ECE 6775 Final Project: FPGA Acceleration of Post-Quantum Cryptography

Aidan McNay (acm289) Barry Lyu (fl327) Edmund Lam (el595) Nita Kattimani (nsk62)

Introduction

With the increasing reliance on digital systems in the modern age, maintaining the security and privacy of important data is more critical than ever. As such, cryptographic algorithms for securly storing and communicating data have widespread usage. To maintain security, these algorithms are centered around computational problems that are infeasible for classical processors to solve; these are known as *NP-hard* problems, which have no known polynomial time solution. Users with a secret key are able to decrypt the encoded messages, but users without would have to solve this NP-hard problem to access the encrypted data, which are designed to take an astronomical number of years to solve with brute-force.

The advent of quantum computers have called the stengths of many of these algorithms into question. For example, both RSA (a popular asymmetric-key algorithm with widespread use) and Diffie-Hellman (an algorithm for securly establishing a common shared key, for use in symmetric-key encryption) rely on the difficulty of factoring large numbers for their security. Algorithms for quantum computers to solve such problems in polynomial time have existed for a while [1], but have never had a computer advanced enough to run them. However, modern advances in quantum computing have demonstrated that computers may be available soon that can crack these algorithms. Even just earlier this week, Google unveiled a new quantum computer, "Willow", that can achieve speedups over the fastest classical processors on select problems by a factor of 10^{30} [2].

While such computers aren't currently able to break modern cryptographic algorithms, many experts suspect it's only a matter of time before current cryptographic algorithms become insecure [3]. To this end, NIST (the National Institute of Standards and Technology) has standardized the use of RSA-2048 only until 2030, and noted that updated strengths are heavily affected by any progress on quantum computing [4]. Additionally, to prepare for the advance of quantum computing, NIST has standardized additional, *quantum-resistant* algorithms [5]. These algorithms are centered around different computational problems for which there is no known algorithm to efficiently solve for both classical and quantum computers. This *post-quantum cryptography* (PQC) will have increasing significance as advances in quantum computers are made. Additionally, since malicious adversaries could already be recording data to later decrypt once sufficient quantum computers are available, PQC algorithms are already being recommended and used for extremely sensitive data, such as government operations [6].

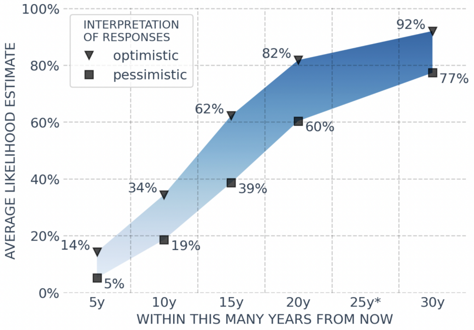


Figure 1: A timeline of when experts believe that RSA-2048 will be able to be broken by a quantum computer in 24 hours [3]

For public-key encryption (where the encrypting and decrypting keys are different), as well as key establishment (for securely establishing a shared secret key over an insecure network), NIST recommends CRYSTALS-Kyber, or simply **Kyber**. Since such an algorithm would be widely used in communication, speeding up its operation would have large impacts for a variety of applications. For our project, we explored implementing Kyber using custom hardware on an FPGA.

Problem Description - Kyber

Kyber is based on the ideas of Learning with Errors (LWE) encryption. LWE encryption is a form of encryption that represents secret information as a set of equations with some errors. In practice, this is often done by adding noise to the encoded secret information in order to conceal it. Kyber uses the ring of polynomials $\mathbb{Z}_q^n$, which represents n-degree polynomials with coefficients in $mathbbZ_q$, the set of integers modulo q. Here, q is a prime number (by default 3329) and n is a variable value that is often a power of 2 (by default 256) [7]. Additionally, kyber works on vectors of R of size $k = 3$ by default.

