Department of Electrical Engineering,
National Tsing Hua University
Special Topic on Implementation
Research Summary

Innovative FPGA Implementation of High-Level Synthesis in Post-Quantum Cryptography: Hardware Acceleration for Falcon Digital Signature

FPGA 實現高階合成於後量子密碼學：Falcon 數位簽章之硬體加速

整合快速傅立葉變換與數論轉換

Major Category: System

Group Number: A400

Advisor: Jiin Lai

Members: Kuan-Hsi Chen, You-Wei Liu, Sheng-Ta Chen,
Yan-Zhi Wang, Bo-Han Chen

Research Period: From (2024/02/05) to (2024/05/05).

Abstract

The development of quantum computing poses a significant threat to conventional cryptographic protocols. In response, this project advances post-quantum cryptography by developing a novel FPGA implementation of the Falcon algorithm, a key candidate renowned for its robustness against quantum attacks and efficient lattice-based signatures. Transitioning from C code to synthesizable Hardware Description Language (HDL) via High-Level Synthesis (HLS), we optimize critical computational elements such as Fast Fourier Transform (FFT) and Number Theoretic Transform (NTT) to enhance performance and security.

At the beginning of the report, we will talk about the falcon algorithm's advantages, along with its main function. We aim to improve the Falcon algorithm by transforming FFT/inverse FFT/NTT/inverse NTT into hardware.

Hardware optimization steps included transforming C code into HLS to make it synthesizable, restructuring FFT, inverse FFT, NTT, and inverse NTT for better data flow and execution time reduction. Additionally, we refined the logic design to minimize FPGA resource usage without sacrificing performance. Further optimization efforts involved streamlining the pipeline stages, enhancing parallel processing capabilities, and optimizing memory usage to reduce latency and increase throughput.

Furthermore, we employ middleware that manages communication and task scheduling between the software interface and the FPGA hardware. This approach allows scalable design, e.g. the increase of the hardware component without the modification on application code.

The project is integrated into an MPSoC with three user-accessible APIs: KeyGen, Sign, and Verify. We plan to deploy these capabilities in a USB dongle format, communicating with PC or notebook via the FIDO2 CTAP2 protocol. This approach aligns with the FIDO Alliance's specifications for hardware-based authentication, which enhances security by reducing reliance on passwords and simplifying authentication processes across various platforms.

The experiment results demonstrate that our hardware-accelerated components substantially outperform traditional software-based methods, setting a new benchmark for future developments in hardware-accelerated cryptography and ensuring scalability and adaptability in the evolving landscape of cybersecurity.

Table of Contents

1. Background

In response to the threat posed by quantum computers to existing cryptographic standards, post-quantum cryptography (PQC) has become a vital research area. The Falcon algorithm [1] stands out in this context for its resistance to quantum attacks. A major challenge with Falcon is its lengthy execution times when implemented in software, limiting its practical use.

2. Purpose

Our research aims to address the challenge of lengthy execution times by transforming the Falcon algorithm from software into synthesizable Hardware Description Language (HDL) for hardware acceleration. We employ High-Level Synthesis (HLS) to accelerate the hardware development process and restructure the HLS code to optimize performance. The heart of our contribution focuses on optimizing critical components such as Fast Fourier Transform (FFT), inverse FFT (iFFT), Number Theoretic Transform (NTT), and inverse NTT (iNTT), enhancing both computational efficiency and security.

A particularly innovative aspect of our work is the integration of PQC with hardware through a hardware/software co-design approach. This methodology enables the Falcon algorithm to be executed more efficiently on hardware platforms. We have finalized the hardware design, generated a bitstream, and developed middleware on the PS (Processing Subsystem) side of a SoC to manage and distribute requests for PL (Programmable Logic) side kernel functions, originating from the Falcon flow. This setup communicates with a KV260 FPGA board through Python's PYNQ API. This integration allows the full Falcon process to be executed on the board, significantly reducing execution times compared to its software-only counterpart.

This project advances the field of post-quantum cryptography by providing a more secure and efficient solution to the threats posed by quantum computers and sets a foundational framework for future research in hardware-accelerated cryptography.

