Zephyr: GPU-Efficient Homomorphic Encryption for Privacy-Preserving Transformer Inference

Sieun Seo
Department of Mathematics
Ewha Womans University
sieun1114@ewha.ac.kr

Chohong Min
Department of Mathematics
Ewha Womans University
chohong@ewha.ac.kr

Abstract

Privacy-preserving machine learning (PPML) enables inference over sensitive data without exposing raw inputs, with CKKS being a widely adopted scheme for approximate arithmetic. However, existing CKKS implementations are primarily based on 64-bit residue number system (RNS) representations, creating a mismatch with modern GPUs optimized for 32-bit integer arithmetic. This mismatch introduces substantial computational overhead, limiting the practicality of encrypted transformer inference.

In this work, we present Zephyr, a GPU-efficient framework for homomorphic transformer inference via 32-bit arithmetic and grafting. Zephyr revisits the design of CKKS under GPU constraints and introduces a grafting-based representation that decouples scale management from the modulus chain. By constructing the RNS basis entirely with 30-bit primes and managing scale through auxiliary graft structures, Zephyr enables flexible rescaling while remaining compatible with efficient 32-bit GPU execution.

Compared to Cheddar (Choi et al., ASPLOS'26), a representative GPU-oriented CKKS design based on fixed 25-30 prime systems, our approach simplifies modulus management and enables more flexible operations across different levels, while reducing rescaling overhead at the cost of additional convolution overhead.

We further optimize ciphertext-ciphertext matrix multiplication (CCMM), a major bottleneck in encrypted transformer inference, by eliminating redundant linear transformations and merging overlapping rotation patterns in attention computation. Our theoretical and empirical analysis demonstrates that grafting-based 32-bit CKKS provides a practical and flexible design point for GPU-accelerated PPML inference.

1 Introduction

Recent advances in transformer-based models have enabled powerful applications across a wide range of domains. However, deploying such models in cloud environments raises significant privacy concerns, as sensitive user data must be processed externally. To address these challenges, privacy-preserving machine learning (PPML) has emerged as an important research direction, aiming to enable secure inference without exposing raw data.

Several recent works have explored privacy-preserving neural network inference using cryptographic techniques. Broadly, these approaches can be categorized into interactive protocols based on secure multi-party computation (MPC) [Wu et al., 2024, Kei and Chow, 2025, Cheng et al., 2025, Dong et al., 2025, Gupta et al., 2023, Zhang et al., 2024, Lu et al., 2023], and non-interactive approaches based on fully homomorphic encryption (FHE) [Chen et al., 2022, Park et al., 2026, Jayashankar et al., 2025, Moon et al., 2025, Park et al., 2025, Zhang et al., 2025, Choi et al., 2026]. MPC-based

Preprint.

methods enable efficient inference through collaborative computation among multiple parties, but require multiple rounds of interaction and communication. In contrast, FHE-based approaches allow computation directly on encrypted data without interaction, providing strong privacy guarantees in outsourced settings.

In this work, we focus on FHE-based inference due to its non-interactive nature and strong privacy guarantees. In particular, the CKKS scheme [Cheon et al., 2017, 2018b] is widely adopted because of its support for approximate arithmetic over real numbers, making it suitable for machine learning workloads. While prior studies demonstrate the feasibility of encrypted inference, their performance remains constrained by the limitations of existing FHE implementations.

Most existing implementations rely on widely used FHE libraries [Al Badawi et al., 2022, Halevi and Shoup, 2020, EPFL-LDS and SA, 2024, Chen et al., 2017, CryptoLab, 2023], which are primarily designed around 64-bit residue number system (RNS) representations. Although such designs are well suited for CPU-based execution, they are not well aligned with modern GPU architectures, which natively favor 32-bit integer operations. As a result, 64-bit arithmetic must be emulated using multiple 32-bit operations on GPUs, leading to substantial computational overhead.

To address this mismatch, Cheddar [Choi et al., 2026] introduces a GPU-oriented 32-bit CKKS implementation based on a 25-30 prime system. By constructing the RNS basis using a combination of 25-bit and 30-bit primes, this approach enables efficient execution on 32-bit hardware and achieves significant performance improvements over prior implementations. However, the fixed structure of the 25-30 prime system introduces several inherent trade-offs, including discretized scale control, increased rescaling overhead, and additional complexity in managing heterogeneous modulus chains.

In this work, we explore an alternative design for efficient 32-bit CKKS execution based on Grafting [Cheon et al., 2025]. Instead of encoding scale implicitly through the RNS modulus chain, we adopt a grafting-based representation that decouples scale from the underlying modulus structure. Building on this idea, we propose a 30-graft system that operates entirely over 30-bit primes while maintaining scale via an auxiliary representation, referred to as the graft modulus. Compared to fixed prime-chain approaches, this design offers a simpler and more flexible execution model while remaining compatible with GPU-efficient execution.

In addition, we revisit the implementation of ciphertext-ciphertext matrix multiplication (CCMM), a core operation in encrypted transformer inference. By restructuring the computation to eliminate redundant intermediate steps, we propose an optimized formulation that reduces redundant transformations and key-switching cost.

Our Contributions. The contributions of this work are summarized as follows:

- A 30-graft CKKS design for GPU-efficient encrypted inference is proposed, integrating grafting-based scale management into a fully 30-bit RNS framework compatible with efficient 32-bit execution.
- The trade-offs between the proposed 30-graft system and prior 32-bit CKKS designs, such as Cheddar's 25-30 prime system, are analyzed, highlighting improvements in rescaling efficiency and modulus alignment while identifying the additional convolution overhead introduced by graft-based execution.
- An optimized formulation of ciphertext-ciphertext matrix multiplication (CCMM) is presented, which eliminates redundant transformations and reduces key-switching overhead in encrypted transformer inference.

2 Preliminaries

2.1 CKKS Scheme

Homomorphic encryption (HE) enables computation directly on encrypted data. Among various HE schemes, CKKS supports approximate arithmetic over real and complex numbers, making it well suited for machine learning applications.

In CKKS, ciphertexts are defined over a residue number system (RNS) modulus $Q = \prod_{i=0}^{L} q_i$ and operate on the ring $\mathbb{Z}_{Q_l}[X]/(X^N + 1)$ at level l. Homomorphic multiplication increases the scale,

which is controlled via rescaling operations that consume modulus primes and reduce the level. As computation proceeds, ciphertexts transition to lower levels, reducing the remaining modulus budget available for subsequent operations.

Efficient computation therefore requires careful management of scale and modulus alignment across ciphertexts. Bootstrapping [Cheon et al., 2018a] restores the modulus chain of low-level ciphertexts, enabling continued computation. Due to its high computational cost, improving level efficiency and reducing evaluation overhead remain key challenges in practical CKKS systems.

We provide a more detailed description of CKKS, including RNS representation, scaling behavior, and level management, in Appendix A.

2.2 Encrypted Matrix Computation

Matrix multiplication constitutes a major portion of the computational workload in transformer-based models, and it remains a central operation in encrypted settings as well.

