Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Compiler-Guided Polynomial-Level Parallelism for FHE-Encrypted Machine Learning Inference

Zijian Duan
Hunan University
Zuoyan Zhang
Hunan University
Tao Dai
Guangdong Provincial Key Laboratory of Power System Network Security
Zhihong Liang
Guangdong Provincial Key Laboratory of Power System Network Security
Jie Zhao
jiezhao@hnu.edu.cn

Hunan University

Research Article

Keywords: ANT-ACE, Fully homomorphic encryption, Parallelism

Posted Date: September 14th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10901622/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Compiler-Guided Polynomial-Level Parallelism for FHE-Encrypted Machine Learning Inference

Zijian Duan[1][0009−0004−2505−9057], Zuoyan Zhang[1][0009−0005−2615−2541], Tao Dai[2,3], Zhihong Liang[2,3], and Jie Zhao[1][0000−0003−2303−9736]

[1] Hunan University, Changsha, China {duanzj1,zyanz,jiezhao}@hnu.edu.cn
[2] Guangdong Provincial Key Laboratory of Power System Network Security
[3] CSG Electric Power Research Institute Co., Ltd., Guangzhou, China
{daitao,liangzh}@csg.cn

Abstract. While compiling machine learning (ML) inference workloads encrypted by fully homomorphic encryption (FHE) has converged on a staged lowering strategy that translates ML computation graphs into ciphertext polynomial programs, polynomial-level parallelism remains underexplored and is typically confined to hard-coded FHE libraries. We introduce CRISP, a compiler-reasoned inter-stage safe parallelization approach that trades runtime SIMD vectorization for compile-time OpenMP parallelism. CRISP enables polynomial-level parallelization while preserving both data dependences and domain-specific cryptographic invariants, ensuring correctness across compilation stages and safety under inter-stage lowering and transformation. CRISP also coordinates compile-time parallelization with polynomial-level fusion, enabling optimization compositions that previously required manual implementation within FHE libraries. We implement CRISP within ANT-ACE, an end-to-end compiler for FHE-encrypted ML inference, and evaluate it on an Intel multi-core CPU platform. Experimental results demonstrate that CRISP enables effective compile-time parallelism, achieving a 2.65× mean speedup on polynomial-level operations, a mean execution-time reduction of 5.63% on ML operators, and an average 137-second reduction in end-to-end ML inference latency.

Keywords: ANT-ACE · Fully homomorphic encryption · Parallelism

1 Introduction

ML is now pervasive across cloud services and edge/device platforms, enabling applications ranging from large-scale data analytics to low-latency inference. In many deployment scenarios, ML inference is executed on infrastructure operated by untrusted providers, where users submit sensitive data for remote processing. Despite its scalability and convenience, this execution model raises serious privacy concerns, as user data may be exposed to untrusted environments.

FHE [13–15] offers a principled solution by enabling computation directly on encrypted data. Clients encrypt their inputs and delegate computation to an untrusted server, which operates exclusively on ciphertexts and returns encrypted

results that decrypt to correct plaintext outputs. Among existing FHE schemes, CKKS [6] and its RNS-CKKS [5] variant are well suited for ML workloads, since they support approximate fixed-point arithmetic over encrypted floating-point values. Thus, a rich ecosystem of CKKS libraries [1, 9, 12, 18, 30, 34] and FHE compilers [3,7,10,11,22,26,33] has emerged to automate encrypted ML inference.

Existing FHE compilers converge on a staged lowering strategy, illustrated in Fig. 1. Given an ML computational graph, these compilers first apply graph-level transformations, then progressively approximate plaintext computation using CKKS arithmetic, and finally lower the program into ciphertext polynomial operations supported by FHE runtime libraries.

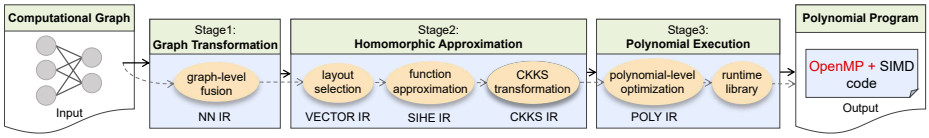


Fig. 1. A staged lowering strategy shared by existing FHE compilers for ML inference. Solid arrows connect stages, while dashed arrows illustrate the pipeline implemented by ANT-ACE [26]. NN: neural network; SIHE: Scheme-independent homomorphic encryption. CRISP enables OpenMP parallelism on top of this lowering strategy.

