Athena: Accelerating KeySwitch and Bootstrapping for Fully Homomorphic Encryption on CUDA GPU

Yifan Yang[1], Kexin Zhang[1], Peng Xu[1(✉)], Zhaojun Lu[1], Wei Wang[2], Weiqi Wang[1], and Kaitai Liang[3,4]

[1] School of Cyber Science and Engineering, Hubei Key Laboratory of Distributed System Security, Huazhong University of Science and Technology, Wuhan 430074, China
{yangyifan,zhangkexin,xupeng,lzj_cse,hustweiqiwang}@mail.hust.edu.cn
[2] Cyber-Physical-Social Systems Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
viviawangwei@hust.edu.cn
[3] Delft University of Technology, Delft, The Netherlands
Kaitai.Liang@tudelft.nl
[4] University of Turku, Turku, Finland
kaitai.liang@utu.fi

Abstract. Fully Homomorphic Encryption (FHE) enables computation over encrypted data, but it faces significant challenges in practical implementation due to its high computational costs, particularly in HMult, HRot, and Bootstrapping operations. This work presents Athena, an accelerated FHE system built on GPUs with a new algorithm-hardware co-design approach. Specifically, to accelerate HMult, HRot, and Bootstrapping, we redesign their common and expensive operation KeySwitch, based on the KLSS method proposed by Kim et al. in CRYPTO'23, and accelerate its core operations, namely NTT, EBConv, and IP. We further optimize the dataflow of Bootstrapping by reducing redundant EBConv and (I)NTT operations, and by improving the global memory I/O in the double-hoisting-based C2S/S2C operation. Moreover, Athena is designed as a general-purpose system that supports various cryptographic parameters. Experimental results demonstrate that Athena significantly improves the performance of KeySwitch and Bootstrapping. In particular, Athena's accelerated KeySwitch optimizes HMult $2.17\times \sim 4.40\times$ and HRot $1.89\times \sim 4.54\times$ compared to TensorFHE (HPCA'23), Poseidon (HPCA'23), and FAB (HPCA'23), respectively. Besides, Athena's Bootstrapping outperforms TensorFHE by nearly $2.74\times$.

Keywords: Fully Homomorphic Encryption · KeySwitch · Bootstrapping · GPU Acceleration

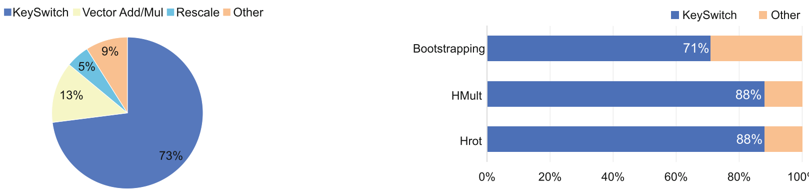
V. Nicomette et al. (Eds.): ESORICS 2025, LNCS 16054, pp. 442–462, 2026.
https://doi.org/10.1007/978-3-032-07891-9_23

1 Introduction

Fully Homomorphic Encryption (FHE) is an advanced cryptography technique that allows users to perform Turing-complete computations on encrypted data. It carries profound implications across a wide range of applications, e.g., privacy-preserving data analysis and secure cloud computing [3,24]. Since Gentry proposed the first FHE [14], both academia and industry have concentrated on improving FHE schemes in terms of practical performance and functionality. Single-Instruction-Multi-Data (SIMD)-like FHE schemes (e.g., BFV [12], BGV [5], and CKKS [9]) enable computations on multiple plaintext elements and support parallel processing across numerous slots. They require up to 100,000x more computational resources than plaintext-based addition and multiplication.

FHE Accelerating Platforms. Several approaches have been proposed to accelerate FHE using CPU [2,26], General-Purpose Graph Processing Unit (GPU) [13,17,28], Field Programmable Gate Arrays (FPGA), and Application Specific Integrated Circuits (ASICs). Among these, CPU-based solutions provide the worst performance primarily due to the CPU's limited computing units and low instruction throughput. While FPGA-based solutions [1,21,22,29] offer more customized designs for computation and memory systems compared to CPUs and GPUs, their limited on-chip resources (even when using multiple FPGA boards [1]) make it challenging to handle complex FHE workloads effectively [1]. ASIC-based solutions [18,19,25] deliver powerful performance in terms of computation latency and throughput but lack flexibility in supporting various FHE schemes and cryptographic parameters. In comparison, GPUs offer significant advantages, including high parallelism, abundant on-chip resources, and ease of software deployment. Therefore, this work considers leveraging GPUs to accelerate FHE computations.

Performance Bottleneck. Recall that SIMD-like FHE schemes are built around three core operations, namely Homomorphic Multiplication (HMult), Homomorphic Rotation (HRot), and Bootstrapping. They enable homomorphic multiplication between ciphertexts, homomorphic rotation of plaintext slots, and refreshing ciphertexts to restore their multiplicative depth, respectively. We observe that KeySwitch is a common and computationally expensive operation to switch secret keys during the execution of HMult, HRot, and Bootstrapping. For



(a) Execution time breakdown of the CKKS-based ResNet-20 inference.

(b) Execution time breakdown of the HMult, HRot, and Bootstrapping.

Fig. 1. Execution time breakdown.

Table 1. Overall comparisons (unit: ms).

Accelerator	Platf.	Modulus	KeySwitch	HMult	HRot	Bootstrapping
100x'21 [17]	NVIDIA V100	64-bit	v2	17.40	16.83	428.93
TensorFHE'23 [13]	NVIDIA A100-40G	32-bit	v2	6.64	6.66	250.45
Phantom'24 [28]	NVIDIA RTX 4090	64-bit	v2	2.29	2.22	×
FAB'23 [1]	Xilinx U280 × 8	64-bit	v2	1.71	1.57	92.4
Poseidon'23 [29]	Xilinx U280	32-bit	v2	3.66	3.31	127.45
Athena	NVIDIA RTX 4090	64-bit	**v3**	**1.51**	**1.47**	**91.46**

Note: (1) Let v2 and v3 denote the Hybrid KeySwitch and the KLSS-based KeySwitch, respectively; (2) All the cryptographic parameters of the above works guarantee at least 128 security level; (3) Athena and Phantom have the same parameters, and we show the best performance of the other works claimed in their papers; (4) Although FAB has the suboptimal performance, its HMult and HRot operations just support 23 multiplication depth, and it just remain 6 multiplicative depth after Bootstrapping; (5) In contrast, all the other works, excluding FAB, support HMult and HRot with ≥ 40 multiplication depth and remain ≥ 16 multiplicative depth after Bootstrapping; (6) The maximum Bootstrapping precision relies on the modulus size, the larger modulus size means that a FHE system can achieve a higher computational accuracy; (7) × means that Phantom does not support Bootstrapping

example, by evaluating the CKKS-based ResNet-20 inference [24] on CIFAR-10 dataset using the OpenFHE [2] library, we see that KeySwitch accounts for over 73% of the total homomorphic inference time, as shown in Fig. 1a. Moreover, KeySwitch also dominates majority time in HMult, HRot, and Bootstrapping as illustrated in Fig. 1b. Therefore, *this work aims to introduce a new perspective to accelerating* KeySwitch *so as to significantly improve FHE performance on GPUs.*

Novelty and Challenge. All prior acceleration efforts, to the best of our knowledge, have focused on the 2nd-generation Hybrid KeySwitch [16]. This type of KeySwitch transforms a ciphertext from a smaller basis to a larger one and then multiplies the transformed ciphertext with a KeySwitch key **swk**. This procedure inevitably requires a large number of (I)NTT operations, resulting in inefficient KeySwitch performance [13,17]. In contrast, this paper adopts the 3rd-generation KLSS KeySwitch [20] to reduce the number of required (I)NTT operations. This reduction comes at the cost of increased computation overhead of EBConv and IP operations in KeySwitch. Moreover, the 3rd-generation KLSS KeySwitch introduces redundant EBConv and (I)NTT operations in the double-hoisting-based C2S/S2C phase of Bootstrapping, which impacts Bootstrapping performance.