In the homomorphic setting, matrix multiplication is implemented as a combination of ciphertext multiplication, rotation, and addition operations. Among these, plaintext-ciphertext matrix multiplication (CPMM) is primarily used in linear layers and accounts for a significant portion of the overall execution time. In contrast, ciphertext-ciphertext matrix multiplication (CCMM) involves ciphertext multiplication and key-switching operations, resulting in substantially higher evaluation cost than CPMM.

Therefore, the overall performance of encrypted neural network inference is largely determined by the efficiency of these matrix operations, with CCMM being an important target for optimization. As a result, several prior works [Moon et al., 2025, Zhang et al., 2025, 2024, Park et al., 2025] have focused on optimizing encrypted matrix multiplication, particularly CCMM, to improve the efficiency of transformer inference under homomorphic encryption.

2.3 Encrypted Transformer Operations

Transformer computations consist of a sequence of linear and non-linear operations, all of which must be evaluated homomorphically in the encrypted setting.

Linear operations are implemented using CPMM and CCMM. In particular, CCMM is used in attention mechanisms, while CPMM is predominantly used in feed-forward layers. These operations are repeatedly executed and collectively determine the overall computational cost.

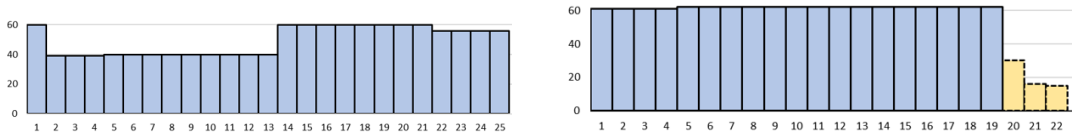
Non-linear functions such as Softmax, LayerNorm, and GELU are evaluated using polynomial approximations, which introduce additional multiplication and rescaling operations. During this process, scale management and modulus transitions accumulate and increase the cost of homomorphic evaluation.

Consequently, efficient encrypted transformer inference requires both optimized matrix operations and efficient scale management.

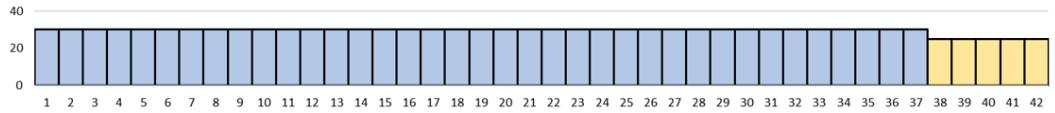
3 GPU-Efficient RNS Design for CKKS

In this section, we review existing RNS-based CKKS designs and discuss the challenges of efficient GPU execution. We then present our GPU-efficient 30-graft system, which combines grafting-based scale management with a 32-bit-friendly modulus structure.

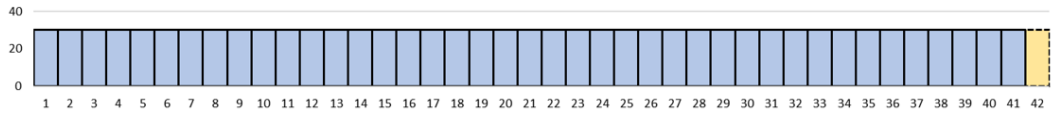
The Residue Number System (RNS) represents large integers as sets of smaller residues, enabling efficient parallel arithmetic and serving as a key component in practical implementations of CKKS. In conventional RNS implementations, as exemplified by Lattigo [EPFL-LDS and SA, 2024], the modulus chain consists of a combination of a general-purpose 2^{40} modulus and heterogeneous bootstrapping moduli of varying sizes. Although these systems are implemented using 64-bit integers, the individual RNS primes are typically smaller than 64 bits, resulting in underutilization of native 64-bit arithmetic.



(a) 64-bit RNS systems: Standard and 15–16–30 graft.



(b) 32-bit RNS system: 25–30.



(c) 32-bit RNS system: Proposed 30-graft.

Figure 1: Level-wise modulus composition of representative CKKS parameter sets. The x-axis corresponds to the RNS prime index i, while the y-axis denotes the bit-size of the corresponding modulus q_i. Blue bars represent main RNS primes, while yellow bars denote auxiliary components.

Alternative RNS Constructions. To address this inefficiency, prior works have explored alternative RNS constructions. BitPacker [Samardzic and Sanchez, 2024] introduces rational rescaling (RR), dynamically adding and removing RNS primes during rescaling to maintain the target scale while minimizing the number of moduli. However, this approach incurs significant memory overhead due to the additional public keys required. In contrast, Grafting [Cheon et al., 2025] designs take a different approach. The **15-16-30 graft system** introduces a grafting technique that enables flexible bit allocation within a 0–60 bit range, as illustrated in Figure 1, reducing the number of RNS components (e.g., from 25 to 23) and achieving a reported 1.2–1.5× speedup. As a result, graft-based designs have been considered among the most efficient approaches for RNS-based homomorphic encryption.

GPU-Efficient 32-bit Arithmetic and Its Challenges. Modern GPUs are significantly more efficient for 32-bit integer arithmetic than for 64-bit operations, where 64-bit multiplication is typically emulated using multiple 32-bit instructions. This architectural difference motivates the use of 32-bit-friendly CKKS constructions. Experimental results in Appendix B show that the performance gap between uint32 and uint64 arithmetic widens as arithmetic intensity increases.

While 32-bit execution offers substantial advantages in compute-bound settings, adapting CKKS to 32-bit arithmetic remains challenging due to constraints imposed by NTT-friendly moduli and scale management.

Prior Design: Cheddar. Based on this observation, Cheddar [Choi et al., 2026] proposes a 32-bit CKKS design based on a **25–30 prime system**. The RNS basis is primarily constructed using 30-bit primes, while introducing 25-bit primes as auxiliary components to support operations involving the conventional 2^{40} modulus and bootstrapping moduli.

For example, a rescaling operation by 2^{40} can be approximated by combining 25-bit and 30-bit primes:

$$\begin{aligned} -40 &= 2 \cdot 25 - 3 \cdot 30 \\ &= 2 \cdot 30 - 4 \cdot 25. \end{aligned}$$

These relations enable flexible scale adjustment through combinations of heterogeneous primes. However, the resulting design fixes the prime composition and tightly couples scale with the modulus chain, leading to a different set of trade-offs in modulus management and scale handling.

In this work, we explore an alternative design that decouples scale from the modulus structure through grafting.

Our Design: 30-Graft System. We propose a **30-graft system** that combines the advantages of the 25–30 system and the 15–16–30 graft system. In our design, each graft modulus is defined as 2^α for $0 \leq \alpha \leq 30$, and is embedded into a ring composed of 30-bit primes:

$$\text{graft: } \mathbb{Z}_{2^\alpha} \hookrightarrow \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \mathbb{Z}_{q_3}.$$

For typical parameter settings with $N \leq 2^{17}$, we have

$$N \cdot (2^\alpha)^2 \leq 2^{77} < q_1 q_2 q_3 \simeq 2^{90},$$

which guarantees that no overflow occurs during multiplication within the $q_1 q_2 q_3$ ring.