This design resembles compilation pipelines for unencrypted ML workloads [4, 17, 32, 36] but diverges fundamentally in the latter stages. Instead of executing on plaintext data, encrypted ML inference must approximate computation using polynomial arithmetic under cryptographic constraints. In CKKS, plaintext values are encoded into polynomials with integer coefficients and encrypted into ciphertexts represented by multiple polynomials modulo a large composite modulus Q. Homomorphic operations are implemented as polynomial arithmetic, typically executed in the Number Theoretic Transform (NTT) domain, where convolutions are reduced to pointwise multiplications for efficiency. During this process, cryptographic parameters such as Q and polynomial degree are selected on the fly to satisfy correctness and precision requirements.

Once parameters are fixed, the final stage must exploit parallelism and locality in ciphertext polynomial programs. Two orthogonal forms of parallelism naturally arise. First, polynomial arithmetic applies independently to each coefficient, exposing *coefficient-level parallelism*, which is commonly exploited via SIMD vectorization and batching. Second, the RNS-CKKS variant decomposes the modulus Q into a product of r pairwise co-prime moduli, representing each ciphertext polynomial as r smaller polynomials, one per RNS channel. Arithmetic across RNS channels is mathematically independent and preserves correctness via the Chinese Remainder Theorem, exposing *RNS-level parallelism*.

Existing compilers largely exploit coefficient-level parallelism by lowering programs to library primitives with hard-coded SIMD implementations. Exploiting RNS-level parallelism within libraries is impractical, as CKKS parameters induce

a large configuration space and RNS execution interacts tightly with compiler transformations such as polynomial fusion and operator scheduling. Runtime libraries lack sufficient semantic context to reason about these interactions safely.

To address this issue, we propose CRISP, a **C**ompiler-**R**easoned **I**nter-stage **S**afe **P**arallelization approach. CRISP not only trades runtime SIMD vectorization for compile-time OpenMP parallelism in polynomial operations, but also integrates compile-time OpenMP parallelism with runtime SIMD parallelism in auxiliary operations mandated by the FHE scheme. We implement CRISP within ANT-ACE [26], an end-to-end FHE compiler that fully automates the staged lowering strategy in Fig.1. CRISP and the ANT-ACE baseline use identical compilation and runtime settings, the only difference is the polynomial-level parallelization strategy applied at the POLY IR level. This controlled comparison isolates and evaluates the performance contribution of CRISP. CRISP takes the lowered POLY IR as input and produces an optimized POLY IR after dependence elimination, variable privatization, loop flattening, and parallel-attribute annotation. The optimized POLY IR is subsequently translated by ANT-ACE's existing POLY2C backend into C source code containing OpenMP parallel loops and the executable binary is generated later by an external C compiler. CRISP targets the POLY IR, the final compiler IR where RNS channels are represented as loops over runtime primitives. At this level, cryptographic parameters have been instantiated, loop bounds over RNS channels are statically determined, and interactions with polynomial fusion are explicit. Unlike traditional compiler parallelization, CRISP goes beyond data-dependence analysis by enforcing domain-specific invariants and correctness of fused polynomial operators. Furthermore, CRISP enables polynomial-level fusion without resorting to hard-coded implementation within FHE libraries, preserving optimization compositions in the SIMD-accelerated approaches.

By selectively annotating parallelizable loops under these constraints, CRISP enables coarse-grained OpenMP parallelism across both polynomial coefficients and RNS channels. We evaluate CRISP on six encrypted ML inference workloads. Experiments on an Intel Xeon Platinum 8538C CPU demonstrate a 2.65× mean speedup on core polynomial operations, an average execution-time reduction of 2.68 seconds on all ML operators in encrypted ResNet-20 [19], and an average 137-second reduction in end-to-end inference latency compared to the fully optimized ANT-ACE using four CPU threads, with parallelism effectiveness demonstrated up to 16 threads. We further validate that CRISP remains compatible with graph-level operator fusion [31] and delivers consistent performance improvements across different CKKS parameter settings.

In summary, this paper makes the following contributions:

- We identify and formalize a three-stage compilation structure shared by existing FHE compilers, clarifying the roles of graph transformation, homomorphic approximation, and polynomial-level execution.
- We show that this staged decomposition limits parallelism exploitation when confined to runtime libraries, motivating compiler-level intervention.

