Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 14, Number 1, pages 22–26, January 2025

Evaluation of Gaussian elimination using HLS for fast public key generation in the Classic McEliece

Masashi Kihara
Graduate School of Science and Engineering
National Defense Academy of Japan
Yokosuka, Japan
em62047@nda.ac.jp

Keisuke Iwai
Department of Computer Science
National Defense Academy of Japan
Yokosuka, Japan
iwai@nda.ac.jp

Takashi Matsubara
Department of Computer Science
National Defense Academy of Japan
Yokosuka, Japan
matubara@nda.ac.jp

Takakazu Kurokawa
Department of Computer Science
National Defense Academy of Japan
Yokosuka, Japan
kuro@nda.ac.jp

***Abstract*—With the recent development of quantum computers and massively parallel computing, the continued use of existing cryptographic techniques is in jeopardy. For this reason, standardization of Post-Quantum Cryptography (PQC) is underway at the National Institute of Standards and Technology (NIST) in the United States and other national standardization organizations. While several cryptosystems have already been finalized as part of the standard, there are still proposals under discussion, including the Classic McEliece Cryptosystem, which is the only code-based cipher remaining in the U.S. NIST proposal. The characteristic feature of the Classic McEliece Cryptosystem is its very large key size, with a maximum public key size of approximately 1.4 MB. Generating a public key requires a very large matrix calculation. Because of the large size of the matrix, the time required for generation is also large, and even after optimization, it is the most time-consuming process. In this paper, we propose an FPGA implementation of the Gaussian elimination method to accelerate the generation of public keys for the Classic McEliece Cryptosystem using HLS. By implementing the method on an FPGA for data centers suitable for HLS, the CPU load can be reduced, and the public key can be obtained from the FPGA. As a result of the implementation, the processing time was about 0.1 seconds per operation with the largest parameter size. Since the speed was about the same as the calculation on a partially optimized CPU, parallelization of this calculation can be expected to result in faster key generation.**

***Keywords*—FPGA, Classic McEliece, Post-Quantum Cryptography, High Level Synthesize,**

I. Introduction

Quantum resistant cryptography must be widely used before quantum computers can be widely deployed. This is because it will no longer be possible to establish secure communication using existing cryptographic methods. Therefore, national standards organizations are evaluating quantum cryptography and trying to establish a standard. Especially in the standardization at NIST in the U.S. [1], some quantum-resistant ciphers have already been published as standard specifications [2]. However, some ciphers still remain under evaluation and are under discussion as PQC Round-4 [3].

Classic McEliece [4] is a type of key encapsulation mechanism (KEM) and is the only remaining code-based cryptosystem in PQC Round-4. Classic McEliece has by far the largest key size. Classic McEliece's public key of approximately 1.4 MB is 1000 times larger than CRYSTALS-KYBER's public key of 1.5 KB, when compared to the maximum parameters. Considering that even the key length of CRYSTALS-KYBER is larger than that of existing KEM, the size of Classic McEliece's public key is enormous.

On the other hand, as advantages, the encryption and decryption processes are fast, and the computational cost other than key generation is lower than that of other proposed post quantum cryptosystems [6]. In addition, it is a very stable cryptosystem and can be said to be reliable even today, although various attack methods have been proposed for more than 40 years since its proposal as a basic cryptosystem.

Classic McEliece takes a large amount of time to generate a pair of keys due to the size of its public key. This has led to proposals [7] for hardware implementations that focus on public key generation, and proposals [8] to reduce the communication overhead by caching the public key itself.

In this paper, we first evaluate the properties of Classic McEliece in software implementation from an implementation with reference code. Then, we evaluate the FPGA implementation with the goal of speeding up key generation by speeding up the Gaussian elimination method using FPGA with HLS.