3. Method

3.1 Hardware acceleration

3.1.1 Synthesizable Implementation

Consider the FFT as an example: initially, we allowed computation loop boundary to be variable, which requires synthesizing logic to handle these boundaries [3], potentially extending execution time. Therefore, we must address the issue of the unbounded loop structure. By using the indexing from the iterative radix-2 FFT algorithm with bit-reversal permutation, each stage loops 256 times. Therefore, we set the boundary at 256 for each stage. Following this, we outline the indexing for each stage:

- *i = n + ((n / index_const) * index_const)*
- *i_gm = n / index_const*

3.1.2 Datapath Restructure

To save the use of memory and area, we separate each stage and reuse the processing element (PE). We connect several functions (FFT here) in a staged fashion with arrays (fin) acting as buffers between the stages. Fig. 3-1 provides a graphical depiction of this process.

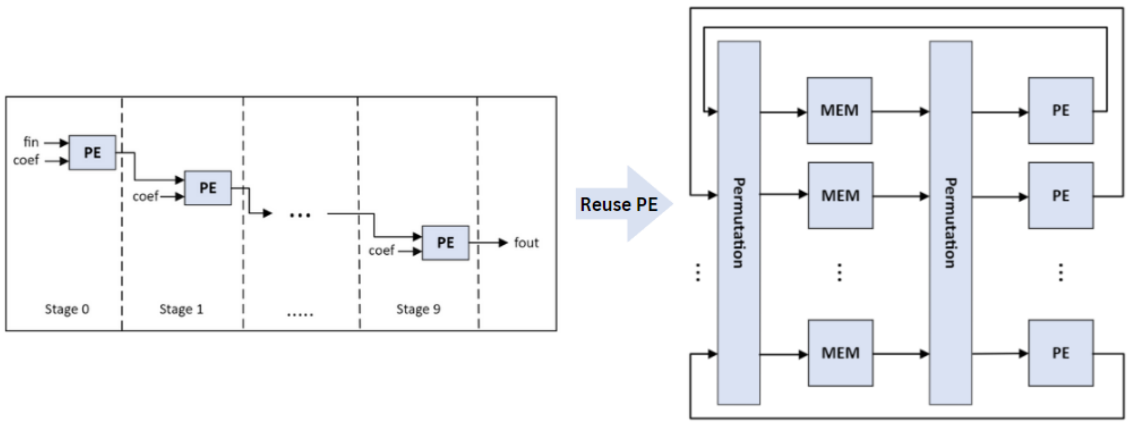

Fig.3-1 Process and structure (reuse PE) of FFT

From Fig. 3-1, we can see that FFT algorithm is calculated by loading data from the memories into the PEs and storing the result again in the memories. This process is iterated until the complete FFT is computed. The reuse of the PEs for different stages reduces the number of butterflies. In our project, we use one PE and one in-place memory buffer. With HLS, we can easily extend the number of PEs [4] to achieve more parallelism.

3.1.3 Complex Multiplication

The multiplication of two complex numbers $(X_r + jX_i)$ and $(Y_r + jY_i)$ is defined as:

$$Z_r = X_r \cdot Y_r - X_i \cdot Y_i$$

$$Z_i = X_r \cdot Y_i + X_i \cdot Y_r$$

where Z_r is the real part of the result and Z_i is its imaginary part.

Instead of using 4 real multipliers, a structure with 3 real multipliers can be obtained by rewriting as [2]:

$$tmp = Y_i \cdot (X_r - X_i)$$

$$Z_r = X_r \cdot (Y_r - Y_i) + tmp$$

$$Z_i = X_i \cdot (Y_r + Y_i) + tmp$$

This structure leads to area reduction although the usage of 5 adders instead of 2 adders is needed in the direct implementation, since multipliers require significantly more area than adders.

