Post-Quantum Cryptography Acceleration

Kalan Brunell
Tufts University
kalan.brunell@tufts.edu

Patrick Johnson
Tufts University
pjohns11@tufts.edu

***Abstract*—This paper analyzes the performance of a Vitis C++ HLS implementation of an ML-KEM accelerator on an AMD Xilinx UltraScale+ FPGA, comparing it to a C software implementation run via the Sniper microarchitectural simulator. We focus on cycle counts, hardware resource usage, and power consumption to evaluate the advantages and disadvantages of using an application-specific accelerator for the ML-KEM process. Our results show that the accelerator significantly reduces cycle counts compared to CPU baselines. Specific latency and throughput optimizations work, however further work must be done to optimize the critical path of the algorithm. Additionally, we find that while the accelerator does consume more power than the CPU baselines, it is still within a reasonable range for data center applications.**

I. Introduction

Public-key cryptography is the primary mode of securing modern network communication. Up until now, two underlying algorithms have dominated the scene, namely, Rivest-Shamir-Adleman and Elliptic Curve Cryptography, more commonly known by their acronyms RSA and ECC. The security hardness of which is sourced from the difficulty of prime number factorization and discrete logarithm problems over a curve, respectively. In essence, these algorithms are nearly effortless for a classical computer to compute in one direction due to the computationally cheap steps, but are completely unfeasible to reverse due to the sheer number of possible computations necessary. However, this security guarantee only stands for classical computers, and in a post-quantum world, Shor's algorithm suggests that these schemes could be compromised with a large enough quantum computer. Additionally, the threat actually precedes the existence of these quantum systems, as encrypted data can be harvested now and decrypted later when the technology allows. This leaves historical, current, and future data at risk.

To combat this, NIST has made efforts to institute a standard of post-quantum cryptography protocols, one of them being ML-KEM. ML-KEM, formerly named CRYSTALS-Kyber, serves the function of a Key Encapsulation Mechanism. Notably, the NIST also selected standard algorithms for digital signatures, encryption, and others, but here, we focus only on ML-KEM.

Contrasting RSA or ECC, ML-KEM's hardness boils down to the Module-Learning with Errors (MLWE) problem, which is understood to be resistant to both classical and quantum attacks. The mechanism features three main processes, each with a distinct purpose. The Keygen process produces a public/private key pair, essential for public-key cryptography. Then, the Encapsulate process uses the public key to create the ciphertext and a shared secret. The third function is Decapsulate, which recovers the shared secret from the ciphertext using the private key. In a real system implementation, this shared secret could be used as a symmetric key for something like AES-GCM.

In this paper, we evaluate a High Level Synthesis (HLS) implementation of an ML-KEM accelerator by simulating it on an AMD Xilinx UltraScale+ FPGA to compare with a C software implementation run via the Sniper microarchitectural simulator. With these experiments, we aim to reveal and analyze the advantages and disadvantages of using an application-specific accelerator for the ML-KEM process.

II. Project Background

A. What is ML-KEM

As discussed, ML-KEM is a Key Encapsulation Mechanism whose threat resistance comes from its use of the Module Learning With Errors (MLWE) problem. In this implementation, MLWE is done on polynomial rings with a large prime number as the modulus, namely 3329 [3]. With this design, MLWE is believed to be secure against both quantum and classical cryptographic attacks, and is hoped to provide the modern and future security that classic algorithms like RSA and ECC cannot. The KeyGen process builds a public/private key pair, which is used by the Encapsulate process. Encap, processed by the sender, uses the receiver's public key to generate a ciphertext and the shared 32-byte secret. Decap is responsible for decrypting the shared secret from the ciphertext via its own (receiver's) private key, from which it can use the shared secret as a symmetric key for a cipher like AES.

