this work. Table 4.3 displays the execution times, in clock cycles, of all the implementations considered in this comparison. The number of employed *digital signal processors* (DSP) for hardware implementations is also shown, providing an indication of the number of MAC operations performed in parallel.

Although hardware accelerators [12, 14] perform better than the NMC solution, they are customized for the FrodoKEM application, accelerating both the matrix product and the matrix generation.

The solutions presented in [12] and [14] strongly optimize the pseudo-random generation required to produce **A**, **S** and **E**. In [14], a lightweight stream cipher is used to significantly enhance the matrix generation.

Moreover, the NMC approach differs structurally from hardware accelerators, as it avoids the overhead of implementing dedicated memories. For instance, the hardware implementation proposed in [12] employs a total of 13.5KB of extra RAM memory dedicated to the accelerator.

This proposal allows us to gain a performance advantage over software implementations while incurring only a small overhead on the periphery of the data RAM. It is worth noting that the Processing Unit inside the C-SRAM is versatile and can be utilized to accelerate any other application that can benefit from the provided level of parallelism.

4.0.1 Conclusion

In today's world, the protection of sensitive information is of utmost importance. With the advent of quantum computing, traditional cryptographic methods may no longer be secure, and this could have alarming consequences for individuals and organizations alike. Therefore, the adoption of Post-Quantum Cryptography

(PQC) is essential to ensure the security of sensitive information. One promising PQC scheme is FrodoKEM, which boasts a high level of security. However, due to its complexity, it requires the handling of large matrices and a significant amount of computations to solve matrix products. Software implementations can be especially time-consuming due to the need to transfer data between the CPU and memory.

To address this issue, this work proposes an implementation of FrodoKEM on a Near-Memory Computing (NMC) architecture. By using a Computational SRAM, which can perform arithmetic operations directly on data stored on the same line of the data RAM, this architecture can accelerate the matrix product of FrodoKEM by a factor of four in terms of clock cycles. The performance gain on the whole key generation function is 12%. Despite the minimal area overhead that comes with adding some digital logic at the periphery of the data RAM, this architecture remains flexible and can accelerate general-purpose operators, benefiting any application that could benefit from NMC.

While the improvement in the performance of the FrodoKEM Key Generation function is impressive, it only solves one of the algorithm's main bottlenecks. Future work could expand to optimize not only the FrodoKEM key generation function but also the encryption and decryption processes. In addition, it is possible to implement new functionalities in the C-SRAM to optimize the matrix generation and possibly the generation and storage of the secret key, thus further improving the algorithm's security. By reducing the need to move the secret key through the busses from the CPU to the memory, we can make the algorithm more robust and secure. Therefore, investing in research and development to enhance post-quantum cryptography is critical to ensure the protection of sensitive information in the future.

Appendix A

Reference implementation

```
int frodo_mul_add_as_plus_e(uint16_t *out, const uint16_t *s, const
    uint16_t *e, const uint8_t *seed_A) {

    int i, j, k, m;
    int16_t A[PARAMS_N];                                          //
    work with one row at time, do not save the entire matrix

    uint8_t seed_A_separated [2 + BYTES_SEED_A];
    uint16_t *seed_A_origin = (uint16_t *) &seed_A_separated;

    for(i = 0; i < BYTES_SEED_A; i++)
        seed_A_separated [i+2] = seed_A[i];

    for(i = 0; i < PARAMS_NBAR PARAMS_N* sizeof(uint16_t); i++)
        out[i] = e[i];

    for(i=0; i<PARAMS_N; i++) {
        for (m = 0; m(PARAMS_N); m++)                             //
    initialize to 0 the array
            A[m] = 0;

        seed_A_origin[0] = (uint16_t) (i);
        shake ((unsigned char*) (A), (unsigned int) (2* PARAMS_N),
    seed_A_separated, 2 + BYTES_SEED_A);

        for (k = 0; k < PARAMS_NBAR; k++) {
            uint16_t sum = 0;
            for(j=0; j < PARAMS_N; j++)
                sum += A[j]*s[k* PARAMS_N + j];
            out[i* PARAMS_NBAR + k] += sum;
        }
```

```
28      }
29      return 1;
30  }
```