– We propose a domain-aware compiler pass that trades coefficient-level parallelism for RNS–coefficient combined parallelism while preserving cryptographic and optimization invariants.
– We demonstrate substantial performance gains across different CKKS parameter settings.

2 Background and Motivation

This section introduces FHE and ANT-ACE. As modern CKKS implementations use RNS representations, we use CKKS and RNS-CKKS interchangeably.

2.1 CKKS-Based FHE Computation

Fig. 2 illustrates how ML operators on plaintext data are approximated by homomorphic computation over ciphertexts under CKKS. Given a (possibly fused) ML operator, it operates on encrypted user inputs through the following process.

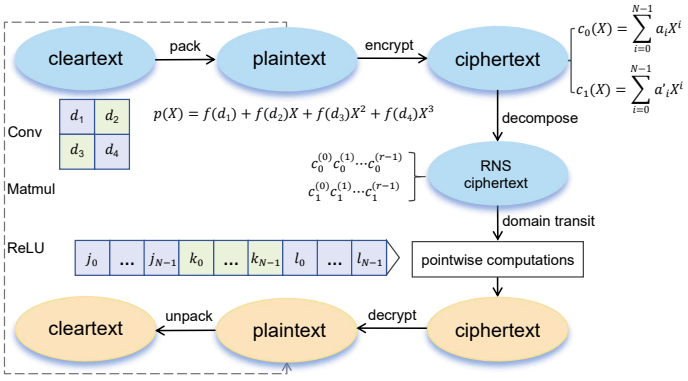


Fig. 2. Approximating ML operators (e.g., convolution, matrix multiplications shown using the dashed arrow) on plaintext data using CKKS-based homomorphic computation over ciphertext polynomials. Ellipses in blue denote inputs, and ellipses in orange represent computation results. Homomorphic computation is implemented as pointwise polynomial arithmetic in the NTT domain. Coefficient-level parallelism is shown on the left of pointwise computation: coefficients at the same polynomial index across SIMD lanes are processed independently and exploited by FHE libraries via vectorization.

Plaintext encoding. Let $d = (d_0, d_1, \dots)$ denote a vector of real-valued inputs. CKKS first packs these values into a *plaintext polynomial* $p(X) = \sum_{j=0}^{N-1} f(d_j) X^j \in \mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, where each $f(d_j)$ is an integer obtained by scaling and rounding a corresponding real value, and N is the polynomial degree, typically a power of two (e.g., 2^{12}~2^{15}). This packing enables SIMD semantics, where arithmetic on $p(X)$ conceptually applies to all encoded values in parallel.

Encryption and ciphertext structure. The plaintext polynomial is encrypted into a CKKS ciphertext $\mathbf{c}(X) = (c_0(X), c_1(X)) \in \mathcal{R}_Q^2$, where $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N + 1)$ and Q is a large composite modulus, typically in the range of 2^{180} to 2^{300} depending on circuit depth and precision requirements. Each ciphertext thus consists of two polynomials modulo Q.

To improve performance and manage large moduli, RNS-CKKS decomposes Q into a product of r pairwise co-prime moduli: $Q = \prod_{i=0}^{r-1} Q_i$, where each Q_i fits in a machine word (e.g., 60–64 bits). Under this decomposition, each ciphertext polynomial is represented as a vector of r smaller polynomials: $c_k(X) \to \left(c_k^{(0)}(X), c_k^{(1)}(X), \ldots, c_k^{(r-1)}(X)\right), \quad k \in \{0, 1\}$, with $c_k^{(i)}(X) \in \mathbb{Z}_{Q_i}[X]/(X^N+1)$. Arithmetic across different RNS channels is mathematically independent and recombined via the Chinese Remainder Theorem, exposing RNS-level parallelism.

Polynomial arithmetic and NTT domain. Homomorphic evaluation of ML operators is implemented using polynomial addition $\oplus$ and multiplication $\otimes$ over ciphertexts. Direct polynomial multiplication in the coefficient domain is expensive, so practical implementations convert ciphertext polynomials into the NTT domain, where multiplication reduces to coefficient-wise products and polynomial arithmetic exhibits both coefficient- and RNS-level parallelism.

Decryption and noise management. The resulting ciphertext result is decrypted to recover a plaintext polynomial, which is then decoded to obtain approximate real-valued outputs. Homomorphic operations introduce noise that grows with circuit depth, particularly under multiplication. To maintain correctness, FHE execution must carefully manage auxiliary operations such as rescaling and bootstrapping. In addition, since ML model weights are public, homomorphic operations may involve ciphertext–ciphertext or ciphertext–plaintext operands.