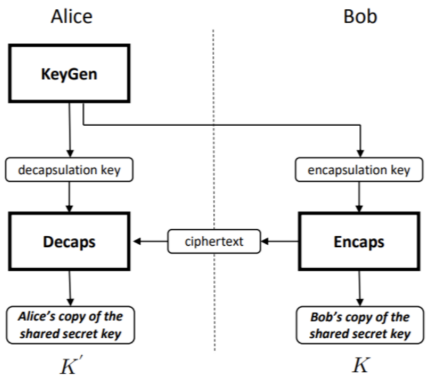

Fig. 1. Alice and Bob key exchange flow for ML-KEM.

$$b_A = \begin{bmatrix} 4 & 1 & 11 & 10 \\ 5 & 5 & 9 & 5 \\ 3 & 9 & 0 & 10 \\ 1 & 3 & 3 & 2 \\ 12 & 7 & 3 & 4 \\ 6 & 5 & 11 & 4 \\ 3 & 3 & 5 & 0 \end{bmatrix} \times \begin{bmatrix} 6 \\ 9 \\ 11 \\ 11 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \\ 1 \\ 1 \\ 1 \\ 0 \\ -1 \end{bmatrix} \pmod{13} = \begin{bmatrix} 4 \\ 7 \\ 2 \\ 11 \\ 5 \\ 12 \\ 8 \end{bmatrix}$$

Fig. 2. Example of the vector multiplication in ML-KEM.

The greater ML-KEM scheme features three security levels consistent with those of NIST: ML-KEM-512, ML-KEM-786, and ML-KEM-1024. The largest difference between these levels is the module rank k, which is responsible for setting the size of the public matrix and secret vector. A larger k value causes larger keys, larger ciphertexts, etc., and provides stronger security, but at the cost of more data to process.

Within Encapsulate and Decapsulate, the majority of the load comes from three computational operations. Number-Theoretic Transform (NTT) is responsible for converting polynomials between representations to reduce the computation load of polynomial multiplication. The Keccak permutation works as a subprocess of the SHAKE-XXX hashing functions to sample pseudorandom values to build the public matrix from the 32-byte seed. Further exploration into these processes reveals their readiness to be parallelized, which motivates much of the work here.

B. Why Hardware Acceleration?

While ML-KEM was designed to be comparable with currently used algorithms on a standard CPU, the underlying mathematical properties lend it to being deployed on an accelerator. The NTT layers expand out, often described as "butterflies, " with each layer independent from the others, making it an excellent candidate for pipelining/parallelization. Similarly, the Keccak process features sequential layers, combined at the end, suggesting it should be collected into one large combination process. Huge architectural steps like these were not all feasible given our timeline and group size, but the data we collected on our small experiments does a great job of supporting these larger concepts.

Another supporting factor of ML-KEM acceleration is its use case in the real world. On the scale of huge supercomputers and network centers, the number of encapsulation and decapsulation processes would be absolutely massive upon full adoption of this standard. Adding hardware accelerators to these systems to handle cryptographic processes like these only makes sense, and clearly tracks similar choices for classic algorithm computation.

III. Experimental Design

A. Accelerator Source and Target

The accelerator of focus here is a Vitis C++ HLS implementation of the ML-KEM process. Another group developed the original design targeting the AMD Alveo U280 FPGA, namely the BSC LOCA group as part of their PQC-Crystals-HLS-Accelerators project [1]. Thus, we did not author the majority of the HLS code, but did adjust it to our use case, which is discussed later in this section. Our focus was on adjusting, evaluating, and benchmarking the design in the context of general-purpose CPUs. Diving into the technical details of the implementation, the design has two outward-facing kernels `k_kem_enc` and `k_kem_dec`, together with a top-level wrapper `mlkem_accelerator` that selects between them at runtime via a configuration byte. Keygen is not exposed to the user; however, expensive operations, such as matrix expansion and centered-binomial sampling, are included in the Encapsulation functionality, so the hardware cost of Keygen is essentially included in the results. Each kernel uses `m_axi` master interfaces for the key, ciphertext, and shared-secret buffers, following the standard Vitis kernel convention. We were able to synthesize and implement the design using the Vivado part xcu280-fsvh2892-2L-e. With this, we chose an initial target clock of 300MHz to match the initial design.