This structure aligns naturally with the RNS-based execution of ciphertexts while preserving an explicit representation of scale. Since the graft modulus remains in the coefficient domain during standard homomorphic operations, convolutional multiplication requires an additional transformation into the NTT domain. Consequently, convolution operations in the 30-graft system incur additional overhead compared to the 25–30 system. The corresponding computational complexity is analyzed in Section 4.

Despite this overhead, the proposed design still benefits substantially from efficient 32-bit GPU execution. As illustrated in Figure 1, the standard 64-bit parameter set requires 25 levels, whereas the proposed 30-graft system requires 42 levels. Under a simplified proportional cost model and assuming the idealized arithmetic advantage shown in Figure 4 in Appendix B, this corresponds to a theoretical speedup factor of approximately

$$8 \cdot \frac{25}{42} \simeq 4.76.$$

Although memory-bound operations may not fully realize this benefit and may require additional optimization, the proposed 30-graft system still provides strong practical motivation for GPU-oriented 32-bit CKKS execution.

4 Comparison between the 25–30 system and the proposed 30-graft system

We compare the structural and computational characteristics of the conventional 25–30 system and the proposed 30-graft system. In standard RNS-based CKKS, modulus chains follow a nested structure, where lower-level moduli are subsets of higher-level ones. The 25–30 system breaks this property by combining heterogeneous 25-bit and 30-bit primes across levels, resulting in non-nested modulus bases.

In contrast, the proposed 30-graft system restores a structured progression by using uniform 30-bit primes consumed sequentially, preserving the nested property of modulus chains. Scales exceeding the native range are handled using auxiliary graft moduli of the form 2^α, enabling controlled scale management. Figure 2 illustrates this structural difference between the two systems.

4.1 ModDown

The 25–30 system incurs higher rescaling cost due to larger changes in the RNS basis between adjacent levels. In contrast, the 30-graft system leverages graft moduli and simpler operations, resulting in lower expected operation complexity. Detailed derivations are provided in Appendix C.

4.2 Challenges in Cross-Level Addition

In homomorphic Transformer inference, addition between ciphertexts at different levels frequently occurs, for example in residual connections such as Add-LayerNorm and in polynomial approximations of nonlinear functions. In conventional RNS-based CKKS implementations, such additions can be performed efficiently. This is because the RNS basis at a lower level is always a subset of that at a higher level, allowing higher-level ciphertexts to be reduced to match lower-level ones without additional overhead.

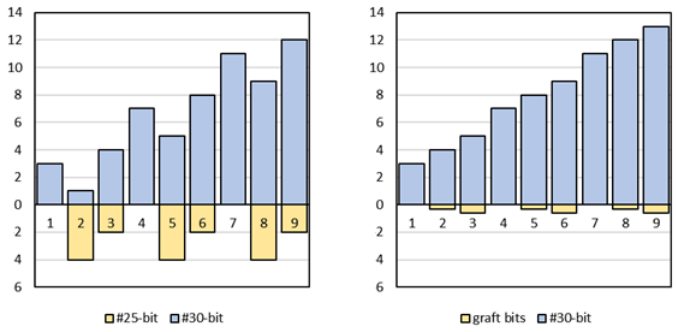


Figure 2: Comparison of modulus structures in the 25–30 and 30-graft systems. The 25–30 system uses heterogeneous primes and results in a non-nested structure, whereas the 30-graft system preserves a nested modulus chain using uniform 30-bit primes with auxiliary graft moduli for scaling.

In contrast, this property does not hold in the 25–30 system. The RNS basis at a lower level is not necessarily contained in that of a higher level. For instance, as illustrated in Figure 2, adding ciphertexts at levels 6 and 8 can be performed by aligning to the level-6 basis. However, when adding ciphertexts at levels 7 and 8, their common basis may only correspond to level 4, forcing the result to drop to a much lower level. Unlike conventional RNS systems, where addition does not consume levels, the 25–30 system may incur excessive level consumption due to this misalignment.

The proposed 30-graft system preserves a structured modulus hierarchy: lower-level 30-bit primes are always contained in higher-level bases. Moreover, graft moduli of the form 2^{α_1} and 2^{α_2} can be aligned by taking the modulus $2^{\min(\alpha_1,\alpha_2)}$, enabling addition in a manner consistent with conventional RNS systems.

4.3 Convolution

In the 25–30 system, convolution can be performed efficiently in the NTT domain. In contrast, the 30-graft system requires additional embedding steps since graft moduli are not NTT-friendly, introducing extra overhead. The detailed complexity analysis is provided in Appendix D.

5 Optimizing Homomorphic Matrix Multiplications

Matrix multiplication dominates the computational cost of LLM inference, both in plaintext and privacy-preserving settings [Zhu et al., 2024, De Castro et al., 2025]. In homomorphic encryption-based PP-LLMs, matrix operations are among the most expensive components, alongside bootstrapping [Moon et al., 2025, Park et al., 2025].

In this setting, matrix multiplication is categorized into two types: ciphertext–plaintext matrix multiplication (CPMM) and ciphertext–ciphertext matrix multiplication (CCMM).

LLM data is determined by the sequence length and hidden dimension, and is typically processed in blocks of size 128×256. In CKKS, such data can be packed into a single ciphertext of size $2^{15} = 128 \times 256$. Various packing strategies exist, including column packing, diagonal packing [Moon et al., 2025], and modified column packing [Park et al., 2025], with the latter providing the best efficiency among them.

CPMM denotes the multiplication between an encrypted x and a plaintext weight W. For a block $x \in \mathbb{R}^{128\times256}$ and a weight matrix $W \in \mathbb{R}^{256\times256}$, the product $y = xW$ can be expressed, under modified column packing, as a linear transformation between packed vectors m_x and m_y of size 2^{15}. This corresponds to multiplication by a transformed matrix $\widetilde{W}$, which requires 256 distinct shifts. Details on constructing $\widetilde{W}$ are provided in Appendix E.

