FAST:An FHE Accelerator for Scalable-parallelism with Tunable-bit

Shengyu Fan
State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
Beijing, China
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
fanshengyu@iie.ac.cn

Xianglong Deng
State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
Beijing, China
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
dengxianglong@iie.ac.cn

Liang Kong
Ant Group
Beijing, China
kongliang.kong@antgroup.com

Guiming Shi
Tsinghua University
Beijing, China
Ant Group
Beijing, China
guimingthu@163.com

Guang Fan
Ant Group
Beijing, China
fanguang.fg@antgroup.com

Dan Meng
State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
Beijing, China
mengdan@iie.ac.cn

Rui Hou
State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
Beijing, China
hourui@iie.ac.cn

Mingzhe Zhang*
Ant Group
Beijing, China
smartzmz@gmail.com

Abstract

Fully Homomorphic Encryption (FHE) enables direct computation on encrypted data, providing substantial security advantages in cloud-based modern society. However, FHE suffers from significant computational overhead compared to plaintext computation, hindering its adoption in real-world applications. While many accelerators have been designed to address performance bottlenecks, most do not fully leverage cryptographic optimization technologies, leaving room for further performance enhancements.

In this work, we propose FAST, an FHE accelerator incorporating recent cryptographic optimizations, including hoisting technology and the gadget decomposition key-switching method (named KLSS method). We analyze ciphertext level consumption throughout application execution and observe that workload requirements vary significantly with different ciphertext levels for both hybrid and KLSS key-switching methods. Additionally, we note the differing computational precision requirements for these key-switching methods. Based on these observations, we designed a versatile framework that supports multiple key-switching methods during a single application execution and integrates hoisting technology. Furthermore, we developed a scalable, precision-tunable multiplier to accommodate the needs of hybrid and KLSS key-switching methods. FAST architecture features a specialized multiplier and novel data organization to exploit cryptographic optimizations effectively. To our knowledge, this is the first accelerator to support hoisting technology and the gadget decomposition key-switching method. Our solution achieves a significant performance improvement, averaging a 1.8× speedup.

CCS Concepts

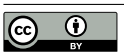
• **Computer systems organization** → **Parallel architectures**; • **Security and privacy** → **Hardware security implementation**.

Keywords

Fully Homomorphic Encryption, Accelerator, CKKS

ACM Reference Format:
Shengyu Fan, Xianglong Deng, Liang Kong, Guiming Shi, Guang Fan, Dan Meng, Rui Hou, and Mingzhe Zhang. 2025. FAST:An FHE Accelerator for Scalable-parallelism with Tunable-bit. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731407

*Corresponding author.



ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731407

1 Introduction

The growing demand for cloud computing and concerns about the risk of privacy data leakage have become the mainstream technological development contradictions. Individual users or companies

can hand over large amounts of data, even sensitive and private data, to cloud computing platforms to improve personal productivity and work efficiency. While this provides great convenience, it undoubtedly increases the risk of privacy exposure.

Fully homomorphic encryption (FHE) is a technology that allows direct computation of encrypted data. Once the data is encrypted, without any decryption operations during the qentire computation process. Therefore, this technology can achieve privacy protection without contradicting the principles of cloud-based services. Due to its unique security characteristics, a technology known as CKKS has become the mainstream scheme, offering encrypted computation on real or complex vectors. It provides opportunities for securing contemporary machine learning (ML) workloads [11, 15, 25, 29].

Although FHE offers outstanding security advantages, it introduces a large computational overhead compared to direct plaintext computation. This overhead mainly originates from two sources: First, to support more homomorphic multiplications, ciphertext data undergoes significant expansion, often by hundreds or even thousands of times, which directly increases the cost of computations. Second, additional operations are required to ensure the correctness and decryptability of ciphertext. These operations significantly impact the overall performance of FHE applications. During program execution, homomorphic operations introduce noise, necessitating noise management to reduce ciphertext levels. While this reduction can slightly decrease ciphertext size, it cannot satisfy the requirement for unlimited computational depth in FHE applications [11]. Therefore, bootstrapping is required to restore the ciphertext levels. However, this process still consumes most ciphertext levels. Currently, bootstrapping has become the main bottleneck in FHE programs. Since it comprises numerous ciphertext rotations and multiplications, the associated overhead is largely due to key-switching operations, which can account for up to 80% of the total cost. Thus, optimization techniques targeting key-switching are essential for enhancing the performance of FHE applications [8, 10, 18, 22].

Some works have begun to adopt algorithm optimizations on the CPU and GPU platforms to accelerate key-switching [8, 16, 18, 22], thereby improving the efficiency of the entire program execution process. Although some ASIC-based accelerator designs have demonstrated excellent acceleration effects, making them a viable option for the future development of FHE [20, 21, 23, 39, 40], these designs did not take into account recent algorithm optimization techniques, and only used one key-switching algorithm during the whole execution of applications, making it difficult to support different algorithm optimization schemes on existing accelerators. Because FHE is a novel scenario with immense commercial value, FHE algorithms are developing and advancing rapidly. To adapt the accelerator designs to the progress and iteration of algorithms, it is necessary to design systems that can better accommodate the runtime state changes of the entire program, rather than focusing solely on specific scenarios, thereby providing opportunities to support algorithm optimizations.

In this work, we analyzed the range of ciphertext levels required during the program execution process. It was found that most of the ciphertext levels are consumed by the bootstrap operation, leaving very few levels for the program operations, typically only eight levels (in fact, only seven levels are usable, as the last two levels need to meet the ciphertext's q_0 requirement to ensure accuracy). Additionally, we further analyzed the computational overhead of different key-switching algorithms at various levels. Through this analysis, we designed hardware that supports different key-switching algorithms. Moreover, we examined the impact of the hoisting technique on the computational overhead of the key-switching algorithm. For the first time, we analyzed different key-switching algorithms and applied them to the design of FHE accelerators to improve the execution efficiency of FHE applications.

Based on the above analysis, we designed **FAST**, an **F**HE **A**ccelerator for **S**calable-parallelism with **T**unable-bits. We found that by employing different key-switching algorithms during program execution, the computational overhead could be reduced. However, these algorithms have varying precision requirements during computation, which necessitates accelerator support for computations of different precisions. Consequently, this design incorporates computing components that support different parallelism and precision requirements while reducing computational overhead through different key-switching methods. More importantly, our accelerator design is the first to discuss how to support mainstream optimization algorithms in custom accelerator designs, providing an opportunity to enhance FHE application performance.

In particular, this work makes the following major contributions:

- We analyzed the computational overhead of different key-switching algorithms during program execution. We also considered the computational overhead when applying hoisting techniques, and clarified the requirements for key-switching algorithms under different ℓ. Furthermore, we analyzed the precision requirements of different key-switching algorithms and the hardware design overhead associated with different computational demands.
- We proposed a software framework that supports the selection of different key-switching algorithms and key management. In addition, we designed a multiplier that can accommodate different precision requirements while offering scalable parallelism.
- We proposed an accelerator architecture in which all computational units can support different parallelism and precision requirements. We also introduce how to support advanced FHE optimization algorithms under certain on-chip capacity and bandwidth constraints.
- Through detailed experimental evaluation, we demonstrated that our design achieves an average of 44.4% reduction in latency compared to state-of-the-art accelerators. Despite the increased computational and storage overhead, our design achieves a 1.13× performance-area improvement.

2 Background on CKKS and Applications

In this section, we introduce the key parameters, notations, and the basic construction and operations of CKKS [11], along with its application. Table 1 summarizes the notation used in this paper.

2.1 CKKS Implementation

2.1.1 CKKS Scheme and Ciphertext Structure. In the CKKS scheme, the message vector $\mathbf{m}$, typically consisting of real or complex numbers, is first scaled by a large factor (Δ) to convert it into a fixed-point (integer) representation. This scaling process introduces minimal rounding errors and ensures that operations on encrypted data are accurate within a defined precision. The scaled message is then encoded into a plaintext polynomial, denoted as $[\![\mathbf{P}]\!]$, which belongs to the polynomial ring $R_Q = Z_Q[X]/(X^N + 1)$, where N is the ring degree and Q is the modulus. The precision of computations within the CKKS scheme is directly affected by the choice of the scaling factor Δ, as it determines the trade-off between precision and the accumulation of rounding errors during homomorphic operations [20]. The *error growth* with the ciphertext executes FHE operations, which limits the number of operations allowed before a re-encryption or bootstrapping to refresh.

The encryption process transforms the encoded plaintext polynomial into two ciphertext polynomials $(\mathbf{c}_0, \mathbf{c}_1)$, each of which contains noise resulting from the encryption process. These ciphertext polynomials have degree N and coefficients modulo Q. The security of the CKKS scheme depends on both N and Q: while increasing N improves security by enlarging the degree of the underlying polynomial ring, it also increases the computational complexity of homomorphic operations. On the other hand, a larger modulus Q allows for more homomorphic multiplications before running out of noise budget, though this can reduce the ciphertext security, as larger Q values can be more vulnerable to attacks.

Given that Q is often extremely large, sometimes on the order of thousands of bits, direct computation on the coefficients becomes impractical. To overcome this limitation, CKKS employs RNS techniques by decomposing Q into L smaller moduli q_i, where each modulus is considerably smaller (28-bit to 64-bit). This decomposition, based on the Chinese remainder theorem (CRT) [11], allows for efficient parallel computation by treating the large polynomial as a set of smaller polynomials, or *limbs*, each operating under its respective modulus q_i. In terms of ciphertext structure, each ciphertext consists of two groups of L limbs, with each limb containing N coefficients. By contrast, the plaintext is encoded into a single group of L limbs. This structure allows for homomorphic operations to be carried out efficiently, as the decomposition enables parallelism, reducing the computational burden associated with working directly on large moduli.