Our Contributions. This paper proposes Athena, a GPU-based system for accelerating CKKS FHE computations. Athena focuses on optimizing the core FHE operations KeySwitch and Bootstrapping through an algorithm-hardware co-design approach. Table 1 compares Athena with previous SOTAs, demonstrating that Athena achieves the best performance across HMult, HRot, and Bootstrapping operations. Our main contributions are summarized as follows:

- We redesign KeySwitch based on the KLSS [20] method and optimize the KLSS-based KeySwitch instance to better exploit GPU architectural features. Specifically, we optimize three key computational kernels of the KLSS-based KeySwitch: (1) constructing a pipeline-efficient NTT; (2) reusing intermediate data during EBConv; and (3) alleviating memory bottlenecks in IP.
- We accelerate Bootstrapping by optimizing its C2S/S2C and EvalMod phases. Specifically, we reduce redundant EBConv and (I)NTT operations introduced by the KLSS-based KeySwitch, merge the C2S/S2C's PtMatMultAdd operation into a separate kernel to reduce the Global Memory (GMem) I/O, and reuse the KeySwitch key **swk** in EvalMod phase. In addition, we reduce the plaintext matrix bandwidth by 65.7% in the C2S/S2C phase by encoding the Discrete Fourier Transform (DFT) matrices into KLSS's temporary basis $\mathcal{T}$.
- Finally, we adopt a flexible component-based design to ensure that Athena supports a wide range of cryptographic parameters and evaluate its performance across various GPUs. Athena accelerates HMult by $1.52\times \sim 2.68\times$, $1.93\times \sim 4.40\times$, $1.07\times \sim 2.41\times$, and $2.17\times$ over Phantom [28], TensorFHE [13], Poseidon [29], and FAB [1] on a range of cryptographic parameters, respectively. Similar speedup factors are observed for HRot and HMult, with an execution latency of about 1.5 ms. Furthermore, Athena outperforms the GPU-based SOTA TensorFHE [13] by $2.74\times$ in Bootstrapping under equivalent cryptographic settings.

2 Background

We introduce some fundamental preliminaries, including CKKS, polynomial operations, the KLSS-based KeySwitch [20], and Bootstrapping, which are the core accelerated target of Athena. Table 2 summarizes the frequently used notations.

2.1 CKKS

CKKS supports the computation over encrypted fixed-point complex values without decryption. In CKKS, a plaintext vector $\mathbf{z} = (z_0, z_1, \ldots, z_{N/2-1}) \in \mathbb{C}^{N/2}$ contains $N/2$ elements. The vector $\mathbf{z}$ is encoded into a polynomial $m(X) = \sum_{i=0}^{N-1} a_i X^i \in R_Q$ using Discrete Fourier Transform (DFT) and then scaled up by a factor $\Delta \approx q_i$, where $\{z_i\}$ are called slots, and $\{a_i\}$ are called coefficients in CKKS. A ciphertext, which can be decrypted to m using the secret key s, is represented as a pair of polynomials $\mathbf{ct}(m, s) = (a, b) \in R_Q^2$. CKKS supports numerous homomorphic operations on encrypted data, and we can combine all operations between two ciphertexts flexibly. These operations are as follows:

- $\mathsf{CAdd}(\mathbf{ct_0}(m_0, s), c) \rightarrow \mathbf{ct}(m_0 + c, s)$ adds the input plaintext c to the input ciphertext $\mathbf{ct_0}$;

Table 2. Notation Summary

Symbol	Description
λ	Security level
N	Polynomial dimension of R_Q
L	Maximum depth of a ciphertext
l	Current depth of a ciphertext
K	Number of special basis prime modulus
r'	Number of temporary basis prime modulus
γ	Gadget decomposition length of basis $\mathcal{PQ}_L$
$\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N+1)$	Cyclotomic polynomial ring
$Q_L = \prod_{i=0}^{L} q_i$	Prime modulus product of a ciphertext on max-depth
$Q_l = \prod_{i=0}^{l} q_i$	Prime modulus product of a ciphertext on l-depth
$P = \prod_{i=0}^{K-1} p_i$	Prime modulus product of the special basis
$T = \prod_{i=0}^{r'-1} t_i$	Prime modulus product of the temporary basis
$d = \lceil (L+1)/K \rceil$	Decomposition number on max-depth $\mathcal{Q}_L$
$\beta = \lceil (l+1)/K \rceil$	Decomposition number on $\mathcal{Q}_l$
$\tilde{d}_l = \lceil (l+1+K)/\gamma \rceil$	Gadget decomposition block size of $\mathcal{PQ}_l$
$m, \bar{m}$	A polynomial in the coefficient or the NTT domain
ξ_{2N}	$2N$-th root unity of $\mathbb{Z}_q$
$\mathcal{I} = \{q_0, \cdots, q_{k-1}\}$	RNS basis $\mathcal{I}$ with k modulus
Δ	Scaling factor of a plaintext
$[a]_{\mathcal{I}}$	a (a single value or a polynomial) in RNS basis $\mathcal{I}$
ϕ	An automorphism of a polynomial
$radix$	The decompose parameter of the C2S/S2C matrix, where $\log radix = \lceil (\log N - 1)/n \rceil$, and n is the number of decomposed matrix.
$\mathbf{ct}(m, s) = (a, b) \in R_Q^2$	A ciphertext encrypted a plaintext m by the secret key s

- $\mathsf{CMult}(\mathbf{ct_0}(m_0, s), c) \rightarrow \mathbf{ct}(m_0 \cdot c, s)$ multiplies the input plaintext c to the input ciphertext $\mathbf{ct_0}$;
- $\mathsf{HAdd}(\mathbf{ct_0}(m_0, s), \mathbf{ct_1}(m_1, s)) \rightarrow \mathbf{ct}(m_0 + m_1, s)$ adds two input ciphertexts homomorphically;
- $\mathsf{Rescale}(\mathbf{ct_0}(m, s)) \rightarrow \mathbf{ct}(m, s)$ controls the current factor from Δ' to Δ'/q_l and reduces the depth of the input ciphertext by 1;
- $\mathsf{HMult}(\mathbf{ct_0}(m_0, s), \mathbf{ct_1}(m_1, s)) \rightarrow \mathbf{ct}(m_0 \cdot m_1, s)$ multiplies two input ciphertexts homomorphically;
- $\mathsf{HRot}(\mathbf{ct}_0(m, s), r) \rightarrow \mathbf{ct}(rot_r(m), s)$ rotates the slots of m to the left by r;
- $\mathsf{HConj}(\mathbf{ct}(m, s)) \rightarrow \mathbf{ct}(\bar{m}, s)$ computes the conjugation of plaintext m;
- $\mathsf{HAuto}(\mathbf{ct}(m, s)) \rightarrow \mathbf{ct}(\phi(m), s)$ computes the automorphism ϕ of plaintext m;
- $\mathsf{Bootstrapping}(\mathbf{ct_0}(m, s)) \rightarrow \mathbf{ct}(m, s)$ raises the depth of $\mathbf{ct_0}$ to a new level $l' = L - l_{boot}$ when the multiplication depth of the input ciphertext $\mathbf{ct_0}$ drops to 0, where l_{boot} denotes the multiplication depth cost of Bootstrapping.