II. Background of Classic McEliece

A. Classic McEliece Cryptosystem

Classic McEliece is a code-based KEM. Classic McEliece generates a code from a generator matrix using the Goppa code, and then mixes errors arbitrarily into the code, as shown in Figure 1, so that only the target with a specific parity check matrix can decode the code. This basic structure is based on the McEliece code. The basic structure of this robust system has not changed since 1978, when the McEliece cipher was proposed [9].

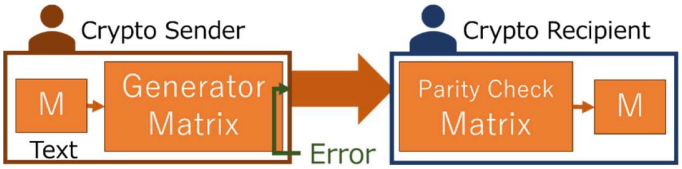

Figure 1: Code-based cryptographic communication

In the Classic McEliece algorithm [8] shown below B, most of the computational work in key generation consists of generating a public key, which is finally obtained from a random GF(2) irreducible polynomial by Gaussian elimination. If the Gaussian elimination reveals that the matrix is not reduced row-echelon form, the key generation is considered to have failed, and the key generation is repeated in the same way until the next random input succeeds. Since the decision renders the previous process useless, more processing time is required in the case of key generation failure.

B. Key Generation Algorithm[8]

The following is a key generation algorithm

1. Generate a random degree t irreducible polynomial $g(x)$
2. Generate a random permutation $(\alpha_0, \alpha_1, \dots, \alpha_{n-1})$ through sorting random prefix.
3. Compute $t \times n$ matrix $H(\alpha_j^{i-1}/g(\alpha_j))$ calculated from $g(x)$ and $(\alpha_1, \dots, \alpha_{n-1})$.
4. Extend to $mt \times n$ matrix T by writing each element as column m bit vectors from H.
5. Gaussian elimination to this matrix T into it systematic form $[I_{mt}, \hat{T}]$. If this fails, return to step 1.
6. Output: Public key: $\hat{T}$, Secret key: $(g(x), (\alpha_0, \alpha_1, \dots, \alpha_{n-1}))$

Table I shows the parameters and key sizes for each strength parameter set. Key sizes for Kyber-1024, the maximum strength of CRYSTALS-KYBER, which has already become the standard, are listed as well. Compared to the key sizes of CRYSTALS-KYBER with similar strengths, Classic McEliece has a very large key.

TABLE I. PARAMETER SETS OF CLASSIC MCELIECE AND KYBER

parameter set	m	n	t	PK size [byte]	SK size [byte]	CT size [byte]
mceliece348864	12	64	3488	261120	6492	96
mceliece460896	13	96	4608	524160	13608	156
mceliece6688128	13	128	6688	1044992	13932	208
mceliece6960119	13	119	6960	1047319	13948	194
mceliece8192128	13	128	8192	1357824	14120	208
Kyber-1024	-	-	-	1568	3168	1568

C. Related Work in Classic McEliece

A hardware implementation of Classic McEliece has already been proposed in [10] and implemented for each parameter respectively. In this implementation, Gaussian elimination is performed using a dedicated operator. There is also an example of SW/HW Co-design implementation [11]. This example shows that the Gaussian elimination method accounts for most of the computation time in public key generation, which takes up most of the computation process. In the software implementation, the Gaussian elimination method is shown to take more time when using CPU extension instructions.

Even if we focus on public key generation, the size of the hardware becomes large, and it is difficult to implement on a small FPGA. Therefore, public key generation is often targeted at servers and other computers with large computing resources. However, there are also implementations[12] aiming at low power consumption and memory saving, such as those for embedded systems, and it would be difficult to adapt them to the real world without improving their efficiency.

