國立清華大學 電機工程學系

實作專題研究成果報告

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

專題領域：系統領域

組　　別：A400

指導教授：賴瑾 教授

組員姓名：陳冠晰、劉祐瑋、陳昇達、王彥智、陳柏翰

研究期間：2024 年 2 月 5 日至 2024 年 5 月 5 日止，共 3 個月

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

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 a middleware that manages communication and task scheduling between the software interface and the FPGA hardware. This approach allows the scalable design: e.g. the increase of the hardware components without the modifications on application code.

The project was originally integrated into a 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.

摘要

量子計算的發展對傳統的加密協定產生了巨大的威脅。為了應對這一項挑戰，本專題透過開發 Falcon 演算法在 FPGA 板上的實現，期望能夠推進後量子密碼學的發展。Falcon 的主要優點便是其強大的抗量子攻擊能力和高效率晶格簽章。本專題從 C code 過渡到使用高階合成 (HLS) 產生的硬體描述語言 (HDL)，我們優化了快速傅立葉變換 (FFT) 和數論轉換 (NTT) 等關鍵演算法，以提升性能和安全性。

在報告的開頭，我們將討論 Falcon 演算法的優勢以及其主要功能。我們的目標是通過將 Falcon 演算法實現為硬體來進行改進，因此我們挑選了一些在過程中會頻繁使用到的演算法，例如快速傅立葉變換(FFT)、數論轉換(NTT)等。

硬體優化的流程涵蓋以下內容，將原始的 C code 轉換成可以合成的 HLS code、重新建構 FFT、iFFT、NTT 和 iNTT 四個演算法以達到改善資料流並減少執行時間。此外，我們還優化了一些邏輯的設計在不犧牲性能的前提下盡量最小化 FPGA 資源的使用。進一步的優化的事項包含簡化流水線的每個階段、增強平行處理的能力和優化記憶體的使用以降低延遲並提高吞吐量(throughput)。

我們的方法採用全面的軟硬體協同設計的方法，包含一個中介(middleware)來處理軟體與 FPGA 硬體之間的溝通和任務排程。透過在 KV260 FPGA 板上使用 Python 的 PYNQ API 進行合成，大大提升了效率和處理速度。

我們最初將實現的硬體合成到 MPSoC 中，提供三個用戶可訪問的 API：KeyGen、Sign 和 Verify。我們計劃將這些功能部署到 USB dongle 的格式中，通過 FIDO2 CTAP2 協議與桌上型電腦或筆記型電腦溝通。這樣的方法符合 FIDO 聯盟的硬體認證規範，透過減少對密碼的依賴並簡化跨平台認證的流程來提高安全性。

實驗結果表明，我們採用硬體加速的方法明顯優於原本軟體的方法，希望能夠為未來使用硬體加速密碼學的計畫提供一個重要的階段性進展，確保在不斷變化的網絡安全格局中的發展性和適應性。

Table of Contents

1. Introduction

1-1. Contributions

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 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.

Our research aims to address this challenge by transforming the Falcon algorithm from C code 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 executing more efficiently on hardware platforms. We have finalized the hardware design, generated a bitstream, and developed middleware on the PS (Processing Subsystem) side of 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 executing on the board, significantly reducing execution times compared to its software-only counterpart.

Our implementation results demonstrate that the hardware accelerated FFT, iFFT, NTT, and iNTT significantly outperform traditional methods, showcasing the effectiveness of our approach in real-world scenarios. This project not only advances the field of post-quantum cryptography by providing a more secure and efficient solution to the threats posed by quantum computers but also sets a foundational framework for future research in hardware-accelerated cryptography.

1-2. Falcon Introduction

We selected Falcon as our target algorithm for its numerous benefits. Falcon ensures high security by utilizing a true Gaussian sampler, which minimizes key leakage even with extensive usage. Its compact signatures, derived from NTRU lattices, offer smaller public keys and shorter signatures compared to other algorithms. Falcon also enhances performance with its fast Fourier sampling, achieving thousands of signatures per second and faster verification times. Additionally, it scales effectively with an $O(n \log n)$ computational cost, providing long-term security at an affordable rate. Furthermore, Falcon's key generation requires less than

30 kilobytes of RAM, significantly boosting efficiency and making it ideal for devices with limited memory. The Falcon API, adhering to NIST guidelines, includes three primary functions:

1. Key Generation: Creates a public and private key pair.
2. Signature Generation: Produces a signature using the private key and a message.
3. Signature Verification: Confirms the signature's validity using the signature, public key, and message.

We used the Vitis-HLS and Vivado tool to synthesize and implement the kernel function from Falcon algorithms on a KV260 FPGA board.

2. Hardware Acceleration

2-1. Determine the Kernel Function

Initially, we utilized profiling tools to determine the most frequently executed sections of Falcon or where most of the processing time is spent. Consequently, we chose FFT, IFFT, NTT, and INTT as our core functions.