Four of these homomorphic operations, e.g. HMult, HRot, HConj, and HAuto, contain the expensive KeySwitch operation. Moreover, HMult and HRot take considerable time in the FHE-based applications. Bootstrapping is the most important and expensive operation in CKKS and is achieved by the composition of various homomorphic operations. Hence, accelerating KeySwitch and Bootstrapping is a major goal of Athena.

2.2 Polynomial Operations

Residue Number System (RNS). The polynomial operations of CKKS rely on cyclotomic polynomial rings $R_Q = \mathbb{Z}_Q[X]/(X^N + 1)$, where N is a power of two. We leverage a series of residue rings with small modulus $R_{q_0} \times R_{q_1} \times \ldots \times$

$R_{q_L} \cong R_Q$ to represent R_Q. Thus, a polynomial in R_Q can be represented as an $(L+1) \times N$ matrix of coefficients.

4-Step Number Theory Transform (NTT). NTT is the main operation to handle polynomial multiplications in lattice-based cryptosystems. Given a polynomial ring R_q, suppose that q is a prime integer satisfying $q = 1 \pmod{2N}$, and twiddle factor ξ_{2N} is the $(2N)$-th root of $\mathbb{Z}_q$. NTT achieves the isomorphism transformation $a(X) \rightarrow \bar{a} = \{a(\xi_{2N}^i)\}_{i\in[0,N-1]}$ from $a \in R_q$ to $\bar{a} \in \mathbb{Z}_q^N$. The computation formula of NTT can be written as a matrix multiplication, namely

$$\bar{\mathbf{a}} = \mathbf{a}^T \times \mathbf{W} \pmod{q} \text{ and } w_{i,j} = \xi_{2N}^{2ij+j} \in \mathbf{W}.$$

The 4-step NTT [10] is a method to decompose N-NTT to many N_1-NTT and N_2-NTT samples, where $N = N_1 \times N_2$. The 4-step NTT has four steps: (1) N_1-NTT computation; (2) Hadamard product; (3) Matrix transpose; (4) N_2-NTT computation. Specifically, the 4-step NTT views the inputted $\mathbf{a}$ as the form of matrix $\mathbf{A}^{N_1 \times N_2}$ and decomposes the twiddle factor matrix $\mathbf{W}$ to $\mathbf{W_1} = \{\xi_{2N_1}^{2ij+j}\}$, $\mathbf{W_2} = \{\xi_{2N}^{2ij+i}\}$, and $\mathbf{W_3} = \{\xi_{2N_2}^{2ij}\}$, where $N = N_1 \times N_2$, and the dimensions of $\mathbf{W_1}$, $\mathbf{W_2}$, and $\mathbf{W_3}$ are $N_1 \times N_1$, $N_1 \times N_2$, and $N_2 \times N_2$, respectively. In this way, we can compute the 4-step NTT as

$$\bar{\mathbf{A}} = ((\mathbf{A} \times \mathbf{W_1}) \odot \mathbf{W_2})^T \times \mathbf{W_3} \pmod{q}.$$

Since both $\mathbf{W_1}$ and $\mathbf{W_3}$ are Vandermond matrices similar as $\mathbf{W}$, we can compute $\times\mathbf{W_1}$ and $\times\mathbf{W_3}$ using the Butterfly method. Compared with the trivial NTT, the 4-step NTT significantly reduces the dependence between internal data. Section 3.1 will describe our 4-step NTT design on GPU in detail.

Exact Basis Conversion (EBConv). EBConv can change the polynomials' basis and was first proposed in [15] without introducing errors. ModUp and ModDown [8] are two basic operations instantiated by EBConv to change and reduce the basis of polynomials, respectively. Since KLSS [20] requires exact basis conversion, we introduce the procedure of EBConv here. Given an input basis $\mathcal{I} = \{q_0, \ldots, q_{r-1}\}$ and an output basis $\mathcal{O} = \{p_0, \ldots, p_{s-1}\}$, $\mathsf{EBConv}_{\mathcal{I}\rightarrow\mathcal{O}}([a]_{\mathcal{I}}) = [a]_{\mathcal{O}}$ converts the RNS representation $[a]_{\mathcal{I}} = (a^{(0)}, \ldots, a^{r-1}) \in \mathbb{Z}_{q_0} \times \cdots \times \mathbb{Z}_{q_{r-1}}$ of an integer $a \in \mathbb{Z}_I$ into $[a]_{\mathcal{O}} \in \mathbb{Z}_{p_0} \times \cdots \times \mathbb{Z}_{p_{s-1}}$ by computing

$$v = \left\lfloor \sum_{i=0}^{r-1} \frac{[a^{(i)} \cdot \hat{q}_i^{-1}]_{q_i}}{q_i} \right\rceil \in \mathbb{Z} \quad \text{and}$$

$$\mathsf{EBConv}_{\mathcal{I}\rightarrow\mathcal{O}}([a]_{\mathcal{I}}) = \left(\sum_{i=0}^{r-1} [a^{(i)} \cdot \hat{q}_i^{-1}]_{q_i} \cdot \hat{q}_i - v \cdot [I]_{p_j} \pmod{p_j} \right)_{0 \le j < s},$$

where $\hat{q}_i = \prod_{i' \neq i} q_{i'} \in \mathbb{Z}$.

2.3 The KLSS-Based KeySwitch

KLSS [20] is an efficient KeySwitch method to reduce (I)NTT execution time via 2D gadget decomposition. KLSS has three bases $\mathcal{P} = \{p_0, \cdots, p_{K-1}\}$, $\mathcal{Q}_L = \{q_0, \cdots, q_L\}$, and $\mathcal{T} = \{t_0, \cdots, t_{r'-1}\}$, where $P = \prod_{p_i \in \mathcal{P}} p_i$, $Q_L = \prod_{q_i \in \mathcal{Q}_L} q_i$, and $T = \prod_{t_i \in \mathcal{T}} t_i$.

The main goal of KeySwitch is to multiply a ciphertext c by a KeySwitch key **swk**, where **swk** consists of $d = (L+1)/K$ parts $\mathbf{swk}_i = (a_i, b_i) \in R^2_{PQ_L}$ where $0 \leq i < d$. Before executing KeySwitch, each polynomial a_i or b_i in $\mathbf{swk}_i$ is split into $\tilde{d} = (K + L + 1)/\gamma$ slices $\mathbf{swk}_i.\mathbf{A}_j$ or $\mathbf{swk}_i.\mathbf{B}_j \in R_{PQ_L[j\cdot\gamma:(j+1)\cdot\gamma]}$, these slices are converted to basis $\mathcal{T}$, and the ciphertext c is similarly divided into d slices $\mathbf{c}_i \in R_{Q_L[i\cdot K:(i+1)\cdot K]}$, where $0 \leq i < d$ and $0 \leq j < \tilde{d}$.

Figure 2 illustrates the dataflow of the KLSS-based KeySwitch [20]. It transforms the ciphertext slices $\mathbf{c}$ from basis $\mathcal{Q}_\mathcal{L}$ to $\mathcal{T}$, computes the Inner Product (IP) between $\mathbf{c}$ and $\mathbf{swk}.\mathbf{A}$ and the IP between $\mathbf{c}$ and $\mathbf{swk}.\mathbf{B}$ both in NTT form, and finally converts the IP results to basis $\mathcal{PQ}_L$ and reduce the converted IP results to basis $\mathcal{Q}_L$ by ModDown, where all basis conversions are executed in coefficient form.