3.1.4 Combine 4 Algorithms in One Hardware

- FFT & NTT

First, the following formula shows the algorithm of the forward transformation of DFT and the forward transformation of $\tilde{a} = \mathrm{NTT}(a)$

$$\mathrm{DFT}: X_k = \sum_{i=0}^{n-1} x[i]\, e^{\frac{-j2\pi ik}{n}}, k = 0,1,2,\dots,n-1$$

$$\mathrm{NTT}: \tilde{a}[i] = \sum_{j=0}^{n-1} x[j]\, \omega^{ij} \bmod q, \text{for } i = 0,1,\dots,n-1$$

NTT is a specialized version of the discrete Fourier transform, as we implement FFT, computing DFT in O(nlogn), we can also implement NTT in O(nlogn). On the left side of Fig. 3-2 shows the block diagram of forward FFT/NTT.

Because of those similarities of FFT/NTT, we try to combine them together to possibly share the logic in processing element. Table 3-1 demonstrates the resource usage of re-structured FFT/NTT, and the resource usage after combining them together.

Table 3-1
Re-structured FFT/NTT

Resource	**DSP**	**FF**	**LUT**
FFT	61	8414	11456
NTT	30	4016	7278
FFT_NTT	41	6841	8637

- FFT&iFFT, NTT&iNTT

Let's look at the inverse FFT:

$$x[n] = \frac{1}{n} \sum_{m=0}^{N-1} X[m]\, e^{\frac{j2\pi mn}{N}}$$

We can see that apart from the difference of the index of twiddle factors, it is divided by N outside the sigma. In our Falcon project, **N** is defined to 1024, which is a power of 2, can be seen as shifting elements in binary.

Fig. 3-2 shows both the forward/inverse FFT/NTT. We can see that FFT/iFFT have the same operators, just in a different order, we can reuse the previous PE which compute FFT/NTT to share their operators. In the next part, we will explain how to also integrate the division by N outside of the inverse FFT/NTT sigma into the kernel.

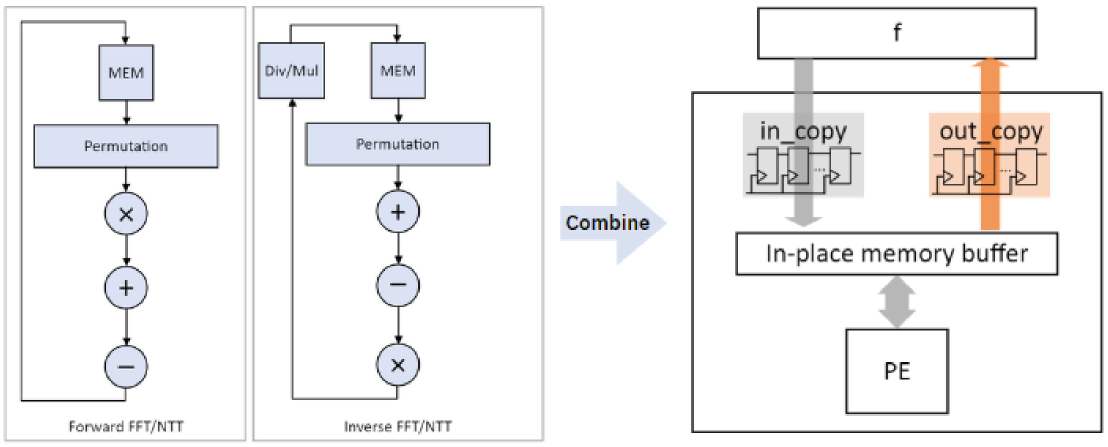

Fig. 3-2 Combination of inverse/forward FFT/NTT

- Output Copy

Since we've allocated an in-place memory buffer to calculate and store data, we must send that data to output after finishing the computation. Fig. 3-2 shows a brief structure about our kernel.

1. f is the user input, which expects to return the calculated forward/inverse FFT/NTT.
2. in_copy is a module looping N times copying data from f to our memory buffer in-place buffer.
3. The module out_copy is crucial here; we implement the calculation (dividing by N) here. Because the computation for this part is placed outside sigma, and the module out_copy is also activated only after the computation inside sigma is completed, the purpose is to transmit the computations completed within sigma from in-place buffer to f through looping N times. Therefore, we can embed the calculation of dividing by N in the outermost layer of inverse FFT/NTT right here.