Algorithm Overview

Background

Before delving further into the algorithm we define some simple notation. We define the space $R = \mathbb{Z}_q^n$ to represent our space of polynomials. We thus can say that R^k represents a vector of k polynomials, and $R^{k \times k}$ represents a matrix of k^2 polynomials.Next, we define β_η for some positive integer η as follows:

$$\text{Sample } \{(a_i, b_i)\}_{i=1}^{\eta} \sim (\{0,1\}^2)^\eta \text{ and output } \sum_{i=1}^{\eta} a_i - b_i$$

If v is some polynomial, we can write $v \sim \beta_\eta$ to say that v is a polynomial with coefficients generated by β_η. We can also thus define a vector of polynomials $v \sim \beta_\eta^k \in R^k$ as a vector of k polynomials with coefficients generated by β_η.

Key Generation

Kyber begins with a public deterministic pseudorandom matrix $A \in R^{k \times k}$ of uniform polynomials. In layman terms, A is a k by k matrix of polynomials with uniformly random coefficients (modulo q). Next is a pseudorandom vector of polynomials $s \sim \beta_\eta^k \in R^k$ that represents the secret key of the algorithm. The public key t is generated by computing $t = A \cdot s + e$ [7], where $e \sim \beta_\eta^k \in R^k$ is a random error term that introduces noise into the key. The fact that it is computationally hard to recover s from t is the crucial result of the Learning with Errors problem [8].

Encryption

Given a message m encoded as a polynomial $m \in R^k$, Kyber encrypts m by a random polynomial $r \in R^k$ and two random error terms $e_1 \sim \beta_\eta^k \in R^k$ and $e_2 \sim \beta_\eta \in R$. The encryption process is as follows:

$$\begin{aligned} u &= A^T \cdot r + e_1 \\ v &= t^T \cdot r + e_2 + m \end{aligned}$$

The encryption process is designed such that multiple encryptions of the same message will result in different ciphertexts (due to different random r), making it difficult to analyze correspondence between messages. The output of the encryption function is (u, v).

Decryption

Given the ciphertext (u, v), the decryption process is as follows:

$$\begin{aligned} v - s^T u &= (t^T r + e_2 + m) - s^T (A^T r + e_1) \\ &= ((As + e)^T r + e_2 + m) - s^T (A^T r + e_1) \\ &= (\cancel{s^T A^T r} + e^T r + e_2 + m) - \cancel{s^T A^T r} - s^T e_1 \\ &= e^T r + e_2 + m - s^T e_1 - \end{aligned}$$

And as the error terms are specifically chosen to be small, the original message m can be recovered from the decryption process.

Components of the Kyber Algorithm

We have identified the following key components of the kyber algorithm through profiling. These components are essential to the algorithm and are the focus of our optimization efforts as they dominate the runtime of the algorithm.

- Random/Noise Sampling (25%)
- Number Theoretic Transform (NTT) and Inverse NTT (30%)

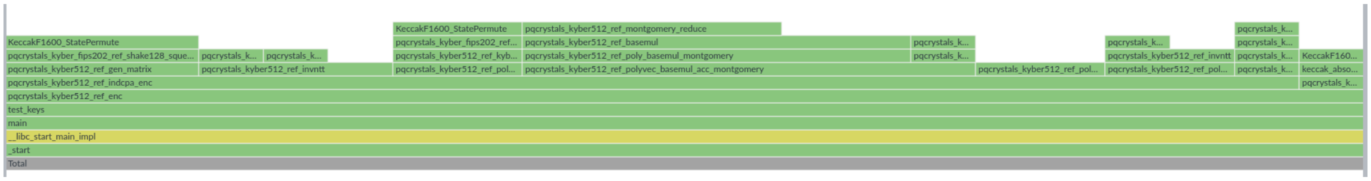


Figure 2: Flame graph of the reference software Kyber implementation

- Montgomery Reduction (35%)

Random/Noise Sampling. Random seeding, generated from expanding a uniformly random matrix, is used during the key generation and encapsulation routine. Randomness is essential to not just the Kyber algorithm, but to many modern cryptographic algorithms as it and avoid determinism in generated keys and ciphertexts, and protects the algorithm against direct/side-channel attacks that exploit predicatability in the system. Noise sampling, on the other hand, is critical to the hardness of solving the learning-with-errors problem, which is the basis of the Kyber algorithm. NTT uses centered binomial noises problem. It is important to note that a deterministic noise generator would compromise the security of the algorithm as it reduces the LWE problem to the weaker learning-with-rounding(LWR) problem.