2.1.2 FHE Operation and Primary Function. The ciphertext $[\![\mathbf{C}]\!]$ is sent to third-party servers, which provide extensive computational resources for outsourced computing. On these servers, the function $f([\![\mathbf{C}]\!])$ is computed by performing the *homomorphic evaluation* of f on the encrypted ciphertext. This process requires a sequence of *primitive FHE operations*, such as addition and multiplication, directly on the ciphertext. Specifically, we denote the additions (multiplications) of two $[\![\mathbf{C}]\!]$ as **HAdd** (**HMult**), the additions (multiplications) of a $[\![\mathbf{C}]\!]$ and a $[\![\mathbf{P}]\!]$ as **PAdd** (**PMult**). We also denote the cyclic rotation of a $[\![\mathbf{C}]\!]$ as **HRot**. Both **HMult** and **HRot** operations change the security key under which the ciphertext was originally encrypted, making it undecryptable by the original key. To restore the original security key and enable decryptable, the ciphertext must undergo a *key-switching* operation using a specific *evaluation key* (*evk*), which re-encrypts the ciphertext under the original key.

Table 1: CKKS parameter notation and descriptions.

Param.	Description
$\mathbf{m}$	Vector of real or complex numbers representing the message.
$[\![\mathbf{P}]\!]$	Plaintext polynomial obtained from encoding $\mathbf{m}$.
$[\![\mathbf{C}]\!]$	Ciphertext $(\mathbf{c}_0, \mathbf{c}_1)$ encrypted from $[\![\mathbf{P}]\!]$.
N	Power-of-two polynomial ring degree.
n	Maximum number of slots in the vector message, $n \leq \frac{N}{2}$.
L	Maximum multiplicative level of $[\![\mathbf{C}]\!]$.
L_{boot}	Number of levels consumed by bootstrapping.
L_{eff}	Maximum achievable level after bootstrapping.
ℓ	Current remaining multiplicative level.
Q	Initial modulus, decomposed as $Q = \prod_{i=0}^{L} q_i$ (RNS).
P	Auxiliary modulus, decomposed as $P = \prod_{i=0}^{\alpha-1} p_i$.
Δ	Scaling factor utilize to $[\![\mathbf{C}]\!]$.
α	Limbs in the RNS decomposition of Q.
$\tilde{\alpha}$	Limbs in the RNS decomposition of PQ for KLSS.
β	Number of limb groups after decomposition in key-switching.
$\tilde{\beta}$	Total number of limb groups used in KeyMult in KLSS.

Additionally, the bootstrapping operation is critical for enabling fully homomorphic encryption, as it allows for arbitrary-depth multiplication, turning the concept of FHE into reality. This operation consists of a series of **HMult**, **CMult**, **PMult**, and other operations, which restore the ciphertext's multiplicative level, enabling further computations. However, bootstrapping consumes a fixed number of multiplicative levels, denoted as L_{boot}, leaving the ciphertext at an *effective* level of L_{eff} after the operation.

These FHE operations are further decomposed into polynomial functions, including the *number-theoretic transform (NTT)*, *base conversion (BConv)*, *automorphisms*, and other element-wise operations. (I)*NTT* is an integer-based Fourier transform, enabling conversion between the *evaluation* and *coefficient representations*, which allows efficient element-wise operations and moduli-related computations. *BConv* facilitates the conversion of polynomials from one modulus to another, supporting computations across different moduli. The automorphism operation maps the i-th coefficient to the $(i \cdot 5^r \mod N)$-th position, enabling efficient **HRot** by a rotation factor of r. These functions enable the continuation of FHE operations while preserving their correctness.

2.1.3 key-switching Methods. As mentioned earlier, `key-switching` is essential for preserving ciphertext usability in FHE and is a major factor in the performance of FHE applications. Fig. 1 introduces two primary key-switching methods to accelerate ciphertext computations, each offering distinct approaches and trade-offs.

Hybrid key-switching method decomposes the limbs into β groups, where each group contains α limbs. The KeyMult operation is performed within these groups, followed by a ModDown operation to reconstruct the original limbs. This approach prioritizes flexibility in managing ciphertext limbs, but the initial KeyMult stage is computationally expensive, requiring numerous *NTT* transformations, which significantly increases processing time [16]. In contrast, the second method, KLSS, reorganizes limbs into groups denoted as $\mathcal{R}_T$, with each group consisting of α' limbs. This value is related to α, $\widetilde{\alpha}$, and the decomposed bit length v, where it is positively correlated with α and $\widetilde{\alpha}$, and negatively correlated with

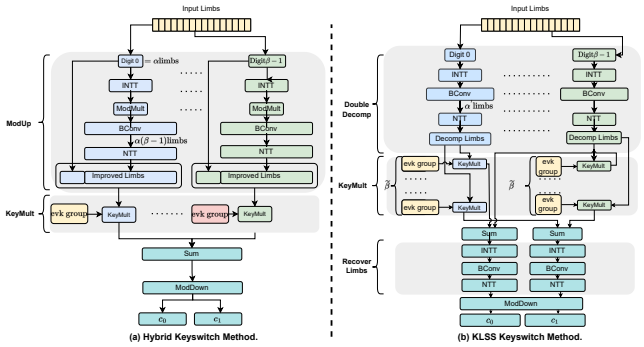


Figure 1: Dataflow diagram for Hybrid and KLSS key-switching methods. (a) Hybrid method's dataflow mainly consists of ModUp, KeyMult, and ModDown steps. (b) KLSS method's dataflow is mainly constructed using double decomposition, KeyMult, Revcover Limbs, and ModDown steps.

v [18, 22]. By utilizing more bits for the parameter v, with 60-bit precision shown to be most efficient in prior research, this method reduces the number of *NTT* operations required, thus optimizing computational efficiency [22]. However, this optimization comes with an increased overhead during the KeyMult stage due to the complexity of managing larger α' limbs. Once the limbs have been processed, they must be restored to their original $\mathcal{R}_Q$ structure, followed by a ModDown operation to recover the final limbs.

While both methods aim to optimize the key-switching process, they differ in their approach to computational trade-offs. The hybrid method emphasizes operational flexibility but comes at the cost of significant *NTT* overhead, while the KLSS method reduces *NTT* operations but increases the complexity in the KeyMult stage. Despite these differences, both methods represent substantial contributions to the efficiency of FHE. These methods contribute to managing noise growth during homomorphic operations and also enhance the practicality of FHE for large-scale applications requiring deep circuit evaluations by minimizing computational overhead.

2.2 FHE Applications

In this section, we provide background on FHE applications, outlining how traditional plaintext computations are transformed into ciphertext operations within the CKKS framework. Below, we describe the various types of computations supported by FAST.

2.2.1 Linear Operations. Traditional linear operations, such as convolution, matrix-vector multiplication, and average pooling, can be directly transformed into CKKS ciphertext operations through linear encoding. In this process, an input privacy vector or image, consisting of a total of n values, is first flattened into a one-dimensional vector. This vector is then encoded and encrypted into a CKKS ciphertext in polynomial form. Non-sensitive data, on the other hand, only requires transformation into a one-dimensional vector, which is subsequently encoded into polynomial form without encryption.

The computational overhead introduced by these linear operations is minimal, apart from the ciphertext expansion into L limbs, which is necessary to support more multiplication. Despite this, these operations maintain high computational efficiency and precision, ensuring that the results of encrypted operations closely match those performed on plaintext data.

2.2.2 Non-linear Operations. Due to its limitations with non-linear functions, CKKS does not natively support non-linear operations, such as ReLU, max-pooling, and sigmoid. To circumvent this, high-degree polynomials typically approximate these operations before being applied in the encrypted domain. For example, the ReLU function can be approximated by a polynomial of up to the 40th order to ensure adequate precision. While this approach allows for secure evaluation of non-linear functions, it also introduces additional computational complexity. The use of high-degree polynomials requires more costly homomorphic operations, but this trade-off is necessary to maintain accuracy in applications like deep learning and privacy-preserving neural networks.

2.2.3 Rotation and Hoisting Techniques. While certain linear operations can be directly mapped to ciphertext, many computations necessitate polynomial rotations to align data for processing. However, this rotation operation is computationally expensive. In practice, multiple rotations on the same ciphertext may be required to satisfy complete computational needs. As discussed earlier, each rotation demands a key-switching operation, which introduces a large overhead due to the first step of limb grouping during ciphertext switching. This grouping generates many NTT operations, further increasing computational costs. To address this, hoisting technology has been introduced. It enables a single grouping operation to handle multiple rotations by rotating the grouped limbs directly. This significantly reduces the number of NTT operations required, lowering computational overhead and improving overall application performance. Hoisting technology has proven to be effective across various applications and is even employed in foundational tasks like bootstrapping operations [12, 19, 37], although increases the evaluation key requirement during the computation.

3 Motivation

This section discusses the computational complexity of two mainstream key-switching methods and the hardware consumption of ALU at different word lengths. Finally, we summarize our observations and highlight the challenges for designing FAST.

3.1 Computation Workload Impact of Different Key-Switching Methods

As introduced in Section 2.1.1, FHE applications require high multiplicative depth to support more multiplications. Thus, the multiplicative level ℓ varies from 1 up to a maximum level L throughout application execution. Most of ℓ is consumed during the bootstrapping operation [18, 20, 33], where **HRot** and **HMult** operations can contribute up to 95% of the computational overhead. Thus, the key-switching operation accounts for approximately 80% of the total execution time, making it a critical target for performance optimization. Although optimization techniques such as KLSS [22] and hoisting [10] have been proposed to accelerate key-switching on GPU and CPU platforms, existing approaches typically use a fixed key-switching method for both **HMult** and **HRot** operations, regardless of variations in ℓ.