B. Synthesis and Co-Simulation

The HLS design was synthesized with AMD Vitis 2023.1. This process generated and returned the full accelerator implementation in Verilog .v files. From here, we performed a full C/RTL co-simulation to measure cycle latency and other metrics for both Encapsulate and Decapsulate. Future work could include generating the full implementation in Vivado, but for our work, the cosimulation provided the correct balance of accuracy and the time it takes to try different HLS configurations. Furthermore, the full implementation requires wrapping the AXI interfaces as well as driving a clock, reset, etc., which was outside of the scope of our small group and timeline. We did attempt this briefly, but without wrapping the AXI ports into a more streamlined system, we found our implementation would need 2000+ IO ports, which is not realistic, nor something available on most FPGAs. That said, because we are only comparing metrics like cycles and hardware budgets, the cosimulation is more than adequate.

C. CPU Baseline

For our software baseline simulation, we chose the PQClean implementation of the ML-KEM-512 algorithm [2], which was chosen for its clean, well-validated, and widely used nature. To obtain comparable benchmarks, we wrote a testbench that performs each of the three steps in series: KeyGen, Encap, and Decap. Furthermore, these three steps were captured in a region-of-interest so that Sniper's benchmarks would be restricted to the cryptographic processes and not anything else. The simulation was run with Sniper version 7.3, which can produce in-depth cycle counts, IPC, and more. Two CPUs were selected within Sniper to provide further context and comparison for our data. Namely, `gainestown.cfg` models a 2.66 GHz Intel Nehalem-class core, and `silvermont.cfg` models an Intel Atom-class in-order core. Notably, these two options offer performance metrics of both an out-of-order and an in-order processor to provide a meaningful comparison.

Lastly, ten trials were run for each simulation with the goal of identifying and mitigating run-to-run variance.

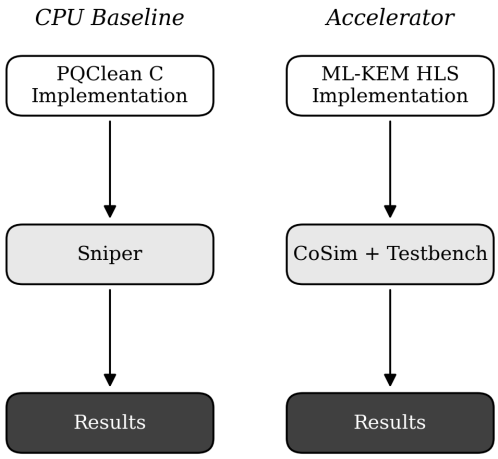

Fig. 3. Two-track simulation. The CPU baseline runs the PQClean reference C implementation via Sniper, while the accelerator track runs the Vitis HLS source via co-simulation.

D. Metrics and Comparison

To meaningfully compare our accelerator implementation and CPU simulation results, we had to narrow down what metrics make sense for the situation. In our case, neither the accelerator nor the CPU is running on actual hardware, which strengthens the comparison (compared to non-simulation vs simulation). However, the two systems run at drastically different clock speeds, so careful measurements were required. The most straightforward, and possibly most telling metric we focused on was *cycle counts*. Independent of clock speeds and other variables, the cycle count can provide valuable context to the raw computational workload. These counts are reported both as a comprehensive view of all three processes (KeyGen + Encap + Decap) and separately for further analysis. On the topic of hardware cost, we also measured FPGA resource usage for our HLS implementation. There is no clear companion on the CPU side to compare these numbers to, but these usage reports do communicate meaningful context about what physical FPGAs are capable of running this accelerator. Furthermore, these usage measurements help illustrate how much headroom there is for further parallelizing the HLS given a specific FPGA. Computation cost aside, we also report power consumption metrics taken from both sides of the comparison. Together, cycle count, resource usage, and power consumption help to fully illustrate the actual improvements and losses of one system versus the other.

It is important to note that for combined metrics such as total cycle count (KeyGen + Encap + Decap), the two implementations handle KeyGen differently. The PQClean C code has an explicit process that matches the official ML-KEM for key generation. However, as mentioned, the HLS implementation has no explicit KeyGen kernel and instead performs computationally similar tasks within the Encapsulate. While this does mean we cannot compare KeyGen vs KeyGen explicitly, we argue that overall cycle, hardware, and power costs are still extremely relevant to implementation comparison.