Appendix B

First implementation

```
int frodo_mul_add_as_plus_e(uint16_t *out, const uint16_t *s, const
    uint16_t *e, const uint8_t *seed_A) {

    int i, j, k, m;
    int16_t A[PARAMS_N];                                          //
    work with one row at time, do not save the entire matrix

    uint8_t seed_A_separated [2 + BYTES_SEED_A];
    uint16_t *seed_A_origin = (uint16_t *) &seed_A_separated;

    for(i = 0; i < BYTES_SEED_A; i++)
        seed_A_separated [i+2] = seed_A[i];

    for(i = 0; i < PARAMS_NBAR PARAMS_N* sizeof(uint16_t); i++)
        out[i] = e[i];

    for(i=0; i<PARAMS_N; i++) {
        for (m = 0; m(PARAMS_N); m++)                             //
    initialize to 0 the array
            A[m] = 0;

        seed_A_origin[0] = (uint16_t) (i);
        shake ((unsigned char*) (A), (unsigned int) (2* PARAMS_N),
    seed_A_separated, 2 + BYTES_SEED_A);

        /******************* Matrix multiplication
    ********************/
        /*-- Write A line in CSRAM > for each line of A N_LINES in
    CSRAM --*/
        for (k=0; k<N_LINES; k++) {
            for(j=0; j<(PARAMS_N/N_LINES); j++){
```

```
26                  csram_write16 (CSRAM_BASE_ADDR, (uintptr_t)a[k], j, A
    [k*N_ELEMENTS + j]);
27              }
28          }
29
30          for (k = 0; k < PARAMS_NBAR; k++) {
31              _cm_bcast8_r(0);                                        //
    internal reg initialized to 0
32
33              /* MAC N-1 times, result in internal register */
34              for(j=0; j<(N_LINES 1); j++) {
35                  _cm_ffmac16_rmm((uint32_t) (h2_cxram_line_t *)s[(k*
    N_LINES + j)], (uint32_t) (h2_cxram_line_t *)a[j]);
36              }
37              /* MAC for the last time, result written in the CSRAM */
38              _cm_ffmac16_mmm ((uint32_t) (h2_cxram_line_t *) res, (
    uint32_t) (h2_cxram_line_t *)s[k*N_LINES + (N_LINES 1)], (uint32_t
    ) (h2_cxram_line_t *) a[ (N_LINES - 1)]);
39
40              for(j=0; j<PARAMS_N/N_LINES; j++){
41                  out[i* PARAMS_NBAR + k] += csram_read16 (
    CSRAM_BASE_ADDR, (uintptr_t) res, j);
42              }
43          }
44      }
45      return 1;
46  }
```

Appendix C

Second implementation

```
1  int frodo_mul_add_as_plus_e(uint16_t *out, const uint16_t *s, const
       uint16_t *e, const uint8_t *seed_A) {
2
3      int i, j, k, m;
4      int16_t A[PARAMS_N];                                          //
       work with one row at time, do not save the entire matrix
5      uint16_t mac_res[PARAMS_NBAR];
6
7      uint8_t seed_A_separated [2 + BYTES_SEED_A];
8      uint16_t *seed_A_origin = (uint16_t *) &seed_A_separated;
9
10     for(i = 0; i < BYTES_SEED_A; i++)
11         seed_A_separated [i+2] = seed_A[i];
12
13     for(i = 0; i < PARAMS_NBAR PARAMS_N* sizeof(uint16_t); i++)
14         out[i] = e[i];
15
16     for(i=0; i<PARAMS_N; i++) {
17         for (m = 0; m(PARAMS_N); m++)                             //
       initialize to 0 the array
18             A[m] = 0;
19
20         for (m = 0; m < (PARAMS_NBAR); m++)                       //
       initialize to 0 the array
21             mac_res [m] = 0;
22
23         seed_A_origin[0] = (uint16_t) (i);
24         shake ((unsigned char*) (A), (unsigned int) (2* PARAMS_N),
       seed_A_separated, 2 + BYTES_SEED_A);
25
```

```
26      /******************* Matrix multiplication
    *******************/
27      /*--- Write A line in CSRAM > for each line of A N_LINES in
    CSRAM ---*/
28      for (k=0; k<N_LINES; k++) {
29          for(j=0; j<(PARAMS_N/N_LINES); j++){
30              csram_write16 (CSRAM_BASE_ADDR, (uintptr_t)a[k], j, A
    [k*N_ELEMENTS + j]);
31          }
32      }
33
34      for (k = 0; k < PARAMS_NBAR; k++) {
35          _cm_bcast8_r(0);                                    //
    internal reg initialized to 0
36
37          /* MAC N-1 times, result in internal register */
38          for(j=0; j<(N_LINES 1); j++) {
39              _cm_ffmac16_rmm((uint32_t) (h2_cxram_line_t *)s[(k*
    N_LINES + j)], (uint32_t) (h2_cxram_line_t *)a[j]);
40          }
41
42          /* MAC for the last time, result written in the CSRAM */
43          _cm_ffmac16_mmm ((uint32_t) (h2_cxram_line_t *) res, (
    uint32_t) (h2_cxram_line_t *)s[k*N_LINES + (N_LINES 1)], (uint32_t
    ) (h2_cxram_line_t *) a[ (N_LINES - 1)]);
44
45          for(j=0; j<PARAMS_N/N_LINES; j++){
46              mac_res[k] += csram_read16 (CSRAM_BASE_ADDR, (
    uintptr_t) res, j);
47          }
48      }
49
50      /* Write partial results in CSRAM and sum to E */
51      for(j=0; j<PARAMS_NBAR; j++){
52          csram_write16 (CSRAM_BASE_ADDR, (uintptr_t) part_res, j,
    mac_res[j]);
53      }
54
55      // add in csram
56      _cm_add16_mmm( (uint32_t) (h2_cxram_line_t *) res, (uint32_t)
     (h2_cxram_line_t *)e[i], (uint32_t) (h2_cxram_line_t *) part_res)
    ;
57
58      // read result line from CSRAM and write final result in out
    array
59      for (k = 0; k < PARAMS_NBAR; k++)
60          out[i* PARAMS_NBAR + k] = csram_read16 (CSRAM_BASE_ADDR,
    (uintptr_t) res, k);
61  }
```

```
62      return 1;
63  }
```

