A Multi-Layer Parallel Hardware Architecture for Homomorphic Computation in Machine Learning

Guozhu Xin, Yifan Zhao, Jun Han, *Member, IEEE,*
State Key Laboratory of ASIC and System
Fudan University, Shanghai 201203, China
Email: junhan@fudan.edu.cn

2021 IEEE International Symposium on Circuits and Systems (ISCAS) | 978-1-7281-9201-7/20/$31.00 ©2021 IEEE | DOI: 10.1109/ISCAS51556.2021.9401623

***Abstract*—Homomorphic Encryption (HE) allows untrusted parties to process encrypted data without revealing its content. People could encrypt the data locally and send it to the cloud to conduct neural network training or inferencing, which achieves data privacy in AI. However, the combined AI and HE computation could be extremely slow. To deal with it, we propose a multi-level parallel hardware accelerator for homomorphic computations in machine learning. The vectorized Number Theoretic Transform (NTT) unit is designed to form the low-level parallelism, and we apply a Residue Number System (RNS) to form the mid-level parallelism in one polynomial. Finally, a fully pipelined and parallel accelerator for two ciphertext operands is proposed to form the high-level parallelism. To address the core computation (matrix-vector multiplication) in neural networks, our work is designed to support Multiply-Accumulate (MAC) operations natively between ciphertexts. We have analyzed our design on FPGA ZCU102, and experimental results show that it outperforms previous works and achieves over an order of magnitude acceleration than software implementations.**

***Index Terms*—Homomorphic encryption, machine learning, parallelism, hardware acceleration, FPGA**

I. Introduction

Nowadays, cloud computing has become the majority in both academia and industry. It allows people to offload the computation tasks to cloud servers without worrying about the size of the workloads, which is quite convenient and scalable. Therefore, many computation-intensive tasks such as AI applications are often deployed on the cloud and they provide helpful services for people such as object detection, recommendations and predictions. However, the provided unencrypted data could be misused or leaked by the cloud service provider, which raises privacy issues. One way to mitigate this issue is to encrypt the data using symmetric key encryption such as Advanced Encryption Standard (AES) [1], but that would cause the cloud server only capable to store the data rather than processing it.

Fortunately, Fully Homomorphic Encryption (FHE), was first achieved in 2009 by Gentry [2], and it allows arbitrary computation on encrypted data. By utilizing the privacy-preserving features provided by HE, people can provide their encrypted data to the cloud server to conduct certain computations such as machine learning, and then get their encrypted result back without leaking any information. With over ten years of research and improvement in FHE, the speed of HE computations is improved significantly, but it is still too slow to meet the needs of reality. Therefore, it is necessary to resort to efficient hardware architectures.

Many works have been done over the years, aiming to make HE schemes more practical. Software libraries such as HElib [3] and Microsoft SEAL [4] have been developed to make HE schemes easy to use, and GPU acceleration such as cuHe [5] has also been proposed. As for the hardware designs, some works [6]–[8] focus on large polynomial multiplications applying the NTT method, which is the basic computation in homomorphic encryption. [9] introduces hardware architectures for homomorphic encryption/decryption. However, since encryption/decryption is usually completed locally while homomorphic computations are often conducted on the cloud and more computation-intensive, it is more reasonable to accelerate homomorphic computation. Previous works such as [10]–[12] propose hardware accelerators for homomorphic computations, but these papers adopt the FV scheme [13] which is not suitable for AI applications requiring approximate computations. Recent work [14] propose pipeline architectures for homomorphic computations applying the CKKS scheme [15] for approximate computations. However, no resource reuse leads to large resource consumption in [15], and no optimization is exploited to support domain-specific applications.

In this paper, we propose a multi-level parallel hardware accelerator for homomorphic computations and the main contributions are as follows.

1) Multi-level parallelism is achieved in this work. In ciphertext levels, two operands can be processed in parallel, which forms the high-level parallelism. The mid-level parallelism is achieved by RNS, and the low-level parallelism is realized by the vector butterfly units, which could process the multiple finite field data in parallel.