3.1.5 Share the Memory

Since FFT uses the double datatype and NTT uses uint16_t, we must utilize two different buffers to store their respective computational results. We consider using a shared buffer, aims to reduce our memory usage. Thus, we use a self-defined datatype **memcell** to save data. A memcell is a union that can save 1 64-bits floating data or 4 16-bits unsigned integer.

Table 3-2
Combine FFT / iFFT / NTT / iNTT (inde: separate buffers, Ver0: combine buffers)

Resource	**BRAM**	**DSP**	**FF**	**LUT**
fiFFNTT inde	32	79	13333	15919
fiFFNTT Ver0	26	78	11774	13543

From Table 3-2, we can see that the reduction of memories and area is significant. Finally, we combined 4 algorithms (FFT, iFFT, NTT, iNTT), named as **fiFFNTT (forward/inverse Fast-Fourier & Number Theoretic Transform)**

Comparing Table 3-1 and Table 3-2, we also discover the increasing usage of DSP, since the inverse FFT&NTT need other calculations outside the main computing loop (sigma), which are dividing by N (iFFT) and Montgomery multiplication (iNTT)

Calculating the inverse Fast Fourier Transform (iFFT) with division operations on double type elements significantly increases DSP usage. The later part will solve this issue by applying a double shifter.

3.1.6 Double Shifter

Based on IEEE-754 double precision, we implement a shifter which can shift the variable with data type double to implement dividing N in iFFT.

Flow:

1. Get the exponent, which is located at bit 62 to 52.
2. Subtract 9 to exponent (which is dividing double by 2^9)
3. Shift back to bit 62 to 52.
4. Handle underflow/overflow:

The above statement replaces the usage of DSPs (11 DSPs) with just some simple logic.

3.1.7 Share the Multiplier

To share the Multiplier, we deconstructed the complex multiplier and monty multiplier in the PE, and we encapsulated the 32-bit integer, 64-bit double adder and multiplier into functions to limit the usage using pragma to realize sharing multiplier and adder.

3.1.8 Embed Falcon Flow

After sharing the memories and multiplier, we introduced our most balanced optimization. In addition to combining four functions, we expand the function coverage to adj_fft and mul_fft. The reason we integrated these two functions is the consecutive usage in Falcon flow. Since the FFT and iFFT implemented in HW have different output orders from the SW implementation, we must add a converter to successfully communicate with SW. Therefore, if the functions are frequently called, the time consumed in conversion is unacceptable. To include these two functions, we can eliminate the delay from HW and SW communicated conversion.