Bibliography

[1] Richard P Feynman. «Simulating physics with computers, 1981». In: *International Journal of Theoretical Physics* 21.6/7 (1981) (cit. on p. 6).

[2] John Mulholland, Michele Mosca, and Johannes Braun. «The Day the Cryptography Dies». In: *IEEE Security Privacy* 15.4 (2017), pp. 14–21. DOI: 10.1109/MSP.2017.3151325 (cit. on p. 8).

[3] P. W. Shor. «Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer». English. In: *SIAM Journal on Computing* 26.5 (1997). Cited By :4523, pp. 1484–1509. URL: www.scopus.com (cit. on p. 8).

[4] L. K. Grover. «A fast quantum mechanical algorithm for database search». English. In: *Proceedings of the Annual ACM Symposium on Theory of Computing*. Vol. Part F129452. Cited By :4223. 1996, pp. 212–219. URL: www.scopus.com (cit. on p. 8).

[5] Daniel J. Bernstein. «Introduction to post-quantum cryptography». In: *Post-Quantum Cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–14. ISBN: 978-3-540-88702-7. DOI: 10.1007/978-3-540-88702-7_1. URL: https://doi.org/10.1007/978-3-540-88702-7_1 (cit. on p. 8).

[6] Frank Arute et al. «Quantum supremacy using a programmable superconducting processor». In: *Nature* 574.7779 (2019), pp. 505–510 (cit. on p. 9).

[7] Michele Mosca. «Cybersecurity in an Era with Quantum Computers: Will We Be Ready?» In: *IEEE Security Privacy* 16.5 (2018), pp. 38–41. DOI: 10.1109/MSP.2018.3761723 (cit. on p. 9).

[8] Erdem Alkim et al. «FrodoKEM learning with errors key encapsulation». In: *NIST PQC standardization: Round* 3 (2020) (cit. on pp. 10, 20, 21).

[9] J. von Neumann. «First draft of a report on the EDVAC». In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75. DOI: 10.1109/85.238389 (cit. on p. 15).

[10] Maha Kooli, Antoine Heraud, Henri-Pierre Charles, Bastien Giraud, Roman Gauchi, Mona Ezzadeen, Kevin Mambu, Valentin Egloff, and Jean-Philippe Noel. «Towards a Truly Integrated Vector Processing Unit for Memory-bound Applications Based on a Cost-competitive Computational SRAM Design Solution». In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 18.2 (2022), pp. 1–26 (cit. on p. 16).

[11] P Prinz, T Crawford, JL Hennessy, and DA Patterson. *Computer Architecture: A Quantitative Approach*. 2018 (cit. on p. 17).

[12] J. Howe, C. Moore, M. O'Neill, F. Regazzoni, T. Güneysu, and K. Beeden. «Lattice-based encryption over standard lattices in hardware». In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2016, pp. 1–6. DOI: 10.1145/2897937.2898037 (cit. on pp. 20, 39).

[13] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. «Standard Lattice-Based Key Encapsulation on Embedded Devices». In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (Aug. 2018), pp. 372–393. DOI: 10.13154/tches.v2018.i3.372-393. URL: https://tches.iacr.org/index.php/TCHES/article/view/7279 (cit. on pp. 21, 39).

[14] James Howe, Marco Martinoli, Elisabeth Oswald, and Francesco Regazzoni. «Exploring Parallelism to Improve the Performance of FrodoKEM in Hardware». In: *Journal of Cryptographic Engineering* 11 (Nov. 2021). DOI: 10.1007/s13389-021-00258-7 (cit. on pp. 21, 39).

[15] Microsoft. *PQCrypto-LWEKE*. https://github.com/Microsoft/PQCrypto-LWEKE (cit. on p. 23).