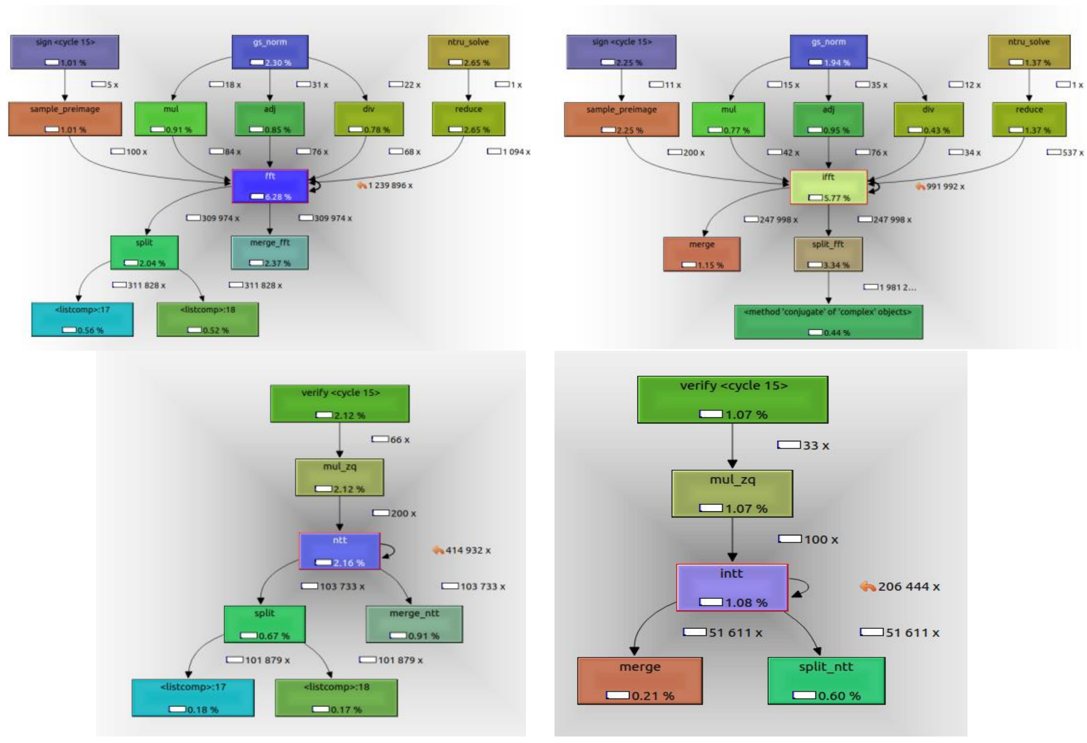
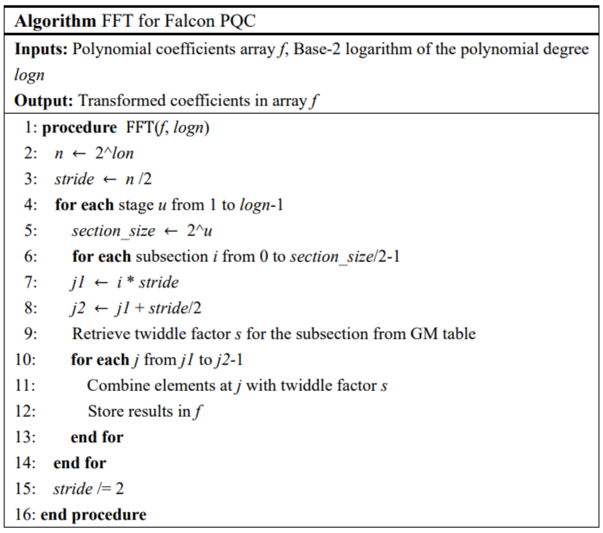


Fig. 2-0 Callgraph of critical functions in Falcon

2-2. Initial Implementation on HLS

The iterative radix-2 FFT algorithm with bit-reversal permutation efficiently computes the Fast Fourier Transform, optimizing memory use and processing steps for real-time applications. This in-place algorithm minimizes extra memory by modifying the input data directly. It breaks the FFT into binary stages, handling pairs of data points effectively when the number of data points, N, is a power of two. Over $\log_2 N$ stages, it segments the dataset, applies size-2 DFTs with twiddle factors, and combines results to complete the FFT.

Both FFT and NTT share similar computational frameworks. The NTT, a variant of the discrete Fourier transform, uses a finite field or ring that includes the necessary roots of unity, enabling exact convolutions of integer sequences without roundoff errors.

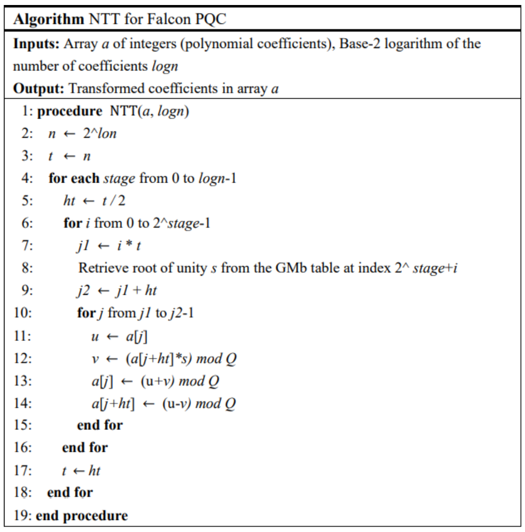


Algorithm FFT for Falcon PQC

Inputs: Polynomial coefficients array *f*, Base-2 logarithm of the polynomial degree *logn*
Output: Transformed coefficients in array *f*

```
procedure FFT(f, logn)
  n ← 2^lon
  stride ← n /2
  for each stage u from 1 to logn-1
    section_size ← 2^u
    for each subsection i from 0 to section_size/2-1
    j1 ← i * stride
    j2 ← j1 + stride/2
    Retrieve twiddle factor s for the subsection from GM table
    for each j from j1 to j2-1
      Combine elements at j with twiddle factor s
      Store results in f
    end for
  end for
  stride /= 2
end procedure
```