Number Theoretic Transform is a generalization of the Discrete Fourier Transform (DFT) that has become increasingly important in Post Quantum Cryptography algorithm. It is fast (with a computational complexity of $O(n \log n)$), memory conserving, and can be implemented in a small code space. Almost all lattice-based crytopgraphy algorithms have been designed with parameters that support this fast multiplication. It plays an important role in Kyber as it is used to convert the polynomial multiplication in the polynomial ring to a multiplication in the number theoretic ring.

Montgomery Reduction is a technique for performing fast modular multiplications. By converting two numbers a and b into Montgomery form, the algorithm can compute $ab \bmod N$ efficiently, avoiding expensive division operations. It is used in many important cryptographic algorithms, including RSA aand Diffie-Hellman key exchange. It is slower than a conventional Barret reduction for single products but is significantly faster for many modular multiplications in a row. In Kyber, both Montgomery and Barret reduction are used to reduce the runtime of the algorithm.

Optimization of NTT

The Number Theoretic Transform (NTT) was the first component of the kyber algorithm that we successfully made synthesizable. Given that kyber has multiple functions, we decided that we would start by implementing and exploring optimizing the NTT and inverse NTT (INTT) components while we were figuring out the scope of our project. The NTT is similar to the Fast Fourier Transform (FFT), a transform that has been documented multiple times to be implemented on FPGAs. Both are used to compute discrete convolutions efficiently by transforming data into a domain where convolution becomes pointwise multiplication. The main difference is that NTT operates over finite fields instead of complex numbers. This similarity makes the NTT highly feasible for FPGA implementation, as it shares the FFT's iterative and predictable computation patterns involving additions, multiplications, and data shuffling, which are well-suited for the FPGA's parallel processing capabilities and configurable logic. Additionally, as mentioned above, NTT takes up a significant 30 % of the run time of the kyber algorithm, so optimizing it could substantially improve the overall performance and efficiency of the kyber algorithm. Our initial NTT implementation (including both NTT and INTT) had a latency of 6916 cycles and used 1 % of BRAM_18K block, 26% of DSP48E blocks, 2% of flip flops, and 5% of look up tables. Due to how much time the NTT component takes and that the design was not incredibly resource intensive, there was plenty of opportunity to optimize. When optimizing, the goal was to enhance performance and reduce latency as much as possible while making sure that our resource utilization made it possible to run our design implementations on an FPGA board. For the NTT function specifically in our implementation, all loops were fully unrolled and pipelined to maximize parallelism and minimize iteration latency. On the other hand, in the INTT function, only pipelining was applied to loops. When we tried to add loop unrolling to the INTT function, it only increased the area and had no impact on latency. Furthermore, including loop unrolling in the INTT function served so purpose. In addition to optimizing the NTT and INTT designs for our NTT implementation, we also unrolled the loops responsible for reading inputs and writing outputs to expedite data handling and improve throughput. For our most optimized design, we array partitioned the input to NTT and INTT, which got our latency down to 1597 cycles, over 4x faster than our baseline NTT optimization. However, this increased our area of DSPE48 blocks all the way up to 92 % and look up tables to 74 % on top of using 6% of the BRAM_18K blocks and 20% of flip flops. 92 % is very high area utilization, and there were some concerns over whether it would possibly run on the FPGA or not, but it ended up working out. These optimizations significantly increased our resources from our 2nd most optimized design, which had a latency of 2070 cycles. The main difference was array partitioning the input of the NTT and INTT functions. Removing this achieved our second most optimized design which was used 1% of the BRAM_18K Blocks, 5% of the DSPE48 Blocks, 10 % of the flip flops, and 65% of the look up tables. It achieved a 3.34x speed up, which is not as great as 4.33x speed up, but managed to achieve that by using significantly less resources. Overall, using loop unrolling, pipelining, and array partitioning, we achieved a significant latency reduction (up to 4.33x speed-up). The juxtaposition between the most optimized and second most optimized designs highlighted the trade-off between latency reduction and resource utilization. While the most optimized design prioritized performance with a 4.33x speed-up at the cost of utilizing 92% of DSP48E blocks, the second most optimized design sacrificed some speed (3.34x speed-up) to achieve a much more resource-efficient implementation, balancing latency improvement with hardware constraints. Although the most optimized design successfully ran on the FPGA board despite its high resource utilization, it was good to have the second, more resource-efficient design as a fallback in case the first design exceeded the hardware's limits.