2) The accelerator is carefully designed to support homomorphic MAC operations in ciphertext levels. By organizing the permutations of the encrypted data in advance, our design can accelerate the matrix-vector multiplications effectively.

II. Preliminaries

A. CKKS Scheme

CKKS scheme [15] is a construction of the homomorphic encryption scheme for approximate arithmetic. In the CKKS scheme, the addition and multiplication are both homomorphic, which means the following properties are satisfied.

This work is supported by the National Natural Science Foundation of China under Grant 61934002.

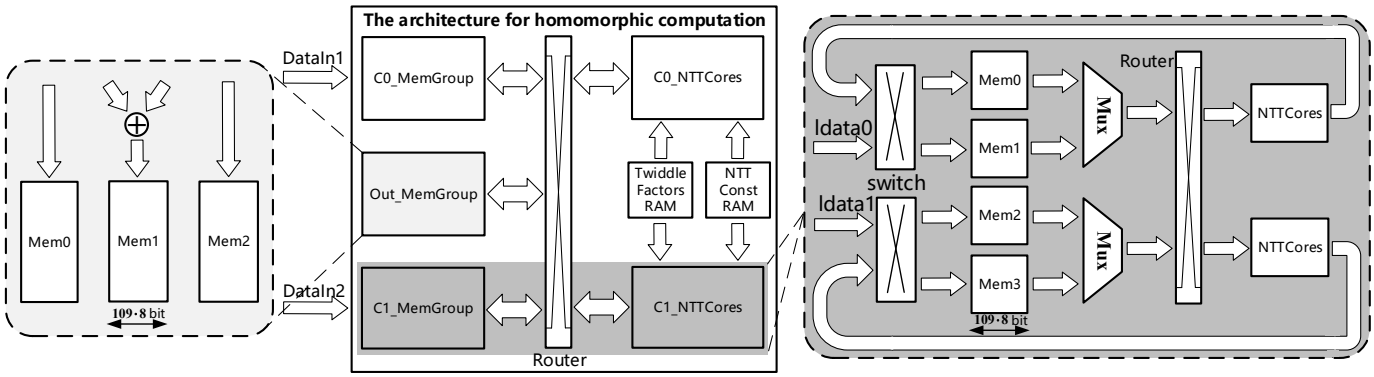

Fig. 1. The multi-level parallel accelerator for homomorphic computations.

$$\textbf{HE.Enc}(a) + \textbf{HE.Enc}(b) = \textbf{HE.Enc}(a + b)$$
$$\textbf{HE.Enc}(a) \cdot \textbf{HE.Enc}(b) = \textbf{HE.Enc}(a \cdot b)$$

The basic elements in CKKS are polynomials defined in $R_q = Z_q[X]/(X^n+1)$, which possess a degree at most $n-1$ and coefficients in Z_q. Polynomial multiplication is the core operations in HE, and its complexity can be reduced from $O(n^2)$ to $O(nlog(n))$ applying NTT methods [16]. CKKS scheme supports approximate computations by introducing a scaling process before the encryption, which causes slight rounding errors in the ciphertext. To support SIMD-style computations, CKKS introduces a batching technique by mapping a plaintext polynomial to a message vector of complex numbers using the complex canonical embedding map. This means multiple data could be packed into one polynomial and they are processed simultaneously during homomorphic computations. The fresh ciphertext consists of two polynomials, and the homomorphic addition and multiplication are defined as follows, where $ct = (c_0,\ c_1)$ and $ct' = (c'_0,\ c'_1)$.

$$ct + ct' = (c_0 + c'_0,\ c_1 + c'_1)$$
$$ct \cdot ct' = (c_0 \cdot c'_0,\ c_0 \cdot c'_1 + c_1 \cdot c'_0,\ c_1 \cdot c'_1)$$

B. Residue Number System (RNS)

When the modulus q becomes too large, the multiplication in the finite field could be quite slow thus harming the overall performance. To mitigate this situation, a technique called RNS is introduced to achieve asymptotic improvements by choosing $q = \prod_{i=0}^{L} p_i$, where integer p_i is coprime with each other. By virtue of the properties of ring isomorphism $R_q \mapsto \prod_{i=0}^{L} p_i$, the multiplications and additions in R_q can be computed on their RNS representations in parallel, which improves the performance significantly.