Fig. 2-1 Pseudo code of FFT

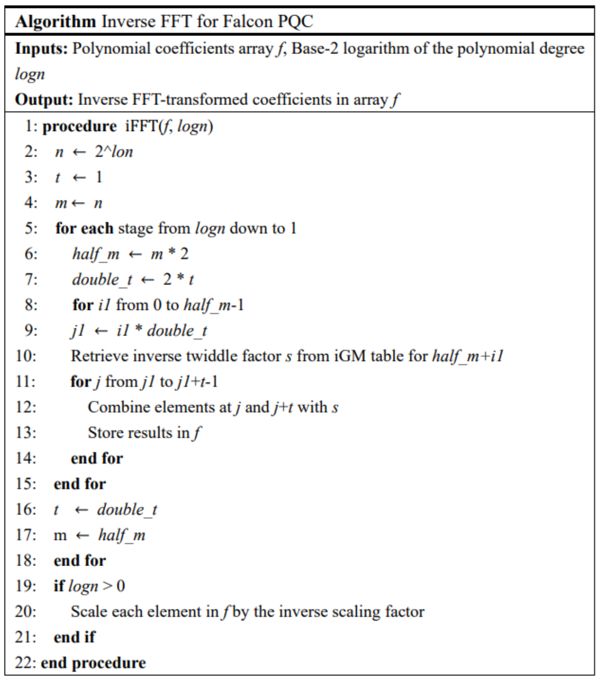


Algorithm NTT for Falcon PQC

Inputs: Array *a* of integers (polynomial coefficients), Base-2 logarithm of the number of coefficients *logn*
Output: Transformed coefficients in array *a*

```
procedure NTT(a, logn)
  n ← 2^lon
  t ← n
  for each stage from 0 to logn-1
    ht ← t / 2
    for i from 0 to 2^stage-1
      j1 ← i * t
      Retrieve root of unity s from the GMb table at index 2^ stage+i
      j2 ← j1 + ht
      for j from j1 to j2-1
        u ← a[j]
        v ← (a[j+ht]*s) mod Q
        a[j] ← (u+v) mod Q
        a[j+ht] ← (u-v) mod Q
      end for
    end for
    t ← ht
  end for
end procedure
```

Fig. 2-2 Pseudo code of NTT

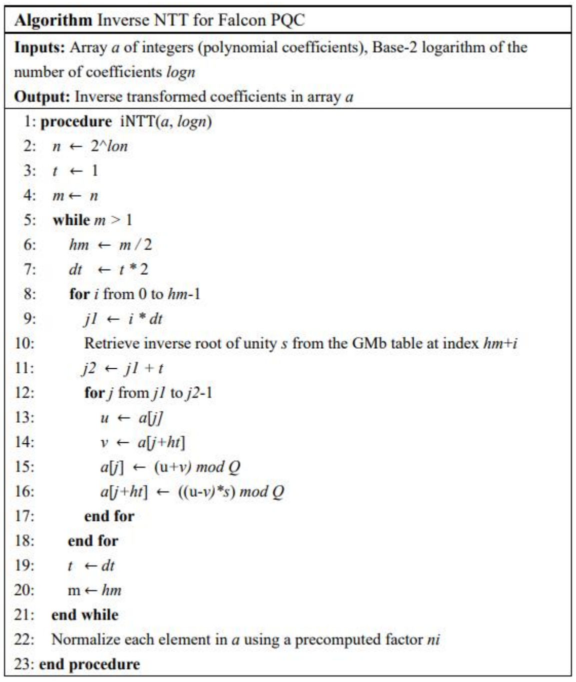


Algorithm Inverse FFT for Falcon PQC

Inputs: Polynomial coefficients array *f*, Base-2 logarithm of the polynomial degree *logn*
Output: Inverse FFT-transformed coefficients in array *f*

```
procedure iFFT(f, logn)
  n ← 2^lon
  t ← 1
  m ← n
  for each stage from logn down to 1
    half_m ← m * 2
    double_t ← 2 * t
    for i1 from 0 to half_m-1
    j1 ← i1 * double_t
    Retrieve inverse twiddle factor s from iGM table for half_m+i1
    for j from j1 to j1+t-1
      Combine elements at j and j+t with s
      Store results in f
    end for
  end for
  t ← double_t
  m ← half_m
  end for
  if logn > 0
    Scale each element in f by the inverse scaling factor
  end if
end procedure
```

Fig. 2-3 Pseudo code of IFFT



Algorithm Inverse NTT for Falcon PQC

Inputs: Array *a* of integers (polynomial coefficients), Base-2 logarithm of the number of coefficients *logn*
Output: Inverse transformed coefficients in array *a*

```
procedure iNTT(a, logn)
  n ← 2^lon
  t ← 1
  m ← n
  while m > 1
    hm ← m / 2
    dt ← t * 2
    for i from 0 to hm-1
      j1 ← i * dt
      Retrieve inverse root of unity s from the GMb table at index hm+i
      j2 ← j1 + t
      for j from j1 to j2-1
        u ← a[j]
        v ← a[j+ht]
        a[j] ← (u+v) mod Q
        a[j+ht] ← ((u-v)*s) mod Q
      end for
    end for
    t ← dt
    m ← hm
  end while
  Normalize each element in a using a precomputed factor ni
end procedure
```

Fig. 2-4 Pseudo code of INTT

The pseudocode above represents an abstraction of the kernel functions used in our hardware. Initially, we need to convert the original C code [1] into HLS for synthesis, addressing certain non-synthesizable elements in the process. A critical modification involves transforming recursive loops into iterative loops [3] because HLS cannot synthesize loops with undetermined endpoints, which is a major issue.