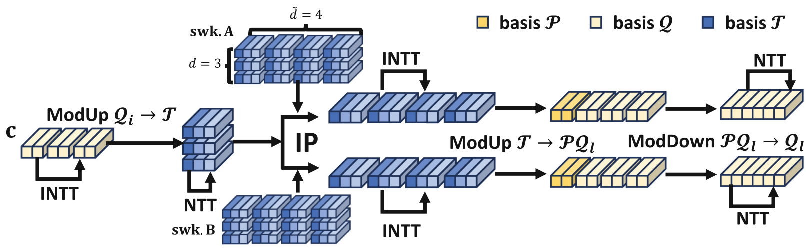


Fig. 2. The KLSS-based KeySwitch dataflow with the parameters $(K, L+1, \gamma, r', d, \tilde{d}) = (2, 6, 2, 3, 3, 4)$. Note that the different colors denote the different bases.

Compared to the Hybrid KeySwitch [16], KLSS significantly improves the performance of KeySwitch by reducing the number of (I)NTT operations. However, its improvement comes at the cost of increased computation overhead on EBConv and IP operations. This paper will design the KLSS-based KeySwitch with high parallelism for significantly enhancing the throughput of KeySwitch on GPU.

2.4 The CKKS Bootstrapping

Bootstrapping is the most critical operation in CKKS to raise the level of a ciphertext from 0 to $L - l_{\mathsf{Boot}}$. It is also the most complex operation in CKKS, as it involves multiple basic homomorphic operations and requires a large amount of memory to store intermediate data. This paper adopts the most advanced Bootstrapping technique [4] to balance the performance, precision, and depth cost. In short, Bootstrapping consists of the following four phases:

- SlotsToCoefficients (S2C) converts the plaintext $m(X)$ from the slots domain to the coefficients domain homomorphically. This phase is equal to multiplying the plaintext DFT matrix and involves $O(radix \log_{radix} N)$ HRot operations by performing the BSGS matrix-vector multiplication [4] with the decomposed DFT matrix [7];
- ModRaise raises the depth of a ciphertext from **ct** to L by bringing the modulus of **ct** from q_0 to Q_L. When decrypting **ct** with the secret key s, we get a different plaintext t satisfying $t(X) = m(X) + q_0 I(X)$;
- CoeffcientsToSlots (C2S) is the inverse of operation S2C to bring the error $I(X)$ introduced in ModRaise from coefficients back to slots;
- EvalMod removes the $q_0 I(X)$ involved by ModRaise. This phase uses the linear polynomial to approximate the nonlinear function $f(x) = \frac{q_0}{2\pi}\sin(\frac{2\pi}{q_0}x) \approx x \pmod 1$ homomorphically.

Moreover, Athena combines C2S/S2C with the feature of KLSS by performing plaintext matrix multiplication on basis $\mathcal{T}$, significantly optimizing the dataflow of Bootstrapping. We will discuss the dataflow optimization in Sect. 3.2.

3 Design of Athena

Figure 3 illustrates an overview of Athena. It includes the polynomial layer, operation layer, and Bootstrapping layer. Athena focuses on optimizing KeySwitch and Bootstrapping. Compared with existing FHE accelerators, Athena accelerates the KLSS-based KeySwitch by redesigning its three important kernels: NTT, EBConv, and IP. Then, Athena optimizes the dataflow of Bootstrapping by combining the C2S/S2C phase with the feature of KLSS and reusing the KeySwitch key of HMult in the EvalMod phase.

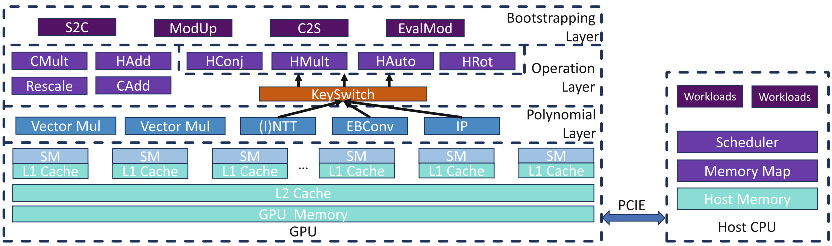


Fig. 3. The system overview of Athena.

3.1 KeySwitchwith High Parallelism

Reduce Pipeline Stalls in NTT.

The SOTA GPU-based work TensorFHE (HPCA'23) [13] states that the various pipeline Read After Write (RAW) stalls between iterations are the main bottleneck of trivial NTT implementation on GPU and take more than 40% time of all the (I)NTT execution time. Hence, TensorFHE implements 32-bit modulus NTT by a naive matrix multiplication and uses the emerging Tensor Core Units (TCUs) to carry the matrix multiplication. However, the TCUs-based method increases the computation complexity of NTT from $O(N \log N)$ to $O(N^2)$. Due to the low computation precision of TCUs, TensorFHE is not applicable to large modulus. For example, the TCUs-based 64-bit modulus NTT involves 64 s8 matrix multiplications and increases computational overhead 4× compared to the TCUs-based 32-bit modulus NTT.

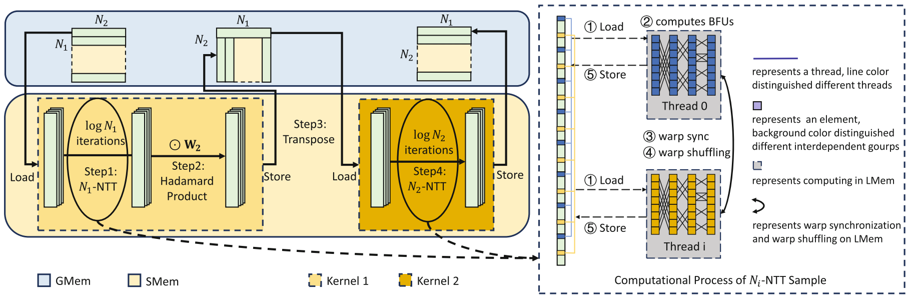


Fig. 4. The 4-step NTT dataflow. Note that: (1) The first three steps are fused into Kernel 1, and the last step occupies Kernel 2; (2) The zoom-in on the right can represent the N_1 and N_2-NTT dataflow; (3) Each warp handles a full N_i-NTT sample, the elements in each N_i-NTT sample are split into different independent groups (eight elements in the same color are in the same group), and each thread processes one interdependent group; (4) Threads are synchronized on warp level, and the data are swapped using warp-shuffling.

Athena addresses the problem of NTT's high RAW stalls on GPU. Specifically, Athena uses 4-step NTT to reduce the data's internal dependence and processes different N_i-NTT on different blocks to achieve NTT pipelines with high performance. To manage the dataflow of N_i-NTT, Athena uses a double-buffering architecture from GLobal Memory (GMem)-Shared Memory (SMem)-Local Memory (LMem) within one block to improve memory access efficiency. To compute N_i-NTT, Athena assigns multiple Butterfly Unit (BFU) based on Shoup modular multiplication tasks of a basic N_i-NTT sample to a single thread and applies the GPU warp-primitive to perform inner-warp data swap. As a result, Athena achieves a highly efficient pipeline across GPU's block-warp-thread level.

We take $N = 2^{16}$ as an example to explain Athena's pipeline-efficient 4-step NTT on the Ada AD102 GPU [23]. Figure 4 shows the procedure of Athena's

4-step NTT. Athena decomposes $N = 2^{16}$ to $N_1 = 2^8$ and $N_2 = 2^8$ and employs two kernels to compute 4-step NTT. It computes N_1-NTT, Hadamard product, and transpose in kernel 1 and computes N_2-NTT in kernel 2. In each kernel, Athena launches 64 blocks, and each block contains 128 threads (equivalent to 4 warps). Athena allocates 1024×8 Bytes SMem for each block and an 8×8 Bytes LMem buffer for each thread to store the intermediate data during the NTT iterations. Athena realizes the four steps of 4-step NTT, mentioned in Sect. 2.2:

- **Step 1**: To compute N_1-NTT, Athena launches kernel 1 and loads four 2^8-NTT samples (containing 1024 elements in total) from GMem to SMem in row-major, and each GPU warp handles one 2^8-NTT sample. Then, each warp sequentially reads its corresponding 2^8-NTT sample, and each thread handles eight elements. During the iterations of N_1-NTT, each thread computes as:
  1. Read its eight dependent elements from SMem to the corresponding LMem buffer;
  2. Execute three times radix-2 butterfly iterations to complete the NTT computation on these eight elements in the LMem buffer;
  3. Synchronize the threads in the same warp via the `__syncwarp()` GPU instruction to maintain data coherency;
  4. Perform on-the-fly data swap between the threads in the same warp using warp-shuffling primitives;
  5. Continue executing NTT iteration until 2^8-NTT completed and write the data back to the SMem.