As mentioned, both sides of the experiment are run in simulations rather than actual hardware. By the nature of microarchitecture simulations and cosimulations, modeling errors are likely to be present in some magnitude. That said, running the trials in this isolated manner actually helps to increase the fidelity of the compared measurements, as noise, such as other processes, data transferring, management, and thermal consequences, is not present. Furthermore, neither side is given any advantage over the other as all trials are simulated. In essence, this means the actual values measured here may not be the same as they would be if run on real hardware; however, for the sake of accelerator vs CPU comparison, this method arguably provides cleaner data.

E. HLS Modifications

Determining what HLS modifications to make was an iterative process, once we got the reference implementation to properly synthesize, implement, and co-sim. We briefly tried to determine what would be the best targets for hardware optimizations, and so unrolling came to mind. However, the reference implementation has already unrolled the NTT/Inverse NTT layers as much as possible, as well as their inner butterfly loops. This is when we noticed that the entire algorithm was built around the ability to modify the K variable, which is primarily controlled by what security level of ML-KEM you are using. In our case, we wanted to test ML-KEM-512 specifically, so we decided to parallelize the 2 polynomials that must be run for K=2 since they are naturally sequential. We do this using module duplication. Within the NTT_Layer function, then call it separately for each polynomial. While this removes the modularity, this process can be expanded to K=3 and K =4 for the other security levels of ML-KEM just as easily. In addition, we could expand it to the InvNTT_layer function if we found success. Afterwards, we determined that the NTT_Layer function was not on the critical path and therefore decided to target a different function using a different method. Basemul_montgomery and Basemul_add are two functions used during the end of the encapsulation/decapsulation process. Basemul_montgomery helps the program multiply the resulting coefficient pairs in the NTT domain using Montgomery reduction. Basemul_add sums the matrices in the NTT domain. Because these two must be done across 128/256 coefficients, widening their buffer width to process 4 coefficients instead of 1, should theoretically give better throughput.

IV. Results

A. NTT Parallelization

Depending on the FPGA chosen for the application, the amount of hardware available varies significantly. In our case, with the original implementation, we had a lot of extra hardware headroom available, and thus, we decided to generate a second configuration of the accelerator that aimed to parallelize the NTT process. By separating the polynomial multiplication into two parallel tracks instead of the default single track, we were able to decrease the number of cycles by about 49%, and consequently, doubled the number of layers. As shown by Fig. 4, we successfully sped up this single process. Overall effects will be examined later.

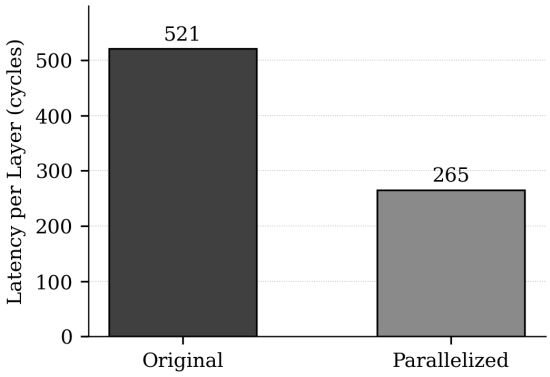

Fig. 4. Per-layer NTT latency for the original and parallelized configurations.

B. Cycle Count Comparison

Table I provides insight into the total cycle counts for the full ML-KEM-512 process (KeyGen + Encap + Decap) across each of our four configurations. These include the original HLS accelerator implementation, the parallelized NTT version, and the two standard CPU baselines. Notably, our parallelized version features almost identical total cycle counts, despite having measurably reduced the cycle counts for each layer.

Also of note is the huge advantage in cycle counts that both accelerator versions display over the standard CPUs. Furthermore, Table II dives into the per-transaction cycle count breakdown. Shown there are our various trials, all returning highly consistent results, with the smaller value representing the Encap processes and the larger being the Decap processes.

TABLE I
Cycle Count Summary: FPGA Accelerator vs. CPU Baseline (KeyGen + Encap + Decap)

Configuration	Total Cycles	Speedup vs. FPGA Original
FPGA Original	11,948	1×
FPGA Parallel NTT	11,967	≈1×
CPU Gainestown	1,195,000	100× (slower)
CPU Silvermont	1,724,000	144× (slower)

Fig. 5. ML-KEM-512 full pipeline cycle count comparison: CPU vs. HLS accelerator.