2-3. Optimization

2.3.1 Synthesizable Implementation

A good architecture will selectively expose and take advantage of parallelism and allow for pipelining. Our final architecture of FFT/iFFT/NTT/iNTT will restructure the code such that each stage is computed in a separate function or module. There will be 10 stages for the butterfly (PE) computations corresponding to the 2-point, 4-point, 8-point, 16-point, … FFT/NTT stages.

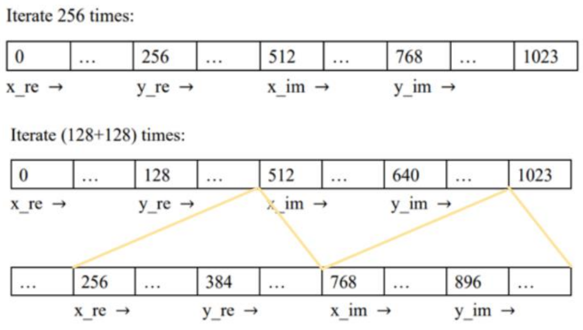


Figure 2-5 Permutation

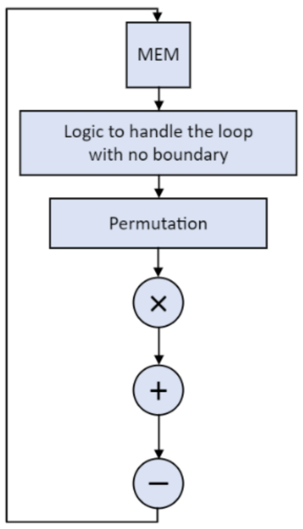


Fig. 2-6 FFT Datapath

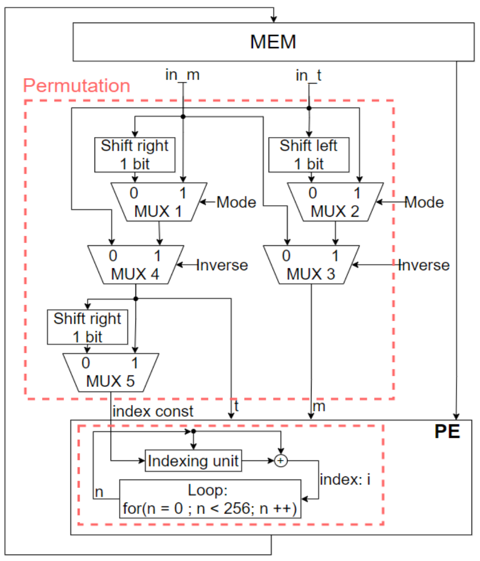


Fig. 2-7 Permutation Logics

Consider the FFT as an example: initially, we allowed the computation loop boundary to be variable, similar to pseudocode. This approach requires synthesizing logic to handle these boundaries (Fig. 2-6), potentially extending execution time. Therefore, we must address the issue of the unbounded loop structure. As shown in Fig. 2-5 (first stage and second stage) 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*

Fig. 2-7 shows our implementation of one-port memory with new custom permutation algorithm logic.

2.3.2 Datapath Restructure

In order to save the use of memory and area, we separate each stage and reuse the processing element (PE). We connect a number of functions (FFT here) in a staged fashion with arrays (fin) acting as buffers between the stages. Fig. 2-8 provides a graphical depiction of this process.

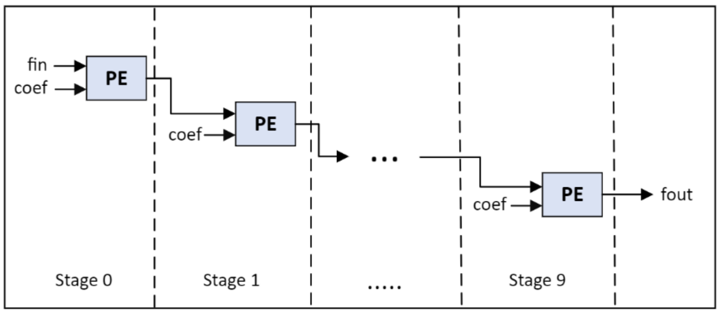


Fig. 2-8 Multiple stage PE

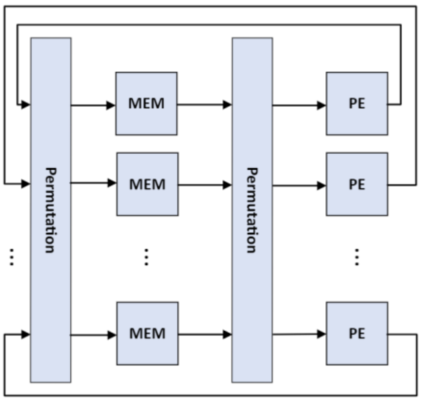


Fig. 2-9 Reuse PE with memories

From Fig. 2-9, 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 [4]. 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 number of PEs to achieve more parallelism.

2.3.3 Complex Multiplication

In the FFT used by Falcon, floating-point (double precision) is used to represent the complex polynomial coefficients, thus requiring the use of 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.

Fig. 2-10 demonstrates a direct implementation of a complex multiplier, which requires 4 real multipliers and 2 real adders.