  For the BFU computations in the above second step, Athena leverages the thread-level memory instruction `cp.async` to concurrently transfer NTT twiddle factors alongside BFU computations to overlap the memory transfer latency, thereby enhancing pipeline utilization. Finally, after eight NTT iterations, kernel 1 completes the N_1-NTT computation and stores the four resulted 2^8-NTT samples in LMem.
- **Step 2**: Compute the Hadamard Product between $(\mathbf{A} \times \mathbf{W_1})$ and $\mathbf{W_2}$ in LMem.
- **Step 3**: Compute the transpose of the result in **Step 2**. Since **Step 4** will read data across blocks, **Step 3** writes the data in LMem back to GMem in transposed form and destroys kernel 1.
- **Step 4**: To compute N_2-NTT, Athena launches kernel 2, reads data from GMem to SMem, computes N_2-NTT like **Step 1**, and writes the results back to GMem.

We state that Athena achieves a highly efficient pipeline across three levels:

- **Block-level**: Distribute different N_i-NTT samples to different blocks and use SMem to store the intermediate data during the iterations;
- **Warp-level**: Distribute N_i-NTT samples in the same block to separate warps and adopt warp-level primitives to perform on-the-fly data swaps and ensure data consistency;
- **Thread-level**: Employ double-buffering strategy to achieve NTT data transfer with high throughput and utilize the `cp.async` instruction to overlap the memory transfer latency of twiddle factors.

Athena achieves parallelism for different modulus and polynomials by launching more blocks. Since the AD102 GPU contains 128 (a multiple of the launched blocks in NTT) Streaming Multiprocessors (SMs), Athena can effectively utilize all SM resources. For other values of N, we adjust the decomposition parameters N_1 and N_2 and the number of sub-NTT samples distributed to each GPU block to obtain the best performance.

Reuse Intermediate Data in EBConv. EBConv contains various computations, like floating-point division FP-Div, 64-bit mulplication u64-Mul, Shoup modular multiplication ModMul, and Barrett reduction ModRed. It is the most complex operation of KeySwitch and introduces many intermediate data to store. Thus, reusing intermediate data in LMem as much as possible is the primary method to improve the performance of EBConv.

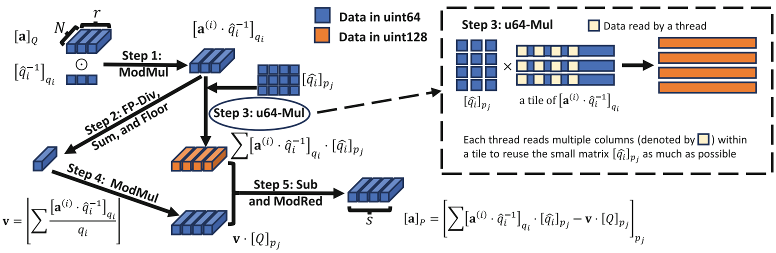


Fig. 5. The EBConv dataflow with high parallelism and data reuse.

Figure 5 illustrates the dataflow of EBConv with an $r \times N$ input matrix $[\mathbf{a}]_Q$ and an $s \times N$ output matrix from basis $\mathcal{Q} = \{q_0, \cdots, q_{r-1}\}$ to basis $\mathcal{P} = \{p_0, \cdots, p_{s-1}\}$. EBConv consists of five steps:

- **Step 1**: Compute Hadamard product between $[\mathbf{a}^{(i)}]_Q$ and $[\hat{q}_i^{-1}]_{q_i}$ by ModMul and output an $r \times N$ matrix;
- **Step 2**: Take the $r \times N$ output matrix of **Step 1** as input, divide the matrix by q_i, sum the r rows of the matrix to obtain an $1 \times N$ matrix, and floor all elements in the $1 \times N$ matrix;
- **Step 3**: Take the $r \times N$ output matrix of **Step 1** as input, multiply the $r \times N$ matrix with the small matrix $[\hat{q}_i]_{p_j}$ by ModMul;
- **Step 4**: Multiply the output of **Step 2** with $[Q]_{p_j}$;
- **Step 5**: Subtract the output of **Step 3** with the output of **Step 4**, apply ModRed to reduce the subtracted result such that all elements are less than p_j, and output $[\mathbf{a}]_P$.

Athena provides high parallelism by partitioning the matrix $\mathbf{a}^{(i)}$ into several tiles with the same size and distributing different tiles to different blocks to balance the parallel workloads. It stores all the intermediate data in the GPU register files to reduce the latency of reading the data as much as possible. In addition, we find that **Step 3** in Fig. 5 involves a small matrix multiplication, where matrix $[\hat{q}_i]_{p_j}$ needs to be loaded many times from GMem. Therefore, Athena lets each thread read multiple columns within a tile to reduce the load times of the matrix $[\hat{q}_i]_{p_j}$.

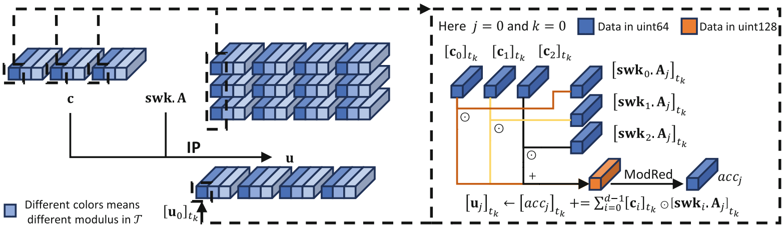


Fig. 6. An example of IP operation. Note that $(d, \tilde{d}, r') = (3, 4, 3)$, $0 \le i < d$, $0 \le j < \tilde{d}$, and $0 \le k < r'$.

Alleviate Memory Bound of IP. Recall that in Sect. 2.3, the KLSS-based KeySwitch adopts IP to compute the Inner Products of the ciphertext polynomial slices $\mathbf{c}$ with the KeySwitch key slices $\mathbf{swk}.\mathbf{A}$ and $\mathbf{swk}.\mathbf{B}$, respectively. Figure 6 gives an example to illustrate the IP operation between $\mathbf{c}$ and $\mathbf{swk}.\mathbf{A}$. In general, IP reads $\mathbf{c}$ and $\mathbf{swk}$ from GMem to LMem and accumulates the i-th product $\mathbf{c}_i \odot \mathbf{swk}_i.\mathbf{A}$ into a 128-bit GPU register acc. After accumulating acc $\beta = \lceil (l+1)/K \rceil$ times, Athena reduces acc by ModRed and writes back the result to GMem, where l is the level of $\mathbf{c}$. Athena distributes the workload of IP operation to several smaller blocks to alleviate the memory bound of the SM's L1 cache. In addition, when executing multiple IP operations with the same $\mathbf{swk}$ (this case will appear when evaluating Chebyshev polynomial in the EvalMod phase of Bootstrapping), Athena can alleviate the memory bound of IP further by reusing the same $\mathbf{swk}$ that has been loaded in LMem.