Implementations

Considering the complexity of the algorithm and FPGA resource constrainsts. We have two different FPGA implementations: The first implementation just accelerates the NTT and Inverse NTT operations, while the second implementation accelerates the entire Kyber algorithm. In this section we will talk about the optimizations we did for each implementation, as well as the trade-offs we made.

0.1 NTT and Inverse NTT

For the first implementation, we focused on accelerating the NTT and Inverse NTT operations. We optimized the NTT and Inverse NTT kernels by aggresively unrolling loops and pipelining the design. We changed our kyber code such that whenever the NTT function is called, the host would stream data to the FPGA, and the FPGA would perform the NTT operation and stream the result back to the host.

This implementation is relatively simple, as we only need to optimize the NTT and Inverse NTT kernels. But since NTT only consists of 30% of the runtime of the Kyber algorithm, the upper bound of the speedup we can achieve is limited at 42%. The added data movement overhead between the host and the FPGA further limits the speedup we can achieve.

0.2 Full Kyber

For our second implementation, we accelerate the entire Kyber algorithm on FPGA. This implementation is more complex as it involves bringing all the reference code to FPGA, so we had to make the entire algorithm compatible with Vivado HLS, including parameterizing functions and fixing variable length arrays. The benefit of this implementation is that we significantly reduce the data movement overhead between the host and the FPGA, which is a significant bottleneck in the first implementation.

The downside, however, is that we are limited by the random generators as we cannot simply replace it with a hardware pseudo-random number generator such as an LFSR. As aforementioned, non-deterministic random noise is essential to the security of the Kyber algorithm.

Another bottlenect of the full kyber implementation is resource utilization. Because of the large number of kernels and relatively limited resources on the FPGA, we cannot optimize the design as much as we would like to.

FPGA Adaptation

Code Changes

A number of code changes were necessary in order to make the FPGA version of the Kyber algorithm synthesizable. The majority of the changes fell into three categories:

1. **Integer and Numeric Types:** Calls to integral types, particularly sized integers like `uint16_t` and `int32_t`, were replaced with synthesizable types like `ap_uint` and `ap_int`. This was performed by using `typedef` to define a number of custom `ap_uint` and `ap_int` types with the appropriate bit widths, such as `bit16_t` for an unsigned 16-bit integer.

2. **Standard Library Functions:** The reference Kyber algorithm used a number of standard library functions, mainly for memory allocation and manipulation. The majority of these were calls to `memset` and `memcpy`, which were replaced with loops that performed the same operations.

3. **Structures and Unknown Array Lengths:** A number of kernels in the Kyber algorithm, particularly in the functions used to implement the shake algorithm, used arrays of structures representing polynomial vectors and matrices. As the original algorithm was implemented in pure C, these lengths were passed in as arguments, and the arrays were represented as pointers. In order to make these kernels synthesizable with constant latency bounds, these had to be rewritten to use template parameters for the array lengths, and using actual arrays as arguments rather than pointers. Additionally, custom types representing polynomial matriceswere replaced with multidimensional arrays to support partitioning and reshaping pragmas.

In order to support these changes, particularly the template parameters, the implementation was refactored to use only header files. This required refactoring the build system, but the main tradeoff was the much higher compilation time due to the entire implementation being compiled as a single translation unit.

Simulation

The simulation of the Kyber algorithm was performed using the test benches we implemented. The test benches perform a number of encode and decode operations on random messages and keys. In order to verify the correctness of the results, it checks that decoding the encoded message results in the original message. Simulation was performed both in CSIM as well as COSIM, with the latter being used to verify the correctness of the HLS implementation in RTL.

