Poseidon: Practical Homomorphic Encryption Accelerator

Yinghao Yang[‡*], Huaizhi Zhang[*], Shengyu Fan[†], Hang Lu[*‡§], Mingzhe Zhang[†], Xiaowei Li[*‡§]
[*] State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS, Beijing, China.
[†] State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China.
[‡] University of Chinese Academy of Sciences, Beijing, China.
[§] Zhongguancun Laboratory, Beijing, China.
{yangyinghao21b, luhang, lxw}@ict.ac.cn, amazing_z@outlook.com, damionfan@163.com, zhangmingzhe@iie.ac.cn

***Abstract*—With the development of the important solution for privacy computing, the explosion of data size and computing intensity in Fully Homomorphic Encryption (FHE) has brought enormous challenges to the hardware design. In this paper, we propose a practical FHE accelerator - "Poseidon", which focuses on improving the hardware resource and bandwidth consumption. Poseidon supports complex FHE operations like Bootstrapping, Keyswitch, Rotation and so on, under limited FPGA resources. It refines these operations by abstracting five key operators: Modular Addition (MA), Modular Multiplication (MM), Number Theoretic Transformation (NTT), Automorphsim and Shared Barret Reduction (SBT). These operators are combined and reused to implement higher-level FHE operations. To utilize the FPGA resources more efficiently and improve the parallelism, we adopt the radix-based NTT algorithm and propose HFAuto, an optimized automorphism implementation suitable for FPGA. Then, we design the hardware accelerator based on the optimized key operators and HBM to maximize computational efficiency. We evaluate Poseidon with four domain-specific FHE benchmarks on Xilinx Alveo U280 FPGA. Empirical results show that the efficient reuse of the operator cores and on-chip storage enables superior performance compared with the state-of-the-art GPU, FPGA and accelerator ASICs. We highlight the following results: (1) up to 370× speedup over CPU for the basic operations of FHE; (2) up to 1300×/52× speedup over CPU and the FPGA solution for the key operators; (3) up to 10.6×/8.7× speedup over GPU and the ASIC solution for the FHE benchmark.**

I. Introduction

Fully Homomorphic Encryption (FHE) is considered to be one of the most promising privacy-preserving methodologies. With FHE, sensitive data like financial records, medical history, private personal habits and so on [10], [19] can be encrypted by their owner and sent to the third-party service provider for direct processing on the encrypted data. As shown in Fig. 1, the user uploads the encrypted data to the cloud server to fulfill the computations, and then downloads and decrypts the data returned from the server for the result. In the whole process, the private data are "available but invisible" to the service provider, without the risk of privacy leakage.

Corresponding Author: Hang Lu. This work is supported in part by the National Natural Science Foundation of China (Grant No. 62002339 and No. 62172387), in part by the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDB44030200), and in part by the Youth Innovation Promotion Association CAS (Grant No. 2021098). This work is also sponsored by CCF-Huawei Populus Grove Fund.

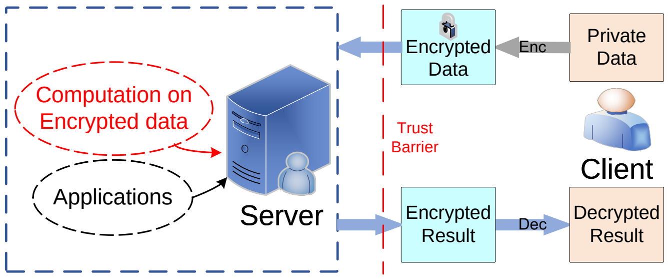

Fig. 1. General concept of FHE.

The development of FHE has been rapid and abundant in recent years. In 2009, Gentry [18] proposed the first FHE scheme. It supports any number of homomorphic additions and multiplications. Several classic FHE schemes have also been proposed to improve Gentry's work, such as BFV [17], BGV [6], TFHE [12], and CKKS [11]. TFHE is an implementation based on the Boolean circuit, while BGV, BFV, and CKKS are based on the arithmetic circuits. The concept of homomorphic calculation is also different for these schemes: BGV and BFV aim at the exactly accurate decrypted result compared with the plaintext result, while CKKS supports approximate computing that allows slight precision loss of the result. Although a collection of FHE algorithms has been proposed, there is still a big gap between the theory and its practical deployment. Even with the highly optimized FHE library, the FHE applications still run 10,000× to 100,000× slower than the unencrypted applications on the CPU. There are two main reasons for such slowdown: (1) Ciphertext size is greatly inflated compared to the plaintext due to the FHE algorithm itself, which causes frequent data movements that further exacerbates the "memory wall" problem; (2) just because of the ciphertext inflation, the computation intensity is also several orders of magnitude larger than the plaintext.

Landscape of Prior Work. In order to narrow the gap mentioned above, recent researches have been devoted to design domain-specific architectures to offload the key operator computation from CPU, and they can be categorized into three types according to the employed hardware platform:

(1) **GPUs**. Some researches in the literature [4], [5], [21], [22], [39], [40] test multiple FHE schemes on different GPU platforms to explore the most matched architecture on the premise of the abundant bandwidth already provided by high-end GPUs. Jung *et al* [21] leverages a single NVIDIA Tesla V100 and achieves up to 257× speedup compared with CPU. Tan *et al*. [37] introduces a new interface for embedding the cryptographic operations into floating-point operations as the "GPU-friendly" cryptographic protocols. Compared to the CPU, it achieves up to 150× speedup. Some open-source libraries [14], [31], [38] are also proposed for the agile FHE software development. As a general-purpose architecture, GPU is not specially designed for FHE. It lacks key operator implementation in the computing cores. Besides, compared with other domain-specific architectures like FPGA and ASIC, large power consumption is also the major weakness of GPU.

(2) **FPGAs**. As a customizable-computing device, the FHE accelerators using FPGA mostly target the highly efficient implementation of the specific FHE operators [7]–[9], [15], [25], [26], [41], such as Number Theoretic Transform (*NTT* hereafter), Inverse Number Theoretic Transform (*INTT* hereafter), Modular Multiplication (*ModMult* hereafter) and Modular Addition (*ModAdd* hereafter) etc. For example, Kim *et al*. [25], [26] targets *ModMult* and *NTT* and achieves up to 118× speedup. Cao *et al*. [7] applies the algorithm in [20] to accelerate the *ModMult* on large numbers. Other works focus on implementing the FHE high-level operators or the complete FHE algorithm on FPGA. Roy *et al*. [33], [34] designed the BFV accelerator based on the multi-core architecture. Y. Doröz *et al*. [16] aims at the fast decryption/encryption operations. HEAX [32], as the most state-of-the-art FPGA accelerator for FHE, proposes a fully pipelined micro-architecture and reports 200× speedup to CPU. FHE applications are both computational and memory intensive. FPGA device however, is limited by the programmable resources and data movement bandwidth. Therefore, the design challenge lies on the efficient reuse of the resources instead of blindly boosting the parallelism, which is ignored by these works.

(3) **ASICs**. F1 [35] is the first ASIC-based FHE accelerator that is programmable and capable of executing complete CKKS-alike FHE algorithm. F1 outperforms the state-of-the-art software implementation by up to 17,000×. Based on F1, CraterLake [36] enables the unbounded multiplicative depth by supporting bootstrapping in hardware and outperforms F1 by 11.2×. BTS [24] further explores the challenges of bootstrapping and selects the proper parameter sets by analyzing the bandwidth and computational requirements. ARK [23] leverages software/hardware codesign to eliminate the off-chip bandwidth bottleneck by the runtime data generation and inter-operation key reuse. Benefited from the flexible resource usage and relatively more freedom in the design phase, the ASIC-based FHE accelerators demonstrate better performance than GPUs and FPGAs. However, current prototypes in the literature still stay in the simulation phase, with impractical assumptions like unattainable bandwidth provided and hundreds of megabytes on-chip storage [23], [24], [36].