GPU Kernel Fusion. Recall that the KLSS-based KeySwitch consists of a series of N-NTT, EBConv, and IP operations. N-NTT is inner-vector data dependent. It means that the input vector of N-NTT must be complete before executing N-NTT. On the contrary, EBConv and IP operations are cross-vector data dependent. It means that a part of the former EBConv's output can be handled by the latter IP operation early if existing the successive EBConv and IP operations. This feature is suitable for fusing the successive EBConv and IP operations. However, in practice, the executing series of N-NTT, EBConv, and IP operations must be INTT $\rightarrow$ EBConv $\rightarrow$ NTT $\rightarrow$ IP $\rightarrow$ INTT $\rightarrow$ EBConv $\rightarrow$ NTT. In other words, an (I)NTT operation must be completely executed before performing EBConv and IP operations. Therefore, Athena splits IP, EBConv, and NTT operations into separate GPU kernels.

To reduce the start and destroy cost of the above kernels and the GMem I/O overhead, Athena fuses the three operations into three separate kernels and parallelizes the workloads on various modulus and polynomials. Specifically, it organizes a 3D GPU grid for each kernel to simplify the computation task of each thread in the kernel. For a 3D grid with (x, y, z) dimensions, the x-dimension handles inner-indexing of a polynomial, the y-dimension parallelizes different modulus, the z-dimension enables executing the operations on different RNS polynomials in batch, and all data are read in row-major to maximize the GPU L2 cache hit rate.

3.2 Bootstrappingwith Dataflow Optimization

Recall that Bootstrapping involves four phases (see Sect. 2.4), in which the C2S, EvalMod, and S2C phases take up almost all the time cost of Bootstrapping [17]. Athena focuses on optimizing the dataflow of the C2S/S2C and EvalMod.

Optimize C2S/S2C Phase. Bootstrapping performs a homomorphically linear transformation $\mathbf{A} \cdot \mathbf{ct}(z)$. The only difference between C2S and S2C is the matrix $\mathbf{A}$. Athena uses the (I)DFT matrix decomposition technique to obtain n sparse diagonal matrices and applies the Baby-Step-Giant-Step (BSGS) strategy [7] to reduce the number of HRot operations in each sparse diagonal matrix multiplication. Athena sets $radix = 2^4$ to balance the depth cost and the number of HRot operations, where $\log radix = \lceil (\log N - 1)/n \rceil$. To reduce the computation overhead of sparse diagonal matrix multiplication, Athena applies the double-hoisting strategy [4] with the BSGS parameters of $bs = 2$ and $gs = 16$ to reduce the number of ModDown operations.

Athena combines the double-hoisting strategy with the feature of KLSS. Specifically, Athena performs all plaintext-matrix-multiplication-addition operations PtMatMultAdd in BSGS on basis $\mathcal{T}$ to reduce the number of (I)NTT and EBConv between bases $\mathcal{PQ}_l$ and $\mathcal{T}$. When implementing the C2S/S2C phase, Athena reads the input ciphertext and rotation keys in a permuted order to avoid data permutation across GPU blocks and fuses PtMatMultAdd into a separate GPU kernel to reduce the I/O between LMem and GMem. In addition, it encodes the plaintext matrix to a smaller basis $\mathcal{T}$ rather than basis $\mathcal{PQ}_l$, significantly reducing the plaintext matrix storage and memory bandwidth by $1 - |\mathcal{T}|/|\mathcal{PQ}| = 65.7\%$.

Optimize EvalMod Phase. EvalMod homomorphically computes $\bmod 1$ function. Athena applies the Chebyshev approximation [6] and the double-angle iteration to approximate $\bmod 1$ function to balance the depth consumption, computation overhead, and precision of EvalMod. Specifically, to realize function $\sin(2\pi x)$, Athena applies the Chebyshev method to approximate function $\cos(2\pi \frac{1}{2^r}(x - 0.25))$ and then computes the double-angle formula $\cos(2x) = 2\cos^2(x) - 1$. To reduce the degree of Chebyshev polynomial and the error caused by Runge's Phenomenon [11], it increases the double-angle iterations to 3 and sets the degree of Chebyshev polynomials to be 32.

Athena merges HMult in the Chebyshev evaluation. Figure 7 illustrates the dataflow of the Chebyshev evaluation. Firstly, it calculates the Chebyshev bases iteratively by computing $T_0(X) = 1$, $T_1(X) = X$, and $T_{2^k}(X) = 2T^2_{2^{k-1}}(X) - 1$.

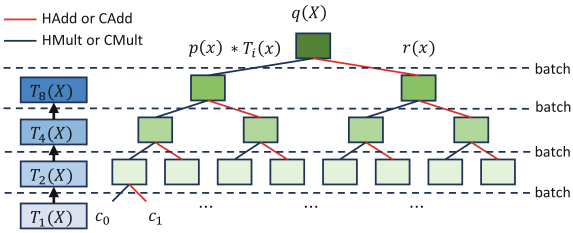


Fig. 7. The dataflow of the 16th-degree Chebyshev evaluation. Note that $T_i(X)$ represents the Chebyshev bases, the binary tree illustrates the computational process, and the tree is evaluated from bottom to top.

Secondly, it constructs a binary tree according to the computation processes of $q(X) = p(X) \cdot T_{2^k}(X) + r(X)$ and adjusts the control flow from the trivial recursive to a bottom-to-top sequential. To enhance the performance of a Chebyshev evaluation, it executes the HMult operations of the same tree level in batch. Specifically, it optimizes the utilization of **swk** in the IP operations by merging the IP operations of the same tree level into the same GPU kernel.

4 Experiment Settings

4.1 Platforms

We implement Athena based on CUDA 12.4, comprehensively evaluate its performance on the server equipped with NVIDIA RTX 4090 and A100 GPUs, Intel Xeon(R) Gold 6338, and 1024GB host memory (optional). The host OS is Ubuntu 22.04. Note that our experiment results mainly rely on RTX 4090 without special mention. We apply Nsight Compute to analyze the pipeline stalls at the micro-architecture level.

4.2 Methodology

We compare Athena with the SOTA, shown in Table 3, and design the following experiments: (1) Analyze the 4-step NTT performance of Athena and compare

Table 3. The SOTAs and their platforms. Note that Phantom is the only open-sourced, and we evaluate it on RTX 4090 for a fair comparison.

Platf.	Accelerator	Hardware	Platf.	Accelerator	Hardware
CPU	Baseline [25]	AMD Ryzen 3975WX	GPU	100x [17]	NVIDIA Tesla V100
FPGA	Poseidon [29]	Xilinx Alveo U280		Phantom [28]	NVIDIA RTX 4090
	FAB [1]	Xilinx Alveo U280 × 8		TensorFHE [13]	NVIDIA A100-SXM-40G
GPU	Athena	NVIDIA RTX 4090			
		NVIDIA A100-PCIe-80G			

the results with TensorFHE [13] and Poseidon [29] on various dimensions; (2) Provide a comprehensive performance of HRot and HMult on various cryptographic parameters and apple-to-apple comparisons with the previous GPU and FPGA-based works; (3) Analyze the optimization of Athena's Bootstrapping and compare it with the SOTAs on the same parameters settings.

5 Evaluation

5.1 NTToptimization Effectiveness

Athena designs an optimized 4-step NTT with Shoup ModMul. This optimization method is also effective in the 4-step NTT with Barrett ModRed in reducing the pipeline stalls. In Fig. 8, we compare our work with the optimized 4-step NTT with Barrett ModRed and the trivial NTT. The results show that our 4-step NTT optimization method can significantly reduce the pipeline stalls 30.6% for both Shoup ModMul and Barrett ModRed compared with the trivial NTT, and our 4-step NTT with Shoup ModMul saves about 28.6% computational overhead compared with both the optimized 4-step NTT with Barrett ModRed and the trivial NTT. Table 4 compares Athena's NTT throughput on RTX 4090 with a CPU baseline [27], TensorFHE [13], and Poseidon [29]. The $\log N$ represents the logarithm of the degree of the polynomials, and Modulus represents the supported maximum modulus size of these works. The results demonstrate that Athena increases the throughput 1057.87×, 1.80×, and 1.53× compared with them, respectively.