III. System Setup

A. Parameter Settings

FHE allows unlimited additions and multiplications, and it is usually achieved by Somewhat Homomorphic Encryption (SHE) combined with an extra procedure called bootstrapping. Bootstrapping operations evaluate the decryption circuit on the twice encrypted ciphertext and decrease the noise to a great extent. However, the bootstrapping method is very expensive in both timing and resources, so SHE is often adopted in reality which supports bounded depth of multiplications. As a lightweight design, we choose a multiplicative depth of one as a minimum requirement to support the matrix-vector multiplications. As suggested in [3], [17], the polynomial degree is set to 4096 and the bit widths of the moduli in RNS are 40, 29, and 40 respectively.

B. Data Arrangement

For matrix-vector multiplication, a naive method requires n^2 multiplications and $n(n-1)$ additions since it encrypts the data one by one and performs the multiplication traditionally. In [4], an approach known as the "diagonal method" is proposed, which only requires n polynomial multiplications and additions. In detail, the $n \times n$ matrix A is represented by n vectors $\tilde{a}_i (0 \le i < n)$ which contain the generalized diagonals of A. Namely, $\tilde{a}_i = (a_{0,i}, a_{1,i+1}, ..., a_{n-1,i-1})$, and $\tilde{a}_i[j] = a_{j,\ i+j \mod n}$. The vector v has cyclic variations that $\tilde{v}_i = (v_i, v_{i+1}, ..., v_{i-1})$ and $\tilde{v}_i[j] = v_{i+j \mod n}$. By doing so, the $M = Av$ can be expressed as $M = \sum_{i=0}^{n-1} \tilde{a}_i \cdot \tilde{v}_i$. Usually, a vector of data can be packaged into one polynomial, and the dot product of vectors can be computed by one polynomial multiplication, so the number of multiplications can be reduced from n^2 to n. Therefore, we can compute the product of 2048×2048 matrix and 2048×1 vector using only 2048 polynomial MAC operations.

IV. Hardware Implementation

A. Multi-level Parallel Accelerator

We propose our multi-level parallel hardware accelerator for homomorphic computation in Fig. 1. As shown on the right side of Fig. 1, we equip NTT cores for each polynomial in both two ciphertexts, which forms the high-level parallelism in ciphertexts. Inside the *NTTCores* block, there are three seperate NTT cores for each RNS component, which forms the mid-level parallelism. Low-level parallelism is achieved by vector butterfly units inside the *NTTCores* block, and it would be discussed later.

In addition, the data access and the NTT/INTT/MAC computations are processed simultaneously in a ping-pong fashion, which further improves the efficiency. The left side of Fig. 1 depicts three RAMs, which are used to store the results of

the MAC operations of polynomials. Previous results would be read out as the partial products for MAC operations, and the new results $res_0 = c_0 \cdot c_0'$ and $res_2 = c_1 \cdot c_1'$ are written to *Mem0* and *Mem2* directly. However, $c_0 \cdot c_1'$ and $c_1 \cdot c_0'$ must be added to get res_1 before written to *Mem1*. A data router is also implemented, which routes data in a certain way according to the types of operations (NTT, INTT or MAC).

B. RNS Implementations

The RNS method is often used to avoid expensive big-integer arithmetic, and it is adopted in many designs [11], [12], [14]. We also implement RNS in our accelerator by partitioning the large modulus into several small ones, thus achieving the mid-level parallelism.