In this paper, we propose a practical FPGA-based FHE accelerator — Poseidon. Apart from just piling up the resources for the highest performance, Poseidon aims at the maximal efficiency under the constraint of limited resources. Upon our observation, there are plenty of resource reuse opportunities in the basic FHE operations. Poseidon fully decomposes these basic operations into more fine-grained individuals which we call the 'operators' in this paper. By recomposing and time-multiplexing these operators, it achieves the most efficient utilization of the FPGA resources and the best balance between the computation parallelism and bandwidth demand. The contributions of this paper are listed as follows:

- *We propose a practical FPGA-based FHE accelerator – Poseidon.* It implements all the basic operations of the CKKS-alike FHE algorithm, including *homomorphic addition*, *homomorphic mulitplication*, *rescale*, *rotation*, *keyswitch*, *modup/down* and even the expensive *bootstrapping*. Widely supported operations enable Poseidon applicable for more complex FHE applications that require a larger multiplicative depth or tough security level.
- *We propose an FHE-specific micro-architectural design paradigm based on operator reuse.* We decompose the aforementioned basic operations into more fine-grained operators, and instantiate the hardware compute unit for each operator. In particular, we implement a hardware-friendly automorphism operator called HFAuto. It converts the data movement on the super-long vector into the swap operation of the sub-vectors in an arbitrary granularity.
- *We implement the overall Poseidon prototype on commercial FPGA platform, Xilinx Alveo U280, to evaluate its performance in practice.* We highlight the following results: 1300× and 52× operator speedup compared with Xeon CPU and GPU respectively; on par with the ASIC-based accelerator like F1.

II. Background and Motivation

A. FHE Basic Operations

The FHE algorithm comprises of a series of "basic operations". Taking CKKS as the example, it mainly consists of the following operations:

① **Homomorphic Addition**: there are two cases for the homomorphic addtion (HAdd for short): ciphertext-plaintext addition and ciphertext-ciphertext addition. According to CKKS, the addition result of the ciphertext $ct_0 = (ct_{0,0}, ct_{0,1})$ and plaintext m is $(ct_{0,0} + m, ct_{0,1})$; the addition result of two ciphertexts $ct_0 = (ct_{0,0}, ct_{0,1})$ and $ct_1 = (ct_{1,0}, ct_{1,1})$ is $(ct_{0,0} + ct_{1,0}, ct_{0,1} + ct_{1,1})$, in which the operator "+" represents the element-wise addition of the two polynomials. Since FHE is performed on the "polynomial ring", the result of the ciphertext addition equals to its modular reduction. For example, $ct_{0,0} + ct_{1,0} = (ct_{0,0}[i] + ct_{1,0}[i])(mod\ q), (ct_{0,0}, ct_{1,0} \in \mathbb{R}_q)$, where q is the modulus.

Operators: it is obvious that HAdd involves two steps, element-wise addition and modular reduction, so we abstract the first operator as **ModAdd** (**MA** for short). MA is the basis for more complex operations as follows.

② **Homomorphic Multiplication with Relinearization**: Similar to HAdd, homomorphic multiplication also contains two types, ciphertext-plaintext multiplication denoted as PMult, and ciphertext-ciphertext multiplication denoted as CMult. The ciphertext and plaintext are both polynomials. PMult directly multiplies ct_0 and the plaintext m to get the result $(ct_{0,0} * m, ct_{0,1} * m)$. However, computing $CMult(ct_0, ct_1)$ requires $\widetilde{ct} = (d_0, d_1, d_2) \bmod q = (ct_{0,0} \cdot ct_{1,0}, ct_{0,0} \cdot ct_{1,1} + ct_{0,1} \cdot ct_{1,0}, ct_{0,1} \cdot ct_{1,1}) \bmod q$. The final result is $ct = (d_0, d_1) + p^{-1} \cdot d_2 \cdot rlk)$, in which rlk is the relinearization key represented as $rlk = (b,a) \in \mathbb{R}^2_{pq} = (-a \cdot s + e + p \cdot s^2, a) \bmod p \cdot q$, where s is the secret key and p is a special integer that relies on the rlk setting. The corresponding p^{-1} is the inverse of p in the modulus pq, represented as $p^{-1} \cdot p \equiv 1 (mod pq)$.

Operators: PMult and CMult involve plenty of polynomial multiplications, i.e. $ct_{0,0} \cdot ct_{1,0}$ or $-a \cdot s$. The coefficients of each polynomial should be firstly transformed to the "point-value" representation through NTT before multiplication. The point values then perform element-wise multiplication and modular reduction. The result is then converted back to the coefficient representation by INTT. We hence abstract three lower-level operators: **NTT/INTT**, **ModMult** (**MM** for short), and **MA** (same as HAdd).

③ **Rescale**: this operation scales down the PMult or CMult result by the scaling factor Δ. The process can not be achieved by simple division in the RNS-based FHE schemes, because the large integers, including the modulus and polynomial coefficients, have already been decomposed into multiple small-integers, a.k.a., the RNS components, by the Chinese Remainder Theorem.

Therefore, the RNS-based Rescale operation is described formally as $ct = (ct^{(j)} = (c_0^{(j)}, c_1^{(j)})_{0 \le j \le l})$ into $ct' = (ct'^{(j)} = (c_0'^{(j)}, c_1'^{(j)})_{0 \le j \le l-1})$, where $c_r'^{(j)} = q_l^{-1} \cdot (c_r^{(j)} - c_r^{(l)}) \bmod q_j$ for $r = 0, 1$ and $0 \le j \le l-1$. The $ct^{(j)}$ is one of the RNS components of a ciphertext, and q_l^{-1} is the inverse of q_l under q_j, which satisfies $q_l^{-1} \cdot q_l \bmod q_j = 1$.

Operators: the Rescale operation is fulfilled by a series of ModAdd and ModMult. Therefore, it contains three lower-level operators as well: **MA**, **MM**, and **NTT/INTT**.

④ **Keyswitch**: This operation leverages an extra key called the *keyswitch keys* on the ciphertext to make it decryptable by the original secret key. In classic procedures [29], there are three sub-operations in Keyswitch: *RNSconv*, *Modup*, *Moddown*, formalized as Eq. 1-3. *RNSconv* aims to transform a_b into a_c and the footnotes represent the different RNS bases. *Modup* extends the b-based a_b to the RNS bases b and c via the RNSconv operation. In contrast, *Moddown* is responsible for reducing the RNS base of the data.

$$RNSconv(a_b, a_c):$$

$$a_c \longleftarrow (\sum_{j=0}^{l-1} [a_b^{(j)} * \hat{q}_j^{-1}]_{q_j} \; * \hat{q}_j \; mod \; p_i)_{i=0,1\cdots,k-1}, \quad (1)$$

Algorithm 1: Primitive NTT

```
Input: Polynomial $a \in \mathbb{R}_q$ of degree $n-1$, $n$-th
       primitive roots $w_n \in \mathbb{Z}_q$, modulus $M$
Result: Polynomial $a_{ntt} \in \mathbb{R}_q$
1  $l = \log_2(n)$;
2  for $r \longleftarrow 1, \cdots, l$ do
3      $mid = 2^{r-1}$;
4      for $i \longleftarrow 0, 2^r, 2^{r+1} \cdots, n-2^r$ do
5          $temp = 1$;
6          for $j \longleftarrow 0, 1, \cdots, mid-1$ do
7              $u = x[i+j]$;
8              $v = x[i+j+mid] \; * \; temp \; \% \; M$;
9              $x[i+j] = (u+v) \; \% \; M$;
10             $x[i+j+mid] = (u-v) \; \% \; M$;
11             $temp = temp \; * \; w_n^{n/2^r} \; \% \; M$;
12         end
13     end
14 end
```

Moddown(a_b, b_c):

$$(a_b, b_c) \longrightarrow (b_c - RNSconv(a_b, a_c)) * [P^{-1}]c, \quad (2)$$

$$Modup(a_b): \quad a_b \longrightarrow (a_b, \; RNSconv(a_b, a_c)). \quad (3)$$

Operators: from the three sub-operations, we can see that each of them is also composed of a series of MA and MM. NTT/INTT is also needed for the point-value transformation, so we abstract the operators similarly to the Rescale as: **MA**, **MM**, **NTT/INTT**.

⑤ **Rotation**: this operation is provided for rotating the ciphertext, which is very commonly used in some AI-related privacy computing, i.e. federated learning. Apart from the plaintext that the rotation can be directly issued, directly rotating the ciphertext violates the correctness of the plaintext. Therefore, the Rotation operation includes two sub-operations: it firstly establishes the "index mapping" relationship between the current and new ciphertexts; then, it issues the Keyswitch for the correct decryption using the original secret key.

Operators: the Keyswitch operation in Rotation is identical as previously specified. The index mapping operation maps the current index i of the ciphertext polynomial to $i \cdot k \bmod N$ as shown in Eq. 4 where k is the parameter associated with the Rotation step, N is the polynomial length, and sgn is the sign under the two different modulo results. Every time the Rotation is invoked, the mapping operation must be issued. Therefore, We abstract this computation as a new lower-level operator termed "**Automorphism**".

$$sgn = \begin{cases} -1 & if \;\; i \cdot k \; mod \; 2N > N, \\ 1 & if \;\; i \cdot k \; mod \; 2N < N. \end{cases} \quad (4)$$

Intuitively, this operator does not cost large computation resources because Eq. 4 only entails one modular reduction

TABLE I
OPERATOR REUSE IN BASIC FHE OPERATIONS.