As shown in Fig. 2, we analyze the computational workload of the Hybrid key-switching method with the parameter **Set-I** and the KLSS method with the parameter **Set-II**. As depicted in Fig. 2(a), the

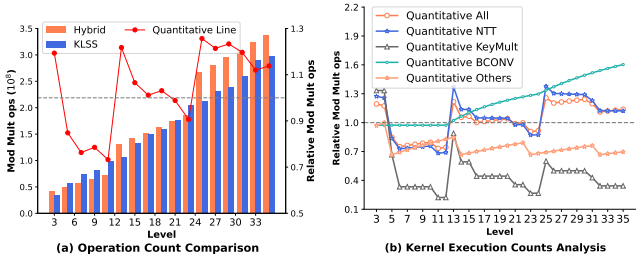


Figure 2: Quantifying the number of modular operations of the hybrid and KLSS key-switching methods. (a) Comparison of ops for the hybrid and KLSS key-switching methods (left) and evaluation of their relative efficiency based on total operation requirements (right). A 'Quantitative Line,' calculated as $\text{hybrid}_{\text{ops}}/\text{KLSS}_{\text{ops}}$, indicates efficiency: If the value is greater than 1, KLSS is more efficient; otherwise, the hybrid method is. (b) Impact of different kernels on the relative efficiency of each method, donates by the 'Quantitative Line'.

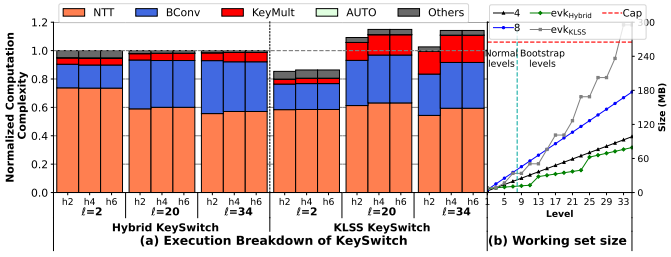


Figure 3: Exploring the impact of hoisting numbers for the different key-switching. (a) Execution breakdown of key-switching, with KLSS total execution operations, are normalized to the hybrid method. *h2* denotes a hoisting number of 2, *h4* denotes 4, and *h6* denotes 6. (b) working set size on various levels. Working set sizes are shown for cases each requiring an evk of hybrid and KLSS methods, and 4, 8 ciphertexts.

comparison reveals that the advantage in computational workload shifts depending on the ℓ.

The KLSS method shows a significant advantage when ℓ is between 25 and 35, achieving a 15.2% reduction in modular multiplication operations. In contrast, when ℓ is between 5 and 12, the hybrid method is superior, reducing modular multiplications by 23.5%. This demonstrates that the performance of the key-switching method does not consistently surpass that of the fixed method as ℓ varies.

As shown in Fig. 2(b), to further investigate the reasons for this shift, we analyzed the impact of each kernel in the key-switching algorithm on the performance of both methods. Between levels 5 to 12, the KLSS method does not significantly reduce the computational complexity of the NTT compared to the Hybrid method. This is because, at this stage, the KLSS method involves more limb groups, resulting in a higher computational requirement for the NTT. Furthermore, between levels 21 to 24, the KLSS method does not reduce the NTT-related operation, while other KeyMult operation computational load increases significantly. As a result, the KLSS method may require more computation than the Hybrid method.

As introduced in Sec. 2.2.2, hoisting technology significantly reduces the computational cost of the NTT operation, greatly improving the performance of FHE applications during consecutive **HRot** operations on the same ciphertext. As shown in Fig. 3(a), we further compare the requirement of the modular operation between the KLSS and Hybrid methods under varying hoisting numbers. The results indicate that the KeyMult operation becomes more dominant as the hoisting number increases. Consequently, KLSS requires more modular multiplication operations, which further decreases the performance superior to the hybrid method.

Based on the above analysis, we conclude that the performance advantage between the KLSS and Hybrid key-switching methods is not fixed for individual **HMult** or **HRot** operations as the ℓ and hoisting number varies.

Although the KLSS method and hoisting technique reduce the computational cost of the key-switching, they impose substantial on-chip storage demands. As shown in Figure 3(b), the KLSS method may require as much as 295 MB at the highest levels. Additionally, as the number of hoisting operations increases, the storage requirements for traditional hybrid methods also scale significantly. Therefore, in selecting an appropriate key-switching algorithm, on-chip storage needs must be carefully considered.

In summary, relying solely on a single key-switching method is suboptimal due to fluctuations in ciphertext levels during FHE application execution and the application of hoisting optimization techniques. Therefore, **dynamically leveraging both the KLSS and traditional hybrid key-switching methods based on the ℓ and hoisting optimization technology, while considering acceptable on-chip memory capacity, offers an opportunity to enhance overall application performance.**

3.2 Impact of Single Configuration Word Length Hardware on Parallelism

The word length of the ALU directly impacts hardware resource consumption in FHE accelerator design. Prior research indicates that a 36-bit word length is sufficient to meet the precision requirements of FHE applications [20]. However, as explained in Sec. 2.1.3, the KLSS key-switching method requires converting limbs to R_T, where each prime is set to a 60-bit word length to reduce the number of NTT operations. Therefore, the varying word length requirements make it challenging for the FHE accelerator to choose an appropriate bit length for the base word length configuration.

Fig. 4 presents both the ALU area and power consumption increase with higher bit lengths, driven by the additional hardware resources needed to handle larger operands. Specifically, the 60-bit ALU requires over 2.9× (2.8×) more chip area and 2.8× (2.7×) more power consumption compared to the 36-bit configuration for the modular multiplier (and multiplier-only designs, respectively). This increase is primarily due to the more complex logic necessary to meet the timing constraints imposed by longer word lengths [20].

Besides, as introduced in Sec. 2.1.3 to support the KLSS key-switching method requires the 60-bit multiplier to efficiently handle its computation demands[1]. However, this word length configuration introduces substantial inefficiencies in chip area and power consumption, particularly when used for the dynamical key-switching

[1]This bit precision requirement cannot be reduced by the bitpacker method proposed in [41], as bitpacker is only applicable to ciphertext computation.

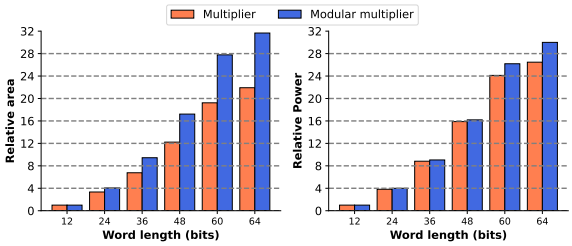


Figure 4: Area (left) and power (right) scaling of major ALUs in FHE: It presents relative area and power consumption for multipliers and modular multipliers across word lengths.

method, where the precision of the 60-bit multiplier exceeds the actual requirements. Additionally, executing 60-bit multiplication on 36-bit ALUs poses a significant challenge, as it requires four 36-bit multiplications using a Booth-like method [24]. This approach either reduces parallelism by 75% or increases chip area, resulting in a 27.5% (30%) overhead in both chip area and power consumption compared to a native 60-bit multiplier. Furthermore, this configuration reduces the overall parallelism of the accelerator by about 50%, since only half of the 36-bit multipliers are utilized to execute the Booth algorithm. Consequently, the system cannot fully leverage its available hardware resources, compromising overall performance.

Therefore, the unified word length configuration is not a suitable choice, as it either significantly increases chip and power overhead or compromises flexibility, ultimately reducing the parallelism of the accelerator. This creates an opportunity to **design an efficient hardware architecture that can flexibly accommodate these differing word length requirements while maintaining high parallelism.** We can optimize the accelerator to support both key-switching methods with efficient resource usage and improved performance by developing a flexible computational unit capable of handling multiple word lengths.

3.3 Challenges

The motivational study suggests that dynamically combining the KLSS and traditional hybrid key-switching methods offers further performance improvement while the FHE accelerator remains the drawback due to the mismatch in ALU computation precision requirements for both key-switching methods. Therefore, to achieve further performance improvement of FHE applications, our primary goal is **to develop a high-performance FHE accelerator capable of efficiently handling both KLSS and traditional hybrid key-switching methods with efficient hardware overhead.**

While we aim to achieve this goal, several challenges arise, which we address as follows. The first challenge is that **identifying the appropriate key-switching method for specific operations across different ciphertext levels during application execution remains a significant challenge.** As demonstrated in Fig. 2 and Fig. 3, the performance of the KLSS and hybrid methods varies across different levels. Furthermore, the performance is influenced by the use of hoisting technology in FHE application design. Besides, the KLSS method and the hoisting technology significantly improve the requirement of the evaluation key, thus it will increase the on-chip memory capacity. Deciding whether to apply optimization techniques and determining the appropriate optimizations, further impacts the performance differences between the two methods at various levels. Consequently, selecting the optimal key-switching method in such a complex and dynamic environment is difficult.

In addition to method selection, **managing evaluation keys for both KLSS and traditional hybrid key-switching methods poses considerable difficulty.** As introduced in Sec. 2.1.3, the KLSS method requires a 60-bit word length to perform key-switching operations efficiently, minimizing the number of NTT operations. Reducing this word length increases the number of NTT operations, which negatively impacts overall application performance. In contrast, previous research has demonstrated that a 36-bit word length is sufficient for the precision requirements of FHE applications [20], and the hybrid method only requires 36-bit for key-switching operations. Consequently, the evaluation keys used in KLSS differ significantly from those used in the traditional hybrid method, complicating key management when both methods are employed. This requires generating and maintaining multiple sets of evaluation keys, adding to the system's complexity.

Besides, from a hardware perspective, **designing a tunable bit-length computation component that balances high parallelism with minimal hardware overhead is highly challenging.** To support both 36-bit and 60-bit precision within the same ALU, a straightforward approach uses multiple 60-bit multipliers to perform the required computations dynamically. Specifically, 60-bit multiplication requires 4 such 36-bit multipliers, resulting in substantial hardware overhead due to the exponential increase in smaller multipliers. Therefore, developing an algorithm that efficiently performs both 36-bit and 60-bit multiplication with fewer 36-bit multipliers is essential. Managing these resources efficiently while maintaining high parallelism remains a design challenge.

4 FAST Framework

In this section, we first introduce the framework of FAST used to decide and manage the evaluation key, and then describe the hardware architecture of the tunable-bit multiplier.

