SVE-Based Acceleration of Homomorphic Encryption Arithmetic on ARM Neoverse Processors

Massimiliano Donati
Dept. Information Engineering
University of Pisa
Pisa, Italy
massimiliano.donati@unipi.it

Guido Falai
Dept. Information Engineering
University of Pisa
Pisa, Italy
g.falai@studenti.unipi.it

Samuele Bartorelli
Dept. Information Engineering
University of Pisa
Pisa, Italy
s.bartorelli@studenti.unipi.it

Daniele Rossi
Dept. Information Engineering
University of Pisa
Pisa, Italy
daniele.rossi1@unipi.it

Sergio Saponara
Dept. Information Engineering
University of Pisa
Pisa, Italy
sergio.saponara@unipi.it

Abstract*—Homomorphic Encryption (HE) enables computations to be performed directly on encrypted data, providing strong cryptographic security guarantees and preserving data confidentiality in untrusted environments such as cloud computing, high-performance computing, and machine learning platforms. Among existing HE schemes, Cheon-Kim-Kim-Song (CKKS) enables efficient approximate arithmetic over encrypted real-valued data, making it particularly well suited for numerical workloads. However, its practical deployment remains constrained by the substantial computational cost of its core ciphertext operations. In this work, we present a software-based acceleration of CKKS arithmetic kernels on ARM Neoverse processors through Single Instruction, Multiple Data (SIMD) vectorization using the Scalable Vector Extension (SVE). Our approach targets the widely adopted Microsoft SEAL library and introduces SVE-optimized implementations of the key polynomial-arithmetic primitives underlying CKKS ciphertext addition, multiplication, and rotation. By leveraging data-level parallelism, vector-length-agnostic programming, and microarchitectural features of the target processor, the proposed optimizations enhance execution efficiency while preserving functional correctness and compatibility with the existing library interface. Experimental evaluation on an ARM Neoverse V2 platform (NVIDIA Grace processor) demonstrates performance improvements of 8–14% for ciphertext multiplication and rotation, and approximately 30% for addition, across multiple CKKS parameter configurations. These results highlight the potential for more efficient processor-only execution of homomorphic encryption workloads through architecture-aware optimization.

***Index Terms*—Homomorphic Encryption, CKKS, ARM, Neoverse, SVE, SIMD, SEAL.**

I. Introduction

Homomorphic encryption (HE) represents a key enabling technology for security-by-design and privacy-preserving computing paradigms. Unlike traditional cryptographic approaches, which protect data only at rest and in transit, HE enables computation to be performed directly on encrypted data without requiring decryption by the evaluator [1]. This capability preserves confidentiality throughout the entire data lifecycle, effectively addressing a fundamental limitation of conventional secure system architectures, where sensitive data must be exposed in cleartext during computation. Consequently, HE is increasingly recognized as a fundamental technology for secure cloud computing, distributed analytics, and machine learning applications deployed in potentially untrusted environments [2].

Among the available HE schemes, the Cheon–Kim–Kim–Song (CKKS) scheme [3] has gained momentum due to its support for approximate arithmetic over real and complex numbers. CKKS enables the evaluation of arbitrary combinations of addition, multiplication, and rotation operations on encrypted data, thereby enabling efficient execution of numerical workloads with controlled precision loss. This makes it particularly suitable for a wide range of computations [4], including signal processing [5] and machine learning [6].

The CKKS scheme is based on lattice cryptography, specifically the Ring Learning With Errors (RLWE) problem, which is widely considered resistant to quantum attacks [7]. It relies on polynomial arithmetic over residue number systems (RNS), with computationally intensive kernels such as modular multiplication and the Number Theoretic Transform (NTT) dominating execution time. Ciphertext operations, including addition, multiplication, and rotation, are executed through these RNS and NTT computations. Additions are relatively inexpensive, while multiplications require multiple NTTs, modular reductions, and rescaling steps, making them

This work was carried out within the AERO project, funded by Horizon Europe programme under grant agreement 101092850

the primary performance bottleneck, and rotations introduce further overhead due to key-switching procedures. CKKS also supports batching, enabling multiple data elements to be encoded into a single ciphertext and processed in parallel as vector slots [8], according to the Single Instruction Multiple Data (SIMD) paradigm.