4. FPGA Implementation

4.1 HW/SW Co-design

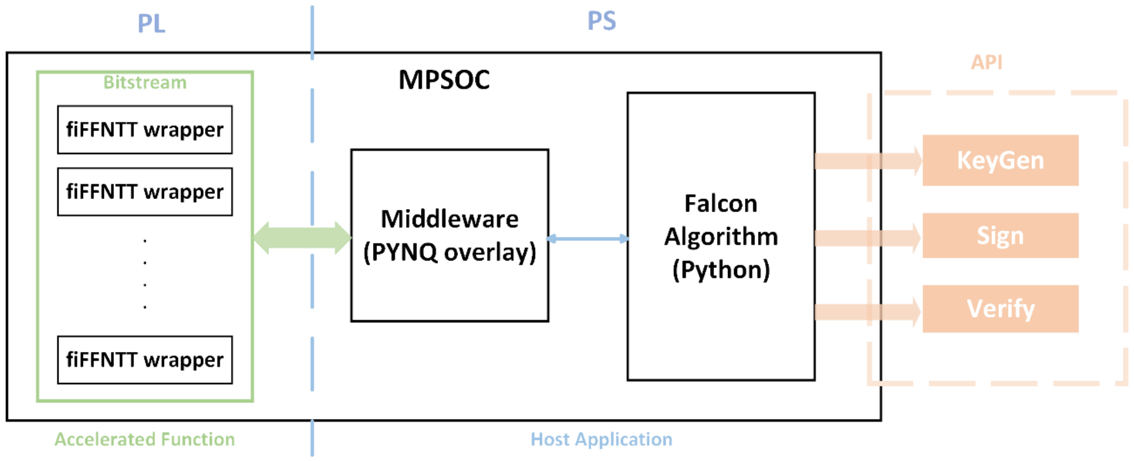

Fig. 4-1 Architecture of the Falcon flow

Fig. 4-1 outlines our architecture where we run Falcon on the host side (PS side) and have multiple fiFFNTT wrappers on the FPGA (PL side), interspersed with a middleware layer. This middleware is designed to receive fiFFNTT requests from the host side and allocate them to the available fiFFNTT hardware that is not currently operating. On the host side, three APIs (KeyGen, Sign, Verify) are exposed to the user.

4.1.1 Middleware

To manage simultaneous requests from Falcon, we developed middleware that preloads and process multiple requests, efficiently managing buffer usage, Initially, we used a custom datatype called `memcell` to accommodate different data types for FFT/NTT operations, but this slowed execution speed. We resolved this by allocating separate buffers for each data type on the software side, where ample memory ensures efficient resource management without impacting hardware capabilities.

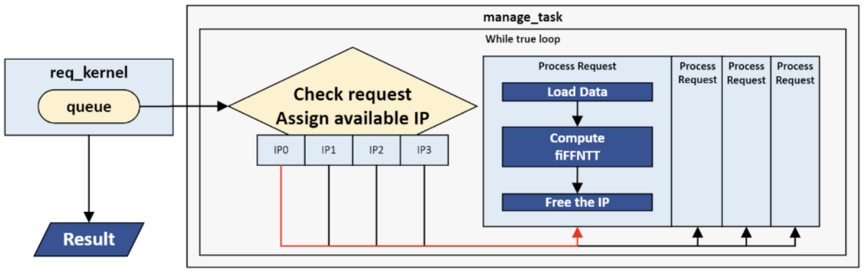

Fig. 4-2 Middleware

5. Result

Table 5-1
Final resource usage (fiFFNTT wrapper includes 2 fiFFNTT kernels)

Resource	**BRAM**	**DSP**	**FF**	**LUT**
fiFFNTT wrapper	48	127	30345	25678

Table 5-2
Final speed (Unit: ms)

Function	**FFT**	**iFFT**	**NTT**	**iNTT**
Python	28.3617	29.9833	32.8956	34.3557
Initial HLS	1.7429	2.2315	3.3797	4.3449
Final optimization	0.1372	0.1631	0.1951	0.1773
Speed up rate	207	184	170	194

Table 5-3
Final speed (Unit: ms)

Version	**KeyGen**	**Sign**	**Verify**
Original software	100.4	2053.3	139.9
HW/SW co-design with middleware	18.9	747.5	60.7