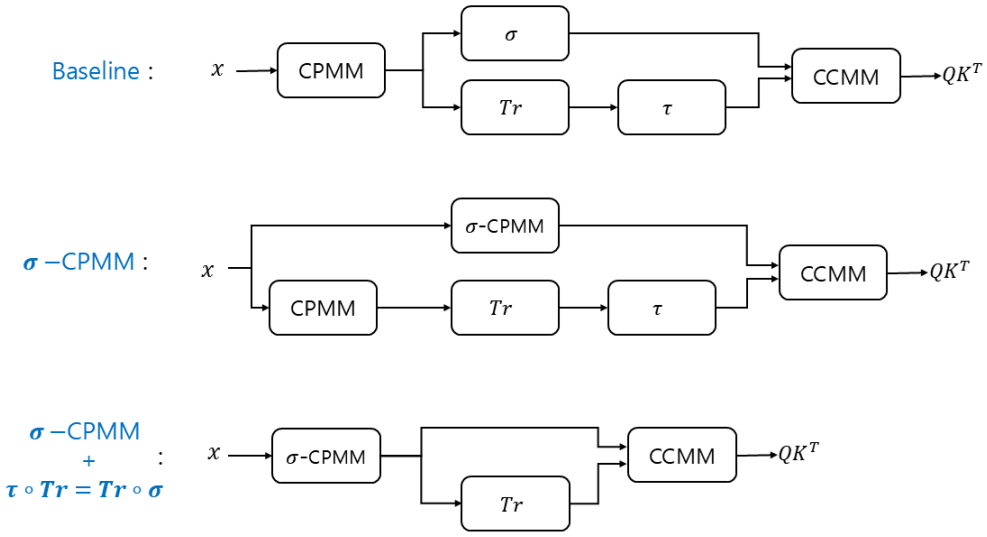


Figure 3: Step-by-step optimization of score computation: In the first step, the transformation σ is merged into CPMM, resulting in a single σ-CPMM operation. In the second step, we exploit the identity $\tau \circ \mathrm{Tr} = \mathrm{Tr} \circ \sigma$ to eliminate the explicit computation of τ, further reducing the computation to a single σ-CPMM operation. This optimization not only improves runtime by eliminating redundant transformations, but also reduces level consumption in homomorphic evaluation.

CCMM denotes multiplication between two encrypted matrices. For $A, B \in \mathbb{R}^{128\times256}$, we apply transformations $A' = \sigma(A)$ and $B' = \tau(B)$, defined by the Transformer head size k. The product is then computed as $A \cdot B = \sum_{l=0}^{k-1} \phi^l(A') \odot \psi^l(B')$.

Table 6 summarizes the number of shifts required for these transformations, including σ, τ, ϕ^l, ψ^l, and transpose operations, as well as those in CPMM.

The computational cost of each transformation is proportional to the number of rotations. CPMM-related transformations require a large number of shifts (e.g., 256) and dominate the overall cost, while operations such as ϕ^l involve relatively few rotations. In addition, τ is approximately twice as expensive as σ, making it a major contributor to CCMM complexity.

5.1 σ−CPMM

In multi-head self-attention, the score matrix QK^T is computed through a sequence of CPMM and CCMM operations. In prior approaches, the computation of $\times W_Q$ and the transformation σ are performed independently, as shown in the baseline configuration in Figure 3.

However, we observe that all rotation patterns required for σ are already included in those of the linear transformation corresponding to W_Q. Therefore, the two operations can be merged into a single linear transformation without increasing the number of rotations (see Algorithm 2). As a result, the σ transformation can be absorbed into the $\times W_Q$ computation without additional cost, as illustrated in the second diagram of Figure 3. Details on constructing the transformed matrix are provided in Appendix F.

5.2 Commutation Relation

On the other hand, the rotation patterns of τ do not overlap with those of $\times W_K$, and thus cannot be directly merged without increasing computational cost. Instead, we exploit the commutation relation

$$\tau \circ \mathrm{Tr} = \mathrm{Tr} \circ \sigma,$$

which allows us to eliminate the explicit application of τ in the computation of K. As a result, the optimized computation of the score matrix reduces the number of required linear transforma-

Table 1: Comparison of QK^T computation for a sequence length of 128 and a hidden dimension of 768. QK^T-1 denotes the configuration where σ is merged into CPMM, and QK^T-2 additionally applies the commutation relation $\tau \circ \mathrm{Tr} = \mathrm{Tr} \circ \sigma$.

Metric	CPMM (Q, K)	σ-CPMM (Q, K)	$QK^\top$ Baseline	$QK^\top$-1	$QK^\top$-2
#KS	93	93	499	470	426
Runtime (ms)	363	363	924	891	853

Table 2: Comparison of the number of key-switching operations in homomorphic encryption implementations for core self-attention operations.

Category	Operation	Bolt	Thor	Powerformer	Zephyr (ours)
CPMM	$\times W_Q,\ W_K,\ W_V$	288	180	122	141
	$\times W_O$	33684	936	731	705
CCMM	$Q \cdot K^T,\ \times V$	168	118	75	78

tions, as illustrated in the third diagram of Figure 3. This optimization not only improves runtime by eliminating redundant transformations, but also reduces level consumption in homomorphic evaluation.

6 Performance Evaluation

In this paper, we presented a GPU-oriented 32-bit 30-graft system and an optimized implementation of $Q \cdot K^T$, a core operation in self-attention. We evaluate the proposed techniques against prior work. Since most FHE-based PP-LLM studies have been conducted on BERT-based models, we select representative systems, including Bolt [Pang et al., 2024], Thor [Moon et al., 2025], and Powerformer [Park et al., 2025], as baselines for comparison. All experimental results are obtained on NVIDIA RTX 4090 and NVIDIA A100 (80GB) GPUs.

6.1 Self-Attention

In evaluating homomorphic encryption implementations, a key metric is the number of key-switching operations, as it is directly proportional to the overall execution time. Table 2 reports a comparison of CPMM and CCMM operations, which constitute the dominant computational cost in self-attention.

In the $Q \cdot K^T$ computation, the σ and τ operations are eliminated, reducing the number of CCMM key-switching operations compared to prior work. In contrast, the number of key-switching operations for CPMM increases.

For linear transformations with N diagonal vectors under the baby-step giant-step (BS–GS) strategy, the configuration that minimizes the total number of switching keys is typically $\sqrt{N} \times \sqrt{N}$. However, the configuration that minimizes the number of key-switching (KS) operations differs from this choice and generally requires a larger number of switching keys. In our design, we prioritize minimizing the total number of SWKs, which leads to an increased number of KS operations. Despite this trade-off, our approach enables efficient self-attention computation. Table 2 shows that our method reduces the number of SWKs required for the QK^T computation.

6.2 End-to-End Inference

We implement end-to-end BERT inference on the proposed 30-graft system, including the optimized CCMM-based QK^T computation. Table 3 reports a comparison of execution times with prior work on the RTE benchmark.

Both Thor and Powerformer, used as baselines in our comparison, are FHE implementations based on the Lattigo RNS system with GPU-accelerated 64-bit integer arithmetic. As reported in Section 3, under an identical implementation assumption, the proposed 30-graft system is expected to achieve approximately ×4.76 speedup. The results in Table 3 show that our approach delivers about ×3.18

Table 3: Breakdown of execution time (in seconds) compared to Thor and Powerformer. All results for the proposed method are measured on NVIDIA A100 GPUs. Softmax and LayerNorm runtimes are not reported for Thor because it employs different implementations for these operations, making direct comparison less meaningful.

Operation	Thor	Powerformer	Ours (A100)	Thor/Ours	Powerformer/Ours
Attention layer	49.77	57.65	12.92	3.85	4.46
Attention score	32.53	28.76	10.00	3.25	2.87
Softmax	–	0.75	0.37	–	2.02
Attention heads	20.63	18.95	10.26	2.01	1.84
Multi-head attention	27.43	22.54	5.41	5.07	4.16
LayerNorm1	–	0.37	0.26	–	1.42
FC1	49.80	59.21	13.92	3.57	4.25
GELU	29.42	8.31	5.21	5.64	1.59
FC2	49.19	43.77	12.41	3.96	3.52
LayerNorm2	–	0.30	0.16	–	1.87
Bootstrappings	337.86	103.72	36.54	9.24	2.83
Total	**623.39**	**344.32**	**107.94**	**5.77**	**3.18**

speedup across linear operations, non-linear operations, and bootstrapping. This gap arises from memory-bound effects and implementation overheads. The observed speedup is influenced by differences in implementation details, making it difficult to assume identical execution conditions across all systems.