To facilitate the practical adoption of HE, several software libraries have been developed. Among them, Microsoft SEAL [9] has become one of the most widely used open-source frameworks, providing an optimized and modular implementation of state-of-the-art schemes such as CKKS. Nevertheless, HE remains significantly slower than cleartext computation, with operations such as ciphertext multiplication often lagging by 2–3 orders of magnitude compared to their plaintext counterparts [10]. This performance gap continues to limit practical adoption and motivates research in both software and hardware acceleration techniques.

A promising direction is the exploitation of data-level parallelism through SIMD vectorization on general-purpose and High Performance Computing (HPC) processors. However, most existing optimizations focus on x86 architectures, while ARM-based server platforms are rapidly emerging in modern infrastructures. In particular, ARM Neoverse processors support Scalable Vector Extension (SVE), which enables vector-length-agnostic SIMD programming and improved portability across different microarchitectures compared to fixed-width extensions such as AVX512 for Intel platforms. Despite this potential, the use of SVE for accelerating HE on CPU-based platforms remains largely underexplored.

In this work, we address this gap by introducing an SVE-based acceleration of CKKS arithmetic kernels on ARM Neoverse processors. Inspired by AVX512-based approaches such as the Intel HEXL library [11], we design optimized SVE microkernels for modular arithmetic and NTT operations targeting ciphertext addition, multiplication, and rotation. The proposed approach leverages data-level parallelism and architecture-specific features while preserving compatibility with the Microsoft SEAL interface, enabling efficient and portable execution of homomorphic encryption workloads on modern ARM-based HPC architectures.

The remainder of the paper is organized as follow: Section II reviews related works; Section III provides some details about CKKS scheme; IV describes the implemented HE kernels; Section V reports the performance results. Finally conclusions are drawn in Section VI.

II. Related works

The performance of homomorphic encryption schemes is largely determined by the efficiency of a small set of arithmetic kernels underlying ciphertext evaluation. In lattice-based schemes such as CKKS, these include polynomial multiplication, forward and inverse NTT, key switching, and RNS modular arithmetic. Since these operations are repeatedly invoked during ciphertext multiplication and rotation, they represent the primary targets for performance optimization.

Several studies have investigated architecture-aware optimizations of these kernels on general-purpose processors. Analyses of the HEAAN scheme, the precursor of CKKS, have shown that the majority of execution time is spent in NTT-based polynomial arithmetic and RNS modular operations, which exhibit significant data-level parallelism but also impose high memory bandwidth requirements [10].

Recent efforts have also focused on vectorized implementations of HE primitives on modern CPU architectures. A notable example is the Intel Homomorphic Encryption Acceleration Library (HEXL), which provides optimized implementations of modular arithmetic and NTT kernels using AVX-512 SIMD instructions on Intel processors [12].

Beyond CPU-based approaches, several works explore Graphics Processing Unit (GPU) acceleration of CKKS kernels, particularly targeting NTT and key switching operations. GPU implementations exploit massive thread-level parallelism and high memory bandwidth to accelerate polynomial transforms and modular arithmetic [13], [14].

Another line of research investigates hardware accelerators for CKKS, including FPGA-based designs and custom architectures. These accelerators typically focus on deeply pipelined implementations of NTT, modular multiplication, and key switching [15]–[17].

Despite these advances, most of the existing work targets GPU platforms, FPGA-based accelerators, or SIMD optimizations for x86 processors. In contrast, architecture-aware vectorization of CKKS kernels on modern ARM server-class processors using SVE remains largely unexplored.

III. Preliminaries on CKKS Scheme

The CKKS scheme is based on the cyclotomic quotient ring $R_q = \mathbb{Z}_q[X]/(X^N + 1)$, where N denotes the polynomial modulus degree and q is the coefficient modulus. Elements of R_q are polynomials of degree N with coefficients within $[0, \ldots, q-1]$. The security of CKKS relies on the hardness of the RLWE problem. The security level (e.g., 128-bit) is roughly proportional to $N/\log q$, while the error distribution also affects security.

In practical implementations, such as Microsoft SEAL, the modulus q is represented as a product of smaller coprime moduli, $q = \prod_{i=0}^{L} q_i$, with each $q_i \leq 2^{60}$. This enables the use of the RNS, where polynomial coefficients are represented as vectors of residues modulo the primes q_i, allowing arithmetic to be executed independently for each modulus on native 64-bit CPU words. Each modulus q_i defines a computation level, and the number of primes sets the maximum multiplicative depth supported by the scheme. On the other hand, the polynomial degree N determines the number of available SIMD slots for batching, equal to $N/2$. Standard libraries typically use power-of-two degrees ($N = 4096$, 8192, 16384, or 32768), so that a single polynomial operation can evaluate multiple data slots simultaneously.

