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 [**?**], 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} [**?**].

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 [**?**]. 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 [**?**]. Additionally, to prepare for the advance of quantum computing, NIST has standardized additional, *quantum-resistant* algorithms [**?**]. 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 [**?**].

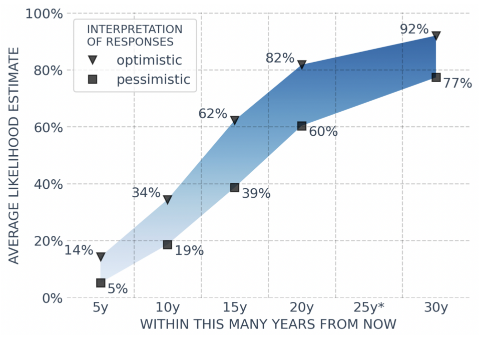


Figure 1: A timeline of when experts believe that RSA-2048 will be able to be broken by a quantum computer in 24 hours [**?**]

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

Components of the Kyber Algorithm

The Kyber algorithm consists of several core components, namely

- Random/Noise Sampling
- Number Theoretic Transform (NTT) and Inverse NTT
- Montgomery Reduction

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 $abmodN$ 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

Implementations

FPGA Adaptation

Code Changes

Simulation

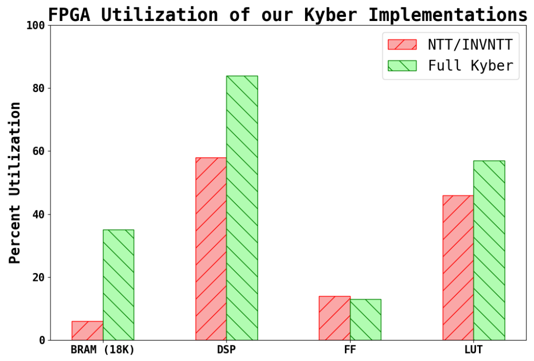
Host Implementation

Evaluation

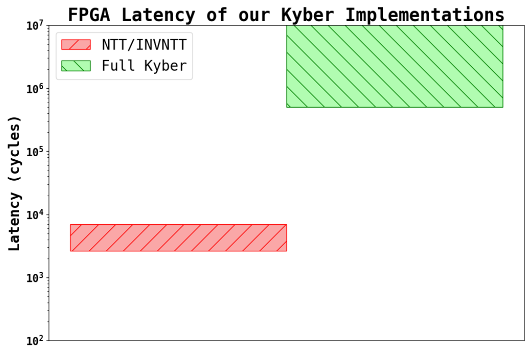
To quantitatively understand our design and the impact of our design choices, we pushed both the implementation of the entire design, as well as the more-optimized NTT kernel (including the NTT and INVNTT operations), through the FPGA flow and implemented on the class Zynq FPGA. This gave us both results 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 **??** and **??**, respectively.



(a) The hardware utilization of the different FPGA implementations



(b) The expected latency of the different FPGA implementations

Figure 2: The synthesis results for our FPGA implementations

For the utilization in Figure **??**, 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. 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.

Experimental

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