Host Implementation

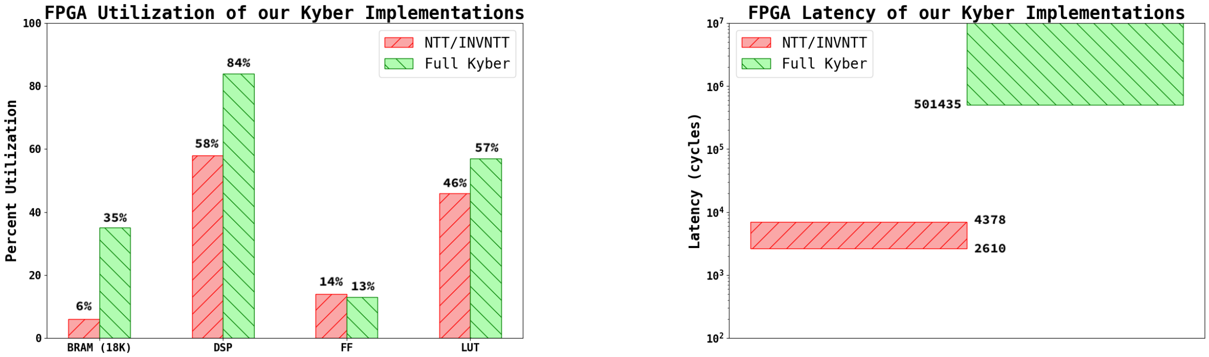
In order to execute the design on FPGA, we created a DUT using HLS streams in order to read the input and output values of the public/secret keys as well as the message in order to encode and decode messages. Following the implementation used in the DigitRec lab, we created a host file that generated a public/private keypair and sent this information to the DUT through the read/write pipes. In order to better measure accurate timing information, it generates 20 random messages and keys and measures the total time used to encode all 20 messages. Inputs and outputs are written and read in batches in order to minimize the communication overhead.

Evaluation

To quantitatively understand our design and the impact of our design choices, we pushed both the implementation of the entire design, as well as the more-optimized NTT kernel (including the NTT and INVNTT operations), through the FPGA flow and implemented on the class Zynq FPGA. This gave us both results after synthesis, as well as a bitstream to use on an FPGA to gain experimental results, allowing us to better gain takeaways about the efficacy of our design.

Synthesis

The first step to gain quantitative results is to use Vivado's HLS flow to push both of our implementations through the FPGA flow. This can tell us how many hardware resources the designs would need, as well as a prediction of how long each design will take. This is shown in Figures 3a and 3b, respectively.



(a) The hardware utilization of the different FPGA implementations

(b) The expected latency of the different FPGA implementations

Figure 3: The synthesis results for our FPGA implementations

For the utilization in Figure 3a, we can see that putting the entire design on FPGA hardware results in more hardware resources overall, as expected. One interesting note is the overall amount of DSP usage; for both designs, a significant number are used, much more than other labs have seen. This is due to the need for multipliers in Montgomery conversion. Having a value in Montgomery form can change performing a modulus over one value (as needed for modular ring operations) to another; by choosing to perform a modulus over a power of two, such moduli can be implemented in hardware quickly. However, converting in and out of this representation involves a multiplication step. This particular operation was one of the key reasons why the NTT kernel couldn't be as optimized. The NTT kernel had an outer loop that required a Montgomery conversion for each iteration, where the number of iterations varied each time it was used. Although we could unroll most uses of this loop, the largest version of this loop could not be unrolled, due to the limited number of DSP blocks available.

While the entire design used more resources in general, we also discovered that the optimized NTT kernel surprisingly used *more* flip-flops (**FF**) compared to the overall design. This seems somewhat surprising, considering that the NTT kernels are included (multiple times) in the entire algorithm, but makes sense once you consider the extra optimizations applied to the NTT algorithms in isolation. Specifically, with just the NTT algorithms, we can exploit loop pipelining and unrolling to achieve much higher performance by doing operations in parallel; however, this comes at the cost of extra flip-flops to store intermediate results in the different pipeline stages, as well as replicated across all loop iterations. This contrasts with the entire design in hardware; due to the lack of available resources, we are able to optimized the design much less, reducing the number of flip-flops needed in this case from the lack of aggressive pipelining/unrolling.