4.1 Aether-Hemera: A Dual-Method Management Framework for key-switching Selection and Key Management

Existing schemes achieve satisfactory performance by relying solely on a single key-switching method, but substantial optimization potential remains. As discussed in Sec. 3.1, dynamically selecting distinct key-switching algorithms based on computational levels during application execution can reduce computational workload. To realize this objective, we propose a dual-method management framework comprising two core components: *Aether* and *Hemera*. Below, we detail the design and functionality of these components.

4.1.1 Aether: key-switching Method Analysis and Decision Tool. While leveraging the KLSS method or hoisting technology in FHE reduces computational workload as introduced in Sec. 3.1, the increased time required for off-chip data transfers by the accelerator directly impacts the overall application performance. Thus, it is necessary to analyze and select of the optimal key-switching method with the computational workload and off-chips evaluation key transfer latency. *Aether* is an analysis software that works as the preprocessing on the server side, it receives the FHE operation

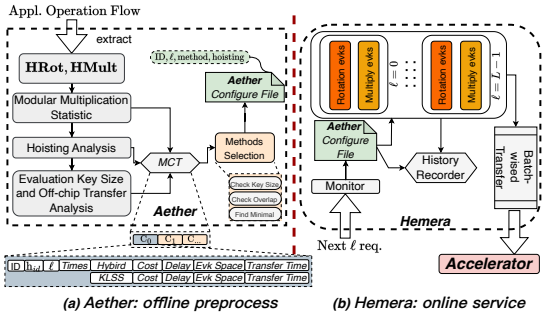


Figure 5: Workflow for the Dual-Method Management Framework for Method Selection and Key Management. Aether serves as the offline preprocessing step, while Hemera manages the evaluation key online with the accelerator.

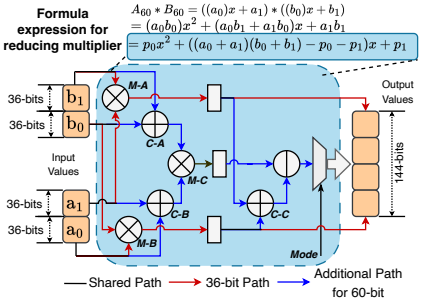


Figure 6: Hardware Architecture of the Tunable-Bit Multiplier. Diagrams are simplified to illustrate three 36-bit base multipliers that support 60-bit multiplication. M-A/B/C: Multiplier-A/B/C (36-bit base multipliers), C-A/B/C: Combiner-A/B/C (partial product aggregation units). When the 36-bit multiplication, the red line activates, while an additional blue line activates for the 60-bit execution.

flow from the applications and generates the *Aether configuration file* (about 1KB) containing the selection of key-switching methods.

Methods Candidate Table (MCT). MCT is the runtime analysis record table of Aether, which contains the metadata for all ciphertexts. Each MCT entry corresponds to one ciphertext and all MCT entries are sequentially organized according to their execution order. The format of each MCT entry is as shown in the bottom portion of Fig. 5(a), which contains the hoisting identifiers (h), the number of times the ciphertext must be executed under a specific hoisting configuration (*Times*), computational cost(*Cost*), relative computational delay(*Delay*), required key space(*Size*), and key transmission time(*Transfer Time*)—recorded separately for both the hybrid and KLSS methods.

Analysis Workflow. As shown in Fig. 5(a), Aether applies a methodical process to determine the optimal key-switching method for each operational context. First, Aether locates the **HRot** and **HMult** operations from the operation flow. Then, it extracts its parameters including ℓ and polynomial length and the ciphertext potential maximum hoisting configuration. After that, Aether sequentially computes modular multiplications for Hybrid and KLSS keyswitch methods, analyzes hoisting parameters, and evaluates evk size and transmission latency. Finally, all analysis results are stored in the MCT.

Key-Switch Selection. After the analysis, Aether sequentially processes each MCT entry as follows. **STEP-1:** Eliminate entries where the required key size exceeds the accelerator chip's reserved key storage capacity. **STEP-2:** From the remaining entries, discard those whose key transmission time is shorter than the execution time of the preceding ciphertext's key-switching operation. **STEP-3:** Among the filtered entries, identify the configuration with minimal execution time, selecting the minimal key size method if two configurations have similar latencies. Selected hoisting numbers and their key-switching methods (Hybrid or KLSS) are indexed by the ciphertext index and recorded in the *Aether configuration file*.

Security. *Aether configuration file* only contains the ciphertext index, ℓ, key-switching method, and the optimal hoisting configuration. These details without revealing any private data, as the messages remain secure under lattice-based cryptography [10, 11]. The security of the ciphertext is preserved since key-switching algorithms operate within the public key framework and the leakage of key-switching methods does not compromise its confidentiality [9].

4.1.2 Hemera: Evaluation Key Management Tool. Hemera is a runtime framework that schedules evk transfers from HBM to accelerators and selects key-switching methods via the *Aether configuration file* and ciphertext indices. As shown in Fig. 5(b), it main contains the *Monitor*, *History Recorder*, *Batch-wised Transfer* and *Evk Pool*.

Evk Pool. It stores the evaluation key address on the HBM, indexed by the ℓ. The pool contains L groups and each group consists of the rotation key for the **HRot** and the Multiply key for the **HMult**. The Monitor subsystem dynamically interfaces with these keys, supplying HBM access addresses to orchestrate low-latency data transfers between the accelerator and memory.

Management Workflow. The *Monitor* executes on the static analysis of FHE computation graphs statically to identify required key-switching operations before execution, extracting both the associated ciphertext index and parameter ℓ. Then, Hemera queries the *Aether configuration file* to obtain pre-optimized hoisting identifiers and key-switching methods, subsequently choosing appropriate evaluation keys address from the evaluation key pool. The history recorder tracks key-switching patterns across subsequent computational levels (contains the configuration of the key-switching method and the suitable hoisting number), enabling proactive adaptation to recurring FHE operation workflows. Hemera implements batched data transmission, synchronized with prefetching mechanisms common in FHE accelerators. It groups 256 consecutive elements as a batch, matching the minimum processing granularity required by individual accelerator computing units. Using this as the basic unit of transmission of HBM can better adapt to the computational logic of on-chip computing components.

Prefetching mechanism. Through historical pattern analysis from the history recorder, Hemera prefetches relevant configurations from the *Aether configuration file* and preloads required evaluation keys. This collaboration between *Aether* and *Hemera* minimizes pipeline stalls while maintaining high accelerator utilization. Notably, the latency of accessing the configuration file is significantly lower than HBM data transfer latency, enabling overlap through prefetching.

4.2 Tunable-Bit Multiplier (TBM)

In this section, we delineate the architecture of the proposed multiplier design, which is designed to support both 36-bit and 60-bit multiplications on the same hardware. This dual-purpose design significantly enhances hardware efficiency by reducing the separate hardware modules for different bit widths, thereby improving the performance by increasing the parallelism. To fully comprehend the intricacies of our design, we introduce how it supports and minimizes the need for 36-bit multipliers for 60-bit configurations.

We select a 36-bit multiplier as the base multiplier to describe our TBM, which supports both two 36-bit multiplications and one 60-bit multiplication. As shown in Fig. 6, each TBM contains two 72-bit-wide input buffers, each capable of storing either two 36-bit values or one 60-bit value. For 36-bit-wide multiplications, each buffer partitions into two 36-bit segments. The lower 36-bit segment from each buffer routes to multiplier B, while the higher 36-bit segment routes to multiplier A, with both results concurrently written to the output buffer. This execution flow enables the TBM to enhance parallelism in fundamental computational operations. For 60-bit precision, the input buffer decomposes into two 36-bit segments: the lower segment retains full precision while the upper segment zero-extends the 24 most significant bits. This precision-tailored decomposition enables timing-aware operand routing—multiplier B computes $a_0 \times b_0$ concurrently with multiplier A processing $a_1 \times b_1$, while multiplier C derives $(a_0 + a_1) \times (b_0 + b_1)$ in pipelined synchronization. The Combiner unit then fuses these three intermediate products through cycle-optimized accumulation logic, implementing a latency-critical variant of the Booth algorithm that reduces multiplication requirement by 33% compared to conventional implementations. The TBM architecture directly supports native 60-bit computations while enabling dual 36-bit multiplications in parallel. Its hierarchical scalability to smaller multipliers (accommodating both 36-bit and 60-bit precisions) achieves dynamic parallelism adaptation, though with inherent area-complexity tradeoffs. Specifically, the design achieves 2× parallelism improvements for 36-bit operations with only 28% area overhead relative to conventional 60-bit multipliers, despite requiring 19% additional control logic.

This flexibility introduces non-trivial integration challenges: decomposing 60-bit operations into four 12-bit multiplier primitives reduces the arithmetic unit count from 25 to 15 instances, but incurs routing overhead from the precision-switching control circuitry. [27, 41]. Such reconfiguration constraints ultimately depend on the host platform's circuit library characteristics, particularly its capacity for mixed-precision operand routing. While TBM enables flexible precision and scalable parallelism, existing accelerators will struggle to integrate these capabilities due to their reliance on fixed parallelism, which may introduce unexpected pipeline stalls and lead to overall performance degradation. Thus, the following section presents our component-level architecture design and respectively introduces their respective dynamic input adjustment mechanisms to meet the parallelism requirements of TBM.