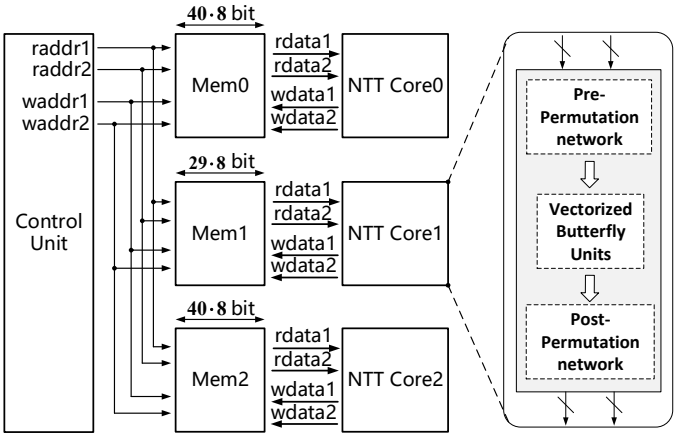

Fig. 2. RNS Implementations.

As Fig. 2 shows, the polynomial with 109-bit modulus is partitioned into three polynomials with 40-bit, 29-bit and 40-bit moduli respectively. The NTT core units, which are composed of permutation networks and vector butterfly units, are equipped for each RNS component. The necessity of the permutation networks would be discussed later. By doing so, all three polynomials in RNS can be computed in parallel. The control unit is also implemented to generate addresses of operands as well as the control signals for NTT cores.

C. Vector Butterfly Units

The polynomial multiplication can be computed applying the NTT-based methods. Namely, the multiplication result $c(x) = a(x) \cdot b(x)$ can be expressed as in equation (1) ($\odot$ denotes point-wise multiplication, $\psi = \sqrt{\omega} \bmod q$ where ω is the n-th primitive root of unity).

$$\begin{aligned} &c = n^{-1}\varphi^{-1} \odot INTT(NTT(a \odot \varphi) \odot NTT(b \odot \varphi)) \\ &(\varphi = (1, \psi, \ldots, \psi^{n-1}),\ \varphi^{-1} = (1, \psi^{-1}, \ldots, \psi^{-(n-1)})) \end{aligned} \tag{1}$$

As we can see, the main operations of polynomial multiplications are NTT/INTT and point-wise multiplications. When computing NTT by decimation in frequency (DIF) method and computing INTT by decimation in time (DIT) method, the bit reversals could be eliminated. In [18], a hybrid butterfly unit is designed to support both DIT and DIF methods. In this paper, we adopt the methods in [18] and design our vector butterfly units with 8 lanes, which form the low-level parallelism.

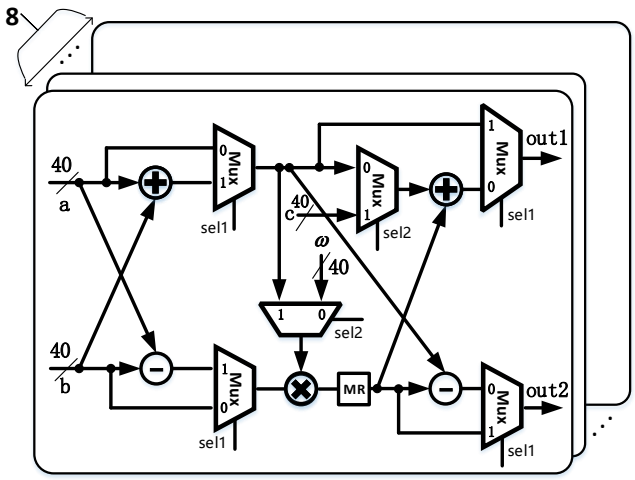

Fig. 3. The vector butterfly units (The pipeline registers are omitted).

As depicted in Fig. 3, we have also optimized the butterfly units to support MAC operations natively by reusing resources compared with [18]. The *MR* unit applies the Barrett reduction method because the moduli are not fixed. Signal *sel2* is used to select either MAC or butterfly operations, and signal *sel1* is used to determine the DIT/DIF mode in butterfly operations. By virtue of our multifunctional butterfly units, the polynomial MAC operations can be performed in the following procedure.