TABLE II
Encapsulation/Decapsulation Transaction Latencies (FPGA Co-Simulation)

Transaction	Original (cycles)	Parallel NTT (cycles)
0	5076	5076
1	6905	6919
2	5043	5043
3	6905	6919
4	5043	5043
5	6905	6919
6	5043	5043
7	6905	6919
8	5043	5043
9	6905	6919
10	5043	5043
11	6905	6919
12	5043	5043
13	6905	6919
14	5043	5043
15	6905	6919
16	5043	5043
17	6905	6919
18	5043	5043
19	6905	6919

C. Hardware Resource Utilization

Table III presents the hardware resource usage of both the accelerator configurations on the Xilinx Alveo U280 (xcu280). This data is as reported by the Vitis HLS post-implementation report. Of note, the parallelized NTT version does occupy slightly more resources, despite its lack of performance improvements; however, still nowhere close to the targeted device's capabilities. This still leaves significant headroom for future experiments. No data is presented here for the CPU baselines due to the incomparable nature of those simulations.

TABLE III
Hardware Resource Usage on Xilinx Alveo U280 (xcu280)

Implementation	BRAM	DSP	FF	LUT
Total Available (U280)	4,032	9,024	2,607,360	1,303,680
Original ML-KEM-512	47 (1%)	303 (3%)	116,926 (4%)	172,674 (13%)
Parallelized NTT	57 (1%)	372 (4%)	129,270 (4%)	190,039 (14%)
Percent Increase	+21%	+23%	+11%	+10%
Widened Basemul	48 (1%)	303 (3%)	119,203 (4%)	175,230 (13%)
Percent Increase	+2%	0%	+2%	+1%

D. Power Consumption

Table IV reports the power usage breakdown for our experiment configurations. These were calculated under the parameters displayed in Table V. Of note, there is a slight increase in power requirements for our parallel NTT configuration, which is consistent with the slight increase in hardware. Within this increase, the majority comes from an increase in dynamic power (LUTs, DSP), which makes sense given the increase in logical computations.

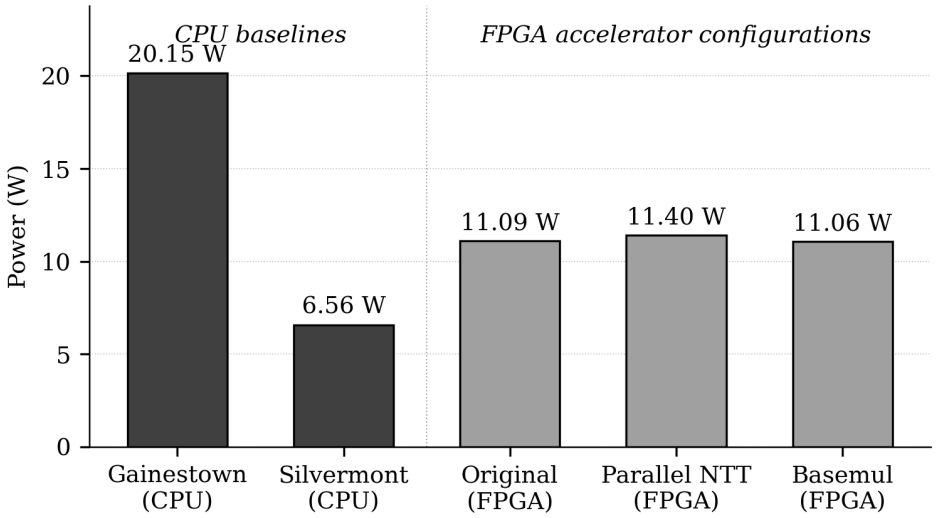

Fig. 6. Average power consumption across all platforms.