As described earlier, all BS–GS configurations for linear operations are designed under the 24GB memory constraint. As a result, the full end-to-end inference can be executed on an NVIDIA RTX 4090 with 24GB memory. Our implementation achieves approximately $1.65\times$ speedup compared to an NVIDIA A100 80GB. This result is based on a detailed comparison with RTX 4090 24GB execution, with further analysis provided in Appendix G.

These results demonstrate that the proposed approach provides overall performance improvements across both low-level cryptographic operations and high-level transformer workloads, despite the additional overhead introduced in certain components. While the benefits of 32-bit arithmetic are most pronounced in compute-bound operations, the combined effect of arithmetic efficiency and algorithmic optimization leads to substantial end-to-end acceleration.

7 Conclusion

In this work, we presented Zephyr, a GPU-efficient framework for privacy-preserving transformer inference based on CKKS. Our approach combines a grafting-based modulus design with GPU-friendly 32-bit execution, enabling a more flexible and structured representation than prior 32-bit CKKS designs.

We introduced a 30-graft system that decouples scale management from the RNS modulus chain while maintaining compatibility with efficient 32-bit GPU arithmetic. We further optimized ciphertext–ciphertext matrix multiplication (CCMM) by eliminating redundant transformations and reducing key-switching overhead.

Our method enables end-to-end inference under constrained GPU memory, supporting execution on an RTX 4090 (24GB) while achieving approximately $1.65\times$ speedup over an A100 (80GB). This highlights the effectiveness of memory-efficient, 32-bit-friendly CKKS design on modern consumer GPUs. Experimental results demonstrate improved efficiency in encrypted transformer workloads, particularly in attention computation and end-to-end inference.

Despite these gains, memory-bound operations benefit less from reduced arithmetic precision, emphasizing the importance of GPU-aware kernel optimization. Future work includes developing more efficient GPU kernels and further co-design of homomorphic computation and model architectures for scalable PP-LLM inference.

References

Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. Openfhe: Open-source fully homomorphic encryption library. In *proceedings of the 10th workshop on encrypted computing & applied homomorphic cryptography*, pages 53–63, 2022.

Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *Advances in Cryptology–EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I*, pages 587–617. Springer, 2021.

Hao Chen, Kim Laine, and Rachel Player. Simple encrypted arithmetic library-seal v2. 1. In *International conference on financial cryptography and data security*, pages 3–18. Springer, 2017.

Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 34–54. Springer, 2019.

Tianyu Chen, Hangbo Bao, Shaohan Huang, Li Dong, Binxing Jiao, Daxin Jiang, Haoyi Zhou, Jianxin Li, and Furu Wei. The-x: Privacy-preserving transformer inference with homomorphic encryption. In *Findings of the association for computational linguistics: ACL 2022*, pages 3510–3520, 2022.

Ke Cheng, Yuheng Xia, Anxiao Song, Jiaxuan Fu, Wenjie Qu, Yulong Shen, and Jiaheng Zhang. Mosformer: Maliciously secure three-party inference framework for large transformers. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 4124–4138, 2025.

Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437. Springer, 2017.

Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part I 37*, pages 360–384. Springer, 2018a.

Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. *Selected areas in cryptography : annual international workshop, SAC proceedings. SAC*, 11349:347–368, 2018b. URL `https://api.semanticscholar.org/CorpusID:52977564`.

Jung Hee Cheon, Hyeongmin Choe, Minsik Kang, Jaehyung Kim, Seonghak Kim, Johannes Mono, and Taeyeong Noh. Grafting: decoupled scale factors and modulus in rns-ckks. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 1098–1112, 2025.

Wonseok Choi, Jongmin Kim, and Jung Ho Ahn. Cheddar: A swift fully homomorphic encryption library designed for gpu architectures. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 35–49, 2026.

CryptoLab. Heaan private ai homomorphic encryption library. `https://heaan.it`, 2023.

Leo De Castro, Daniel Escudero, Adya Agrawal, Antigoni Polychroniadou, and Manuela Veloso. Encryptedllm: Privacy-preserving large language model inference via gpu-accelerated fully homomorphic encryption. In *International Conference on Machine Learning (ICML)*, 2025.

Ye Dong, Wen-jie Lu, Yancheng Zheng, Haoqi Wu, Derun Zhao, Jin Tan, Zhicong Huang, Cheng Hong, Tao Wei, Wen-Guang Chen, et al. Puma: Secure inference of llama-7b in five minutes. *Security and Safety*, 4:2025014, 2025.

EPFL-LDS and Tune Insight SA. Lattigo v6. Online: `https://github.com/tuneinsight/lattigo`, 2024. August 2024.

Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. Sigma: Secure gpt inference with function secret sharing. *Cryptology ePrint Archive*, 2023.

Shai Halevi and Victor Shoup. Design and implementation of helib: a homomorphic encryption library. *Cryptology ePrint Archive*, 2020.

Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, pages 364–390, Cham, 2020. Springer International Publishing. ISBN 978-3-030-40186-3.

Kyoohyung Han, Minki Hhan, and Jung Hee Cheon. Improved homomorphic discrete fourier transforms and fhe bootstrapping. *IEEE Access*, 7:57361–57370, 2019. doi: 10.1109/ACCESS.2019.2913850.

Siddharth Jayashankar, Joshua Kim, Michael B Sullivan, Wenting Zheng, and Dimitrios Skarlatos. A scalable multi-gpu framework for encrypted large-model inference. *arXiv preprint arXiv:2512.11269*, 2025.

Andes YL Kei and Sherman SM Chow. Shaft: Secure, handy, accurate and fast transformer inference. In *NDSS*, 2025.

Seonghak Kim, Minji Park, Jaehyung Kim, Taekyung Kim, and Chohong Min. Evalround algorithm in ckks bootstrapping. In *Advances in Cryptology–ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part II*, pages 161–187. Springer, 2023.

Wen-jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Cheng Hong, Kui Ren, Tao Wei, and WenGuang Chen. Bumblebee: Secure two-party inference framework for large transformers. *Cryptology ePrint Archive*, 2023.

Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. Thor: Secure transformer inference with homomorphic encryption. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 3765–3779, 2025.

Qi Pang, Jialin Zhu, Hanna Möllering, Can Wang, Honglin Yuan, Xiao Huang, Cong Ding, Wenmei Xiong, and Xiaoqian Jiang. Bolt: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 130–147. IEEE, 2024.

Dongjin Park, Eunsang Lee, and Joon-Woo Lee. Powerformer: Efficient and high-accuracy privacy-preserving language model with homomorphic encryption. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11090–11111, 2025.