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

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

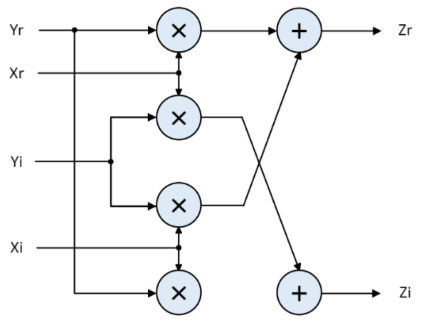


Fig. 2-10 Direct complex multiplier

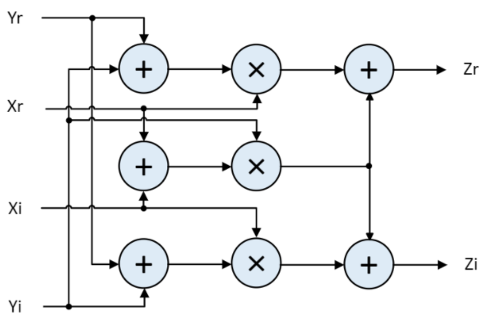


Fig. 2-11 Improved complex multiplier

Fig. 2-11 demonstrates the implementation of a complex multiplier based on it. This structure leads to area reduction although the usage of 5 adders instead of 2 adders needed in the direct implementation, since multipliers require significantly more area than adders.

2.3.4 Combine 4 Algorithms in One Hardware

- FFT & NTT

The following formula shows the algorithm of the forward transformation of DFT and the forward transformation of NTT:

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

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

NTT is a specialized version of the discrete Fourier transform, as we implement FFT, computing DFT in $O(n \log n)$, we can also implement NTT in $O(n \log n)$. On the left side of Fig. 2-12 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 2-1 demonstrate the resource usage of re-structured FFT/NTT, and the resource usage after combining them together.

Table 2-1

Comparison of separate FFT/NTT and combined version

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 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. 2-12 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.

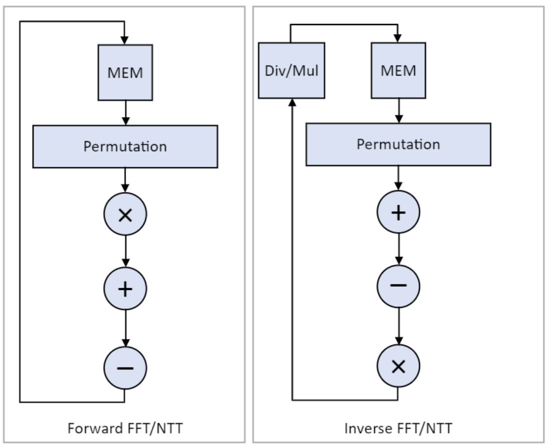


Fig. 2-12 Forward/Inverse FFT/NTT

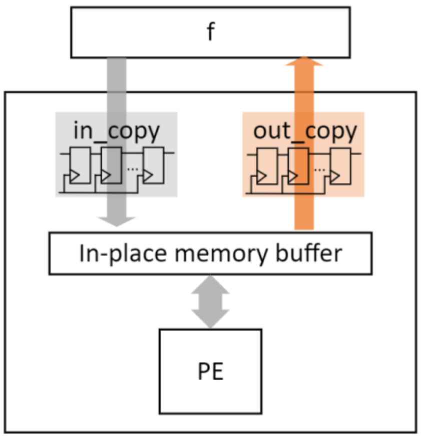


Fig. 2-13 Brief kernel structure

- Output Copy

Since we've allocated an in-place memory buffer in our kernel to calculate and store data, we need to send those data to output after finishing the computation. Fig. 2-13 shows a brief structure about our kernel.

1. *f* is the user input, which expect 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 since 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.

2.3.5 Share the Memories

Since FFT utilizes the datatype of 64-bit floating-point(double) and NTT employs unsigned short integer, we initially use two separate 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.

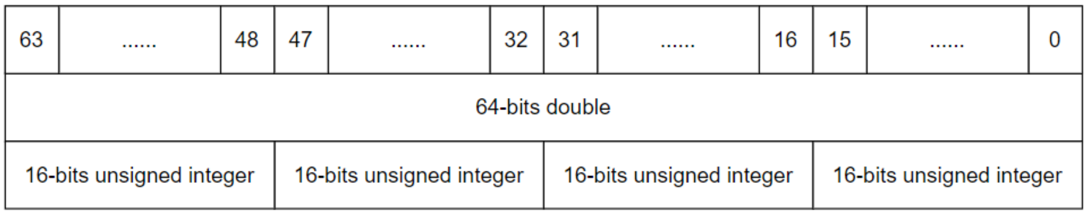


Fig. 2-14 Self defined datatype memcell

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

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

From Table 2-2, 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 2-2 and 2-1, 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. Later part will solve this issue by applying a double shifter.

2.3.6 Double Shifter & Negation

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.