TABLE IV
POWER USAGE COMPARISON

Component	Original(W)	P. NTT(W)	%Change	Basemul(W)	%Change
Clocks	0.901	0.872	-3.22	0.919	+2.00
CLB Logic	2.640	2.846	+7.80	2.717	+2.92
LUT as Logic	2.374	2.507	+5.60	2.450	+3.20
LUT as Dist. RAM	0.151	0.210	+39.07	0.148	-1.99
Register	0.079	0.086	+8.86	0.081	+2.53
LUT as Shift Reg.	0.025	0.029	+16.00	0.027	+8.00
CARRY8	0.011	0.014	+27.27	0.011	0.00
F7/F8 Muxes	0.000	0.000	0.00	0.000	0.00
BUFG	0.000	<0.001	—	<0.001	—
Signals	3.647	3.641	-0.16	3.515	-3.61
Block RAM	0.137	0.174	+27.01	0.143	+4.38
DSPs	0.470	0.564	+20.00	0.470	0.00
Static Power	3.296	3.303	+0.21	3.296	0.00
Total	11.090	11.401	+2.80	11.060	-0.27

Once again, the % change columns for each version are in reference to the original implementation on the left.

TABLE V
THERMAL AND ENVIRONMENTAL PARAMETERS

Parameter	Value
Ambient Temp (C)	25.0
θ_{JA} (C/W)	0.4
Airflow (LFM)	250
Heat Sink	medium (Medium Profile)
θ_{SA} (C/W)	0.5
Board Selection	medium (10"x10")
# of Board Layers	12 to15 (12 to 15 Layers)
Board Temperature (C)	25.0

Table V's environmental thermal parameters are automatically pulled from Vivado's information on the U280, however specific things like the Heat Sink and Airflow were estimated based on the typical use case of this accelerator, which is in a data center environment.

E. Basemul Pipelining Experiment

The second attempt at optimizing the original accelerator design by targeting the `basemul_montgomery` and `basemul_add` modules. These processes work hand-in-hand with the NTT layers as they handle the coefficient-wise polynomial multiplication. Because of this, we expected them to be excellent candidates for parallelization. Table VI displays the small amount of data we were able to collect from our Vitis HLS trials. Notably, our changes (outlined in Experimental Design) did not have any noticeable effect on the cycle cost of these processes. That said, our code change certainly translated to an architecture change, shown by the adjustments in resource cost.

TABLE VI
BASEMUL MODULE LATENCY AND RESOURCE UTILIZATION (VITIS HLS SYNTHESIS ESTIMATES)

Module	Variant	Cycles	BRAM	DSP	FF	LUT
`basemul_montgomery`	Original	529	—	30	1,197	454
`basemul_montgomery`	Basemul	528	—	30	1,233	498
`basemul_add`	Original	515	1	—	69	227
`basemul_add`	Basemul	515	—	—	232	496

V. Analysis

Shown in Table I is how the CPU baselines performed against our FPGA implementation. Cycle count is a good comparison in our case because clock speed can be changed, however for our project the FPGA was run at 300 MHz and Gainestown was run at 2.66 GHz and Silvermont was run at 1.0 GHz. From here, we can extrapolate execution times of 0.0398 ms for the original FPGA implementation, 0.448 ms for the Gainestown CPU, and 1.724 ms for the Silvermont CPU. This means that our FPGA implementation is about 11 times faster than the Gainestown CPU and 43 times faster than Silvermont when looking at execution time instead of cycle counts. We can also look at Figure 6 for the power analysis to see that the FPGA implementations all used relatively equal total power, but used nearly half the power of Gainestown and about 70% more than Silvermont. What this essentially tells us is that FPGA acceleration of ML-KEM-512 is feasible for real-world applications.

While a throughput increase of the overall encap/decap algorithm was not achieved, the results are still promising. To start, we successfully halved (49% decrease) the latency of the NTT layers as expected. Because we optimized for ML-KEM-512 (K=2), we could use module duplication to make both polynomials run in parallel, therefore halving the latency.