5 ACCELERATOR ARCHITECTURE

5.1 Overview

In this section, we present the FHE CKKS accelerator, named FAST, which is engineered to support both 36-bit and 60-bit precision

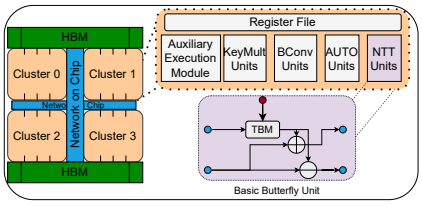


Figure 7: Organization of FAST. FAST consists of four vector clusters connected via a lane-wise NoC.

computations. Our primary design goal is to accommodate a variety of key-switching methods and to leverage hoisting technology, thereby enabling practical execution of FHE workloads with performance improvement compared with prior accelerators such as SHARP [20], ARK [21], and Craterlake [40]. The architecture of FAST is modular, consisting of four clusters, each equipped with 256 lanes. The global data distribution policy between clusters mirrors the approach used in SHARP [20] and ARK [21]. To support both 36-bit and 60-bit computations, FAST employs TBM for fundamental multiplication, thereby facilitating scalable parallelism across each lane. We detail how FAST effectively manages varying computation precision and parallelism across different components.

Our architecture is structured into five primary components. NTTU accelerates polynomial multiplication by converting limbs between coefficient and point representations. BConv Userves the second phase of the BConv operation, while AutoU facilitates limb rotation, supporting the **HRot** operation. KMU is specifically designed to multiply ciphertext limbs with evaluation keys during key switching, and also supports plaintext-related homomorphic operations, and executes the initial step of the BConv operation. Additionally, an auxiliary execution module is included to maintain bootstrapping precision and optimize on-chip memory usage for storing evaluation keys. This modular approach ensures efficient handling of diverse computational requirements and parallelism, thereby enhancing the overall performance and scalability of FAST, and the main architecture of FAST is shown in Fig. 7.

5.2 NTT Unit (NTTU)

In our solution, we employ a radix-2 pipelined FFT architecture to accelerate the (inverse) NTT. To optimize for reduced horizontal bisection bandwidth and minimize wiring complexity in the NTT unit (NTTU), we adopt the ten-step NTT method, similar to **SHARP** [20]. This approach enhances hardware efficiency in terms of area and power consumption. The method pipelines the entire N-point (I)NTT process by recursively executing the well-known four-step 2D-FFT, thus mapping the total N elements into a 3D matrix with dimensions $\sqrt{N} \times \sqrt[4]{N} \times \sqrt[4]{N}$. This matrix representation enables two distinct phases. The first phase performs a column-wise four-step NTT on submatrices of $\sqrt[4]{N} \times \sqrt[4]{N}$ elements, while the second phase handles row-wise operations. Between these phases, an *inter-lane-group transpose* is facilitated through direct cluster-wide wire connections. Each phase comprises $\sqrt[4]{N}$ lane groups, with each group executing a sequence of operations: *butterfly* → *transpose* → *twisting* → *butterfly*. Each butterfly step requires a *butterfly unit*, which consists of $\frac{\sqrt[4]{N}}{2} \log \sqrt[4]{N}$ modular multipliers. These multipliers are interconnected in a butterfly network to multiply input data with the twiddle factors. The *transpose* and *twisting* steps employ a

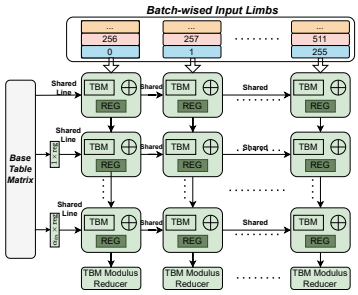


Figure 8: Architecture of the array used in the BConv Units consists of rows that share the same input values from the base table matrix. After the last row, the output values undergo a modulus reduction operation. The BConvU contains two arrays, and we illustrate one of them for description.

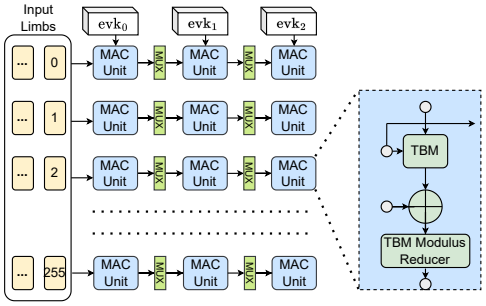


Figure 9: Architecture organization of the KeyMult Unit for executing the KeyMult and element-wise operations.

quadrant-swap transpose unit, which recursively decomposes data into submatrices, allowing full pipelining as detailed in [39]. This unit comprises $\log \sqrt[4]{N}$ layers of smaller transpose units, and it also handles the multiplication of twisting factors.

To support varying computational precision requirements, we replace the basic multiplier with the TBM in the Montgomery modular multipliers. Thus, our NTTU supports different computation precisions with varying levels of parallelism, maintaining the same architectural topology as **SHARP**. For 60-bit computation precision, each lane group processes $\sqrt[4]{N}$ elements, with $\sqrt{N}$ elements being fetched from on-chip memory in a single cycle. These $\sqrt{N}$ elements are organized as one batch, forming a polynomial from $\sqrt{N}$ batches. The execution follows a sequential format: $[b_0, b_1, b_2, \ldots, b_{\sqrt{N}-1}]$. For 36-bit precision, as the TBM can support two multiplier elements, our NTTU enhances parallelism from $\sqrt{N}$ to $2\sqrt{N}$. Each lane group processes $2\sqrt[4]{N}$ elements on the same butterfly network, computing two batches per cycle. The execution flow for two polynomials is represented as $[(b_0^0, b_0^1), (b_1^0, b_1^1), \ldots, (b^0_{\sqrt{N}-1}, b^1_{\sqrt{N}-1})]$, where b_m^n represents batch m of polynomial n. For example, each TBM receives a high 32-bit element from batch 0 of polynomial 0 and a lower 32-bit element from batch 0 of polynomial 1, with both sharing the same twiddle factor.

Our NTTU design enables flexible parallelism and accommodates multiple computational precision requirements for distinct key-switching methods: the 36-bit model operates for hybrid methods, while the 60-bit model supports KLSS methods. This component also executes on-the-fly operations, such as *of-limbs generation*, exclusively in the 36-bit model.

5.3 Base Conversion Unit (BConvU)

In FAST, the Base conversion operation splits into two stages: element-wise modular multiplication followed by large matrix-matrix multiplication. The latter is optimally accelerated via a 2D systolic array, which is the targe of the BConvU.

As shown in Fig. 8, we design two 2D systolic arrays to execute the multiplication of the limbs matrix with the base table matrix. Each cell of the BConvU performs one multiply-accumulate (MAC) operation per cycle, passing the sum result to the neighboring cells below. Modular reduction operation is performed in the lowest-level cells to avoid the overflow. When the dimensions of the systolic array are smaller than those of the limbs matrix or base table matrix, blocked matrix multiplications are required.

BConvU performs matrix-matrix multiplication for the input limbs matrix with shape $(N \times \alpha_{in})$ and the base table matrix with shape $(\alpha_{in} \times \alpha_{out})$. Height of the array is determined by the number of α_{in}, with each row sharing the same values. The values from the column are parallelized across different rows. Besides, each row includes a buffer to hold input values, ensuring the correctness of the pipeline. Partial sum results from the lowest row are sent to the modular reduction unit, which shares the same output modulus.

The width of our systolic array is 256, resulting in BConvU containing 256 columns. Our design also supports different computation precision requirements by integrating the TBM into each MAC cell. The input limbs matrix is organized into two different models to accommodate these requirements. For the 60-bit precision requirement, the input limbs matrix is organized into multiple batches, each containing 256 elements that contribute to different columns of the systolic array. The elements from each batch are fetched from the highest row and then passed from the top to the bottom row cycle-by-cycle. For the 36-bit precision requirement, the TBM can parallelize the computation of two values, increasing the parallelism from 256 to 512. Although BConvU only serves 256 columns, each column needs to receive two elements, which can be easily merged from consecutive batches. The high 36 bits are set as the elements from batch 0, while the lower 36 bits are from batch 256. Thus, our design supports different computation precision requirements with scalable parallelism.

5.4 KeyMult Unit (KMU)

As shown in Fig. 9, FAST introduces a unit responsible for element-wise key multiplication and additions in CKKS, which are critical for the KLSS key-switching method and hoisting technology. For the hybrid method, key multiplication involves a vector-vector multiplication, where each vector element contains $\alpha\beta$ limbs. In a non-hoisting environment, there is no data reuse for input limbs and evaluation keys, but hoisting technology allows for the reuse of input limbs. In the KLSS method, key multiplication involves a vector-matrix multiplication (with the input limbs matrix having shape $1 \times \beta$ and the evaluation key having shape $\beta \times \widetilde{\beta}$), where each element contains α' limbs. This structure enables input limb reuse. Therefore, key multiplication can be optimized by reusing input limbs, providing opportunities for hardware design improvements.

KMU is implemented as a 2D systolic array with an output-stationary architecture. The array's width is set to 3 (determined by

the parameter β), and its height is 256 (corresponding to the base parallelism, where each limb is split into multiple batches, each containing 256 elements). Each MAC unit in the KMU includes one TBM, a reduction unit, and an adder unit, which together perform the multiplication of input limb elements with elements of the evaluation key. Sharing of the input limb coefficients between neighboring MAC units within the same row occurs only when KLSS or hoisting technology is applied.

TBM used in KMU supports calculations with different precision requirements, enabling flexible parallelism. For 60-bit precision, all input data is processed in a single batch, whereas for 36-bit precision, the input is divided into two batches, allowing parallelism to scale up to 512. When input limbs are shared, key inputs for each column come from distinct key groups, and the input weights consist of consecutive batches. When no sharing occurs, input weights and keys are mapped one-to-one. Each column array performs batch calculations with an interval of 1, such as $(b_0, b_1, b_2), (b_3, b_4, b_5), \ldots$, and the intervals within each column are fixed at 3.

Besides, it also servers the element-wise operations such as **HAdd**, **PMult**, **PAdd**, **CMult**, and **CAdd** for achieving the more high efficiency and parallelism.

5.5 Automorphism Unit (AutoU)

AutoU transforms limb elements using the mapping function ϕ_r, defined as $i \mapsto (i \cdot 5^r) \mod N$. It process multiplies the index i of each element by 5^r and reduces it modulo N, rearranging the limbs according to the mapping. It is essential for ciphertext rotation. AutoU is based on Benes network [7] and it is always used in the key-switching operation, handling elements with two different bit-widths. Bit length is set to 72-bit. For 60-bit coefficients, it processes 256 elements per cycle, while for 36-bit coefficients, it processes 512 elements per cycle, with the upper and lower 36-bit segments placed in two consecutive batches to meet the 72-bit word length.