III. EVALUATING GAUSSIAN ELIMINATION IN C LANGAGE

A. Reference Key generation in CPU

The reference code submitted to NIST is a package written in C. It contains the required packages except for Secure Hash Algorithm 3 (SHA-3), which is contained in a directory broken down by parameter. As a hash algorithm, Classic McEliece uses as input the numbers obtained from SHAKE256, a function that provides a hash of arbitrary length bits of SHA-3. There are two implementations in the proposal [8], one using systematic form and the other using partially semi-systematic form. The implementation using partially semi-systematic form has been shown to be faster in software implementation. The size of the matrix required for public key generation in Classic McEliece is $mt \times n/8$. The maximum parameter for this is 1644×1024. Since each is held as a bitstring in char format, its capacity is 1.6 MB. Not many CPUs have a cache size that can accommodate this. Matrix operations require many memory references, so they are not suitable for CPU processing.

However, recent CPUs have extended instructions that can perform SIMD operations more efficiently. These are called vector instructions and can efficiently process multi-bit length computations. Even consumer products, especially high-end products, support these instructions, and reference code has been submitted in the form of code that supports the AVX instructions. Other additional implementations include extended instructions such as SSE.

Another common optimization is the -O option of the GNU Compiler Collection (GCC). This option performs optimizations in stages from O0 to O3. The generally recommended level of this option is O2. At option level O3, strong optimizations such as loop transformations and memory access transformations are applied. This option is a stronger optimization for calculations with frequent memory accesses, such as matrix operations.

B. Valication & Evalution In CPU

In this study, the rate of the public key generation process is first checked in a typical CPU environment. Since the Zynq UltraScale was used in the previous study [11], the processor used in the evaluation is the ARM processor for embedded applications built into the Zynq. Here, the same verification is performed on a CPU with general extended instructions. The environment used for the verification is as follows

- Profile retrieval: Flame Graph[13]
- CPU: Intel i7-9700
- OS: Ubuntu 22.04 on WSL 2

First, we measure the actual execution time for each parameter set. The execution times for each parameter set are shown in Table II . These numbers are obtained from "omp_get_wtime();", a standard OpenMP function. Basically, cryptographic processing time is often evaluated in terms of the number of CPU cycles [6]. This is because it is difficult to measure the real time of ordinary ciphers because they can be processed so quickly. However, Classic McEliece takes a long time to process, so we measured the real time.

The actual execution times for each parameter showed that GCC's optimization options were strongly adapted to the Intel Core i7-9700 processor. Compared to O0 with no options, the O3 option reduced the execution time by a maximum of nearly 1/90th. In addition, the process is even faster when the extended instruction AVX is used, and Gaussian elimination is performed in tens of milliseconds.

TABLE II. ACTUAL EXECUTION TIME FOR EACH PARAMETER SETS

parameter sets	AVX [s]	O3 [s]	O2 [s]	O0 [s]
mceliece348864	0.001998	0.009063	0.186995	0.878694
mceliece460896	0.005808	0.032082	0.684219	3.569551
mceliece6688128	0.014522	0.098582	1.662650	7.937640
mceliece6960119	0.013467	0.087751	1.429807	6.993926
mceliece8192128	0.015053	0.102067	2.004971	9.399704

Profiles were obtained for each parameter set. The profiles obtained from the smallest and largest parameter sets to observe the changes are shown in Figure 2 below. This figures shows the percentage with the horizontal axis at 100, and the functions called are stacked on the vertical axis. In other words, the longer a function takes to execute, the longer it is displayed horizontally. It can be seen that the blue arrows are the functions that perform public key generation and account for almost half of the total in KEM.

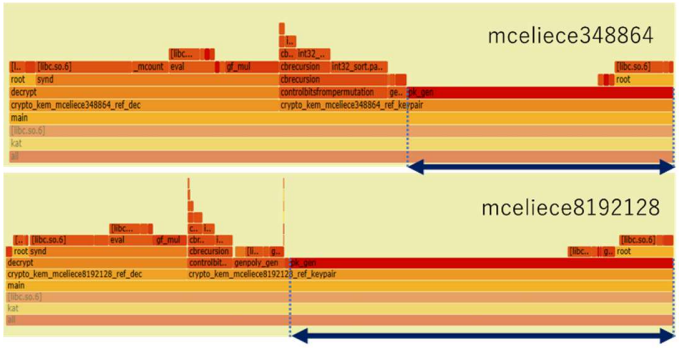

Figure 2: Flame Graph with normal optimization of KEM

Next, we show the profiles for each parameter set when optimized using the AVX extension instructions. As in the previous section, the following figures show the results for the smallest and largest parameter sets. As in Figure 2, the blue arrows indicate the functions responsible for public key generation, and although the overall execution time decreases when the AVX extension instruction is used, public key generation accounts for nearly 90% of the process.