The scheme of a generic CKKS operation is shown in the Figure 1. The cleartext vector $m = [z_0, \ldots, z_{n-1}] \in \mathbb{C}^{N/2}$ is encoded into a polynomial $p(X) \in R_q$ using canonical

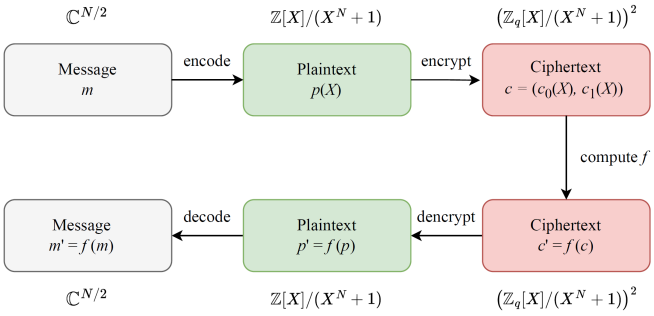

Fig. 1. Simplified scheme of a generic CKKS operation.

embedding, where the polynomial coefficients represent scaled approximations of the original complex numbers. A scaling factor $\Delta = 2^k$ (typically $k \in [30, \ldots, 50]$) and a rounding phase ensure an approximate fixed-point representation, controlling precision loss during computations.

The ciphertext encrypting a plaintext polynomial $p(X)$ typically consists of two polynomials $c = (c_0(X), c_1(X))$ with $c_0, c_1 \in R_q$. In detail, given a secret key polynomial $s(X)$, a random polynomial $a(X)$, and a small polynomial error $e(X)$ sampled from a Gaussian source, the encryption is defined as $c_0(X) = a(X) \cdot s(X) + e(X) + p(X)$ and $c_1(X) = -a(X)$.

Homomorphic operations are applied directly to ciphertexts, producing $c' = f(c)$.

During decryption, the resulting plaintext is recovered by computing $p'(X) = c'_0(X) + c'_1(X) \cdot s(X) = m'(X) + e(X)$. The corresponding cleartext $m' = f(m)$ is then obtained by dividing each coefficient $p'(X)$ by Δ and applying the inverse canonical embedding to map the polynomial back to a vector of complex numbers.

The noise term ensures security but grows during homomorphic operations, limiting the computational depth before the result becomes corrupted. Homomorphic addition, multiplication, and rotation affect the noise and scaling factor differently.

A. Homomorphic Addition

Given two ciphertexts $c_1 = (c_{1,0}, c_{1,1})$ and $c_2 = (c_{2,0}, c_{2,1})$ encrypted under the same parameters and scaling factor, their sum is computed by coefficient-wise modular addition (1).

$$c_{\text{add}} = (c_{1,0} + c_{2,0}, c_{1,1} + c_{2,1}) \in R_q^2 \tag{1}$$

This operation corresponds to the element-wise addition of the underlying cleatext vectors. The scaling factor remains unchanged, while the noise increases slightly, approximately equal to the sum of the individual noise terms of the operands.

B. Homomorphic Multiplication

Multiplication of c_1 and c_2 produces a new ciphertext encrypting the product of the underlying cleartexts involving polynomial multiplication (2).

$$c_{\text{mul}} = (c_{1,0} \cdot c_{2,0}, c_{1,0} \cdot c_{2,1} + c_{1,1} \cdot c_{2,0}, c_{1,1} \cdot c_{2,1}) \in R_q^3 \tag{2}$$

The result contains three polynomial components. To restore the ciphertext to its standard two-component form, a *relinearization* step is applied. It uses an evaluation key (i.e. an encryption of $s(X)^2$) to fold the extra component back into the original two through modular arithmetic. This helps to keep the ciphertext size constant and control noise growth.

Multiplication of two ciphertexts also increases the scaling factor to $\Delta_{mul} \approx \Delta^2$ and amplifies the noise. To deal with it, a *rescaling* step divides the ciphertext coefficients by the last modulus in the modulus chain, restoring the scale approximately to its original value $\Delta_{\text{res}} \approx \frac{\Delta_{\text{mul}}}{q_{\text{last}}} \approx \Delta$. After rescaling, the last modulus q_L is removed from the chain, leaving $q' = \prod_{i=0}^{L-1} q_i$. This limits the remaining multiplicative depth, as each prime q_i supports one level of computation. Multiplications can continue only until the modulus chain is exhausted; beyond that, expensive *bootstrapping* is needed to refresh levels [18].