However, looking at Table II, we can see that latency of the overall program is identical for encap and nearly identical for decap. This leads me to believe that there is a bottleneck somewhere in the Dataflow architecture, rather than this being a result of Amdahl's law (NTT making up a small portion of the overall program). One explanation is that because the NTT layers are already unrolled so much, a different parallel process is bottlenecking that stage and decreasing its latency will therefore not have an effect on the overall throughput. The synthesis report did not tell us what parts of the program were sequential and what were parallel, so we had to make an educated guess based on the code and the cycle counts when trying to figure out what would be the bottleneck. Notable steps on the critical path (long latencies without the ability to use hardware unrolling) include Readmem, Split, Basemul_Montgomery, and Basemul_Add. These have the highest latency/initiation interval. Thus, we tried to optimize Basemul by widening buffer width. Table VI shows how the cycle count did not improve even when trying to optimize for something on the critical path. Note that all of the instantiations of Basemul_montgomery and Basemul_add had similar results to the one shown in Table VI. What is not shown, although, is the throughput. Because we increased the bandwidth of these functions, it doesn't necessarily increase the time it takes to compute each process. The best measure would be to run a co-simulation and testbench, however the local computers ran out of RAM to run it. Therefore, settings need to be changed and the program needs to be optimized further before we can fully test it. Another possible explanation for lack of throughput increase could simply be data starvation. We focused on increasing the bandwidth of the basemul modules, but if the data is not being fed into those modules fast enough, then we won't see a throughput increase. Many of the ports are set up to process 1 coefficient at a time, so to see more improvement we might have to widen buffers throughout the entire program.

During the process of testing the HLS, we also discovered a bug in the original code where they did not specify stream depths for the AXI master port bundles, so we added them manually to get our data. However, because of how long the HLS synthesis takes to run, we didn't get to experiment with different stream depths, which could have an effect on the latency of the program.

VI. Future Work

- Additional CPU Baselines We chose to compare our accelerator implementation with two CPU configurations preloaded in Sniper. Strategically, the two examples spanned in-order and out-of-order processor styles, but neither of them is particularly modern in architecture. Simulating our PQClean version on a more modern configuration would likely reveal further valuable comparisons in a modern context.
- Run the Co-Simulation on the Basemul Widening after fixing the RAM usage issue. In addition, we could combine the two optimizations to see if basemul increases the throughput enough where NTT would be on the critical path and thus the latency improvement would have an effect.
- The HLS synthesis and implementation take 2 hours together. If we had more time, we could have analyzed more implementations such as ML-KEM-786 and ML-KEM-1024, which would have provided more insight into how the accelerator performs across different security levels.

VII. Conclusion

Our goal for this project was to see how an HLS FPGA implementation of ML-KEM-512 would perform against a CPU baseline. Then, see if we can optimize the program further using hardware's parallelizability. We successfully implemented the accelerator on the AMD Xilinx UltraScale+ FPGA, and ran a co-simulation to get cycle counts for the encapsulation and decapsulation processes. We also ran a Sniper simulation of the PQClean C implementation of ML-KEM-512 on two different CPU configurations to get cycle counts for the same processes. We found that both accelerator implementations had significantly lower cycle counts and faster execution times than the CPU baselines, however our optimization of parallelizing the NTT layers did not have an effect on the overall cycle count of the program. Our attempt to widen the buffer for Basemul to increase throughput directly also failed, most likely due to data starvation, although we were not able to confirm due to time constraints. This paper, while incomplete in some ways, shows that FPGA acceleration and optimization have promising results for throughput intense applications of ML-KEM. At the very minimum, we have shown that FPGA

acceleration is significantly faster due to the parallelizability of the algorithm's matrix computations. As ML-KEM and other post-quantum cryptography standards are adopted, it is likely that hardware acceleration will be a key part of their real-world deployment, and thus, further research into this area is crucial.

REFERENCES

[1] BSC LOCA, "PQC-Crystals-HLS-Accelerators," GitHub repository, 2024. [Online]. Available: https://github.com/bsc-loca/PQC-Crystals-HLS-Accelerators.

[2] PQClean Project, "Clean, portable, tested implementations of post-quantum cryptography," GitHub repository. [Online]. Available: https://github.com/PQClean/PQClean

[3] National Institute of Standards and Technology, "Module-Lattice-Based Key-Encapsulation Mechanism Standard," FIPS Publication 203 (Final), Aug. 2024. [Online]. Available: https://csrc.nist.gov/pubs/fips/203/final