In addition to the hardware utilization of each design, we also compare about the predicted latency of each design; how many cycles each design should take. This is shown as a range for both designs in Figure 3b. For the NTT algorithms, the latency is uncertain, as it depends on which operation we're performing. Performing an NTT operation takes **2610 cycles**, whereas performing an INVNTT operation takes **4378 cycles** due to an extra multiplication reduction step needed at the end. However, for the entire design, the lowest latency was **501453 cycles** (on the order as expected), but the tools could not compute an upper latency bound. When we fixed the offending loops to have constant bounds, the maximum possible latency was astronomical (on the order of 1000 seconds). This is because the algorithm currently uses rejection sampling to achieve a random number in the modular ring for Kyber. In practice, many rejections are likely not needed (or algorithmic changes could be made to use a modulus instead of random sampling, albeit affecting the distribution), so our design can be best approximated by the lower bound.

Experimental

In addition to theoretical results, we additionally used Vivado's HLS tools to generate bitstreams for both implementations, such that they could be used on the FPGA. This allowed us to write host-side code to run on the ARM core and interact with our designs to measure their actual execution time, the results of which are shown in Figure 4. Measurements were done by running the desired algorithm 20 times by feeding 20 inputs, then reading the resulting 20 outputs, in order to amortize the communication latency.

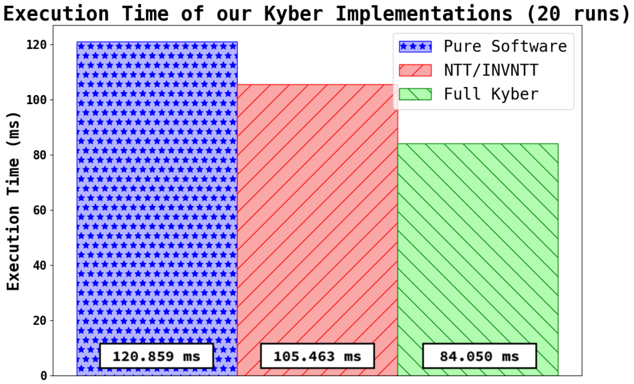


Figure 4: The measured execution time of the different Kyber implementations on the FPGA

From the figure, we can see that the NTT algorithms in hardware (taking **105.463 ms**) resulted in some speedup over a pure software implementation (**120.859 ms**), but were overall limited in their speedup from the fact that the majority of the Kyber algorithm was still in software. Even though we significantly optimized the NTT/INVNTT portions of the algorithm, they only composed an overall (**28.354 ms**) of the algorithm, with the other (**77.108 ms**) representing the remaining (unoptimized) software algorithm.

Having the entire algorithm in hardware (**84.050 ms**) resulted in more speedup over software than just the NTT kernels, but ultimately not as much as we hoped. This is due to two main factors:

- The ARM core is running at a faster frequency than the FPGA fabric. Therefore, software algorithms, even though running on general-purpose hardware, may be more competitive due to their higher frequency
- Kyber is composed of many different kernels and parts of the algorithm, as shown in the flame graph above. Each of these different operations requires different hardware to implement, resulting in low levels of hardware re-use. Because of this, many resources are used to simply implement the entire algorithm, leaving fewer available to parallelize the operations for higher performance, as was achieved in lab. We were ultimately unable to optimize large portions of the design for this reason.

Project Management

Milestones

The project was initially divided into four main milestones for each week, but as the project progressed it split into two main goals. First, we focused on optimizing a single kernel, the NTT kernel, which was the most computationally intensive part of the Kyber algorithm. Second, we also focused on optimizing the entire Kyber algorithm as a whole. As such, the main milestones of the project were

1. Synthesis of the NTT kernel on the FPGA.
2. Synthesis of the entire Kyber algorithm on the FPGA.
3. Optimization of the NTT kernel.
4. Optimization of the entire Kyber algorithm.