C. Homomorphic Rotation

Rotation of the packed slots in a ciphertext c_1 by k positions produces a cyclic shift of the underlying cleartext vector by k positions. The operation is implemented using a Galois automorphism σ_k of the polynomial ring R_q. This requires special rotation keys r_k derived from the secret key $s(X)$ (3).

$$c_{1,\text{rot}} = (c_{1,0}^{\text{rot}}, c_{1,1}^{\text{rot}}) \approx \sigma_k(c_1) + \text{relin}(r_k). \tag{3}$$

Here, σ_k maps $X \mapsto X^k \mod (X^N + 1)$ and acts on each polynomial in the ciphertext by permuting its coefficients (4).

$$\sigma_k(c_{1,j}(X)) = \sum_{i=0}^{N-1} c_{1,j,i} X^{ik \bmod N}, \quad j = 0, 1 \tag{4}$$

The rotated ciphertext is relinearized using r_k to preserve the standard two-polynomial structure.

After rotation, the scaling factor remains approximately constant, while the noise increases slightly due to the linear transformations involved.

IV. Implementation of Accelerated Arithmetic

The proposed optimization strategy accelerates the arithmetic kernels that dominate the execution time of ciphertext addition, multiplication, and rotation by exploiting the data-level parallelism and the SVE execution units of the ARM Neoverse processor. These kernels operate on vectors of 64-bit unsigned polynomial coefficients.

Unlike architecture-agnostic implementations such as Microsoft SEAL, which avoid architecture-specific SIMD optimizations, the proposed approach employs SVE-optimized microkernels to efficiently execute CKKS polynomial arithmetic while preserving the library interface, delivering improved performance on modern ARM-based HPC platforms. In addition, SVE enables vector-length-agnostic SIMD computation, where vector registers with runtime-determined length (VL) process arrays under predicate control, eliminating scalar tail handling and ensuring portability across ARM platforms.

A. Vectorized addition

According to (1), the homomorphic addition of two ciphertexts reduces to element-wise modular additions of the N coefficients of the two ciphertext polynomials.

Algorithm 1 implements a vectorized microkernel for this operation. Coefficients are processed in batches of $BS = VL/64$ elements in parallel, where VL denotes the SVE vector length in bits, and 64 bits correspond to the width of each coefficient. For each batch, coefficients from the two input arrays are loaded into vector registers, summed using vector addition, and conditionally reduced modulo q_j using predicated subtraction when the partial sum exceeds the modulus. To perform a complete ciphertext addition, the microkernel is applied independently to the two ciphertext polynomials for each modulus in the modulus chain.

Algorithm 1 Vectorized Modular Addition

Require: coefficient arrays $A[0..N-1], B[0..N-1]$, output array $C[0..N-1]$, modulus q_j, polynomial degree N
Ensure: $C[i] = A[i] + B[i] \bmod q_j$
1: **for** $i \leftarrow 0$ to $N-1$ **step** BS **do**
2: $v_a \leftarrow \texttt{svld1}(A[i])$
3: $v_b \leftarrow \texttt{svld1}(B[i])$
4: $v_c \leftarrow \texttt{svadd}(v_a, v_b)$
5: $\texttt{mask} \leftarrow \texttt{svcmpge}(v_c \geq q_j)$
6: $v_c \leftarrow \texttt{svsub\_m}(\texttt{mask}, v_c, q_j)$
7: $\texttt{svst1}(C[i], v_c)$
8: **end for**

B. Vectorized multiplication

Algorithm 2 Vectorized Modular Multiplication with Barrett Reduction

Require: coefficient arrays $A[0..N-1], B[0..N-1]$, output array $C[0..N-1]$, modulus q_j, polynomial degree N, barrett constants $cr_{\text{lo}}, cr_{\text{hi}}$
Ensure: $C[i] = A[i] \times B[i] \bmod q_j$
1: # Constant vectors preparation
2: $v_{mod} \leftarrow \texttt{svdup}(N)$
3: $v_{cr0} \leftarrow \texttt{svdup}(cr_{lo})$
4: $v_{cr1} \leftarrow \texttt{svdup}(cr_{hi})$
5: **for** $i \leftarrow 0$ to $N-1$ **step** BS **do**
6: $v_a \leftarrow \texttt{svld1}(A[i])$
7: $v_b \leftarrow \texttt{svld1}(B[i])$
8: $z_{lo} \leftarrow \texttt{svmul}(A[i], B[i])$ # mul lower 64 bits
9: $z_{hi} \leftarrow \texttt{svmulh}(A[i], B[i])$ # mul upper 64 bits
10: # Barrett reduction
11: $v_c \leftarrow$ **Vectorized_barrett**$(z_{lo}, z_{hi}, v_{mod}, v_{cr0}, v_{cr1})$
12: $\texttt{svst1}(C[i], v_c)$
13: **end for**