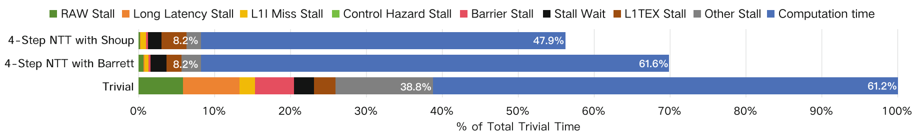


Fig. 8. The pipeline stalls breakdown of three kinds of NTT implementations.

Table 4. (I)NTT throughput on various dimensions (unit: OPS).

	Accelerator	Modulus	$\log N$			
			13	14	15	16
NTT	CPU baseline in [27]	64	6613	3390	1578	769
	TensorFHE [13]	32	3599792	3140055	–	–
	Poseidon [29]	32	–	–	–	548856
	Athena	64	**6410257**	**3606853**	**1562500**	**813504**
INTT	CPU baseline in [27]	64	8695	4118	2057	977
	TensorFHE [13]	32	3592672	3137670	–	–
	Poseidon [29]	32	–	–	–	548856
	Athena	64	**6463341**	**3597122**	**1712329**	**838926**

5.2 KeySwitchOptimization Effectiveness

To accelerate HMult and HRot, Athena focuses on optimizing their key operation KeySwitch. To show the optimization effectiveness more clearly, we compare Athena with the previous works in performing HMult and HRot operations. Table 5 presents the performance of TensorFHE [13], Phantom [28], Poseidon [29], FAB [1], and Athena on various parameters. All these parameters guarantee that CKKS has at least 128-bit security. Since Phantom is the only existing work providing the source code, we evaluate Phantom on all these parameters. The detailed explanation of the parameters is summarized in Table 2. Note that Table 5 does not present the previous work 100x [17] since its performance is much worse than Phantom [28] and TensorFHE [13].

Athena outperforms the state of the art GPU open-source library Phantom [28] $1.52\times \sim 2.68$ and $1.51\times \sim 2.71\times$ in executing HMult and HRot operations, respectively. Compared with the SOTA NVIDIA-GPU work TensorFHE [13], Athena speeds up $\geq 2.90\times$ for both HMult and HRot. When comparing with the SOTA FPGA works Poseidon [29] and FAB [1], Athena outperforms FAB $2.17\times$ and $1.89\times$ for HMult and HRot, respectively. Although FPGA has the customized memory I/O model, which allows FPGA to be much better than GPU in mitigating the memory bound problem, Athena just be slightly slower than Poseidon in executing HRot. Moreover, Athena supports various parameters

Table 5. Comparison: HMult and HRot on various parameters (unit: μs).

Params	Assigned Values							
$\log N$	15		16					
$L+1$	20	23	40	47	44			24
K	4	1	8	1	1			8
γ	8	3	8	3	3			8
r'	8	3	10	3	3			10
Compared Works	Phantom					TensorFHE	Poseidon	FAB
`HMult`	630.7	1696.2	2289.1	11310.3	10009.4	6648.4	3663.0	1710.0
`HRot`	615.5	1678.1	2215.2	11218.5	9911.5	6656.3	3311.3	1570.0
Our Work	Athena							
`HMult`	**318.4**	637.3	**1509.5**	4463.6	3724.3	3440.6 (on A100)		788.1
Speedup	1.98×	**2.66**×	1.52×	2.53×	**2.68**×	1.93×	1.06×	2.17×
`HRot`	**344.8**	671.8	**1467.4**	4399.0	3657.7	3419.6 (on A100)		830.4
Speedup	1.78×	**2.50**×	1.51×	2.55×	**2.71**×	1.92×	0.97×	1.89×

Note: (1) TensorFHE [13] and Phantom [28] are based on GPU, and Posiedon [29] and FAB [1] are based on FPGA; (2) FAB and Phantom support 64-bit modulus, and TensorFHE and Poseidon only support 32-bit modulus; (3) When comparing with TensorFHE, we evaluate Athena on NVIDIA A100 for a fair comparison, while the other results are based on RTX 4090 (not only the parameters listed in Table 5). In contrast, Poseidon supports only one suit of parameters. In addition, Athena has the best performance on the parameters $(L+1, K, \gamma, r') = (40, 8, 8, 10)$ in the case of $\log N = 16$. In this case, Athena outperforms TensorFHE 4.40× and 4.54× in HMult and HRot, respectively. Compared with Poseidon, Athena's HMult and HRot achieve 2.43× and 2.27× improvement, respectively. In this case, we do not compare Athena with FAB since the latter only supports 23 multiplication depth, which is much less than the former's maximum depth of 39.

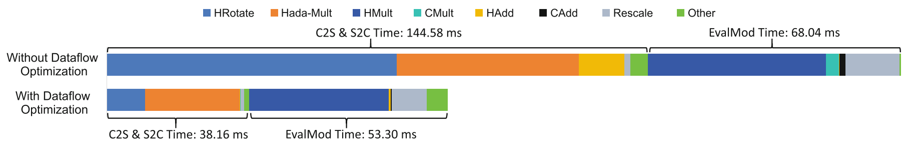


Fig. 9. Homomorphic-operation-level time breakdown of Athena's Bootstrapping on the parameters $(\log N, L+1, K, \gamma, r', radix) = (16, 35, 7, 7, 10, 2^4)$.

5.3 BootstrappingOptimization Effectiveness

After applying Athena's KeySwitch to realize Bootstrapping, Athena further optimizes the dataflow of Bootstrapping. Figure 9 illustrates the time breakdown analysis of Athena's Bootstrapping with or without the dataflow optimization. The left of the dashed line represents the total time cost of C2S and S2C, and the right part represents the total time cost of EvalMod. The results show that Athena's Bootstrapping with the dataflow optimization achieves 2.32× improvement. Specifically, Athena's dataflow optimization significantly reduces the number of EBConv and (I)NTT operations and memory I/O overhead between GMem and LMem in PtMatMultAdd. And Athena's dataflow optimization also improves the utilization of **swk** in multiple HMult's IP kernels, thereby improving the performance of EvalMod.

Table 6. Bootstrapping comparisons (unit: ms). Note that Set-1 and Set-2 denote the parameters $(\log N, L+1, K, \gamma, r', radix) = (16, 35, 7, 7, 10, 2^4)$ and $(\log N, L+1, K, \gamma, r', radix) = (16, 24, 8, 8, 12, 2^4)$, respectively.

Set-1			Set-2	
Athena	100x [17]	TensorFHE [13]	Athena	FAB [1]
91.46	428.93	250.45	52.23	92.40

Table 6 compares Athena with the previous works in the aspect of Bootstrapping performance. The results show that Athena accelerates Bootstrapping performance 4.68× and 2.74× compared with the GPU-baseline 100x [17] and TensorFHE [13] on the same parameter Set-1, respectively. To compare FAB fairly, we evaluate Athena's Bootstrapping on the parameters Set-2, which FAB provides. Although FAB employs 8 FPGA boards, Athena still achieves 1.77× improvement with relatively limited hardware resources. Note that in this part, we do not compare Athena with Poseidon [29] and Phantom [28], since Poseidon's parameters make Bootstrapping key having a huge and impractical size ($\geq$ 150 GB), and Phantom does not support Bootstrapping.