Timeline

Week of 11/17: Research and Planning

1. Research on the Kyber cryptographic algorithm. Everyone
2. Find a suitable C/C++ implementation of Kyber in software. Everyone
3. Implement test cases for the HLS implementations. Barry

4. Creating a build system for the HLS implementations. Aidan, Edmund

5. Begin converting software implementation for synthesizability.

    (a) Converting types to HLS compatible types. Aidan

    (b) Removing standard library calls. Barry

    (c) Converting arguments to template parameters. Edmund

6. Preliminary work synthesizing NTT kernel. Nita

Week of 11/24: Preliminary HLS Implementation and Results

1. Finish initial HLS implementation of NTT. Aidan, Nita

2. Initial optimizations of NTT Kernel Barry

3. Synthesize and verify correctneses of HLS implementation on CSIM, and COSIM RTL simulation. Aidan

4. Refactor project to create fully synthesizable Kyber algorithm. Edmund

Week of 12/01: Optimization and Final Results

1. Optimize HLS implementation of NTT to minimize latency and area. Nita

2. Create zeboard host software for NTT kernel. Aidan

3. Create zedboard host software for full Kyber algorithm. Edmund

4. Further optimizations on full Kyber algorithm. Barry

5. Finish DUT and verify full Kyber algorithm with CSIM and COSIM. Barry

Week of 12/08: Finalized Results, Report, and Presentation

1. Refactor zedboard host software to use batched testing for averaged results. Aidan, Edmund

2. Finalize results and create presentation. Everyone

3. Record final presentation and upload. Everyone

4. Create final report. Everyone

Challenges and Setbacks

The main challenge faced by the team was in the compatibility of the reference software implementation with HLS and FPGA. Although the implementation was already written in C, many of the kernels used arrays of unknown lengths in arguments, and a number of functions used pointer arithmetic and other non-synthesizable constructs. This required longer than expected to convert the software implementation into a synthesizable form, which was why we decided to focus on the NTT kernel first. We eventually resolved these issues by using many template arguments as many of the array lengths were known at compile time, and we also had to rewrite some of the functions to eliminate standard library calls and pointer arithmetic. This also required us to move all implementations into header files for linking purposes, which required refactoring the build system as well.

Conclusion

Acknowledgements

References

[1] P. Shor, “Algorithms for Quantum Computation: Discrete Logarithms and Factoring,” in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, Revised in 1996, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700. [Online]. Available: https://ieeexplore.ieee.org/document/365700.

[2] H. Neven, *Meet Willow, our state-of-the-art quantum chip*, Dec. 2024. [Online]. Available: https://blog.google/technology/research/google-willow-quantum-chip/ (visited on 12/10/2024).

[3] Global Risk Institute, *2024 Quantum Threat Timeline Report*, Accessed December 10th, 2024, Dec. 2024. [Online]. Available: https://globalriskinstitute.org/publication/2024-quantum-threat-timeline-report/.

[4] E. Barker, *Recommendation for Key Management*, en, May 2020. DOI: https://doi.org/10.6028/NIST.SP.800-57pt1r5. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-57pt1r5.

[5] G. Alagic *et al.*, *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*, en, Jul. 2022. DOI: https://doi.org/10.6028/NIST.IR.8413-upd1. [Online]. Available: https://doi.org/10.6028/NIST.IR.8413-upd1.

[6] *Quantum Computing Cybersecurity Preparedness Act*, 44 U.S.C. § 3502, 3552 and 3553 Chapter 35, Dec. 2022. [Online]. Available: https://www.govinfo.gov/app/details/PLAW-117publ260/summary.

[7] J. Bos, L. Ducas, E. Kiltz, *et al.*, “Crystals-kyber: A cca-secure module-lattice-based kem,” in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2018, pp. 353–367.

[8] Z. Brakerski, A. Langlois, C. Peikert, O. Regev, and D. Stehlé, “Classical hardness of learning with errors,” in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, 2013, pp. 575–584.