The multiplication of two ciphertexts is performed by computing three polynomials via dyadic product decomposition in the NTT domain, as expressed in (2), where polynomial multiplication reduces to coefficient-wise modular products.

The vectorized microkernel, reported in Algorithm 2, processes coefficients in batches, using SVE to handle multiple coefficients in parallel. For each batch, the low and high 64-bit parts of the product are computed separately, producing a 128-bit intermediate result. These intermediates are then reduced to 64-bit coefficients modulo q_j using a vectorized Barrett reduction with precomputed ratio constants, where cr_{lo} and cr_{hi} correspond to the high and low 64-bit parts of the ratio $cr = \left\lfloor \frac{2^{128}}{q_j} \right\rfloor$. This approach avoids costly full-width division, following the SIMD-oriented Barrett reduction approach commonly used in accelerated HE arithmetic [11]. Finally, the reduced coefficients are stored back to memory with predicated vector stores.

A complete ciphertext multiplication requires, for each modulus in the modulus chain, four calls to the multiplication microkernel and one additional call to the addition microkernel (i.e., Algorithm 1) to compute the cross-term sum.

C. Vectorized rotation

The rotation of the ciphertext is based on the transformation in (3), which combines the Galois automorphism and the relinearization using the corresponding rotation keys to restore the standard ciphertex representation.

Unlike addition and multiplication, this operation cannot be expressed purely as element-wise modular arithmetic. Instead, it consists of a permutation of the coefficients followed by a key-switching procedure involving forward and inverse NTT transforms and modular arithmetic.

The proposed implementation, shown in Algorithm 3, accelerates the arithmetic primitives within the key-switching pipeline, while the Galois permutation stage and the control-flow components of the key-switching procedure remain scalar.

In particular, modular additions and multiplications are performed using the vectorized kernels described in Algorithm 1 and Algorithm 2, respectively. The same Barrett reduction scheme used in the multiplication kernel is also employed in the modular reduction step to ensure that the coefficients remain within the valid range modulo q_j. Furthermore, the forward and inverse NTT computations are partially vectorized by accelerating the butterfly subkernels, which represent the fundamental computational structure of these transforms.

V. Results

A. Experimental setup

The experimental evaluation was carried out on an NVIDIA Grace platform, which utilizes ARM Neoverse V2 cores, each equipped with four 128-bit SVE units. The system ran Ubuntu 22.04 LTS (kernel 5.15), with GCC 12.3 and CMake 3.27 used to build binaries. To ensure cycle-accurate measurements, we relied on the hardware performance counter `PERF_COUNT_HW_CPU_CYCLES` via the Linux `perf` tool. This approach enables the collection of metrics that support future comparisons across different systems, even when they operate at different clock frequencies.

All measurements were collected on a single CPU core. The accelerated implementation exploits the SVE execution units

Algorithm 3 Vectorized rotation

Require: ciphertext c, rotation step k, rotation keys rk
Ensure: Rotated ciphertext c_{rot}

Scalar Galois coefficient permutation
$c' \leftarrow$ **Galois_permutation**(c, k)
for each modulus q_j in the modulus chain **do**
    # Vectorized NTT transformation
    $\hat{c} \leftarrow$ **Vectorized_NTT**$(c', q_j, \dots)$
    # Vectorized multiplication by rotation key
    $t_0 \leftarrow$ **Vectorized_Modular_Multiply**$(\hat{c}_0, rk_0, q_j)$
    $t_1 \leftarrow$ **Vectorized_Modular_Multiply**$(\hat{c}_1, rk_1, q_j)$
    # Vectorized sum of intermediates
    $u \leftarrow$ **Vectorized_Modular_Addition**(t_0, t_1, q_j)
    # Vectorized inverse NTT transformation
    $v \leftarrow$ **Vectorized_Inverse_NTT**$(u, q_j, \dots)$
    # Vectorized Barrett reduction
    $w \leftarrow$ **Vectorized_Barrett**(v, q_j)
end for
# Scalar key switch finalization
$c_{\text{rot}} \leftarrow$ **Switch_Key_Inplace**(w, rk)
return c_{rot}

available within the datapath of a single Neoverse V2 core, while the baseline SEAL implementation runs on the same single-core configuration without SVE-specific vectorization.