1) Set $sel2 = 1, sel1 = 0, c = 0$, and perform the constant multiplication before NTT.
2) Set $sel2 = 0, sel1 = 1$, and perform NTT in DIF mode.
3) Set $sel2 = 1, sel1 = 0$, and perform the MAC operations by setting c to previous partial product.
4) Iterate processes $1-3$ until the accumulations complete.
5) Set $sel2 = 0, sel1 = 0$, and perform INTT in DIT mode.
6) Set $sel2 = 1, sel1 = 0, c = 0$, and perform the constant multiplication after INTT.

The parallel processing of MAC operations is quite straightfoward. However, the permutation pattern of data is different in each stage of the NTT butterfly network, so additional works should be done. To handle this issue, we apply the methods proposed in [18] and implement permutation networks placed before and after the vector butterfly units. Taking the 40-bit q in our RNS system as an example, we first implement a two-port RAM with a depth of 512 and a data width of 320 bit to store a complete polynomial. Then eight data are read from each port, and the total sixteen data are permutated before they are sent to the butterfly units. When the butterfly operations are finished, the processed sixteen data are permutated again before written to the RAM.

As Fig. 4 shows, (a_0, a_2), (a_1, a_3) are pairs of operands in stage 1 of the DIT butterfly network while (a_0, a_4), (a_1, a_5) form pairs in stage 2. There are two types of permutations, which are discussed below.

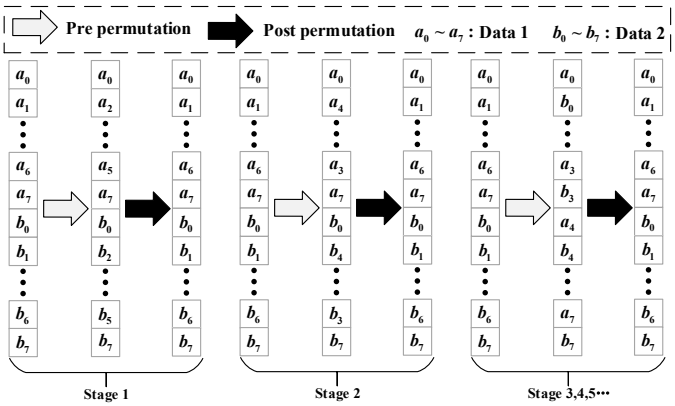

Fig. 4. Permutation patterns of each stage in the DIT butterfly network.

1) When stage $i < log8 = 3$, the permutations are within one data line read from RAM, and the two read addresses of the data RAM are adjacent. We also notice that permutation is not needed in stage 0.
2) When stage $i \geq log8 = 3$, the permutation pattern is fixed, and the corresponding data between the two data lines are paired together. When stage $i > log8 = 3$, two read addresses of RAM are no longer adjacent to match the patterns in the butterfly network.

Our vector butterfly units are fully pipelined, and the data access pattern is fixed and has no issues of data dependency in NTT/INTT/MAC operations, so the data results are generated every clock cycle after a pipeline filling delay.

V. RESULTS

We evaluate our accelerator for homomorphic computations on ciphertext on Xilinx Zynq UltraScale+ ZCU102 Evaluation Board. Table I shows the detailed area consumptions of the core modules, such as butterfly units and NTT cores. There are few research works about the hardware acceleration for homomorphic computation applying the CKKS scheme, and what we found related to our work till now is [14]. Since the resource sharing method is not applied in [14], for a fair comparison with our NTT cores, we have calculated and listed the overall resource consumptions of NTT, INTT and multiplication modules with eight vector lanes in [14] at the bottom of Table I. As we can see, although the consumptions of the DSP block are the same, our NTT cores save a lot of area resources compared with HEAX [14].