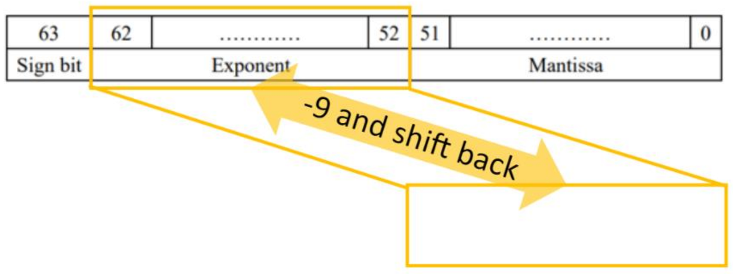


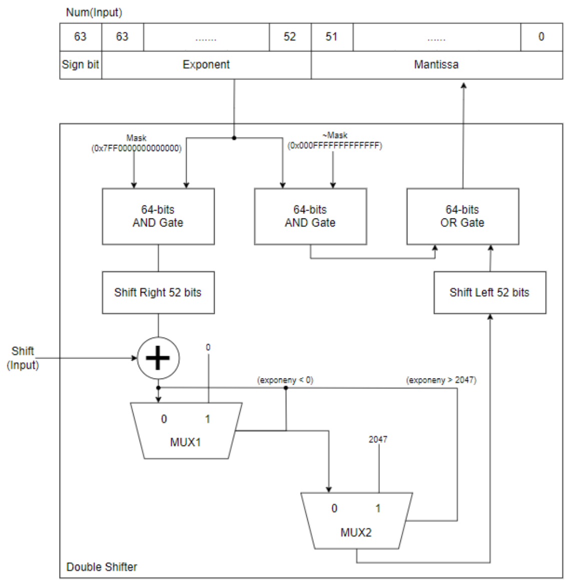
Fig. 2-15 schematic diagram of shifter

Flow:

1. Get the exponent, which located at bit 62 to 52.
2. Subtract 9 to exponent (which is dividing double by 2^9)

3. Shift back to the location from bit 62 to 52.
4. Handle underflow/overflow.

Above statement replaces the usage of DSPs (From Table 2-5, **fiFFNTT_Ver1** decrease 11 DSPs) with just some simple logic.



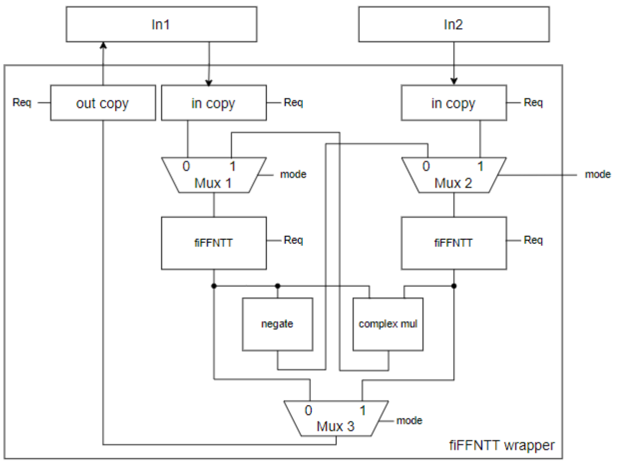


Fig. 2-16 Double Shifter Fig. 2-17 fiFFNTT wrapper

Next, in double subtraction, we share the existing design of an adder plus negation, which can be accomplished by simply toggling the sign bit with a few logic gates, avoiding the need to create an extra subtractor and increase resource usage.

2.3.7 Share the Multiplier

To share the Multiplier, we deconstructed the complex multiplier and Montgomery multiplier in the PE, and we encapsulated the 32-bit integer, 64-bit double adder and multiplier into functions in order to limit the usage using pragma to realize sharing multiplier and adder. (**fiFFNTT_Ver2** in table 2-5)

```
#pragma HLS ALLOCATION function instances=d_mul limit=3
#pragma HLS ALLOCATION function instances=d_add limit=6
#pragma HLS ALLOCATION function instances=u_add limit=2
```

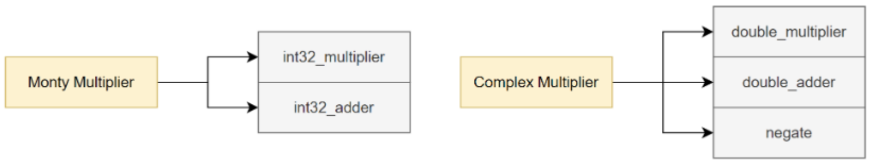


Fig. 2-18

2.3.8 Embed Falcon's Computation Process

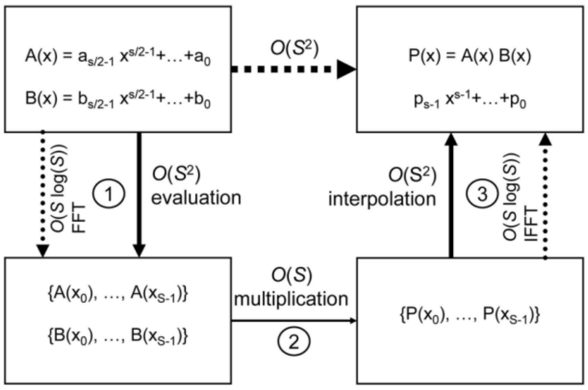
After sharing the memories and multiplier, we introduced our most balanced optimization (ver2). This version not only combines 4 functions bus also extends the functionality to include *adj_fft* and *mul_fft*. We integrated these 2 functions due to their sequential use in the Falcon computation process. Since the FFT and iFFT implemented in hardware produce outputs in a different order from the software implementation, we added a converter to facilitate successful communication with the software. The frequent calling of these functions makes the time spent on conversion excessively long and increases hardware/software communication overhead. By incorporating these 2 functions, we can eliminate the delays caused by hardware and software conversion communication.

Table 2-3

Operation mode (according to Fig. 2-17)

Mode	Function	Mux1	Mux2	Mux3	Reqest
0	FFT	0	1	0	FFT
1	iFFT	0	1	0	iFFT
2	NTT	0	1	0	NTT
3	iNTT	0	1	0	iNTT
4	adj_FFT	0	0	1	FFT -> iFFT
5	mul_FFT	0 -> 1	1	0	FFT -> iFFT

- **adj_FFT** is the adjoint operation by FFT. To do the adjoint operation, we first do the FFT, then do the conjugate in the frequency, and finally do inverse FFT to change it back to time domain.
- **mul_FFT** calculate the multiplication of two polynomial by FFT which reduce the complexity to $O(n \log n)$. To do the multiplication of two polynomial, we first do FFT to both polynomial, then do the complex multiplication in frequency domain, and finally do inverse FFT to change the result back to time domain.