	MA	MM	NTT	Automorphism	SBT
Modup	✓	✓			✓
Moddown	✓	✓			✓
HAdd	✓				
PMult	✓	✓	✓		✓
CMult	✓	✓	✓		✓
Rotation	✓	✓	✓	✓	✓
Keyswitch	✓	✓	✓		✓
Rescale	✓	✓	✓		✓
Bootstrapping	✓	✓	✓	✓	✓

in terms of the current index i. However, the degree of the polynomial (N) usually ranges from 2^{12} to 2^{17}, which means mapping to the new index will be issued many times. Worsestill, conventional automorphism computation cannot be vectorized, because the new index does not comply with a fixed pattern. It will result in conflicts between different indices for the in-situ storage. Section III-B will elaborate our proposed HFAuto to resolve this problem.

⑥ **Bootstrapping**: for the FHE application that requires deeper multiplicative depth, this operation issues the decryption using the encrypted secret key to refresh the ciphertext to a lower noise level. `Bootstrapping` is the most complex operation in FHE, and we employ the state-of-the-art packed `Bootstrapping` algorithm [30] that contains a combination of `PMult`, `CMult`, `HAdd`, `Keyswitch`, and `Rescale` operations.

Operators: since `Bootstrapping` is comprised of many other basic operations, the operators abstracted also coincide with them, including, **MA**, **MM**, **NTT/INTT**, and **Automorphism**. The complexity stems from the highly frequent reuse of these operators to fulfill the whole `Bootstrapping` operation, which is just the objective Poseidon targets.

The abstracted operators all involve the modular reduction; for example, NTT/INTT takes the modulo of the point values as the final step; autormorphism uses modular reduction to compute the new index mapping. Therefore, modular reduction can also be reused among the abstracted operators. We instantiate the modular reduction alone as an individual operator, termed as "Shared Barrett Reduction (**SBT** for short)". TABLE I summarizes the operator reuse in Poseidon for different basic operations of FHE.

B. Acceleration Opportunities

(1) Resource Reuse. Simply boosting the computation parallelism is surely able to improve the FHE performance. However, the increased resources also requires a matched bandwidth to support high-speed computation. Especially for FHE, the ciphertext computation consumes an extremely large bandwidth. The optimal solution should balance the two sides through the resource reuse mechanism to bridge the gap between frequent data movement and the limited bandwidth provided, instead of blindly piling up the resources.

If we decompose the basic operations into lower-level operators, different operations might have many overlaps. For example, `HAdd` and `PMult` both involve MA; `Keyswitch` and `Bootstrapping` both involve NTT/INTT and automorphism. It is hence unnecessary to implement each basic operation in the accelerator. By time-multiplexing and dynamically combining different operators at runtime, the higher-level FHE operations could also be achieved but in a hardware-efficient manner.

(2) Operator Optimization. The complexity of the operators described in Section II-A are also different. For example, MA only requires a simple comparison of the ciphertext (as will be shown in Figure 3), while NTT on the contrary is very computationally intensive; Automorphism aims at the data mapping on the super-long vector and is difficult to process in parallel. The diversity of the operators makes the pipeline critical path latency hard to harness as well.

However, there are also opportunities to optimize the time-consuming operators. For example in NTT/INTT, one of the important operands - "twiddle factor" has exponential characteristics and satisfies the exponential operation rules. By fusing multiple twiddle factors, expensive modular arithmetic could be decreased. This procedure is called "NTT-fusion" in Poseidon. Similarly in Automorphism, by dividing a long vector into sub-vectors, the data mapping will be transformed into the mapping between sub-vectors, which is beneficial to increase the computation parallelism.

We will start detailing these methodologies in the next section.

III. METHODOLOGY

As two most expensive operators in CKKS-alike FHE, NTT and Automorphism entail a large amount of computation and data movement. The computation pattern also influences the on-chip data accesses in the consecutive pipeline stages and the critical path latency. This section describes the optimization method targeting the two key operators.

A. NTT-fusion

The basic computation pattern of NTT includes a series of recursive "Twiddle, Accumulation, and Modulo" (**TAM** hereafter) operations, i.e. $(a_1 + a_2 \cdot w_1) \bmod q$, where q is the modulus; a_1 and a_2 are two coefficients of a high-degree polynomial; w_1 is the twiddle factor. Modular reduction (denoted by 'mod') is very costly, so we adopt the radix-based ($radix = 2^k$) FFT idea [13] to optimize the NTT operator. k is the radix, and if we set k to 3, the three-phase TAM with 24 modular reductions in the conventional NTT transforms into one-phase "fused TAM" with only 8 modular reductions [13]. However, it also brings some additional overhead because of the increase in the twiddle factors. As shown in TABLE II, the number of twiddle factors (W) increases substantially compared with the conventional unfused NTT, which further leads to the increase of MA and MM. Therefore, the selection of k implies a potential tradeoff between the reduced modular reductions and the increased overhead introduced by MA and MM. We will take an in-depth analysis for the optimal k in Section V.

TABLE II
COMPARISON BETWEEN CONVENTIONAL NTT AND NTT-FUSION. W IS THE NUMBER OF TWIDDLE FACTORS; K IS THE RADIX.

k	W (unfused)	W (fused)	Mult/Add (unfused)	Mult/Add (fused)
2	2	2	8 / 8	12 / 12
3	4	5	24 / 24	56 / 56
4	8	13	64 / 64	240 / 240
5	16	34	160 / 160	992 / 992
6	32	85	384 / 384	4160 / 4160

B. HFAuto

Lemma. If $X = \lfloor \{a \bmod (C * R)\}/C \rfloor$, where a, C, and R are positive integers. Then, $X = \lfloor a/C \rfloor \bmod R$.

Proof. Given a positive integer q and any integer n, there is $n = k \cdot q + s$, where k, s are integers and $0 \le s < q$. We have $a = k_1 \cdot C + s_1$, where $k_1 = k_2 \cdot R + s_2$, $0 \le s_1 < C$ and $0 \le s_2 < R$. Thus, substituting k_1 into a, we have $a = k_2 \cdot R \cdot C + s_2 \cdot C + s_1$. Then, substituting a into X, we have $X = \lfloor \{k_2 \cdot R \cdot C + s_2 \cdot C + s_1\} \bmod (C \cdot R)/C \rfloor = \lfloor a/C \rfloor \bmod R$. *QED.*

Automorphism introduced in Section II imposes significant difficulties for the hardware design due to the large scale of polynomial degree N and the N-based coordinate mapping. Therefore, we segment the N-element vector into several sub-vectors to reduce the mapping domain from N to N/C, where C is the length of each sub-vector after segmentation ($C = 512$ in our implementation). We use $R = N/C$ to denote the number of the segments, and i, j to denote the original coordinates, i.e., $index = i \cdot C + j$. Thus, the automorphism formula $index = (index \cdot k) \bmod N$ is changed into $index = \{(i \cdot C + j) \cdot k\} \bmod N$. We use (I, J) to represent this new $index$, where $I = \lfloor \{(i \cdot C + j) \cdot k\} \bmod N \ / \ C \rfloor$, and $J = \{(i \cdot C + j) \cdot k\} \bmod N \bmod C$. Since $N = C \cdot R$, we can obtain $I = \lfloor \{(i \cdot C + j) \cdot k\} \bmod (C \cdot R)/C \rfloor$, $J = \{(i \cdot C + j) \cdot k\} \bmod C = (j \cdot k) \bmod C$. Referring to the previous lemma, we have $I = \lfloor \{(i \cdot C + j) \cdot k\} \bmod (C \cdot R)/C \rfloor = \{(i \cdot k) + \lfloor j \cdot k/C \rfloor\} \bmod R$. In this way, the mapping of I-coordinates in automorphism can be achieved by the *row* mapping twice, i.e., Stage ❶ and Stage ❷ introduced in Section IV corresponding to $(i \cdot k) \bmod R$ and $\lfloor j \cdot k/C \rfloor \bmod R$, respectively. Similarly, the mapping of J-coordinate can be achieved by the *column* mapping detailed in Stage ❸.

The benefit of this optimization is twofold. Firstly, it turns the original automorphism that requires the element-by-element mapping on the entire vector into the cost-effective mapping between multiple small sub-vectors. By expanding the mapping granularity (from 1 to C), an increased parallelism is attained. Secondly, the parallel read/write of sub-vectors simplifies the original data path, which greatly boosts the performance and design flexibility as well.