5.6 Register File (RF) Organization

FAST introduces large register files based on SHARP architecture, providing one 72-bit word for each lane every cycle. We simplify control and addressing by using small lane-group-wise counters to eliminate the need for sending cluster-wide address signals each cycle, thereby capitalizing on sequential access. FHE accelerators rely on a massive on-chip memory capacity to reuse data and reduce the off-chip memory bandwidth requirement. To support the evk incremental of the KLSS method, and the hoisting technology, we need to select suitable parameters to reflect the algorithm efficiency and also minimize evk storage requirements.

As shown in Fig. 3 (b), a ciphertext is size 19.7MB, and an **evk** is 79.3MB for the hybrid method and 295.3MB for the KLSS method, at the level is 35. Therefore, the KLSS method is not a good choice at the highest level, although it requires less computation workload. To meet the memory capacity for supporting the KLSS and hoisting methods, we design the on-chip memory size as 245MB, thus it provides more opportunities to set larger *bs* and hoisting technology to reduce the computation complexity of the *coeffToSlot* step. However, it also contains sufficient space to support the KLSS method in the *gs* step of *coeffToSlot* and *EvalMod* steps. Consequently, the on-chip memory capacity configuration is enough to support the KLSS method and hoisting technology while maintaining acceptable off-chip bandwidth requirements.

Table 2: Parameter sets used in FAST. Moduli (q_i) is given in bits. *KSW* denotes the key-switching method.

Set	N	n	L	L_{eff}	α	$\tilde{\alpha}$	q_i	*KSW*
I	2^{16}	2^{15}	35	8	12	-	36	Hybrid
II	2^{16}	2^{15}	35	8	5	9	36	KLSS

Table 3: Area and Peak Power consumption of FAST.

Component	Area(mm^2)	Peak Power(W)
4×NTTUs	60.88	142.7
4×BConvUs	28.89	86.6
4×KMUs	10.58	27.67
4×AUTOUs	0.6	0.8
4×AEM	8.67	10.7
Register Files	123.9	29.4
HBM	29.6	31.8
NoC	20.6	27.0
Total	283.75	337.5

5.7 Auxiliary Execution Module (AEM)

To better support the execution of FHE applications on accelerators, several auxiliary modules are incorporated into the accelerator design to ensure the accuracy of computations.

5.7.1 Double-Prime Scaling Unit (DSU). This work adopts the 36-bit as the ciphertext bit-width. Although this configuration allows direct computation with acceptable accuracy loss [20], the double rescale is required during the bootstrapping operation to maintain ciphertext precision. Thus, this work employs the same DSU design as SHARP, which consists of four multipliers, two adders, and two modulo units supporting 512 parallelism.

5.7.2 Evaluation Key Generator (EKG). Interestingly, it was discovered that the evaluation key consists of two parts (a, b), with the second part, b, generated based on a. Thus, only part a of the key needs to be stored on-chip, while part b can be generated dynamically from a. To leverage this, this work utilizes the same PRNG module as prior accelerators [20, 40], known as the Evaluation Key Generator. This approach significantly reduces the key storage cost, providing a foundation for the use of KLSS and hoisting technology.

6 Methodology

6.1 FAST Evaluation

We implement the architecture in RTL to evaluate and synthesize our design to estimate area and power. We use the TSMC 7nm Predictive Process Design Kit (PDK) for synthesis. Besides, we utilize FinCACTI [43] to assess the long wiring and SRAM components, with details of the HBM stacks obtained from prior work [20, 21]. All components and networks operate fully pipelined at 1 GHz.

To rigorously evaluate system performance and power efficiency, we developed a cycle-accurate simulator base on [3, 20]. We first test individual components through functional simulation to ensure algorithmic correctness. Next, we translate each application into a cryptographically structured operation trace for FHE computations, preserving the original execution order and dependencies. These operations are partitioned into hardware-aligned kernels,

Table 4: Comparing hardware information of FAST against prior works. BW: bandwidth (TB/s), Cap: capacity (MB).

	Off-chip BW	Bit Width	Lanes	On-chip Mem BW	On-chip Mem Cap	Area (mm^2)
BTS [23]	1	64	2048	292+38.4	512	373.6
CLake [40]	1	28	2048	84	282	222.7
ARK [21]	1	64	1024	20+72	588	418.3
SHARP [20]	1	36	1024	36+36	198	178.8
$SHARP_{LM}$ [20]	1	36	1024	36+36	281	215
$SHARP_{8C}$ [20]	1	36	2048	72+72	198	250
$SHARP_{8C+LM}$ [20]	1	36	2048	36+36	281	290
FAST	1	60	1024	72+72	281	283.75

which are then mapped to their corresponding components (e.g., NTTU, BConvU, memory controllers (HBM)) with cycle-level timing precision. Finally, the simulator generates a detailed breakdown of execution metrics, including component-level latency, off-chip data transfer latency between the accelerator and HBM, and total system runtime, enabling fine-grained performance/power tradeoff analysis. We also utilize the on-the-fly limb extension (OF-Limb) as proposed in ARK [21], and we also incorporate the minimum key-switching (Min-KS) technology in scenarios where hoisting is not used [21] to decrease the off-chip bandwidth requirement. We re-implement the hybrid and KLSS methods as introduced in [16, 18, 19, 22].

6.2 Benchmarks

To implement both KLSS and traditional hybrid key-switching methods, we adopt a similar technology as described in [17] to ensure the security and correctness of the applications. The parameters used in both methods are shown in Table 2, and both parameters achieve the 128-bit security requirement.

We evaluate the performance of FAST on the following benchmarks: **Bootstrap.** We evaluate the state-of-the-art fully-packed bootstrapping method as presented in prior works [20, 21]. This method consists of four main steps: `ModRaise`, `CoeffToSlot`, `EvalMod`, and `SlotToCoeff`. The `ModRaise` step enhances the multiplicative level to L. The `CoeffToSlot` and `SlotToCoeff` steps perform homomorphic Discrete Fourier Transform (DFT) operations, utilizing pairs of **HRot** and **PMult**. The `EvalMod` step conducts approximate modular reduction using a polynomial function. While the latter three steps predominantly consume levels to achieve L_{eff}, we set L_{eff} to 8, in alignment with SHARP and ARK. Note, that each **HMult** and **PMult** consumes two levels as the double rescale for maintaining the precision. **ResNet-20.** We performed inference on the CNN model for an encrypted $32 \times 32 \times 3$ image using the CKKS implementation of the ResNet-20 model [25]. **HELR.** We also executed an ML workload training a binary classification model with a batch size of 256 images (**HELR256**) or 1024 images (**HELR1024**), over 32 iterations [15]. $\mathbf{T_{mult,a/s}}$**.** It is a important metric, referred to as *amortized mult time per slot* [19], leverages fair comparison for parameter selection and bootstrapping latency.

7 Results

7.1 Hardware Overhead of FAST

FAST supports 60-bit computation for a more efficient key-switching method. As shown in Tables 3 and 4, compared with SHARP (36-bit version), its on-chip computation circuit overhead increases by 2 times, while the on-chip memory increases by 1.41 times due to the hoisting technology and KLSS method, which significantly increase the evaluation key requirement. Compared with the recent 60-bit FHE accelerator, ARK, FAST reduces chip overhead by 27.3%, thus reducing the on-chip memory requirement. Although the chip overhead increases compared to the SOTA accelerators, FAST is the first to explore both KLSS and hoisting technologies on accelerators.

Table 5: Execution time (ms) of FAST versus prior works. (LM: Large Memory, 8C: 8-Cluster)

Method	Bootstrap	HELR256	HELR1024	ResNet-20
BTS	22.88	-	28.4	1910
CLake	6.32	3.81	-	321
ARK	3.52	-	7.42	125
SHARP	3.12	1.82	2.53	99
$SHARP_{LM}$	2.94	1.72	2.44	93.88
$SHARP_{8C}$	2.16	1.33	1.89	72.34
$SHARP_{LM+8C}$	2.03	1.26	1.83	68.59
FAST	1.38	1.12	1.33	60.49

Table 6: Comparing $\mathbf{T_{mult,a/s}}$ of FAST against prior works.

	Slots	$T_{A.S.}$(ns)	Speedup	Speedup per area
$F1_{32}$	1	470	87	20.5
BTS_{64}	2^{15}	45.7	8.4	10.4
ARK_{64}	2^{15}	14.3	2.6	3.6
$CLake_{28}$	2^{15}	17.6	3.3	2.4
$SHARP_{36}$	2^{15}	12.8	2.4	1.4
$SHARP_{60}$ [5]	2^{15}	11.7	2.2	2.3
$FAST_{60}$	2^{15}	5.4	-	-

7.2 Performance Comparison

As shown in Table 5, FAST achieves 23.17×, 13.29×, 3.4×, and 1.85× higher performance on average compared to BTS, CLake+, ARK, and SHARP, respectively. All of the applications have a significant portion of their execution times consumed by bootstrapping, reaching up to 94.5% for HELR256, with an average of 87.73%. This underscores the importance of accelerating bootstrapping in FHE. Thus, bootstrapping is the most critical operation. FAST improves its performance by integrating hoisting technology in the `CoeffToSlot` and `SlotToCoeff` stages, while utilizing the KLSS method in the `EvalMod` and `SlotToCoeff` stages. Besides, it also provides more parallelism for the hybrid keyswitch method execution. We also compare the performance per area to evaluate the efficiency of FAST. As shown in Table.4 and 5, FAST achieves 1.13× in terms of the performance per chip area, while the bootstrapping application achieves 1.4× improvement.