Jaiyoung Park, Sejin Park, Jai Hyun Park, Jung Ho Ahn, Jung Hee Cheon, Guillaume Hanrot, Jung Woo Kim, Minje Park, and Damien Stehlé. Scaling up privacy-preserving ml: A ckks implementation of llama-2-7b. *arXiv preprint arXiv:2601.18511*, 2026.

Nikola Samardzic and Daniel Sanchez. 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*, pages 137–150, 2024.

Sieun Seo and Chohong Min. Optimal design of key switching parameters for efficient ckks bootstrapping. *IEEE Access*, 2025.

Hyewon Sung, Sieun Seo, Taekyung Kim, and Chohong Min. Evalround+ bootstrapping and its rigorous analysis for ckks scheme. *IEEE Access*, 2025.

Haoqi Wu, Wenjing Fang, Yancheng Zheng, Junming Ma, Jin Tan, Yinggui Wang, and Lei Wang. Ditto: Quantization-aware secure inference of transformers upon mpc. *arXiv preprint arXiv:2405.05525*, 2024.

Jiawen Zhang, Xinpeng Yang, Lipeng He, Kejia Chen, Wen-jie Lu, Yinghao Wang, Xiaoyang Hou, Jian Liu, Kui Ren, and Xiaohu Yang. Secure transformer inference made non-interactive. *Cryptology ePrint Archive*, 2024.

Linru Zhang, Xiangning Wang, Jun Jie Sim, Zhicong Huang, Jiahao Zhong, Huaxiong Wang, Pu Duan, and Kwok-Yan Lam. Moai: Module-optimizing architecture for non-interactive secure transformer inference. In *The Fourteenth International Conference on Learning Representations*, 2025.

Rui-Jie Zhu, Yu Zhang, Ethan Sifferman, Tyler Sheaves, Yiqiao Wang, Dustin Richmond, Peng Zhou, and Jason K. Eshraghian. Scalable matmul-free language modeling. *arXiv preprint arXiv:2406.02528*, 2024.

A Detailed Description of CKKS

Homomorphic encryption (HE) enables computation directly on encrypted data without requiring decryption. Among various HE schemes, CKKS supports approximate arithmetic over real and complex numbers, making it particularly suitable for machine learning workloads.

RNS Representation. In CKKS, the ciphertext modulus is represented in residue number system (RNS) form as

$$Q = \prod_{i=0}^{L} q_i,$$

where each q_i is a prime modulus. A ciphertext at level l operates over the modulus

$$Q_l = \prod_{i=0}^{l} q_i.$$

Higher levels correspond to a larger remaining modulus budget.

Ciphertexts are defined over the ring

$$\mathbb{Z}_{Q_l}[X]/(X^N + 1),$$

where N is typically a power of two to enable efficient polynomial arithmetic.

Scaling and Rescaling. Plaintexts are encoded with a scaling factor that determines numerical precision. During homomorphic multiplication, the scale grows multiplicatively. To control this growth, CKKS applies a rescaling operation, which reduces both the scale and the modulus by removing one or more RNS primes.

As computation proceeds, ciphertexts transition to lower levels, reducing the remaining modulus chain available for further operations. This tight coupling between scale and modulus levels is a defining characteristic of CKKS.

Efficient Polynomial Arithmetic. CKKS implementations rely on the Number Theoretic Transform (NTT) to accelerate polynomial multiplication. By converting polynomials into the transform domain, convolution can be performed efficiently via pointwise multiplication.

Level Alignment. Many homomorphic operations require operands to share the same modulus basis and scale. Therefore, efficient level alignment strategies are critical for minimizing overhead in practical implementations.

Bootstrapping. Bootstrapping [Cheon et al., 2018a] refreshes a ciphertext by restoring its modulus chain, enabling further homomorphic computation. Since bootstrapping involves a large number of expensive operations, maximizing the effective level after bootstrapping is essential for overall system efficiency.

As a result, prior works [Bossuat et al., 2021, Chen et al., 2019, Han et al., 2019, Seo and Min, 2025, Sung et al., 2025, Han and Ki, 2020, Kim et al., 2023] have focused on reducing computational overhead and improving level efficiency in CKKS-based systems.

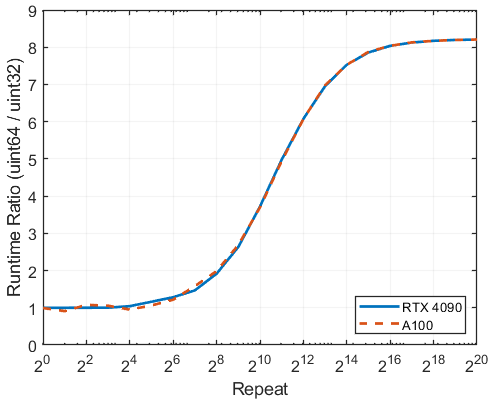


Figure 4: Runtime ratio between uint64 and uint32 repeated multiplication under increasing arithmetic intensity on RTX 4090 and A100 GPUs.

B GPU Efficiency of 32-bit vs 64-bit Arithmetic

Modern GPU architectures exhibit significantly higher efficiency for 32-bit integer arithmetic compared to 64-bit operations. This is because 64-bit multiplication is typically implemented using multiple 32-bit instructions, leading to increased computational overhead.

Experimental Setup. To quantify this effect, we evaluate a repeated multiplication benchmark on NVIDIA RTX 4090 and A100 GPUs. In this benchmark, each loaded value is repeatedly multiplied to control the arithmetic intensity of the kernel, allowing us to isolate compute-bound behavior from memory-bound execution.

Results. Figure 4 shows the runtime ratio between uint64 and uint32 multiplication as a function of arithmetic intensity. As the number of repeated multiplications increases, the overhead of emulated 64-bit arithmetic becomes increasingly pronounced across both GPU architectures.

Discussion. When arithmetic intensity is low, memory accesses and kernel launch overheads dominate execution time, limiting the performance gap between uint32 and uint64 arithmetic. As arithmetic intensity increases, the advantage of 32-bit arithmetic becomes significantly more pronounced, highlighting its importance for compute-bound workloads.

These results demonstrate that while 32-bit CKKS execution can provide substantial performance improvements, memory-bound operations may not fully benefit from reduced arithmetic precision alone.

C Operation count of ModDown

We analyze the cost of rescaling (ModDown) operations across different designs. A ModDown operation that multiplies by m primes and divides by k primes over an RNS system with l basis primes proceeds through a sequence of operations summarized in Table 4. The operation complexity denotes the number of modular multiplications required in each step.

We denote the total operation count as

$$\text{ModDown}_{OP}(l, m, k) = \left(\frac{m+l}{2} \log N + k(m+l+1-k) + m + 2l \right) N.$$

Comparison across designs. Compared to standard RNS and graft-based designs, the 25–30 system exhibits larger changes in the RNS basis between adjacent levels, resulting in higher rescaling cost. For a commonly used 2^{40} scaling factor, the 25–30 system employs $(m, k) = (2, 3)$ and $(3, 4)$

Table 4: Operation complexity of the ModDown procedure over an RNS basis with l primes, multiplying by m primes and dividing by k primes.