The SEAL-exposed operations for the programmer `evaluator.add()`, `evaluator.multiply()`, and `evaluator.rotate_vector()` were benchmarked by running each 100 times on freshly generated ciphertexts from random cleartext data. Decryption was performed after each operation to verify the correctness of SVE-based kernel implementations. In addition, to reduce the impact of noise and sporadic system effects, outliers were removed using the interquartile range (IQR) method, and the mean of the remaining values was used for the reported results.

To quantify the benefits introduced by architecture-aware SIMD optimizations, two configurations were benchmarked:

- Native SEAL: standard library compiled in release mode with default optimizations level `-O3` and without the proposed SVE microkernels.
- SVE-accelerated SEAL: standard library with kernels replaced by SVE-optimized versions, compiled with flags `-march=armv8.2-a+sve2 -msve-vector-bits=128 -O3 -funroll-loops`.

The results reported below were obtained for the CKKS parameter configurations listed in Table I. These choices allow assessing how performance scales with both polynomial degree and modulus chain length. All experiments were conducted on ciphertexts at the first level of the modulus chain q and using symmetric-key encryption. Experiments with asymmetric encryption showed no significant differences in clock counts and are therefore omitted.

TABLE I
CKKS PARAMETER SETS USED IN THE EXPERIMENTS

N	q_i **(bits)**	Δ	λ **(bits)**
8192	{60, 40, 40, 60}	2^{40}	128 bit
16384	{60, 40, 40, 40, 60}	2^{40}	128 bit
32768	{60, 40, 40, 40, 40, 60}	2^{40}	128 bit

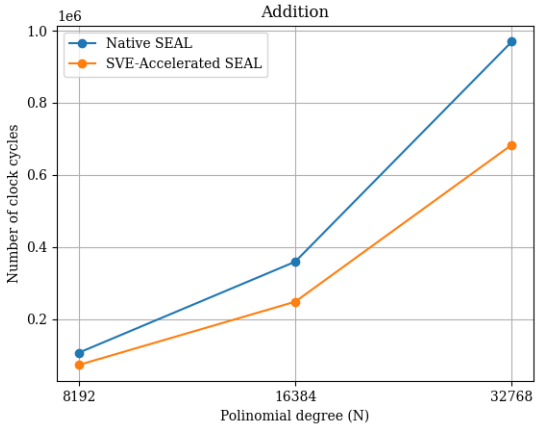

Fig. 2. CPU cycles comparison for addition across polynomial degrees.

B. Performance comparison

In general, SVE vectorization reduces the execution time of the evaluated ciphertext operations by exploiting the intrinsic data-level parallelism of the arithmetic primitives underlying CKKS. It leverages the element-wise processing of polynomial coefficients across multiple moduli, enabling effective SIMD execution on coefficient vectors. The optimized kernels reduce the number of CPU cycles required for ciphertext evaluation. These performance improvements are observed for all polynomial degrees ($N = 8192$, 16384, and 32768), with speedup depending on operation complexity and the proportion of arithmetic primitives that effectively leverage SIMD execution.

Addition of ciphertexts is computed by coefficient-wise modular addition, with independent operations across all moduli. This operation is particularly well-suited to SIMD acceleration, achieving significant performance improvements. Figure 2 reports the CPU cycles required to perform ciphertext addition across the evaluated polynomial degrees. For $N = 8192$, addition achieves a reduction of approximately 31.8% in CPU cycles (1.47× speedup). For $N = 16384$, the reduction is about 30.9% (1.45× speedup), and for $N = 32768$, it reaches 29.5% (1.42× speedup). The speedup is observed to remain nearly constant across all tested configurations. The slight decrease in relative speedup for larger N can be attributed to memory access and cache effects, as well as fixed loop control overheads, which limit SIMD efficiency.