$A(x) = a_{s/2-1} x^{s/2-1}+\ldots+a_0$
$B(x) = b_{s/2-1} x^{s/2-1}+\ldots+b_0$
$O(S^2)$
$P(x) = A(x) B(x)$
$p_{s-1} x^{s-1}+\ldots+p_0$
$O(S \log(S))$ FFT
1
$O(S^2)$ evaluation
$O(S^2)$ interpolation
3
$O(S \log(S))$ IFFT
$\{A(x_0), \ldots, A(x_{S-1})\}$
$\{B(x_0), \ldots, B(x_{S-1})\}$
$O(S)$ multiplication
2
$\{P(x_0), \ldots, P(x_{S-1})\}$

Fig. 2-19

Table 2-4

Resource/Speed comparison (latency unit: cycle)

Resource	BRAM	DSP	FF	LUT	LATENCY
Initial implementation					

FFT	8	25	4242	4261	119600
IFFT	8	25	4700	4706	143267
NTT	1	3	1428	2609	224273
INTT	1	4	2026	3084	278408
PE Optimization					
FFT	40	61	8414	11456	4933
IFFT	80	62	13140	13027	12817
NTT	5	30	4016	7278	7355
INTT	5	31	4787	7594	15674

Table 2-5

Integrating 4 algorithms in one hardware (latency unit: cycle)

Version	**Resource**				**Latency**	
	BRAM	**DSP**	**FF**	**LUT**	**FFT/iFFT**	**NTT/iNTT**
fiFFNTT Ver0	26	78	11774	13543	14730	20421
fiFFNTT Ver1	26	67	11253	13303	14668	8976
fiFFNTT Ver2	22	56	14423	12483	14720	9019

Operating at a frequency of 100 MHz (10ns/cycle).

3. FPGA Implementation

3-1. Integrate into MPSOC

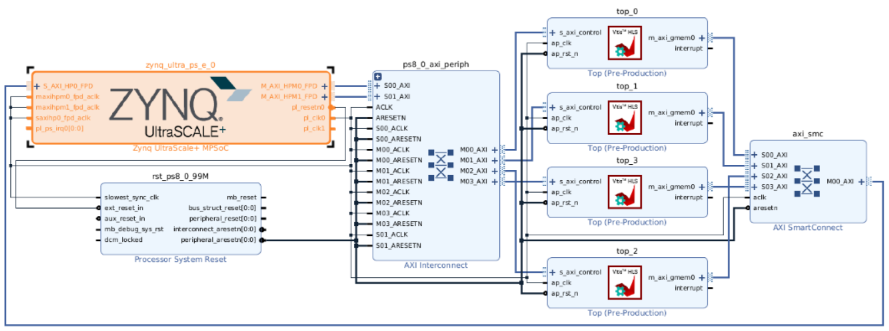


Fig. 3-1 Block Diagram

Table 3-1

Execution time (Unit: ms / execution)

Function	**FFT**	**iFFT**	**NTT**	**INTT**
Python	28.3617	29.9833	32.8956	34.3557
Initial HLS	1.7429	2.2315	3.3797	4.3449
fiFFNTT Ver2	0.1372	0.1631	0.1951	0.1773
Speed up rate	207	184	170	194

For the execution time measurement, we accounted for variance in Python by measuring 10 times and then calculating the average.

3-2. HW/SW Co-Design

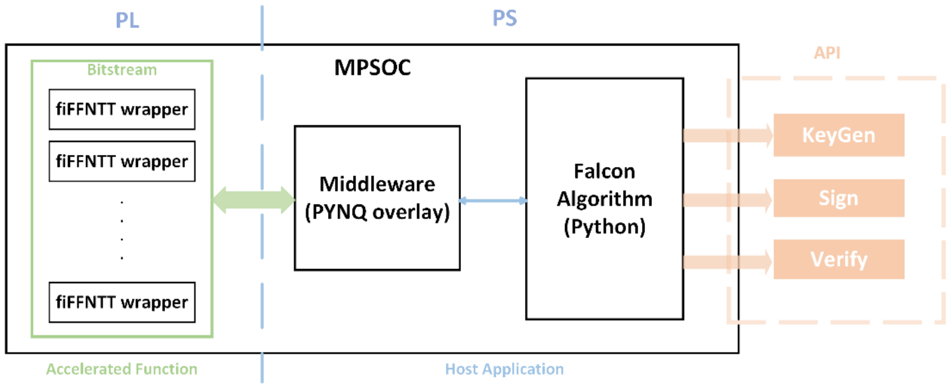


Fig. 3-2 Architecture of the Falcon flow

Fig. 3-2 outlines our architecture where we run an implementation of Falcon in Python [1] 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.

3.2.1 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.