6. Future Work and Conclusion

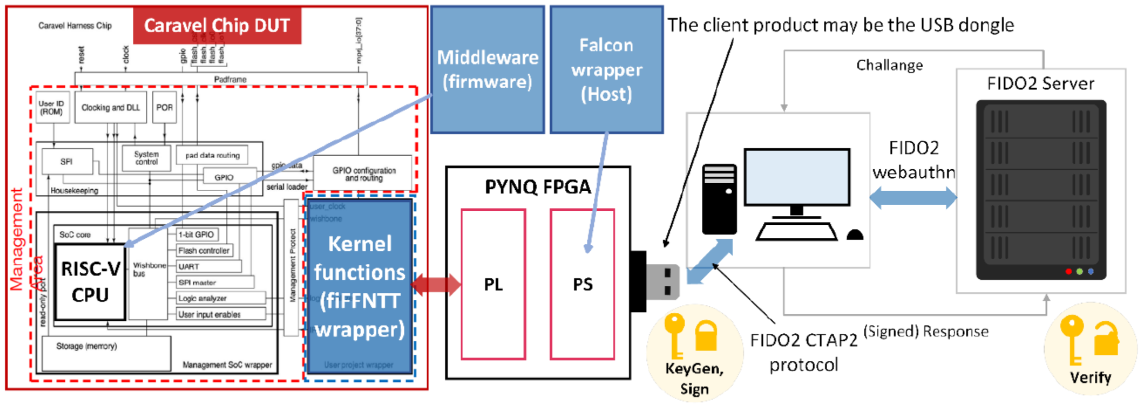

Fig. 6-1 System view

During the chip verification phase, we integrated multiple kernels into the PL side of the MPSoC to handle concurrent requests, and middleware to manage requests from Falcon. To minimize the high communication overhead between hardware and software (as illustrated by the green double arrow in Fig. 4-1), the middleware will be implemented as firmware on the RISC-V CPU within the Caravel SoC.

Additionally, the kernel will be integrated into the user project on the Caravel SoC which will be included in the tape-out of the Caravel Chip DUT. This integration provides user-accessible APIs and is encapsulated in an FPGA USB dongle. The dongle facilitates PC communication via FIDO2 CTAP2, adhering with FIDO Alliance specifications for hardware-based authentication. This enhances security and simplifies authentication across different platforms.

In conclusion, our research has significantly advanced the field of post-quantum cryptography by accelerating the Falcon algorithm through hardware. We have not only streamlined the development process but also greatly improved the computational efficiency and security of critical cryptographic operations such as FFT, iFFT, NTT, and iNTT.

Our innovative hardware/software co-design approach has successfully integrated the Falcon algorithm with hardware platforms, notably reducing execution times through effective management and distribution of cryptographic tasks, further underscore our project's practical implications, allowing for real-time cryptographic processing that significantly outperforms traditional software-based methods.

However, there remains a challenge: the middleware still operates as software on the PS side, leading to notable hardware/software communication overhead. Future efforts will focus on transforming the middleware into firmware on the hardware side to further reduce these inefficiencies.

Our project provides a robust response to the threats posed by quantum computing and establishes a foundational framework for future innovations in hardware-accelerated cryptography, paving the way for more secure and efficient cryptographic solutions.

7. Review and Reflection

From this project, we have learned a lot. In the first half of the semester, we made some preparations for the project in the next semester, including getting familiar with the entire Caravel SoC environment, learning embedding programming through labs, practicing remote operation of FPGA through Python, and getting familiar with Verilog and HLS, etc. In the second half of the semester, after determining the topic, we first studied the entire Falcon algorithm. Since we did not have relevant background knowledge, we searched a lot of literature, learned basic cryptography knowledge, and all the mathematical theories applied in Falcon. Then we selected several important kernel functions to try to accelerate through HLS.

Very few reference materials have done HLS for Falcon, and they only rewrite the C code without optimizing the entire architecture, result in the enormous amount of hardware resources usage and long execution time. So, we spent a lot of time thinking about how to optimize the original kernel function. After completing the hardware architecture of the kernel function, we need to connect the Falcon running on the software side with our hardware. In response, we need to design middleware to handle communication on both sides. During the process, we found that the data type has a great impact on the overall performance, and the entire flow of Falcon will use different data types. If not handled well, the performance may even be worse than the original Falcon.

In addition to learning professional knowledge, we also learned how a team can work together to solve a difficult problem, including how to use software such as GitHub, Slack, and HackMD for collaboration, how to find the necessary literature, how to divide and conquer, and so on. In conclusion, this course has been very rewarding, and we would like to thank Professor Jiin Lai for his guidance.

8. Reference

[1] Fouque, Pierre-Alain, et al. "Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU Specification." v1.2, 1 Oct. 2020, https://falcon-sign.info/

[2] Pedro Paz, Mario Garrido, “Efficient Implementation of Complex Multipliers on FPGAs Using DSP Slices,” April 2023

[3] Michael Schmid, Dorian Amiet, Jan Wendler, Paul Zbinden, and Tao Wei, “Falcon Takes Off - A Hardware Implementation of the Falcon Signature Scheme,” 2023

[4] Zeynep Kaya, Mario Garrido, “Memory-Based FFT Architecture with Optimized Number of Multiplexers and Memory Usage,” IEEE Transactions on Circuits and Systems, August 2023