We evaluate the FAST architecture against SHARP with enhanced resource configurations[2] shown in Table.4 and 5, specifically, the $SHARP_{LM}$ ($215mm^2$), which features an increased on-chip memory capacity from 198MB to 281MB and integrates direct hoisting technology. Our results show that FAST achieves an average performance improvement of 1.76× and a 1.35× increase in performance per area compared to $SHARP_{LM}$. We also assess the $SHARP_{8C}$($250mm^2$) architecture with an 8-cluster configuration (the internal memory bandwidth also achieves 72TB/s of each

[2]We acquired execution time and area data for the 8-cluster configuration from [20]. We also model performance under large memory conditions by comparing the effects of reduced computational workload.

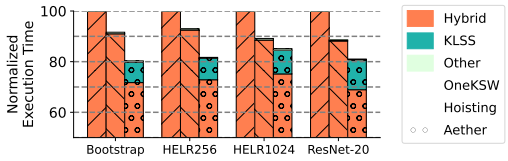


Figure 10: Execution Time Breakdown: Proposed Scheme vs. Prior Implementations on FAST. "Hybrid" and "KLSS" denote respective execution times. "OneKSW" uses only the hybrid method, "Hoisting" leverages hoisting technology, and "Aether" integrates hoisting and the KLSS method.

Table 7: Average power consumption, energy, and Energy-Delay Product (EDP) of FAST for all workloads.

Application	Avg. Power(W)	Energy(J)	EDP
Bootstrap	120	0.16	0.2
HELR256	118	6.5	360
HELR1024	154	0.16	2.2
ResNet-20	160	0.20	14.8

memory), which includes a more modular multiplier and enhanced on-chip bandwidth. In this scenario, FAST demonstrates an average performance improvement of 1.34× and a 1.18× increase in performance per area relative to SHARP_{8C}. When comparing the SHARP_{LM+8C}(290mm^2), which both integrate large on-chip memory and an 8-cluster configuration, FAST still achieves an average performance improvement of 1.27× and a 1.3× increase in performance per area. These findings indicate that the hardware design of FAST significantly enhances parallelism and effectively balances computational workload and off-chip data transfer latency. This is achieved by strategically assigning hosting and KLSS keyswitch methods by utilizing the Aether-Hemera. Although the Hemera tool operates online, its interaction with the configuration files generated by the offline Aether tool ensures a seamless workflow. The time needed for Hemera to read these configuration files (less 900ns) is significantly shorter than for evaluation key transmission (about 80us). Thus, the online Hemera key management tool does not adversely affect overall performance.

As shown in Table 6, we compare $\mathbf{T_{mult,a/s}}$ with prior FHE accelerators. FAST_{60} achieves a 17.6× performance improvement on average and a 6.7× improvement in terms of performance per area. Furthermore, as shown in Table 7, FAST increases power consumption by 1.7× compared with SHARP, with these workloads averaging 138.5W and reaching a maximum of 160W for HELR1024, primarily due to the significant increase in power consumption by the NTTU. In terms of energy consumption, FAST achieves 1.76J on average, which is a 22.8% reduction on average for SHARP, and even reaches a 44.1% reduction for the bootstrapping application. It also shows a 58.8% reduction on average for the Energy-Delay Product compared with SHARP[3].

7.3 Execution Time Breakdown

In this section, we perform a quantitative analysis of various keyswitch schemes, including the hybrid keyswitch, direct utilization of hosting, and leveraging Aether to decide whether to use

[3]We assume the power consumption of SHARP at 94.7W for all workloads, which is obtained from [20].

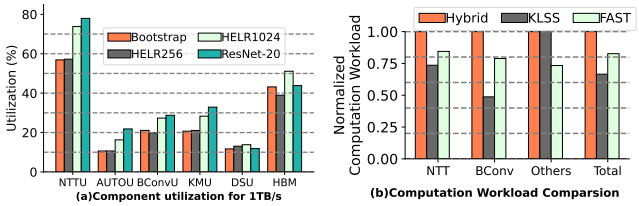


Figure 11: (a) Utilization of components of FAST. (b) Computational ModOps Comparison for the Bootstrap.

hosting and KLSS methods. As illustrated in Fig. 10, the hosting technology reduces keyswitch execution time by 10%. The performance improvement is limited as directly using hosting technology significantly increases the evaluation key requirement, leading to increased off-chip communication time. Aether optimizes the choice of hosting and KLSS methods at different levels, balancing computation execution time and evaluation key transmission time. Thus, it achieves a 1.24× performance improvement compared to the baseline. Besides, Aether replaces some hybrid keyswitch methods with KLSS, reducing the original hybrid execution time by 56.96%.

7.4 Utilization of Hardware Components

Fig. 11(a) presents the utilization of the main hardware units, showing that FAST is primarily a compute-bound accelerator. The NTTU activates with an average utilization of 66.47%, as it is used in most of the HE operations, converting data into polynomial representation. BConvU and KMU achieve 24.3% and 25.7% utilization, respectively, due to the increased BConv-related operations and evaluation key multiplication operations introduced by the KLSS method during the key-switching execution. For the bandwidth set similar to prior work (1TB/s), we also observe that most of the time is consumed by data transfer from off-chip memory via HBM, averaging 44.3%. This indicates that FHE applications are also memory-bound workloads [3].

7.5 Computation Workload

As shown in Fig.11(b), we evaluate the impact of different key-switching algorithms on computational complexity in bootstrap applications. The KLSS method significantly reduces NTT and BConv's computational complexity with unlimited memory capacity. However, it requires large on-chip storage and introduces transmission delays. Thus, using KLSS at higher levels is not optimal. In contrast, FAST employs different key-switching algorithms at varying ℓ. Compared to traditional methods, it reduces total computation by 17.3%, decreases NTT computation by 16%, increases BConv computation by 21.2%, and reduces element-level computations, such as key multiplication operations, by 26.7%. Moreover, FAST also enhances parallelism, resulting in better performance.

7.6 Efficiency Study

To evaluate how FAST achieves superior performance improvements, Fig. 12 presents the performance study of application performance enhancements due to each optimization. We use FAST as the baseline, and gradually reduce each optimization method and ultimately as an accelerator with a 36-bit ALU. Our results show that by utilizing the Aether-Hemera KeySwitch method selection and evaluation key management tool, $\text{FAST}_{\text{without TBM}}$ achieves

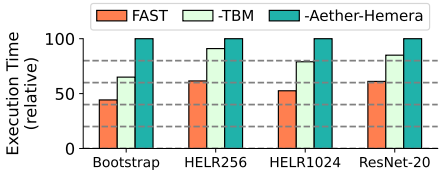


Figure 12: Performance changes from the gradual reduction of TBM, and Aether-Hemera to the FAST with 36-bit ALU.

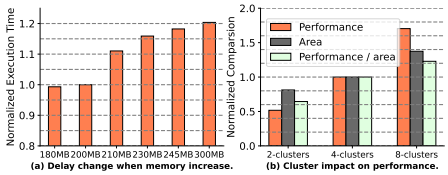


Figure 13: Performance, area, and performance per area of FAST for bootstrapping, evaluated across various scratchpad SRAM sizes and cluster numbers.

a 1.3× performance improvement compared with the 36-bit ALU accelerator. When integrated with the TBM, FAST further attains a 1.45× performance boost by significantly enhancing the parallelism of the accelerator, thereby reducing the execution time of each component and balancing off-chip communication time via the Aether-Hemera tool. These findings demonstrate that our TBM hardware design, combined with the Aether-Hemera tool, effectively enhances FHE application performance. Although leveraging four 36-bit ALUs achieves similar performance with TBM, it increases 1.5× area overhead for each multiplier group.

7.7 Sensitivity

On-chip memory capacity directly impacts application performance. Larger on-chip storage space enables adopting more efficient algorithms in Baby-Step Giant-Step (BSGS) during bootstrapping. Besides, it enhances the effectiveness of hoisting technology and the KLSS method, leading to improved accelerator efficiency. As shown in Fig. 13(a), increasing on-chip memory reduces the computational workload while increasing the evk transfer time for the HBM. As a result, the overall performance of bootstrapping does not improve significantly. Conversely, reducing the on-chip memory capacity leads to a noticeable performance drop, as it increases the computational workload, decreases the hoisting number, and forces the switch from KLSS to the hybrid method. This demonstrates that while large on-chip memory is necessary to support advanced optimization techniques, such as hoisting technology and the KLSS method, excessive memory does not always yield performance gains as the off-chip bandwidth also limits the performance.

Additionally, we explore the relationship between compute units and application performance, as shown in Fig. 13(b). As the number of clusters decreases, performance also decreases, with a 48.3% reduction in performance, which is limited by the computation units. Conversely, increasing the number of clusters boosts performance by 1.7×, and area increases by 1.37×, while it increases the 12% pipeline stall as the HBM fetches the evaluation keys. These results demonstrate that FAST can significantly improve performance by scaling computation components, showing good scalability.

8 Related Work

FHE Acceleration. Recent advancements in FHE target both algorithmic efficiency and hardware optimization. On the algorithmic side, key innovations include the KLSS algorithm, which streamlines key-switching via decomposed keys to reduce NTT operations [22], hoisting techniques that cut ModUp costs during bootstrapping [8, 10], and optimized BSGS algorithms for faster bootstrapping [26, 28]. Improved ciphertext encoding further reduces computational overhead for matrix operations [6]. Hardware efforts leverage GPUs for parallelization and memory optimization [13, 18, 19] and FPGAs for custom NTT acceleration [2, 4, 14, 32, 36, 45, 46]. To overcome bandwidth bottlenecks, custom accelerators now combine high-throughput dataflow designs with optimized on-chip storage, enabling scalable FHE deployment [1, 20, 21, 34, 35, 39, 40, 42]. Together, these strides bridge the gap toward practical FHE adoption.