IV. FHE ACCELERATOR - POSEIDON

A. Overall Architecture

The overall architecture of Poseidon is shown in Fig. 2. It primarily includes the memory system including the scratchpad and high bandwidth memory (HBM hereafter), and the computing cores including MM, MA, NTT, Automorphism

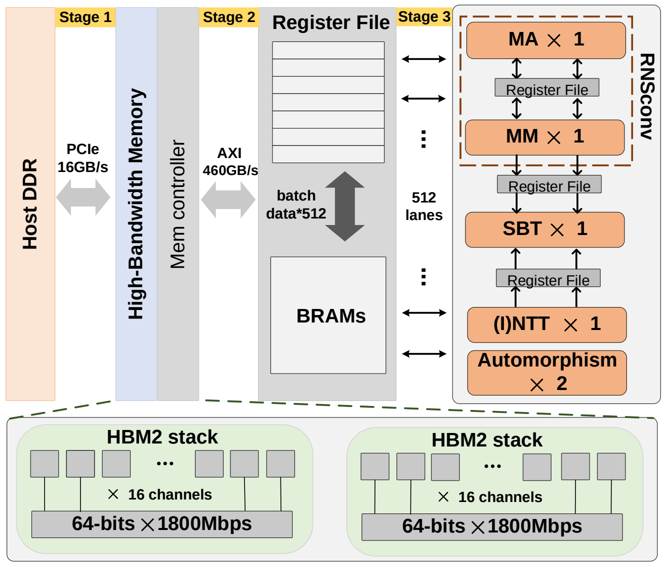

Fig. 2. Poseidon overall architecture.

and SBT. To utilize FPGA resources efficiently, we share the SBT cores with NTT and MM cores. In addition, given the specialty of *Modup* and *Moddown*, we cascade MA and MM cores to implement *RNSconv*, instead of instantiating additional cores separately.

Poseidon decomposes all higher-level operations into basic operator-level computations to maximize parallelism and improve versatility. The size of the vector lane is set to $C = 512$ as shown in the figure, which matches the data width of the scratchpad cluster. The scratchpad is directly connected to the HBM2 memory controller, and the data are categorized in the scratchpad and provided to the designated cores afterward. Since all cores are fully pipelined, the throughput will possibly attain $C = 512$ data/cycle in theory. We use the RNS-based FHE scheme to limit the data width to 32 bits, thereby avoiding the hardware overhead of dealing with super large-width data arithmetic. CraterLake [36] verifies that the data width as minimum as 28 can just obtain sufficient prime moduli under the modulo-chain length $L = 60$. We choose the slightly larger 32 bits to normalize the HBM accesses and simultaneously support deeper multiplication depth.

Memory System. The Poseidon on-FPGA memory system mainly comprises register files and BRAMs. It also leverages the abundant off-FPGA bandwidth provided by HBM to accelerate the data movement. The HBM architecture involves two HBM2 stacks, with each stack having 16 channels. Each channel is 64-bit width with the bit rate of up to 1800 Mbps, which provides a theoretical bandwidth of 460 GB/s. The overall data flow is organized as follows: in Stage ❶, the data are loaded from the host DDR to the HBM via the PCIe interface; in Stage ❷, HBM transfers the data to the register file and BRAM directly connected to the cores; the cores obtain the data from the scratchpad to accomplish computation and write back the data in Stage ❸. A polynomial vector can be segmented by the number of HBM channels, and we can abstract the multi-channel HBM into a vector memory

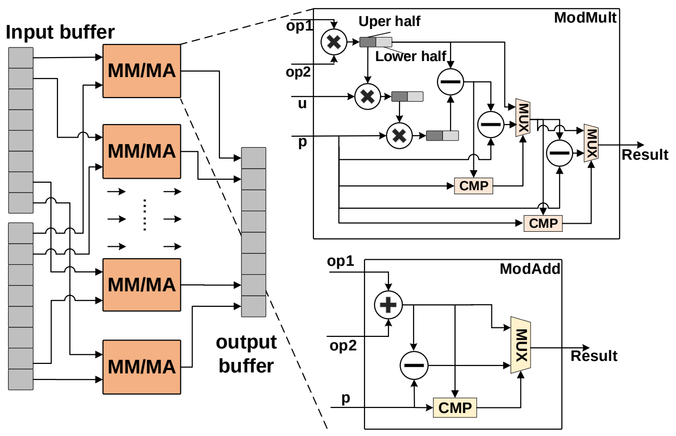

Fig. 3. MA/MM core architecture. We implement fine-grained decomposition to reduce the resource consumption of ModMult. Following Barrett Reduction algorithm, we implement a subtractor to perform ModAdd.

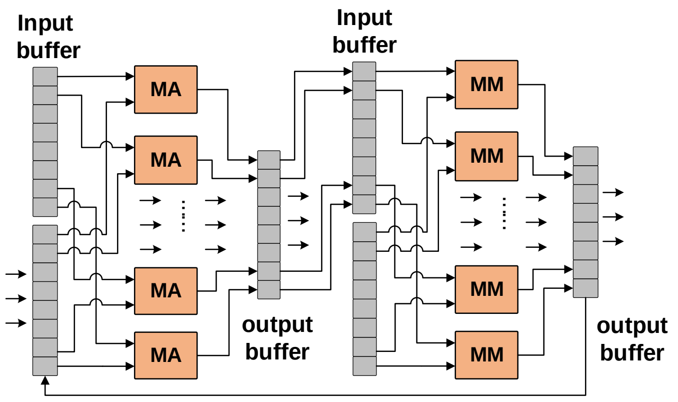

Fig. 4. RNSconv architecture, designed by cascading MA and MM core.

with multiple read/write ports and no access conflicts. The parallelism of the on-chip storage and computational cores is also designed in accordance with the HBM channels, thereby unifying the data management in the accelerator. In Section V, we employ Xilinx Alveo U280 [42] to provide the HBM support for Poseidon.

B. Computational Cores

(1) MA and MM. The ModAdd operation aims to add two polynomials element-wise, and then obtain the modulo result. Since each input polynomial has already performed modular reduction, its value is no larger than the modulus q. Therefore, ModAdd can be accomplished by comparing with q and subtracting q if the addition result is greater than q. As shown in Fig. 3, two conditions are handled: the result is less than or greater than q, following Eq. 5:

$$(a+b)\ mod\ q = \begin{cases} a+b, & a+b<q, \\ a+b-q, & a+b \geq q. \end{cases} \tag{5}$$

Compared with ModAdd, ModMult of two ciphertexts is more complicated, following Eq. 6:

$$\begin{cases} res_{a+b} = a*b, \\ p = res_{mult}/q, \\ res_{multmod} = res_{mult} - (p*q). \end{cases} \tag{6}$$

Given two polynomials a and b, multiplying a with b obtains the intermediate result denoted as $res_{mult} = a*b$. The result will be divided by the modulus q to obtain the quotient p. res_{mult} subtracted by $p*q$ is the final ModMult result. Fig. 3 shows the hardware implementation of MM. Since implementing division using FPGA is very expensive, we employ Barrett Reduction [20] for the division result. By introducing the middle term u to transform division into multiplication and shifting, the upper and lower half hence represents the $k+1$ and $k-1$ bits for a datum with $2k$-bit width; thus, Barrett Reduction avoids the high overhead in ModMult.

(2) RNSconv. It is used to accelerate the `Keyswitch` operation introduced in Section II, shared by *Modup* and *Moddown* operations. *Modup* comprises the vector-scalar multiplication and element-wise accumulation, while *Moddown* is the combination of the vector subtraction and vector-scalar multiplication. Our design does not include the vector-scalar cores, but the same functionality can be accomplished using MA and MM cores. In specific, *RNSconv* is implemented by cascading the MM and MA cores with additional routing and control logic. The hardware design details are shown in Fig. 4. It has two data routes. The first one is the MA core that fetches two groups of data from the input buffer and takes the result as the input of the MM core to complete *Modup*. The another one is the MM core that takes the result as the input of the MA core to complete *Moddown*.

(3) NTT/INTT. As the most complex and time-consuming operator, NTT/INTT core directly determines the accelerator performance. Our design adopts the “NTT-fusion” concept introduced in Section III to decrease the modular reductions and iterations in conventional NTT. We explore the performance in different parameter settings in NTT-fusion and select the appropriate $k = 3$ to achieve optimal performance and resource utilization in tandem. Conventional NTT would require $log_2 8 = 3$ phases with 24 unfused TAMs in total, if taking 8 operands as input. However, only 1 phase with 8 fused TAMs is required in NTT-fusion. For the NTT/INTT core implementation, we simply hardcode the computing circuit in RTL under this setting.