Multiplication of ciphertexts is more complex than addition, as it involves dyadic products (64×64 bits) of polynomial coefficients combined with Barrett reductions (128→64 bits) in the NTT domain. Although the primitives still benefit from SVE vectorization, the multi-stage structure of the algorithm

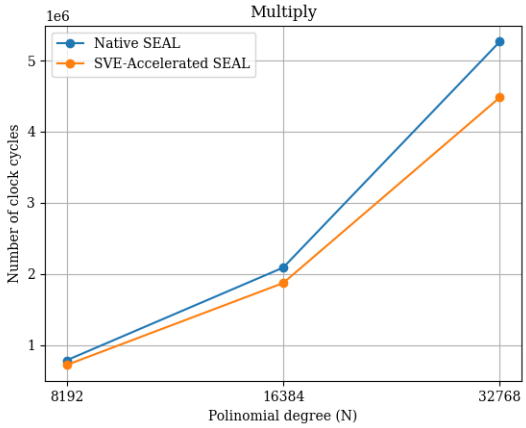

Fig. 3. CPU cycles comparison for multiplication across polynomial degrees.

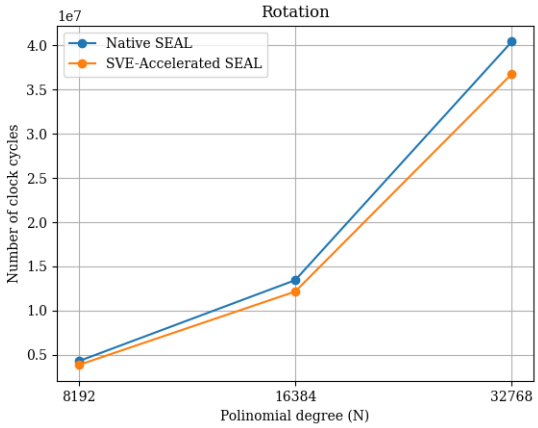

Fig. 4. CPU cycles comparison for rotation across polynomial degrees.

introduces memory accesses and synchronization points, and handling the 128-bit intermediates increases the computational cost, limiting the exploitable SIMD parallelism and reducing the achievable performance. Nevertheless, as shown in Figure 3, the SVE-optimized implementation provides improvements across all parameter sets: for $N = 8192$, multiplication achieves an 8.4% reduction (1.09× speedup); for $N = 16384$, 10.4% (1.12× speedup); and for $N = 32768$, 14.9% (1.17× speedup). Interestingly, the speedup modestly increases with polynomial degree, as the vectorizable modular arithmetic operations grow relative to fixed overheads such as loop control, memory access, and synchronization.

Rotation of a ciphertext is implemented through a Galois automorphism followed by a key-switching procedure. This operation is among the most computationally expensive ciphertext operations, involving polynomial permutations, modular arithmetic, and additional NTT transformations during key-switching. While the optimized arithmetic kernels accelerate parts of the computation, rotation remains harder to fully vectorize with SIMD due to algorithmic components such as coefficient permutations and control-flow-intensive stages of the key-switching pipeline. Figure 4 reports the measured performance of right rotations by one position. The SVE implementation achieves a 10.1% reduction of CPU cycles (1.11× speedup) for $N = 8192$, 9.6% (1.11× speedup) for $N = 16384$, and 9.0% (1.10× speedup) for $N = 32768$. SVE acceleration yields a consistent yet modest improvement across all N, with only minor variations due to the relative impact of fixed key-switching and control-flow overhead.

VI. Conclusion

This paper presented an SVE-based acceleration of main CKKS ciphertext operations on ARM Neoverse processors within the Microsoft SEAL framework. By exploiting the data-level parallelism available in RNS polynomial arithmetic, the proposed approach introduces vectorized microkernels for modular addition and modular multiplication with Barrett reduction, together with vectorized butterfly stages in the forward and inverse NTT transforms.

These optimizations accelerate the arithmetic kernels underlying ciphertext addition, multiplication, and rotation while preserving compatibility with the SEAL interface. Experimental results on an ARM Neoverse V2 platform show speedups of up to about 1.4× for ciphertext addition and between 1.09× and 1.16× for multiplication and rotation across different CKKS parameter configurations.

These results highlight that architecture-aware SIMD optimizations based on ARM SVE can provide meaningful benefits for homomorphic encryption workloads executed entirely on ARM Neoverse processors, even when GPUs or dedicated accelerators are not available, while narrowing the performance gap with established x86-based software-only acceleration approaches. Although the achieved speedups remain lower than those typically obtainable with GPU- or hardware-accelerated solutions, they still demonstrate appreciable gains achievable through software-only optimization.