FHE Applications. FHE's evolution has spurred applications across domains. Machine learning leverages FHE for privacy-preserving logistic regression, K-means clustering [15, 48], and secure inference on convolutional neural networks (CNNs) [25, 26, 30]. Natural language processing employs FHE for text classification and sentiment analysis [31, 38], while federated learning integrates FHE to safeguard collaborative model training [44, 47]. These use cases highlight FHE's critical role in balancing data utility and security. By improving FHE performance, industries can advance secure, scalable intelligent systems—bridging privacy and innovation.

9 Conclusion

In this work, we tackled the computational overhead challenges of FHE with a focus on key-switching operations. We introduced **FAST**, a novel FHE accelerator that addresses key challenges in supporting hybrid and KLSS key-switching methods, supporting scalable parallelism and tunable precision, incorporating mainstream optimization algorithms for enhanced efficiency. Our detailed experimental evaluation shows a 44.4% average reduction in latency compared to state-of-the-art accelerators, achieving a 1.13× performance-area improvement. These findings demonstrate the potential of our design in enhancing the practicality and efficiency of FHE for secure cloud computing.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. We thank Dr. Lei Chen, Dr. Yilan Zhu, and Dr. Geng Yang for their invaluable support in improving the quality of this paper. This work is supported by the National Natural Science Foundation of China for Distinguished Young Scholars under Grant No. 62125208, the National Key R&D Program of China under Grant No. 2023YFB4503200, and the Strategic Priority Research Program of Chinese Academy of Sciences under Grant No. XDB0690100.

References

[1] Rashmi Agrawal, Jung Ho Ahn, Flavio Bergamaschi, Ro Cammarota, Jung Hee Cheon, Fillipe DM de Souza, Huijing Gong, Minsik Kang, Duhyeong Kim, Jongmin Kim, et al. 2023. High-precision RNS-CKKS on fixed but smaller word-size architectures: theory and application. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 23–34.

[2] Rashmi Agrawal, Anantha Chandrakasan, and Ajay Joshi. [n. d.]. HEAP: A Fully Homomorphic Encryption Accelerator with Parallelized Bootstrapping. ([n. d.]).

[3] Rashmi Agrawal, Leo de Castro, Chiraag Juvekar, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. MAD: Memory-Aware Design Techniques for Accelerating Fully Homomorphic Encryption. In *The 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE.

[4] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 882–895.

[5] Aikata Aikata, Ahmet Can Mert, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy. 2023. REED: Chiplet-based accelerator for fully homomorphic encryption. *Cryptology ePrint Archive* (2023).

[6] Youngjin Bae, Jung Hee Cheon, Guillaume Hanrot, Jai Hyun Park, and Damien Stehlé. 2024. Plaintext-Ciphertext Matrix Multiplication and FHE Bootstrapping: Fast and Fused. In *Annual International Cryptology Conference*. Springer, 387–421.

[7] Václad E Beneš. 1964. Optimal rearrangeable multistage connecting networks. *Bell system technical journal* 43, 4 (1964), 1641–1656.

[8] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 587–617.

[9] Zvika Brakerski and Vinod Vaikuntanathan. 2014. Lattice-based FHE as secure as PKE. In *Proceedings of the 5th conference on Innovations in theoretical computer science*. 1–12.

[10] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. 2019. Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 395–412.

[11] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2019. A full RNS variant of approximate homomorphic encryption. In *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*. Springer, 347–368.

[12] Austin Ebel, Karthik Garimella, and Brandon Reagen. 2023. Orion: A Fully Homomorphic Encryption Compiler for Private Deep Neural Network Inference. *arXiv preprint arXiv:2311.03470* (2023).

[13] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. Tensorfhe: Achieving practical computation on encrypted data using gpgpu. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 922–934.

[14] Serhan Gener, Parker Newton, Daniel Tan, Silas Richelson, Guy Lemieux, and Philip Brisk. 2021. An fpga-based programmable vector engine for fast fully homomorphic encryption over the torus. In *SPSL: Secure and Private Systems for Machine Learning (ISCA Workshop)*.

[15] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2019. Logistic Regression on Homomorphic Encrypted Data at Scale.. In *AAAI Conference on Artificial Intelligence (AAAI)*. 9466–9471.

[16] Kyoohyung Han and Dohyeong Ki. 2020. Better bootstrapping for approximate homomorphic encryption. In *Topics in Cryptology–CT-RSA 2020: The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24–28, 2020, Proceedings*. Springer, 364–390.

[17] Intak Hwang, Jinyeong Seo, and Yongsoo Song. 2023. Optimizing HE operations via Level-aware Key-switching Framework. *Cryptology ePrint Archive* (2023).

[18] Shutong Jin, Zhen Gu, Guangyan Li, Donglong Chen, Cetin Kaya Koç, Ray CC Cheung, and Wangchen Dai. 2024. Efficient Key-Switching for Word-Type FHE and GPU Acceleration. *Cryptology ePrint Archive* (2024).

[19] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 114–148.

[20] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[21] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse. *arXiv preprint arXiv:2205.00922* (2022).

[22] Miran Kim, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. 2023. Accelerating HE operations from key decomposition technique. In *Annual International Cryptology Conference*. Springer, 70–92.

[23] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. BTS: An accelerator for bootstrappable fully homomorphic encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 711–725.

[24] Shiann-Rong Kuang and Jiun-Ping Wang. 2009. Design of power-efficient configurable booth multiplier. *IEEE Transactions on Circuits and Systems I: Regular Papers* 57, 3 (2009), 568–580.

[25] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022. Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions.. In *International Conference on Machine Learning (ICML)*. 12403–12422.

[26] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. 2022. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access* 10 (2022), 30039–30054.

[27] Yongwoo Lee, Seonyoung Cheon, Dongkwan Kim, Dongyoon Lee, and Hanjun Kim. 2024. Performance-aware scale analysis with reserve for homomorphic encryption. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 302–317.

[28] Yongwoo Lee, Joon-Woo Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and HyungChul Kang. 2022. High-precision bootstrapping for approximate homomorphic encryption by error variance minimization. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 551–580.

[29] Qian Lou, Bo Feng, Geoffrey Charles Fox, and Lei Jiang. 2020. Glyph: Fast and accurately training deep neural networks on encrypted data. *Advances in neural information processing systems* 33 (2020), 9193–9202.

[30] Qian Lou and Lei Jiang. 2021. Hemet: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture. In *International conference on machine learning*. PMLR, 7102–7110.

[31] Amirhossein Ebrahimi Moghaddam, Buvana Ganesh, and Paolo Palmieri. 2024. Privacy-Preserving Sentiment Analysis Using Homomorphic Encryption and Attention Mechanisms. In *International Conference on Applied Cryptography and Network Security*. Springer, 84–100.

[32] Kevin Nam, Hyunyoung Oh, Hyungon Moon, and Yunheung Paek. 2022. Accelerating N-Bit Operations over TFHE on Commodity CPU-FPGA. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 98, 9 pages. https://doi.org/10.1145/3508352.3549413

[33] Jaiyoung Park, Michael Jaemin Kim, Wonkyung Jung, and Jung Ho Ahn. 2022. AESPA: Accuracy preserving low-degree polynomial activation for fast private inference. *arXiv preprint arXiv:2201.06699* (2022).

[34] Adiwena Putra, Yi Chen, John Kim, Joo-Young Kim, et al. 2023. Strix: An End-to-End Streaming Architecture with Two-Level Ciphertext Batching for Fully Homomorphic Encryption with Programmable Bootstrapping. *arXiv e-prints* (2023), arXiv–2305.

[35] Brandon Reagen, Woo-Seok Choi, Yeongil Ko, Vincent T Lee, Hsien-Hsin S Lee, Gu-Yeon Wei, and David Brooks. 2021. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 26–39.

[36] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. 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*. 1295–1309.

[37] Lorenzo Rovida and Alberto Leporati. 2024. Encrypted image classification with low memory footprint using fully homomorphic encryption. *Cryptology ePrint Archive* (2024).

[38] Lorenzo Rovida and Alberto Leporati. 2024. Transformer-based language models and homomorphic encryption: An intersection with bert-tiny. In *Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics*. 3–13.

[39] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 238–252.

[40] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data.. In *ISCA*. 173–187.

[41] Nikola Samardzic and Daniel Sanchez. 2024. BitPacker: Enabling High Arithmetic Efficiency in Fully Homomorphic Encryption Accelerators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 137–150.

[42] Kaustubh Shivdikar, Yuhui Bao, Rashmi Agrawal, Michael Shen, Gilbert Jonatan, Evelio Mora, Alexander Ingare, Neal Livesay, José L Abellán, John Kim, et al. 2023. GME: GPU-based Microarchitectural Extensions to Accelerate Homomorphic Encryption. *arXiv preprint arXiv:2309.11001* (2023).

[43] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P Jouppi. 2008. *CACTI 5.1*. Technical Report. Technical Report HPL-2008-20, HP Labs.

[44] Qipeng Xie, Siyang Jiang, Linshan Jiang, Yongzhi Huang, Zhihe Zhao, Salabat Khan, Wangchen Dai, Zhe Liu, and Kaishun Wu. 2024. Efficiency optimization techniques in privacy-preserving federated learning with homomorphic encryption: A brief survey. *IEEE Internet of Things Journal* 11, 14 (2024), 24569–24580.

[45] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, and Xiaowei Li. 2023. Poseidon: Practical Homomorphic Encryption Accelerator. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 870–881.

[46] Tian Ye, Rajgopal Kannan, and Viktor K Prasanna. 2022. Fpga acceleration of fully homomorphic encryption over the torus. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[47] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. 2020. {BatchCrypt}: Efficient homomorphic encryption for {Cross-Silo} federated learning. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 493–506.

[48] Peng Zhang, Teng Huang, Xiaoqiang Sun, Wei Zhao, Hongwei Liu, Shangqi Lai, and Joseph K Liu. 2022. Privacy-preserving and outsourced multi-party k-means clustering based on multi-key fully homomorphic encryption. *IEEE Transactions on Dependable and Secure Computing* 20, 3 (2022), 2348–2359.