Data Access Pattern. The parallel NTT cores also require the parallelized data input to achieve the best throughput. The BRAMs in FPGA is responsible for the storage of these data including the twiddle factors and the cyphertexts. Hence, BRAM data access efficiency is vital for the efficiency of the NTT cores as well. In Poseidon, NTT-fusion effectively reduces the NTT phases and TAMs. If we set k to 3 and the given polynomial degree is 4096 (2^{12}), for example, the required phases are 4 ($log_2 4096/k$) instead of 12 ($log_2 4096$),

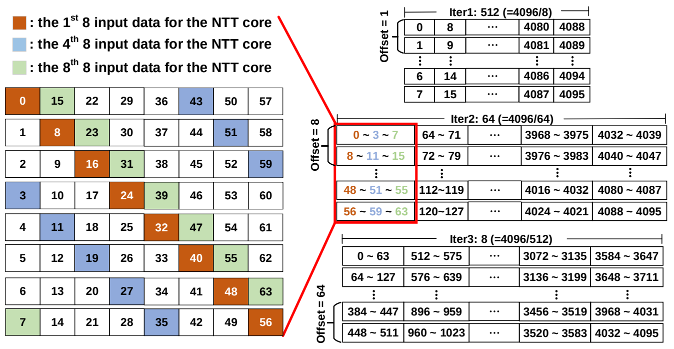

Fig. 5. Data access pattern in Poseidon. We show the front three iterations (or phases) in TABLE III.

TABLE III
DATA ACCESS PATTERN COMPARISON. PRIOR ACCELERATORS FOLLOW THE CONVENTIONAL NTT (12 PHASES FOR 4096 CIPHERTEXT), WHILE POSEIDON ADHERES TO NTT-FUSION (ONLY 4 PHASES). INDEX OFFSET DENOTES THE DATA ACCESS PATTERN.

Prior accelerators			**Poseidon (k = 3)**		
ITERs	TAMs	Index Offset	ITERs	Fused TAMs	Index Offset
1	4096	1	1	4096	1
2	4096	2	/	/	/
3	4096	4	/	/	/
4	4096	8	2	4096	8
5	4096	16	/	/	/
6	4096	32	/	/	/
7	4096	64	3	4096	64
8	4096	128	/	/	/
9	4096	256	/	/	/
10	4096	512	4	4096	512
11	4096	1024	/	/	/
12	4096	2048	/	/	/
Total: 12 phases, 4096*12 TAMs			Total: 4 phases, 4096*4 TAMs		

as compared in TABLE III. If we simply assume each phase is an iteration executed by one NTT core, it requires 12 iterations in conventional NTT while NTT-fusion only requires 4. Specified in TABLE III, if we index 4096 input data, conventional NTT will offset 2^{iter-1} indices to access each input data for each phase, while Poseidon will offset $2^{(iter-1)*k}$. To achieve the highest pipeline efficiency, the target input data should be ready when each phase begins. Fig. 5 presents the data access pattern which takes the first three iterations of TABLE III as the example. In iteration 1, it loads in the operands with index 0$\sim$7, 8$\sim$15, ... 4088$\sim$4095; then, in iteration 2, it does not take in the data sequentially but periodically with the fixed offset (i.e., index 0, 8, 16, 24, 32, 40, 48, 56); in iteration 3, the index offset is 64 because it needs to reorder the index for the next phase.

The NTT core in Poseidon takes 8 operands as input. The BRAM storage of the 8 input operands is also specially manipulated for the best efficiency. Also taking iteration 2 as the example, the required 8 operands (marked in red) are diagonally stored as shown in Fig. 5. This is because each BRAM could only read/write one data in one cycle. Since one core needs to load 8 operands in parallel, the operands are interleaved and stored in 8 different BRAMs. Therefore, the NTT cores can directly load the operands without any delay at the beginning of each phase. In order to match the throughput of other operator cores, we instantiate 64 NTT cores of 8-input which can process 512 data in parallel.

(4) Automorphism. Automorphism essentially performs polynomial coordinate mapping. Its software implementation is relatively simple, while the hardware design is untoward because the mapping domain of each coordinate is the entire polynomial vector. It will inevitably result in two coordinates mapped in different vector lanes. A simple solution is to store the whole polynomial vector in registers; however, the degree of the polynomial may attain 2^{17} and the hardware resources will be burdened. Therefore, we design a hardware-friendly automorphism core based on the method in Section III. Fig. 6 shows the logic of the automorphism core that takes the parameter setting as $C = 512$, $R = 4$, and $k = 3$. According to the simplified mapping formula, automorphism can be divided into four stages. The first two stages implement row mapping, and the last two stages implement column mapping:

Stage ❶: row_i to $row_{(ik\ mod\ R)}$. This stage reads $C = 512$ data per cycle from BRAMs according to the address selection circuit, and write them to FIFO.

Stage ❷: $FIFO_{(i,j)}$ to $FIFO_{(i+jk/C\ mod\ R,j)}$. This stage cyclically shifts the data in each FIFO.

Stage ❸: Switching data dimension. We use a similar idea with the memory access pattern of the NTT core to map the physical row data to the logical rows, and realize the two-dimensional data access on BRAMs.

Stage ❹: $column_i$ to $column_{(ik\ mod\ C)}$. This stage is similar to Stage ❶, where the data in column i will be read and write to the column $ik\ mod\ C$.

V. EVALUATION

A. Experimental Setup

Platform. Poseidon is a practical FHE accelerator, so we build a real-world experimental environment based on the x86 CPU system. The Poseidon accelerator is implemented in the Xilinx Alveo U280 FPGA plugged into the PCIe slot of the mainboard. We installed Xilinx off-the-shelf developing toolkit Vivado [2] and Vitis [1] version 2021.1 on the host side. These tools will be working in conjunction with the Xilinx Runtime [3] environment and OpenCL framework to interact with the Alveo U280 through PCIe. Alveo U280 owns HBM. The intermediate data of the FHE applications are transferred between on-chip storage and the HBM. Only the results are sent back to the host.

Baseline. In our experiments, the CPU (Intel Xeon Gold 6234) running at 3.3 GHz with a single thread is selected as the baseline. Besides, we also compare Poseidon with the state-of-the-art GPU [21], FPGA [25], [26], [32], and 4 FHE accelerator ASICs [23], [24], [35], [36].

Benchmark. We use the following 4 benchmarks (as shown in TABLE V) for evaluation:

(1) Logistic regression (LR). It is the HELR [19] algorithm implementation based on the CKKS scheme. In combination

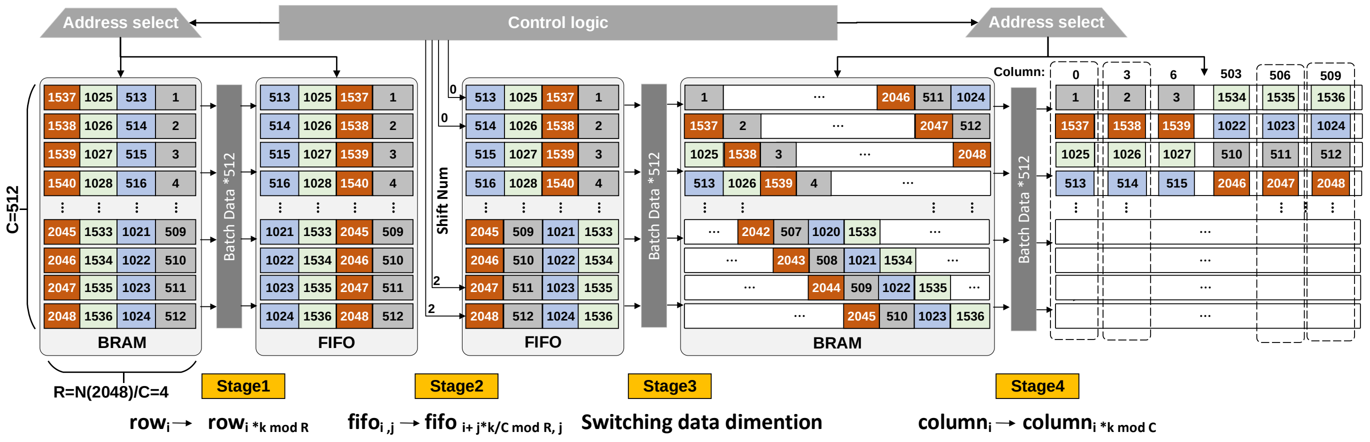

Fig. 6. Automorphism architecture in Poseidon. Only one dual-port BRAM is needed. The result of Stage 4 will be written back to HBM directly.

with `Bootstrapping`, we use the multiplication depth of $L = 38$ and evaluate the average performance of 10 iterations supported by two `Bootstrapping` operations.