ModDown operation	Operation complexity
$l\times$ multiplication	lN
$k\times$ INTT	$k\left(\frac{\log N}{2}+1\right)N$
$(k,\, l+m)\times$ base conversion	$kN+k(l+m-k)N$
$(l+m-k)\times$ NTT	$(l+m-k)\frac{\log N}{2}N$
$(l+m-k)\times$ multiplication	$(l+m-k)N$
Total	$\left(\frac{m+l}{2}\log N+k(m+l+1-k)+m+2l\right)N$

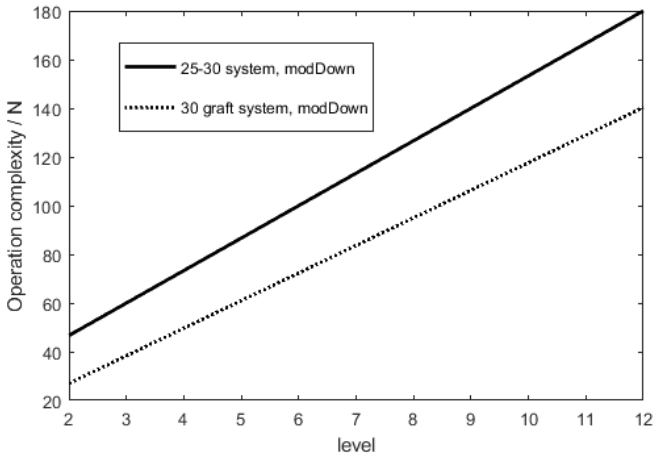


Figure 5: Operation complexity of ModDown as a function of level l. The proposed 30-graft system consistently achieves lower complexity than the 25–30 system due to reduced RNS basis changes and simpler rescaling operations.

with frequencies $\frac{2}{3}$ and $\frac{1}{3}$, respectively. The expected operation complexity is therefore

$$\left(\left(\frac{16}{3}+\frac{\log N}{2}\right)l+\frac{7}{6}\log N+\frac{7}{3}\right)N.$$

In contrast, the proposed 30-graft system handles graft moduli using simple 32-bit integer multiplications rather than modular operations. Depending on the graft parameter α, it uses $(m,k)=(0,1)$ or $(1,2)$ with frequencies $\frac{2}{3}$ and $\frac{1}{3}$, respectively, leading to an expected complexity of

$$\left(\left(\frac{10}{3}+\frac{\log N}{2}\right)l+\frac{\log N}{6}+\frac{1}{3}\right)N.$$

Figure 5 shows that the proposed 30-graft system consistently achieves lower ModDown complexity than the 25–30 system across all levels.

D Operation count of Convolution

Over the ring $Q=q_1\cdots q_l$, where each $q_i\simeq 2^{25}$ or 2^{30}, the 25–30 system stores all RNS components in the NTT domain, enabling convolution to be implemented as a Hadamard product between NTT-domain elements. Therefore, the operation complexity is lN.

In contrast, since the graft modulus 2^α is not an NTT-friendly prime, the 30-graft system requires an embedding step into a larger ring with NTT friendly primes.

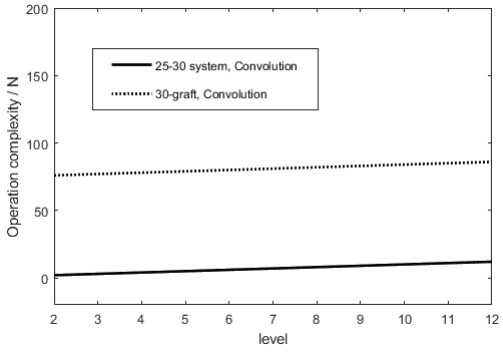
$$\text{graft}: \mathbb{Z}_{2^\alpha}\hookrightarrow \mathbb{Z}_{q_1}\times\mathbb{Z}_{q_2}\times\mathbb{Z}_{q_3}$$

Table 5: Operation complexity of convolution in the proposed 30-graft system.

Convolution operation	Operation complexity
$(l-1)\times$ Hadamard product	$(l-1)N$
$6\times$ NTT	$6 \cdot \frac{\log N}{2} N$
$3\times$ Hadamard product	$3N$
$3\times$ INTT	$3\left(\frac{\log N}{2}+1\right)N$
Total	$\left(l+\frac{9}{2}\log N+5\right)N$

Multiplication is then performed in the q_1, q_2, q_3 NTT domain, followed by a reduction back to the 2^α modulus. This introduces additional computational overhead. The detailed procedure and corresponding operation complexity in the proposed 30-graft system are summarized in Table 5.

Comparing the operation complexity across levels between the 25–30 system and the proposed 30-graft system, the graph below shows that the 30-graft system incurs a consistently higher cost by the NTT–INTT execution time.



E Details of the CPMM algorithm

The size of the weight matrix is determined by the hidden dimension (e.g., 512, 768, or 8192). By decomposing the input into 128×256 blocks, the weight matrix can also be partitioned into 256×256 blocks, and the matrix multiplication can be computed as a sum of block products.

Let $x \in \mathbb{R}^{128\times256}$ and $W \in \mathbb{R}^{256\times256}$. Then, $y = x \cdot W$ is computed as

$$y\,[i]\,[j] = \sum_{k=0}^{255} x\,[i]\,[k] \cdot W\,[k]\,[j]\,.$$

Under the modified column packing, let m_x and m_y denote the packed vectors corresponding to x and y. If $y[i][j] = m_y$ [row] and $x[i][k] = m_x$ [col], then

$$\text{row} = [j]_{64} \cdot 512 + \left\lfloor \frac{j}{64} \right\rfloor \cdot 128 + i \text{ and}$$
$$\text{col} = [k]_{64} \cdot 512 + \left\lfloor \frac{k}{64} \right\rfloor \cdot 128 + i.$$

The shift, defined as the difference between col and row, takes 256 possible values as shown below.

Table 6: Linear transformations required for Transformer inference and their corresponding shift counts in homomorphic implementation. We assume a head size of $k = 64$ and modified column packing.

Linear transform	Shift	#Shifts
CPMM	$128 \times 0, \cdots, 128 \times 255$	256
σ	$512 \times 0, \cdots, 512 \times 63$	64
τ	$-63, \cdots, 63$	127
ϕ^l	$512 \cdot l$	1
ψ^l	$l, -l$	2
Transpose	$-512 \times 63, \cdots, 512 \times 63$	127

$$\begin{aligned}\text{shift} &= 128 \cdot \left(\left([k]_{64} \cdot 4 + \left\lfloor \frac{k}{64} \right\rfloor\right) - \left([j]_{64} \cdot 4 + \left\lfloor \frac{j}{64} \right\rfloor\right)\right) \\ &\in 128 \cdot \{-255, \cdots, 0, \cdots, 255\} \\ &= 128 \cdot \{0, \cdots, 255\} \pmod{2^{15} = 128 \times 256}\end{aligned}$$

Therefore, there exists a weight representation $\tilde{W} \in \mathbb{R}^{256 \times 2^{15}}$ such that

$$m_y = \sum_{d=0}^{255} \tilde{W}[d] \odot \text{Rot}^{128 \times d}(m_x).$$

Here, the matrix $\tilde{W}$ is determined by Algorithm 1.