The maximum clock frequency of our accelerator is 150 MHz, and the performance of the operations are shown in Table II. Thanks to the multi-level parallel design, our work achieves around $2.18\times$ acceleration in the NTT/INTT operations compared with [14]. Compared with a single-threaded Intel Xeon(R) Silver 4108 running at 1.80 GHz [14], our work achieves $27.04\times$ and $25.81\times$ accelerations in NTT and INTT operations respectively. We have also evaluated the performance of the matrix-vector product operations, and it achieves a throughput of 18 op/s.

TABLE I
AREA CONSUMPTIONS AND COMPARISONS

This Work	**LUT**	**FF**	**DSP**	**BRAM**
Butterfly Unit (29bit)	645	230	12	0
Butterfly Unit (40bit)	1037	336	15	0
NTT Cores (109bit)	28735	7229	336	0
Total	176012	36600	1344	550
HEAX [14]	**ALM**	**REG**	**DSP**	**BRAM**
NTT/INTT/MUL	95827	263372	336	–
Total	246323	723188	1185	1731

TABLE II
PERFORMANCE OF BASIC OPERATIONS

Operations (op/s)	**CPU [14]**	**HEAX [14]**	**This Work**
NTT	7222	89518	195313
INTT	7568	89518	195313
Homomorphic MAC	–	–	19453
Matrix-Vector Product $(2048, 2048) \times (2048, 1)$	–	–	18

VI. CONCLUSION

In this paper, a multi-level parallel hardware accelerator for homomorphic computation is proposed. It is highly optimized for matrix-vector multiplications existing in machine learning by accelerating polynomial MAC operations in parallel. Compared with previous works, our work achieves better performance in the NTT/INTT operations and saves a lot of area by resource reusings. Compared with software implementations, our work also achieves over an order of magnitude performance improvement.

REFERENCES

[1] D. Joan and R. Vincent, "The design of rijndael: Aes-the advanced encryption standard," in *Information Security and Cryptography*. springer, 2002.
[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.
[3] "Microsoft SEAL (release 3.5)," https://github.com/Microsoft/SEAL, Apr. 2020, microsoft Research, Redmond, WA.
[4] S. Halevi and V. Shoup, "Algorithms in helib," in *CRYPTO*. Springer, 2014, pp. 554–571. [Online]. Available: https://www.iacr.org/archive/crypto2014/86160286/86160286.pdf
[5] W. Dai and B. Sunar, *cuHE: A Homomorphic Encryption Accelerator Library*. Springer International Publishing, 2015.
[6] J. Cathébras, A. Carbon, P. Milder, R. Sirdey, and N. Ventroux, "Data flow oriented hardware design of rns-based polynomial multiplication for she acceleration," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 69–88, 2018.
[7] E. Öztürk, Y. Doröz, E. Savaş, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 3–16, 2016.
[8] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. Cheung, D. Pao, and I. Verbauwhede, "High-speed polynomial multiplication architecture for ring-lwe and she cryptosystems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 1, pp. 157–166, 2014.

[9] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 353–362, 2019.

[10] S. S. Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "Hepcloud: An fpga-based multicore processor for fv somewhat homomorphic function evaluation," *Computers, IEEE Transactions on*, vol. 67, no. 11, pp. 1637–1650, 2018.

[11] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 387–398.

[12] F. Turan, S. S. Roy, and I. Verbauwhede, "Heaws: An accelerator for homomorphic encryption on the amazon aws fpga," *IEEE Transactions on Computers*, 2020.

[13] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[14] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.

[15] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.

[16] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2012, pp. 139–158.

[17] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, "Homomorphic encryption security standard," HomomorphicEncryption.org, Toronto, Canada, Tech. Rep., November 2018.

[18] G. Xin, J. Han, T. Yin, Y. Zhou, J. Yang, X. Cheng, and X. Zeng, "Vpqc: A domain-specific vector processor for post-quantum cryptography based on risc-v architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 8, pp. 2672–2684, 2020.