(2) LSTM. It is the Long-Term Short-Term (LSTM) model in natural language processing [27] which aims to iterate the formula $y_i + 1 = \sigma(W_0 y_i + W_1 x_1)$ several times. σ is a nonlinear activation function approximated by a cubic polynomial, and W is a matrix with dimensions 128×128. It requires 50 `Bootstrapping` operations in total during one inference.

(3) ResNet-20. This benchmark is the inference of an image on the ResNet-20 model implemented with FHE [28].

(4) Packed bootstrapping. This benchmark adopts the most advanced fully packed bootstrapping algorithm [30]. The high noise-level ciphertext with the multiplication depth $L = 3$ will be refreshed to the low noise-level ciphertext with the multiplication depth $L = 57$.

These benchmarks are common in reality, i.e., "federated learning" used to train/test a machine learning model, "medical image prediction" with privacy using deep neural networks. The performance of the FHE accelerator is imperative to the user experience and quality of service in these scenarios.

B. Accelerator Performance

(1) FHE Basic Operations. The performance of the basic operations in FHE, i.e., `PMult`, `CMult`, `NTT`, `Keyswitch`, `Rescale`, and `Rotation`, directly determines the performance of the accelerator. The accelerator architectural design in return determines the performance of the basic operations. In Poseidon, we implement these operations efficiently through operator reuse (Note that NTT is actually a basic operator, but we take it out separately due to its high complexity). We use "operations per second" as the performance metric. Since our FPGA baseline - HEAX [32] does not report the results under our parameter setting, we estimate the highest performance based on its hardware design for comparison.

As shown in Table IV, Poseidon achieves better acceleration on complex operations, i.e., `PMult` (349×), `NTT` (1348×), `Keyswitch` (780×), and `Rotation` (774×). Compared with the state-of-the-art GPU, Poseidon still achieves several times improvements. Compared with HEAX, the most advanced FPGA-based prototype, Poseidon achieves 3× and 50× speedup on `Keyswitch` and `NTT`, respectively.

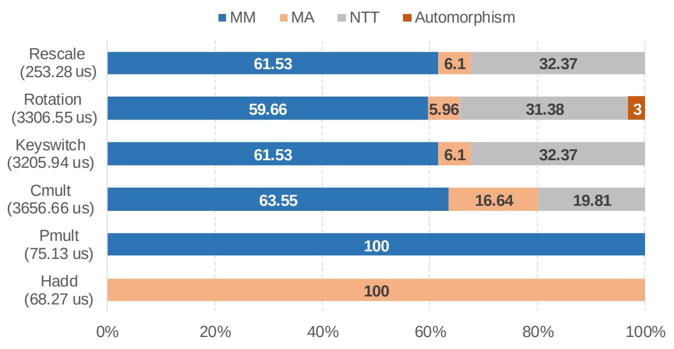

Fig. 7. Operator core analysis. The ciphertext parameters are set to $N = 2^{16}, L = 44$.

TABLE IV
PERFORMANCE COMPARISON OF FHE BASIC OPERATIONS. WE USE "OPERATIONS PER SECOND" AS THE PERFORMANCE METRIC.

	CPU (Xeon)	Over 100x (GPU) [21]	HEAX (FPGA) [32]	Poseidon (FPGA)	speedup
HAdd	35.56	4807	4,161	13,310	**374**×
PMult	38.14	7,407	4,161	13,310	**349**×
CMult	0.38	57	119	273	**718**×
NTT	9.25	/	237	12,474	**1,348**×
Keyswitch	0.4	/	104	312	**780**×
Rotation	0.39	61	/	302	**774**×
Rescale	6.9	1,574	/	3,948	**572**×

Besides, we also analyze the key FHE operators. Fig. 7 clearly shows the basic operations are comprised of the key operators as mentioned before. `HAdd` only involves MA; `PMult` only involves MM, while `Rotation` involves all 4 operators. MM is also the most frequently used operator in `Rescale`, `Rotation`, `Keyswitch` and `CMult`. The analysis also takes the data movement overhead into consideration.

(2) Full-system performance. This experiment evaluates the full-system performance of Poseidon with state-of-the-

TABLE V

FULL-SYSTEM PERFORMANCE COMPARISON WITH SOTA ACCELERATOR PROTOTYPES. WE USE ACTUAL BENCHMARK EXECUTION TIME IN MS AS THE METRIC.

	LR [19]	LSTM [27]	ResNet-20 [28]	Packed Boot-strapping [30]
F1+ (ASIC)	639	2,573	2,693	58.3
CraterLake (ASIC)	119.52	138.0	249.45	3.91
BTS-1 (ASIC)	39.9	/	1,910	/
BTS-2 (ASIC)	28.4	/	2,020	/
BTS-3 (ASIC)	43.5	/	3,090	/
ARK (ASIC)	7.717	/	294	/
over100x (GPU)	775	/	/	/
Poseidon (FPGA)	**72.98**	**1,848.89**	**2,661.23**	**127.45**

TABLE VI

COMPARISON OF THE STORAGE RESOURCE CONSUMPTION.

	HBM	Scratchpad	Running Fre. (GHz)
	Capacity / Bandwidth		
	(GB / TB/s)	(MB / TB/s)	
F1+ [35], [36] (ASIC)	16 / 1	256 / 29	1
CraterLake [36] (ASIC)	16 / 1	256 / 29	1
BTS [24] (ASIC)	16 / 1	512 / 38.4	1.2
ARK [23] (ASIC)	32 / 2	512 / 20	1
Poseidon (FPGA)	**8 / 0.45**	**8.6 / 3.4**	**0.45**

art accelerator prototypes including GPU and ASIC. As is shown in TABLE V, Poseidon performs 10.6× higher speedup than over 100x [21] for the benchmark LR. For the same benchmark, Poseidon's performance is quite close to other accelerator ASICs. For the other three benchmarks, Poseidon's performance is close to and even better than F1+ [35], [36]. However, there is still a large gap between Poseidon and some ASICs like CraterLake [36] and ARK [23]. Frequent data movement will bring significant overhead, so CraterLake and ARK apiece provides 256 MB and 512 MB register files to the computing cores as a scratchpad, which improves the data reuse and reduces the interactions with the off-chip storage. Poseidon does not instantiate such large on-chip storage considering the practical implementation on FPGA and ASIC, and only provides the scratchpad with 8.6 MB capacity and 3.4 TB bandwidth, which is much less than the two accelerators. However, Poseidon still achieves impressive performance due to the efficient hardware design of the operator cores and the resource reuse strategy.

Fig. 8 and Fig. 9 show the execution time breakdown of the benchmarks at two different granularities: the FHE basic operations and the key operators. As can be seen from Fig. 8, `Keyswitch` and `CMult` are the two FHE operations with the highest proportion, accounting for more than 65% across all benchmarks. Fig. 9 further shows that most of the time is consumed by MM and NTT operator cores. The analysis of the two granularities is consistent because the key computations in `Keyswitch` and `CMult` are just MM and NTT. Therefore, we conclude that the performance of FHE depends highly on MM and NTT operators. The operator reuse in Poseidon provides enough flexibility in supporting diverse FHE operations, not only in one particular algorithm but also in a wide range of general-purpose FHE algorithms.

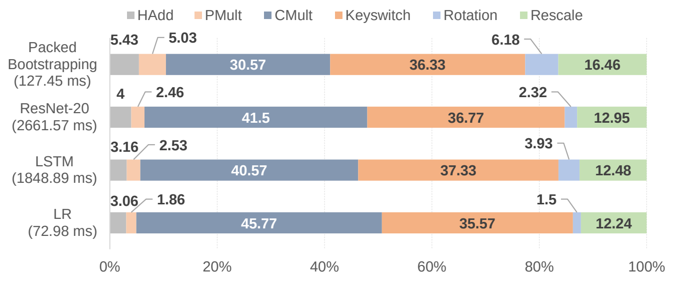

Fig. 8. Analysis of the basic operations in benchmarks. We evaluate the execution time, and Keyswitch and CMult occupy the largest proportion.

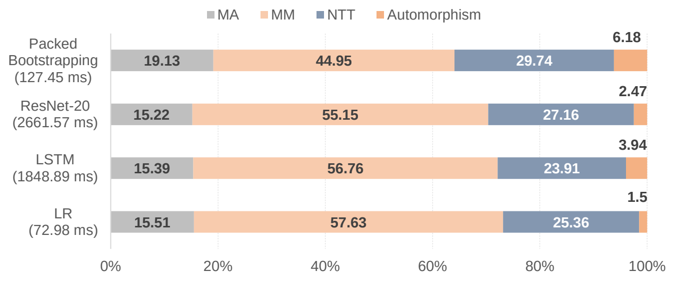

Fig. 9. Execution time analysis of 4 key operators in FHE basic operations. MM and NTT occupy the largest proportion.