Future work will investigate additional optimized CKKS kernels and subkernels, broader comparisons with existing x86 SIMD acceleration approaches, and alternative HE libraries such as OpenFHE and PALISADE. It will also analyze the impact of the proposed optimizations on power consumption and evaluate the effect of platform-specific compiler optimization flags, including processor-targeted options such as `-mcpu`, on ARM Neoverse platforms.

Acknowledgment

Work partially supported by the Italian Ministry of Education and Research (MUR) in the framework of the FoReLab project (Departments of Excellence). Computational resources provided by computing@unipi, a Computing Service provided by University of Pisa.

References

[1] C. Moore, M. O'Neill, E. O'Sullivan, Y. Doröz, and B. Sunar, "Practical homomorphic encryption: A survey," in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014, pp. 2792–2795.

[2] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Computing Surveys*, vol. 51, no. 4, 2018.

[3] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, ser. Lecture Notes in Computer Science, vol. 10624. Springer, 2017, pp. 409–437.

[4] W. Liu, "A survey on approximate homomorphic encryption and its applications," *Computers & Security*, vol. 103, p. 103775, 2025. [Online]. Available: https://doi.org/10.1016/j.csi.2023.103775

[5] M. Y. Ren, "Ckks speech fully homomorphic encryption method based on gpu acceleration," *International Journal of Network Security*, vol. 27, no. 1, pp. 185–199, 2025, preprint / Journal article. [Online]. Available: https://ijns.jalaxy.com.tw/contents/ijns-v27-n1/ijns-2025-v27-n1-p185-199.pdf

[6] L. Wu, X. Wang, J. Liu, Y. Su, Z. Tu, and W. Liu, "Homomorphic encryption for machine learning applications with ckks algorithms: A survey of developments and applications," *Computers, Materials & Continua*, vol. 85, no. 1, pp. 89–119, 2025. [Online]. Available: https://doi.org/10.32604/cmc.2025.064346

[7] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology – EUROCRYPT 2010*, ser. Lecture Notes in Computer Science, vol. 6110. Springer, 2010, pp. 1–23.

[8] N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," *Designs, Codes and Cryptography*, vol. 71, no. 1, pp. 57–81, 2014.

[9] "Microsoft SEAL (release 4.1)," https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.

[10] W. Jung, E. Lee, S. Kim, K. Lee, N. Kim, C. Min, J. H. Cheon, and J. H. Ahn, "Heaan demystified: Accelerating fully homomorphic encryption through architecture-centric analysis and optimization," *arXiv Preprint*, 2020. [Online]. Available: https://arxiv.org/abs/2003.04510

[11] F. Boemer, S. Kim, G. Seifu, F. D. de Souza, V. Gopal *et al.*, "Intel HEXL (release 1.2)," https://github.com/intel/hexl, 2021.

[12] F. Boemer, S. Kim, G. Seifu, F. de Souza, and V. Gopal, "Intel hexl: Accelerating homomorphic encryption with avx512-ifma52," in *IEEE/ACM Conference*, 2021.

[13] O. Ozerk, C. Elgezen, A. C. Mert, E. Ozturk, and E. Savas, "Efficient number theoretic transform implementation on gpu for homomorphic encryption," Cryptology ePrint Archive, Paper 2021/124, 2021.

[14] Y. Zhai, M. Ibrahim, Y. Qiu, F. Boemer, Z. Chen, A. Titov, and A. Lyashevsky, "Accelerating encrypted computing on intel gpus," *arXiv preprint arXiv:2109.14704*, 2021.

[15] P. N. Duong and H. Lee, "Pipelined key switching accelerator architecture for ckks-based fully homomorphic encryption," *Sensors*, vol. 23, no. 10, p. 4594, 2023.

[16] S. Fan, X. Deng, Z. Tian, Z. Hu, L. Chang, R. Hou, D. Meng, and M. Zhang, "Taiyi: A high-performance ckks accelerator for practical fully homomorphic encryption," *arXiv preprint arXiv:2403.10188*, 2024.

[17] S. Di Matteo, M. L. Gerfo, and S. Saponara, "Vlsi design and fpga implementation of an ntt hardware accelerator for homomorphic seal-embedded library," *IEEE Access*, vol. 11, pp. 72 498–72 508, 2023.

[18] J. Bossuat, J. R. Troncoso-Pastoriza, and J. Hubaux, "Bootstrapping for approximate homomorphic encryption," in *Lecture Notes in Computer Science*. Springer, 2022.