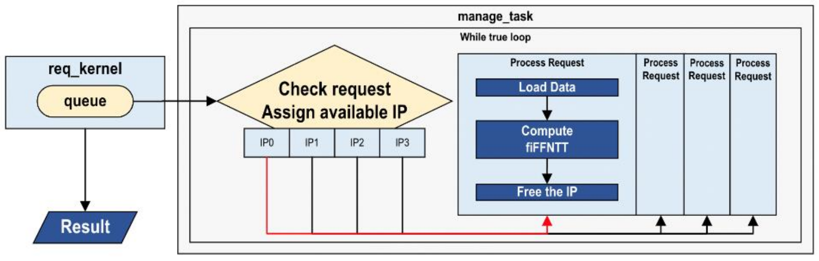


Fig. 3-3 Middleware

After using middleware, we can schedule the 4 kernels concurrently. Code illustrated below:

```
task1 = req_kernel(ntt_in, tmp, 0x2)  # non-blocking
task2 = req_kernel(intt_in, tmp, 0x3) # non-blocking
task3 = req_kernel(fft_in, tmp, 0x0)  # non-blocking
task4 = req_kernel(ifft_in, tmp, 0x1) # non-blocking
```

```
result1 = await task1
result2 = await task2
result3 = await task3
result4 = await task4
```

Fig. 3-4

3-3. Experimental Results

Table 3-2

Final resource usage (fiFFNTT wrapper includes 2 fiFFNTT kernels)

Resource	BRAM	DSP	FF	LUT
fiFFNTT warpper	78	127	30345	25678

Table 3-3

Final speed (Unit: ms)

Flow	KeyGen	Sign	Verify
Original software	100.4	2053.3	139.9
HW/SW with middleware	18.9	747.5	60.7

Above, the KeyGen section, we only focus on comparing the time it takes to compute the public key, as the time required to solve NTRU varies and cannot serve as a reliable benchmark for optimization. Although the way the middleware manages requests has greatly accelerated the entire Falcon flow, running the middleware on the PS side as software would still result in a hardware/software communication overhead issue. Further optimization is possible, so we plan to transform the middleware into firmware and place it on the hardware side to reduce communication overhead.

4. Conclusion

4-1. Future Work and Conclusion

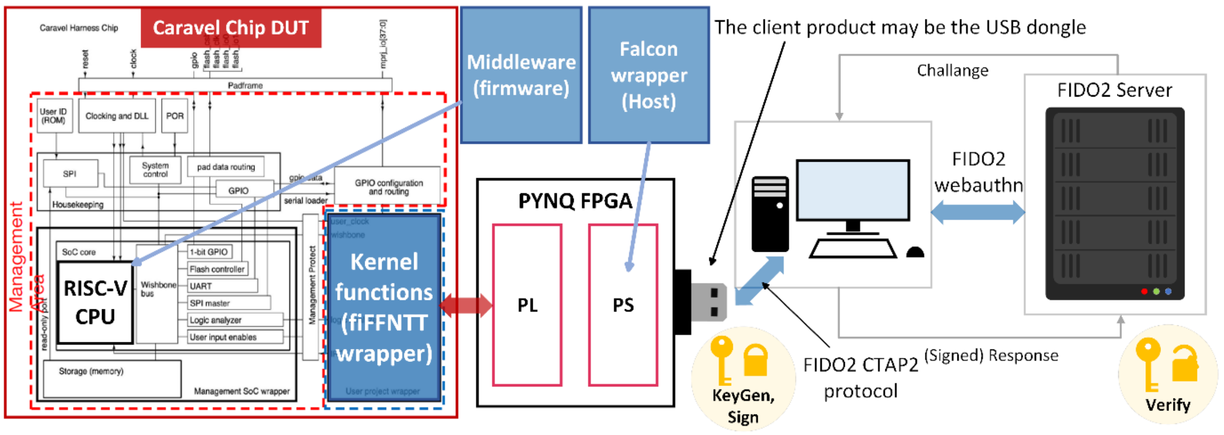


Fig. 4-1 Overall system view

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. 3-2), 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.

5. 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

6. Plan Management and Teamwork

6-1. Plan Management

The initial collaboration invitation came from the ITRI, so we started having weekly meetings during the winter break, which continued until early February when we began studying the Falcon algorithm and post-quantum cryptography. After the school term started, we began the design phase. Every week, we held regular meetings with our professor to discuss out progress and implementation, seek advice, and solve problems.

6-2. Teamwork

- 陳冠晰/Kuan-Hsi Chen:

  Tasked with the entire design and development of all hardware, starting form the initial C code through the initial implementation of HLS to the final fiFFNTT wrapper. Implemented and managed all aspects of kernel optimization, verified kernel functionality on FPGA board, and assessed results within Falcon. Additionally, modified Falcon’s Python code to implement 3 APIs (KeyGen, Sign, Verify). Furthermore, developed, and designed middleware to facilitate communication between the software and hardware interfaces (PS-PL). Also responsible for authoring the final report and designing the accompanying poster.
- 劉祐瑋/You-Wei Liu:

  Enhanced the performance of the middleware and optimized the kernel, along with refining Python’s implementation of the Falcon flow. Extensive verification of the updated functionalities was conducted on the FPGA board to ensure reliability and efficiency of the system under real-world operating conditions.
- 王彥智/Yan-Zhi Wang:

  Optimized the kernel and conducted on-board verification for the kernel and Falcon, also wrote an abstract report. The abstract report not only consolidated all our contributions but also condensed the contents of the final report. Also, draw some of schematic diagrams in the final report.
- 陳昇達/Sheng-Ta Chen:

  Optimized the kernel and conducted on-board verification for the kernel and Falcon, also participated in the writing of the final report. This report detailed the optimization strategies and procedures and wrote down the result of improvements, providing thorough documentation.
- 陳柏翰/Bo-Han Chen:

  Optimized the kernel and conducted on-board verification for the kernel and Falcon, and organize the data from the optimization process, also participated in the writing of the abstract report. Besides drawing the schematic diagram of the middleware and took part in the design of the poster.