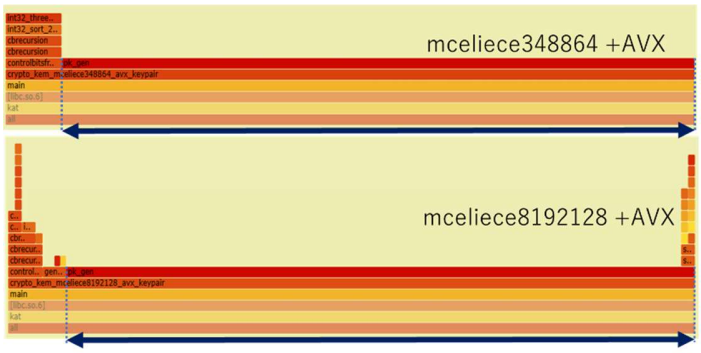

Figure 3: Flame Graph using KEM with AVX

Gaussian elimination is included in the function "pk_gen" and accounts for most of it. The percentage of Gaussian elimination in "pk_gen" may decrease with the use of extended instructions, since matrix computation is accelerated, but the percentage of "pk_gen" is used for the purpose of comparison with [11]. Table III shows the execution times for the maximum and minimum parameter sets for the extended instructions and for the normal optimization. The obtained results show that most of the time is spent on public key generation. In addition, at most 70% of the time in the normal GCC optimization implementation was spent on public key generation in KEM. From these results, it can be seen that studies focusing on Gaussian elimination, as in [14], contribute to the speedup of Classic McEliece. The results of this measurement show different trends for the ARM-based Zynq UltraScale processor and the Intel Core i7-9700. More time needs to be allocated to public key generation on Zynq UltraScale's ARM processor than on the vector instructions on the Intel CPU. Therefore, we found that the ARM-based embedded CPU required more hardware implementation than this regular consumer CPU.

TABLE III. PERCENTAGE OF EXECUTION TIME FOR PUBLIC KEY GENERATION TO KEM

	Function pk_gen (including Gaussian elimination)
	Exec. time [%]
mceliece348864 + AVX	91.9
mceliece348864	48.4
mceliece8192128 + AVX	91.6
mceliece8192128	79.0
[11] Vectorized	97.6
[11] Baseline	81.8

IV. High Level Synthesise

A. HLS Implementation

In this study, we implemented a dedicated hardware in HLS to efficiently process this huge matrix, which can be easily ported from the reference implementation in C. In addition, the implementation can be made more portable by using FPGAs suitable for data centers. In addition, the implementation can be made more portable by using FPGAs suitable for data centers.

For this evaluation, we used the Alveo U250 from AMD/Xilinx. This FPGA, which is designed for server-side applications, allows the corresponding FPGA to perform processing through a software interface called XRT. Using this runtime, processing can be passed from the CPU to an external processor like OpenCL or CUDA. HLS, like a parallel computer, converts the for instructions in the C language into parallelization and pipelining by compiler directives. HLS also performs optimization for high-speed processing by performing appropriate memory transfers and buffer retention for the specified parallelization or pipelining.

The versions of each library and runtime used are as follows:

- v++: v2023.1
- vitis_hls: v2023.1
- XRT: 2.16.204

v++ is operated from the CUI in the same way as the GCC compiler. Vitis_hls is a GUI-based software that performs High Level Synthesis from C, C++, and OpenCL. Using these two high level synthesis software, we performed logic synthesis and evaluated the execution time. These operations can be thought of as the equivalent of blocking and unrolling, which are often performed in matrix calculations for high-performance computing.