C. Bandwidth Utilization

High-performance FHE accelerator requires optimized operator core design, and the performance of the operator cores further requires the support of sufficient memory bandwidth. Poseidon uses the HBM+FPGA architecture, and we provide a detailed evaluation of its bandwidth utilization. TABLE VII shows the lowest bandwidth utilization of each FHE operation. The bandwidth consumed by `HAdd` and `PMult` is significantly higher than other operations due to its relatively simpler computing logic but large-volume data access from HBM. `Rescale` has the lowest bandwidth utilization, because of its frequent reuse of the small-scale data stored in the 8.6 MB scratchpad of Poseidon. In addition, We also evaluate the average bandwidth utilization of the entire benchmark, and obtain nearly 43% $\sim$ 59% utilization. The evaluation concludes that complex operations are not the major bandwidth consumer; instead, simpler operations usually consume larger bandwidth due to the unified and frequent data access.

D. Poseidon Specifics

(1) Parameter Selection—NTT Fusion. In radix-based NTT, parameter k denotes how many TAMs will be fused at a time in Poseidon. This parameter directly affects the NTT computations in that higher k indicates fewer modular reductions. However, a higher k setting also spawns new twiddle factors and extra computations that will undermine the accelerator performance in return. In order to explore the finest setting, we evaluate several metrics scaling with k, including the hardware resource utilization and execution time. As shown in Fig. 10, the four metrics uniformly denote an inflection at nearly $k = 3$, marked as bold. For the hardware metrics like *#Regs*, *#DSPs*,

TABLE VII
LOWEST AND AVERAGE BANDWIDTH UTILIZATION ANALYSIS OF BASIC OPERATIONS AND WHOLE BENCHMARKS.

	LR [19]	**LSTM [27]**	**ResNet-20 [28]**	**Packed Bootstrapping [30]**
HAdd (%)	97.79	97.69	97.76	63.29
PMult (%)	97.65	97.15	97.48	97.48
CMult (%)	44.72	55.55	50.15	72.35
Keyswitch (%)	36.8	47.47	42.05	63.29
Rotation (%)	65	32.39	58.67	48.67
Rescale (%)	26.16	29.98	26.83	26.83
Bootstrapping (%)	46.39	56.43	52.18	/
Average (%)	**42.78**	**51.99**	**48.08**	**59.07**

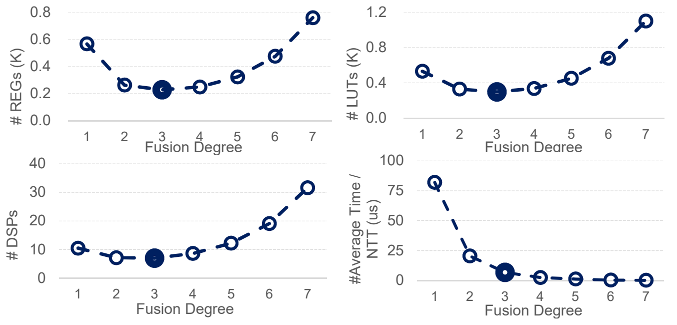

Fig. 10. Parameter Selection – k. We evaluated the FPGA resource usage (in actual #) and the Average Execution Time per NTT (bottom right), scaled by k. The optimal point emerges at $k = 3$, where it consumes the lowest resources with the highest speed.

and #$LUTs$, the $k = 3$ inflection denotes the most optimized resource in FPGA, while for the performance metric NTT (in microseconds), it denotes a relatively shorter execution time.

(2) HFAuto. Differing from other operators, automorphism features a more complex data access pattern. Straightforward hardware implementation can hardly adapt to the fast data access and efficient pipeline design in Poseidon. HFAuto resolves this problem, so this experiment aims to quantify the performance of HFAuto in the automorphism operator cores. As shown in TABLE VIII, the overhead of straightforward design (denoted by Auto) consumes fewer resources than HFAuto because it only processes a single index map in one cycle. However, its latency is much more significant than HFAuto due to the less-optimal parallelism in performing automorphism. Poseidon leverages HFAuto to obtain high performance in exchange with more resource consumption.

We also build an ablation study experiment in terms of different automorphism core designs in Poseidon. As shown in Table IX, compared with Poseidon-HFAuto, the performance of Poseidon-Auto may degrade up to an order of magnitude.

(3) Scalability. Fig. 11 shows the lane scaling from 64, 128, 256 to 512, with respect to the execution time and Energy Delay Product (EDP). We use ResNet-20 as the benchmark to show the trend. Others behave similarly. We can see that the accelerator performance increases in accordance with the lanes. This is reasonable because intuitively the performance should upgrade with the increased parallelism. Meanwhile, the performance growth gradually becomes slower due to the bandwidth limitation. The behavior of EDP is similar to the execution time. Therefore, We choose lanes=512 to maximize the performance and balance the bandwidth consumption in Poseidon.

TABLE VIII
RESOURCE UTILIZATION COMPARISON OF THE AUTOMORPHSIM OPERATOR CORE DESIGN.

	FF	**DSP**	**LUT**	**BRAM**	**Latency (cycles)**
Auto	88	0	0	0	131,073
HFAuto	572	0	25,751	512	517

TABLE IX
HFAUTO PERFORMANCE IN POSEIDON.

	LR	**LSTM**	**ResNet-20**	**Packed Bootstrap-ping**
Poseidon-Auto (ms)	729.8	14,150.2	10,543.1	1,127.2
Poseidon-HFAuto (ms)	72.98 (10×)	1,848.89 (7.6×)	2,661.23 (3.9×)	127.45 (8.8×)

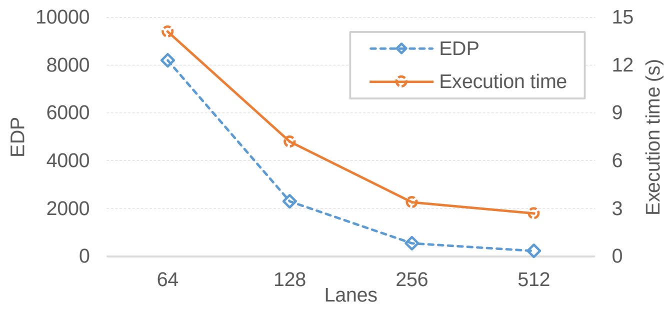

Fig. 11. Sensitivity of the lanes.

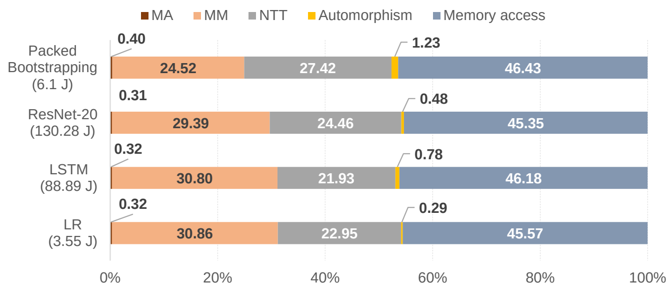

Fig. 12. Energy consumption and breakdown. MM and NTT operator cores take the major proportion besides the memory access.

E. Energy

(1) Energy Consumption and Breakdown.

As shown in Fig. 12, memory access takes up most of the energy consumption. For the operators, as previously proved in Fig. 9, MM and NTT take most of the execution time, which reflects the reason for their large energy consumption. Although MA occupies a certain proportion of the execution time, its energy consumption is minimal due to its simple computing logic.

(2) Energy Efficiency.

TABLE X
EFFICIENCY ANALYSIS. WE USE ENERGY DELAY PRODUCT (EDP) AS THE METRIC. LOWER IS BETTER.

	Over 100x (GPU)	**BTS-2 (ASIC)**	**ARK (ASIC)**	**Crater-Lake (ASIC)**	**Poseidon (FPGA)**
LR	180.19	0.092	0.017	3.028	**0.18**
ResNet-20	/	545.95	24.31	17.08	**236.17**
LSTM	/	/	/	6.04	**113.36**
Packed Bootstrapping	/	/	/	0.0038	**0.51**

TABLE XI
FPGA RESOURCE UTILIZATION OF POSEIDON.