Algorithm 1 CPMM matrix packing

Require: $W \in \mathbb{R}^{256 \times 256}$
Ensure: $\tilde{W} \in \mathbb{R}^{256 \times 2^{15}}$ such that $m_y = \sum_{d=0}^{255} \tilde{W}[d] \odot \text{Rot}^{128 \cdot d}(m_x)$

```
for j = 0 to 255 do
    for k = 0 to 255 do
        d ← ((j mod 64) · 4 + ⌊j/64⌋ − (k mod 64) · 4 − ⌊k/64⌋) mod 256
        for i = 0 to 127 do
            W̃[d][(k mod 64) · 512 + ⌊k/64⌋ · 128 + i] ← W[j][k]
        end for
    end for
end for
return W̃
```

F Details of the $\sigma-$CPMM algorithm

Let $x \in \mathbb{R}^{128 \times 256}$ and $W \in \mathbb{R}^{256 \times 256}$. Then, $y = \sigma(xW)$ is computed as

$$\begin{aligned}y[i][j] &= (xW)[i]\left[\left\lfloor \frac{j}{64} \right\rfloor \cdot 64 + [i+j]_{64}\right] \\ &= \sum_{k=0}^{255} x[i][k] \cdot W\left[\left\lfloor \frac{j}{64} \right\rfloor \cdot 64 + [i+j]_{64}\right][j].\end{aligned}$$

Under the modified column packing [Powerformer], let m_x and m_y denote the packed vectors corresponding to x and y. If $y[i][j] = m_y$ [row] and $x[i][k] = m_x$ [col], then

$$\text{row} = [j]_{64} \cdot 512 + \left\lfloor \frac{j}{64} \right\rfloor \cdot 128 + i \text{ and}$$
$$\text{col} = [k]_{64} \cdot 512 + \left\lfloor \frac{k}{64} \right\rfloor \cdot 128 + i.$$

The shift, defined as the difference between col and row, takes 256 possible values as shown below. This property is common to both CPMM and σ-CPMM.

$$\begin{aligned}\text{shift} &= 128 \cdot \left(\left([k]_{64} \cdot 4 + \left\lfloor \frac{k}{64} \right\rfloor \right) - \left([j]_{64} \cdot 4 + \left\lfloor \frac{j}{64} \right\rfloor \right) \right) \\ &\in 128 \cdot \{-255, \cdots, 0, \cdots, 255\} \\ &= 128 \cdot \{0, \cdots, 255\} \quad \left(\text{mod } 2^{15} = 128 \times 256\right)\end{aligned}$$

Therefore, there exists a weight representation $\tilde{W}_\sigma \in \mathbb{R}^{256 \times 2^{15}}$ such that

$$m_y = \sum_{d=0}^{255} \tilde{W}_\sigma [d] \odot \text{Rot}^{128 \times d}(m_x).$$

Here, the matrix $\tilde{W}_\sigma$ is determined by Algorithm 2.

Algorithm 2 σ-CPMM matrix packing

Require: $W \in \mathbb{R}^{256 \times 256}$
Ensure: $\tilde{W} \in \mathbb{R}^{256 \times 2^{15}}$ such that $m_y = \sum_{d=0}^{255} \tilde{W}[d] \odot \text{Rot}^{128 \cdot d}(m_x)$

for $j = 0$ to 255 **do**
  for $k = 0$ to 255 **do**
    $d \leftarrow ((j \bmod 64) \cdot 4 + \lfloor j/64 \rfloor - (k \bmod 64) \cdot 4 - \lfloor k/64 \rfloor) \bmod 256$
    for $i = 0$ to 127 **do**
      $t \leftarrow \lfloor k/64 \rfloor \cdot 64 + (i + k) \bmod 64$
      $\tilde{W}_\sigma[d][(k \bmod 64) \cdot 512 + \lfloor k/64 \rfloor \cdot 128 + i] \leftarrow W[t][j]$
    end for
  end for
end for
return $\tilde{W}$

G End-to-End Inference for 24GB GPU Deployment

In the $Q \cdot K^T$ computation, the σ and τ operations are eliminated, reducing the number of CCMM key-switching operations compared to prior work. In contrast, the number of key-switching operations for CPMM increases.

This trade-off arises from our hardware-aware design. For linear transformations with N diagonal vectors under the baby-step giant-step (BS–GS) strategy, the configuration that minimizes the total number of switching keys is typically $\sqrt{N} \times \sqrt{N}$. However, minimizing the number of key-switching (KS) operations generally requires a larger number of switching keys. In our design, we prioritize minimizing the total number of SWKs, which leads to an increased number of KS operations but significantly reduces memory requirements.

24GB Memory-Constrained Design. Unlike prior work that assumes high-memory GPUs such as NVIDIA A100 with 80GB memory, our implementation targets a constrained environment using an NVIDIA RTX 4090 with 24GB memory. By carefully designing BS–GS decompositions to minimize SWKs, our approach enables full end-to-end inference within this memory budget. The corresponding configurations are summarized in Table 7, demonstrating that practical deployment is feasible on commodity GPUs.

Table 7: BS–GS decomposition of matrices executable within a 24GB memory budget.

Category	Operation	Diagonal size	BS–GS decomposition	#SWKs
Transformer	CPMM	256	16×16	30
	CCMM	64	8×8	14
	Transpose	127	16×8	22
	σ	64	8×8	14
	τ	127	8×16	22
	$\{\phi^l\}$	64	8×8	14
	$\{\psi^l\}$	128	$2 \times 8 \times 8$	15
	Subtotal			**77**
Bootstrapping	CTS 1, STC 1	16	8×2	8
	CTS 2, STC 2	31	8×4	10
	CTS 3, STC 3	31	8×4	10
	CTS 4, STC 4	15	8×2	8
	Subtotal			**36**
			Total	**91**

Table 8: Comparison of end-to-end BERT inference runtime on NVIDIA A100 80GB and RTX 4090 24GB.

Operation	Ours (4090)	Ours (A100)	Runtime ratio
Attention layer	7.17	12.92	1.80
Attention score	5.96	10.00	1.67
Softmax	0.22	0.37	1.68
Attention heads	6.12	10.26	1.67
Multi-head attention	4.58	5.41	1.18
LayerNorm1	0.14	0.26	1.85
FC1	7.81	13.92	1.78
GELU	3.04	5.21	1.71
FC2	7.07	12.41	1.75
LayerNorm2	0.09	0.16	1.77
Bootstrappings	22.91	36.54	1.59
Total	**65.11**	**107.94**	**1.65**

Performance on RTX 4090 vs A100. We further compare the runtime of our implementation on RTX 4090 (24GB) and A100 (80GB). Table 8 shows that the RTX 4090 consistently outperforms the A100 across all components, achieving approximately $1.65\times$ speedup in total runtime.

This result highlights that, despite having significantly smaller memory, modern consumer GPUs can provide superior performance when paired with 32-bit-friendly CKKS designs.