6 Related Works

The GPU-Based Works. 100x [17] is the first GPU-based work to consider CKKS Bootstrapping and successfully accelerated it to more than 100× compared with CPU. It discusses the performance bottleneck in the 2nd CKKS [16] and achieves good performance. TensorFHE [13] points out that NTT is the performance bottleneck of the CKKS HMult, Bootstrapping, and workloads. It profoundly analyzes the serious pipeline stall problem of naive-NTT on GPU and adopts TCUs to implement matrix-multiplication-like NTT to reduce the pipeline stall. It also focuses on improving the throughput of homomorphic operations through kernel-level batch. Phantom [28] is the first GPU-based open-source library that simultaneously implements the 2nd CKKS scheme on GPU and supports various cryptographic parameters. In summary, these works, except Phantom [28], do not support various cryptographic parameters and explore the parameter selection on hardware overhead, while Phantom [28] does not support Bootstrapping.

The FPGA-Based Works. Several previous works have accelerated FHE on FPGA [1,29]. FPGA offers flexible memory access modes that are well-suited to handling memory-bound workloads caused by the large amount of intermediate data in FHE. With customized computing units, FPGAs often have advantages in optimizing core operations such as modular arithmetic and NTT. However, the limited on-chip resources make it challenging for FPGA to handle complex workloads effectively. Although some FPGA-based works [1,29] achieve performance comparable to the GPU-based works, they are restricted to specific FPGA boards, notably, FAB [1] requires eight FPGA boards to match the performance of a single GPU. Furthermore, upgrading FPGA hardware to improve performance is often complex and costly.

7 Conclusion

FHE has profound implications for various fields, including secure data analytics and cloud computing. This paper presents Athena, a GPU-based FHE system. Athena optimizes the 3rd-generation KeySwitch algorithm KLSS by redesigning its three kernels and optimizes the dataflow of Bootstrapping. The results show that Athena provides the best performance on the core operations HMult, HRot, and Bootstrapping, and provides up to 4.40× and 2.74× improvement over the SOTA GPU-based work in HMult and Bootstrapping performance, respectively.

Acknowledgements. We sincerely thank the anonymous reviewers for their constructive comments. This work was supported by National Key Research and Development Program of China (Grant No. 2022YFB4501500 and 2022YFB4501502). Peng Xu and Zhaojun Lu are co-corresponding authors.

References

1. Agrawal, R., et al.: FAB: an FPGA-based accelerator for bootstrappable fully homomorphic encryption. In: IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 – March 1, 2023, pp. 882–895. IEEE (2023)
2. Badawi, A.A., et aal.: Openfhe: open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915 (2022). https://eprint.iacr.org/2022/915
3. Bian, S., et al.: HE3DB: an efficient and elastic encrypted database via arithmetic-and-logic fully homomorphic encryption. In: Meng, W., Jensen, C.D., Cremers, C., Kirda, E. (eds.) Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26–30, 2023, pp. 2930–2944. ACM (2023)
4. Bossuat, J.-P., Mouchet, C., Troncoso-Pastoriza, J., Hubaux, J.-P.: Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In: Canteaut, A., Standaert, F.-X. (eds.) EUROCRYPT 2021. LNCS, vol. 12696, pp. 587–617. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77870-5_21
5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. ACM Trans. Comput. Theory **6**(3), 13:1–13:36 (2014)
6. Chen, H., Chillotti, I., Song, Y.: Improved bootstrapping for approximate homomorphic encryption. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11477, pp. 34–54. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17656-3_2
7. Cheon, J.H., Han, K., Hhan, M.: Faster homomorphic discrete fourier transforms and improved FHE bootstrapping. IACR Cryptol. ePrint Arch. p. 1073 (2018). https://eprint.iacr.org/2018/1073
8. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full RNS variant of approximate homomorphic encryption. In: Cid, C., Jr., M.J.J. (eds.) Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11349, pp. 347–368. Springer (2018). https://doi.org/10.1007/978-3-030-10970-7_16
9. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10624, pp. 409–437. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70694-8_15
10. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Math. Comput. **19**, 297–301 (1965). https://api.semanticscholar.org/CorpusID:121744946

11. Epperson, J.F.: On the runge example. Am. Math. Mon. **94**(4), 329–341 (1987)
12. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch. p. 144 (2012). http://eprint.iacr.org/2012/144
13. Fan, S., Wang, Z., Xu, W., Hou, R., Meng, D., Zhang, M.: Tensorfhe: achieving practical computation on encrypted data using GPGPU. In: IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 – March 1, 2023, pp. 922–934. IEEE (2023)
14. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing, pp. 169–178 (2009)
15. Halevi, S., Polyakov, Y., Shoup, V.: An improved RNS variant of the BFV homomorphic encryption scheme. In: Matsui, M. (ed.) CT-RSA 2019. LNCS, vol. 11405, pp. 83–105. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-12612-4_5
16. Han, K., Ki, D.: Better bootstrapping for approximate homomorphic encryption. In: Jarecki, S. (ed.) CT-RSA 2020. LNCS, vol. 12006, pp. 364–390. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-40186-3_16
17. Jung, W., Kim, S., Ahn, J.H., Cheon, J.H., Lee, Y.: Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 114–148 (2021)
18. Kim, J., Kim, S., Choi, J., Park, J., Kim, D., Ahn, J.H.: SHARP: a short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In: Solihin, Y., Heinrich, M.A. (eds.) Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17–21, 2023, pp. 18:1–18:15. ACM (2023)
19. Kim, J., et al.: ARK: fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In: 55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1–5, 2022, pp. 1237–1254. IEEE (2022)
20. Kim, M., Lee, D., Seo, J., Song, Y.: Accelerating HE operations from key decomposition technique. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 14084, pp. 70–92. Springer (2023). https://doi.org/10.1007/978-3-031-38551-3_3
21. Lu, Z., et al.: An FPGA-based key-switching accelerator with ultra-high throughput for FHE. In: Xiong, J., Wille, R. (eds.) Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2024, Newark Liberty International Airport Marriott, NJ, USA, October 27–31, 2024, pp. 167:1–167:9. ACM (2024)
22. Lu, Z., Yu, W., Xu, P., Wang, W., Zhang, J., Feng, D.: An NTT/INTT accelerator with ultra-high throughput and area efficiency for FHE. In: De, V. (ed.) Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC 2024, San Francisco, CA, USA, June 23–27, 2024, pp. 158:1–158:6. ACM (2024)
23. NVIDIA: NVIDIA ADA GPU ARCHITECTURE (2023). https://images.nvidia.cn/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf
24. Rovida, L., Leporati, A.: Encrypted image classification with low memory footprint using fully homomorphic encryption. Int. J. Neural Syst. **34**(5), 2450025:1–2450025:16 (2024)
25. Samardzic, N., et al.: Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In: Salapura, V., Zahran, M., Chong, F., Tang,

L. (eds.) ISCA 2022: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 – 22, 2022, pp. 173–187. ACM (2022)

26. Microsoft SEAL (release 4.1) (2023). https://github.com/Microsoft/SEAL, microsoft Research, Redmond, WA
27. Shen, S., Yang, H., Liu, Y., Liu, Z., Zhao, Y.: CARM: CUDA-accelerated RNS multiplication in word-wise homomorphic encryption schemes for internet of things. IEEE Trans. Comput. **72**(7), 1999–2010 (2023)
28. Yang, H., Shen, S., Dai, W., Zhou, L., Liu, Z., Zhao, Y.: Phantom: a CUDA-accelerated word-wise homomorphic encryption library. Cryptology ePrint Archive, Paper 2023/049 (2023)
29. Yang, Y., Zhang, H., Fan, S., Lu, H., Zhang, M., Li, X.: Poseidon: practical homomorphic encryption accelerator. In: IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 – March 1, 2023, pp. 870–881. IEEE (2023)