Gaussian elimination can be roughly decomposed into two loops: forward elimination and backward substitution. The number of loops is predetermined by the parameters of Classic McEliece. As shown in a previous study [11], we also optimize access by storing the matrix columns together as a cache. The frequency suitable for HLS can be maintained by unrolling for the divided loops and parallelizing them. At this time, the array is divided into an appropriate number of arrays for the number of loops to achieve higher speed.

This operation is as shown in Figure 4. Pipelining is performed for each forward erase and backward assignment. Blocking avoids excessive resource consumption by limiting memory accesses. However, since pipelining is performed by HLS, it is not clear whether the process is strictly overlaid as shown in Figure 4.

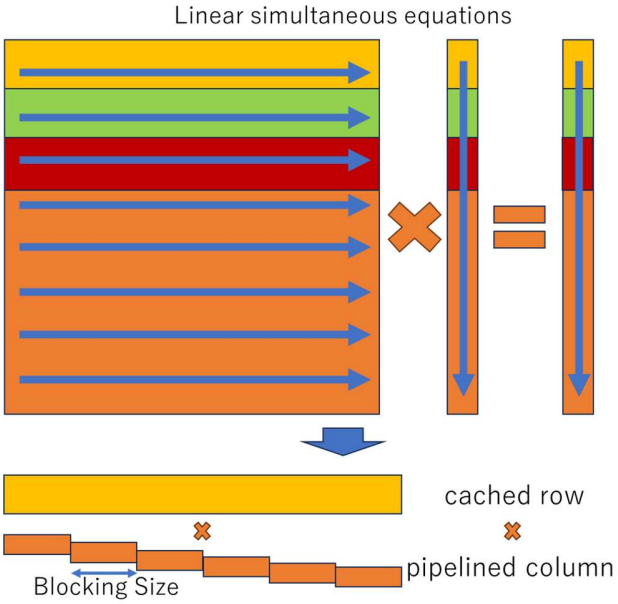

Figure 4: Caching equations with HLS

B. Classic HLS Implemantation Result

Gaussian elimination was implemented for each parameter set. The FPGA used was the Alveo u250 [15], an accelerator card for data centers from AMD. The environment used for HLS was v++ v2023.1, following the developer workflow for this accelerator card.

The resource usage of the implementation is shown in Table IV. Table V shows a comparison with the HLS implementation proposed in the previous study [11]. The implementation was found to result in almost the same amount of percentage resource usage compared to [11]. However, the AMD Alveo u250 used in this experiment is an FPGA for data centers, and the resource amount itself is higher than the Zynq UltraScale series, which is marketed for embedded applications. Although the previous study only mentions a percentage, the implementation in this paper probably has increased resource usage. Instead, it is expected that the computation is completed earlier in actual runtime because of the extra resource usage. The execution time was found to be approximately 0.1 second, even for the largest parameter set.

In reality, this is expected to be worse in practice because of the communication overhead, but this overhead can be absorbed by parallelizing the module itself and communicating alternately. In the previous study [11], this overhead is hidden by combining the implemented module and CPU latency.

TABLE IV. RESOURCE UTILIZATION OF HLS IMPLEMENTATION: 300 [MHz].

	BRAM	DSP	FF	LUT
m3488	222	7	35330	87660
m4608	636	3	82405	186029
m6688	1028	0	95979	181552
m6960	700	37	199472	332825
m8192	1084	0	139421	310003

TABLE V. RESOURCE UTILIZATION COMPARISON WITH STUDY [11].