	LUT (k)	**FF (k)**	**DSP**	**BRAM**	**Latency (cycles)**
MA ($\times 1$)	50	68	0	0	3
MM ($\times 1$)	170	160	1536	0	5
NTT ($\times 1$)	358	344	4032	1024	21
Automorphism ($\times 2$)	52	2	0	1024	517
SBT ($\times 1$)	98	403	3072	0	11

We use EDP as the efficiency metric. TABLE X lists the efficiency of Poseidon and the baseline prototypes. For the benchmark LR, Poseidon is $1000\times$ higher than the GPU-based accelerator - over100x. Compared with the accelerator ASICs, Poseidon outperforms CraterLake and BTS for LR and ResNet-20. Since the power consumption of FPGA is usually much higher than the ASIC with advanced technology node, Poseidon demonstrates lower efficiency than the ASICs for other benchmarks.

F. FPGA Resource Utilization

Based on the high-performance HBM+FPGA platform, Poseidon includes five types of operator cores: MA, MM, Automorphism, NTT, and SBT, with a parallelism of 512 lanes. TABLE XI shows the resource consumption of different cores in detail. Poseidon consumes more DSPs because MM, NTT and SBT all need to perform complex arithmetic in FHE-like multiplications. Poseidon also implements the automorphism operator to support `Rotation` operation, which also causes a slight increase in FPGA resources. However, compared with other FPGA-based prototypes - Kim et al. [25], [26] and HEAX [32] in TABLE XII, Poseidon demonstrates less resource consumption due to its optimized hardware implementation.

VI. DISCUSSION

FHE itself is both computational and memory intensive. The performance enhancement relies highly on the optimization on both sides, including the manner in how the costly operators as well as the bandwidth bottleneck are avoided, which in turn requires the meticulous design of the accelerator pipeline and proper key design parameter settings. Poseidon involves three key design parameters: the fusion degree of NTT (denoted by 'k'), the degree of parallelism (512 by default), and the scratchpad volume (8.6 MB for 512 lanes). Each of them implies an implicit design tradeoff.

TABLE XII
RESOURCE UTILIZATION COMPARISON. FOR FAIRNESS, WE COMPARE TWO SELECTED COMPUTATIONS IN FHE - MODMULT AND A SINGLE TAM.

		Kim [25]	**Kim [26]**	**HEAX [32]**	**Poseidon**
Mod Mult	LUT	1988	/	1663	523
	REG	1810	/	4256	2000
	DSP	12	/	22	9
Single TAM	LUT	/	5368	2066	594
	REG	/	4927	6297	973
	DSP	/	19.95	10	9.25

The fusion degree affects the pipeline critical path latency because NTT is the most time-consuming operator, and a larger fusion degree benefits the computation throughput while it also burdens the storage of the twiddle factors. We set k as 3 to balance the hardware overhead and the performance.

Theoretically, the higher the parallelism, the better the acceleration performance for FHE because of its SIMD-like computation pattern. However, in practical implementation, increasing the parallelism cannot always induce a better performance due to the limited bandwidth a particular hardware platform could provide. An optimal design regime should also balance the peak computational resources and the data movement efficiency, and we explored that implementing 512 lanes on our U280 FPGA achieves the best tradeoff.

Apart from prior studies that leverage costly on-chip storage (i.e., 256 MB [35], [36] and 512 MB [23], [24]), Poseidon only uses an 8.6 MB scratchpad to cache the intermediate data, but leverages elaborate dataflow planning with HBM to collaborate with the main pipeline. Poseidon is also fully compatible with other memory-access technologies, i.e., Near Data Processing (NDP). The optimized operator computation in Poseidon enables the deployment of less expensive computational units near the vast storage device, i.e., SmartSSD by Samsung and Xilinx, with even less on-chip storage space for the scratchpad. We hope Poseidon will act as a jumping board for future improvements in increasing FHE performance.

VII. CONCLUSION

In this paper, we present an HBM+FPGA solution for FHE acceleration. We decompose the operators of various FHE operations in detail and realize the acceleration by reusing the key operators - MA, MM, NTT, Automorphism and SBT. In order to make full use of the HBM bandwidth, we use the radix-based NTT algorithm - NTT-fusion, and explore the best selection of its key parameter. Besides, we propose a highly-parallelized automorphism acceleration method - HFAuto that is convenient for the FPGA implementation to achieve a high-performance FHE accelerator. Based on the above methodologies, we propose a practical FHE accelerator - "Poseidon." It provides support for complex FHE operations, including Rotation, Keyswitch, Bootstrapping and so on through efficient hardware design and operator reuse under limited FPGA resources. We evaluate Poseidon on Xilinx Alveo U280 FPGA and prove that it behaves better than state-of-the-art GPU, FPGA and the accelerator ASICs. We hope this work can inspire new ideas for future FHE accelerator design.

REFERENCES

[1] "Vitis rev:2021.1," https://www.xilinx.com/products/design-tools/vitis.html.

[2] "Vivado rev:2021.1," https://www.xilinx.com/products/design-tools/vivado.html.

[3] "Xilinxruntime rev:2021.1," https://www.xilinx.com/products/design-tools/vitis/xrt.html.

[4] S. Akleylek, Ö. Dağdelen, and Z. Yüce Tok, "On the efficiency of polynomial multiplication for lattice-based cryptography on gpus using cuda," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 155–168.

[5] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 70–95, 2018.

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[7] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan, and N. Hanley, "Optimised multiplication architectures for accelerating fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2794–2806, 2015.

[8] X. Cao, C. Moore, M. O'Neill, N. Hanley, and E. O'Sullivan, "High-speed fully homomorphic encryption over the integers," in *International Conference on Financial Cryptography and Data Security*. Springer, 2014, pp. 169–180.

[9] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan, and N. Hanley, "Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction," *Cryptology ePrint Archive*, 2013.

[10] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1243–1255.

[11] 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.

[12] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *international conference on the theory and application of cryptology and information security*. Springer, 2016, pp. 3–33.

[13] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[14] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

[15] Y. Doröz, E. Öztürk, and B. Sunar, "Evaluating the hardware performance of a million-bit multiplier," in *2013 Euromicro Conference on Digital System Design*. IEEE, 2013, pp. 955–962.

[16] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1509–1521, 2014.

[17] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.

[18] 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.

[19] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 9466–9471.

[20] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[21] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.

[22] A. Khedr and G. Gulak, "Homomorphic processing unit (hpu) for accelerating secure computations under homomorphic encryption," May 21 2019, uS Patent 10,298,385.

[23] J. Kim, G. Lee, S. Kim, G. Sohn, J. Kim, M. Rhu, and J. H. Ahn, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," *arXiv preprint arXiv:2205.00922*, 2022.

[24] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 711–725.

[25] S. Kim, K. Lee, W. Cho, J. H. Cheon, and R. A. Rutenbar, "Fpga-based accelerators of fully pipelined modular multipliers for homomorphic encryption," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2019, pp. 1–8.

[26] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. A. Rutenbar, "Hardware architecture of a number theoretic transform for a bootstrappable rns-based homomorphic encryption scheme," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 56–64.

[27] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim *et al.*, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, pp. 30 039–30 054, 2022.

[28] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim *et al.*, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, pp. 30 039–30 054, 2022.

[29] J.-W. Lee, E. Lee, Y. Lee, Y.-S. Kim, and J.-S. No, "High-precision bootstrapping of rns-ckks homomorphic encryption using optimal minimax polynomial approximation and inverse sine function," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021, pp. 618–647.

[30] C. Mouchet, J.-P. Bossuat, J. Troncoso-Pastoriza, and J. Hubaux, "Lattigo: A multiparty homomorphic encryption library in go," in *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2020.

[31] nucyper. Nufhe, a gpu-powered torus fhe implementation. [Online]. Available: https://github.com/nucypher/nufhe

[32] 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.

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

[34] 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.

[35] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[36] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data." in *ISCA*, 2022, pp. 173–187.

[37] S. Tan, B. Knott, Y. Tian, and D. J. Wu, "Cryptgpu: Fast privacy-preserving machine learning on the gpu," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1021–1038.

[38] vernamlab. Cuda-accelerated fully homomorphic encryption library. [Online]. Available: https://github.com/vernamlab/cuFHE

[39] W. Wang, Z. Chen, and X. Huang, "Accelerating leveled fully homomorphic encryption using gpu," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014, pp. 2800–2803.

[40] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 64, no. 3, pp. 698–706, 2013.

[41] W. Wang, X. Huang, N. Emmart, and C. Weems, "Vlsi design of a large-number multiplier for fully homomorphic encryption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 9, pp. 1879–1887, 2013.

[42] Xilinx, "Product brief of smartssd," https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.