		Latency (ns)	BRAM (%)	FF (%)	LUT (%)
m3488	this work	2.45E+07	4	1	5
	[11]	-	20.4	3.7	6.5
m4608	this work	5.51E+07	11	2	10
	[11]	-	41.4	5.4	8.3
m6688	this work	1.84E+08	19	2	10
	[11]	-	63.3	8.2	14.4
m6960	this work	1.07E+08	13	5	19
	[11]	-	55.8	5.5	8.8
m8192	this work	9.52E+07	20	4	17
	[11]	-	64.8	7.4	9.8

V. CONCLUSIONS

We evaluated the Classic McEliece cipher using the reference code submitted to NIST, and examined hardware that performs Gaussian elimination at high speed to improve its efficiency. As a result of implementation, we were able to perform Gaussian elimination for a 1644×1024 matrix in approximately 0.1 second with a maximum parameter n = 8192. This is equivalent to GCC option -O3 on the CPUs compared in this study. However, the CPU extension instructions were found to be even faster than this.

The speedup of the Gaussian elimination method can be utilized in future implementations, and in the case of SW/HW cooperative implementation, it can be flexibly deployed, such as using unorganized codes in some parts. In addition, if HLS is used for the entire public key generation part of HW, speeding up the Gaussian elimination method, which is a heavy process, will directly lead to speeding up the process.

REFERENCES

[1] NIST, “Post-Quantum Cryptography Standardization,” [Online]. Available: https://csrc.nist.gov/pqc-standardization

[2] NIST, “NIST Releases First 3 Finalized Post-Quantum Encryption Standards,” [Online]. Available: https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards

[3] NIST, “Round 4 Submissions,” [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions

[4] “Classic McEliece,” [Online]. Available: “https://classic.mceliece.org/”

[5] “CRYSTALS Cryptographic Suite for Algebraic Lattices,” [Online]. Available: https://pq-crystals.org/kyber/

[6] C. Paar, Bochum and T. Lange,” Eindhoven, eBACS: ECRYPT Benchmarking of Cryptographic Systems,” ECRYPT 2024. [Online]. Available: https://bench.cr.yp.to/call-kem.html

[7] Y. Zhu et al., "Mckeycutter: A High-throughput Key Generator of Classic McEliece on Hardware," 2023 60th ACM/IEEE Design Automation Conference (DAC), pp. 1-6, 2023.

[8] Classic McEliece: conservative code-based cryptography: guide for implementors, 23 October 2022. [Online]. Available: https://classic.mceliece.org/mceliece-impl-20221023.pdf

[9] R.J.McEliece, “A public-key cryptosystem and algebraic coding theory,” Coding Thv, vol. 4244, pp. 114-116, 1978.

[10] Po-Jen Chen, Tung Chou, Sanjay Deshpande, Norman Lahr, Ruben Niederhagen, Jakub Szefer, and Wen Wang, “Complete and Improved FPGA Implementation of Classic McEliece,”, IACR Transactions on Cryptographic Hardware and Embedded Systems,pp. 71-113, 2022.

[11] V. Kostalabros, J. Ribes-González, O. Farràs, M. Moretó and C. Hernandez, "HLS-Based HW/SW Co-Design of the Post-Quantum Classic McEliece Cryptosystem," 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), pp. 52-59, 2021.

[12] Johannes Roth, Evangelos Karatsiolis, and Juliane Krämer, “Classic McEliece Implementation with Low Memory Footprint,” CARDIS 2020: Smart Card Research and Advanced Applications, pp. 34-49, 2020.

[13] Yihong Zhu, Wenping Zhu, Chen Chen, Min Zhu, Zhengdong Li, Shaojun Wei, Leibo Liu, "Compact GF(2) systemizer and optimized constant-time hardware sorters for Key Generation in Classic McEliece," https://eprint.iacr.org/2022/1277, 2022.

[14] Brendan Gregg, “Flame Graphs,” Brendan’s site, “https://www.brendangregg.com/flamegraphs.html”

[15] ”Alveo™ U250 Data Center Accelerator Card,” 2024. [Online]. Available: https://www.amd.com/ja/products/accelerators/alveo/u250/a-u250-a64g-pq-g.html