2.2 ANT-ACE Compilation Pipeline

Correctly approximating ML computation using CKKS requires extensive domain-specific reasoning, including cryptographic parameter selection, data layout decisions, fixed-point scaling, and bootstrapping placement. These challenges motivate the use of a compiler framework that automates such tasks.

ANT-ACE [26] serves as a natural implementation platform for our work, as it is, to the best of our knowledge, the only end-to-end FHE compiler that supports automatic parameter selection, fixed-point scaling, and optimized bootstrapping placement. In contrast, existing frameworks either lack automatic parameter selection [3, 33], do not support fixed-point scaling [8], or omit bootstrapping optimization [10, 11].

As shown in Fig. 1, ANT-ACE adopts a staged lowering pipeline. In the graph transformation stage, the NN IR performs graph-level operator fusion, recently enhanced by FHEFusion [31]. The homomorphic approximation stage spans multiple IRs: the VECTOR IR determines data packing layouts, the SIHE IR approximates nonlinear functions, and the CKKS IR selects cryptographic parameters and inserts auxiliary operations like rescaling and bootstrapping.

These stages produce a well-structured ciphertext program that is lowered into the polynomial execution stage, represented by the POLY IR and backed by FHE runtime libraries. At this level, ciphertext operations are expressed as loops over RNS channels that invoke hand-optimized runtime primitives, exposing the coefficient- and RNS-level parallelism.

This separation of concerns allows CRISP to focus on the polynomial execution stage, where two forms of orthogonal parallelism arise naturally but remains underexploited. By integrating our approach at this level, we can reason explicitly about RNS execution and polynomial fusion, while remaining compatible with upstream compilation stages.

3 Trading SIMD for OpenMP in Single Polynomial Ops

FHE exposes two fundamental operators, homomorphic addition $\oplus$ and homomorphic multiplication $\otimes$. Each operator can be applied either between two ciphertexts or between a ciphertext and a plaintext, resulting in four polynomial operations in total: plaintext–ciphertext addition, plaintext–ciphertext multiplication, ciphertext–ciphertext addition, and ciphertext–ciphertext multiplication. For brevity, we denote these variants as `PAdd`, `PMul`, `CAdd`, and `CMul`, respectively.

Among them, `CMul` is unique in that multiplying two CKKS ciphertexts (c_0, c_1) and (d_0, d_1) produces a degree-expanded ciphertext with three components: $(c_0, c_1) \times (d_0, d_1) \rightarrow (e_0 = c_0 d_0,\ e_1 = c_0 d_1 + c_1 d_0,\ e_2 = c_1 d_1)$. This degree expansion necessitates subsequent rescaling and relinearization to restore the ciphertext to its standard two-component form. In ANT-ACE, these steps are fused into `CMul` during the upstream CKKS approximation stage, resulting in a fused polynomial operation. In contrast, the remaining three polynomial operations do not increase ciphertext degree and thus do not require rescaling or relinearization. We refer to these operators as single polynomial operations, since each is implemented as a single polynomial-level computation without auxiliary cryptographic transformations.

3.1 Reasons for Compilation Instead of Runtime Library

While these operations appear structurally similar, the appropriate parallelization strategy depends on the surrounding compilation context. For example, a polynomial multiplication may appear as part of `CMul`. In this case, its loop participates in intermediate-value reuse across operations, and aggressive outer-loop parallelization may introduce redundant memory traffic or interfere with fusion.

The compiler can distinguish such cases because it has access to the IR and the program's global dataflow. In contrast, a runtime library only observes individual operator calls and cannot determine whether an operation participates in fusion, reuses intermediate values, or has already undergone loop restructuring. As a result, applying a fixed parallelization strategy in a runtime library may lead to suboptimal performance or violate assumptions relied upon by earlier

compilation stages. This context-sensitive optimization motivates implementing CRISP as a compiler transformation rather than a runtime-library optimization.

In this section, we focus on single polynomial operations and demonstrate how their polynomial-level computation can be parallelized by trading SIMD vectorization for OpenMP parallelism. We also explain in detail why CRISP selects OpenMP parallelism in place of SIMD vectorization rather than combining the two forms of parallelism. Without loss of generality, we use `CAdd` as a running example. The POLY IR code generated by ANT-ACE for `CAdd` is shown in Listing 1.1. The implementations of `PAdd` and `PMul` follow an analogous structure and are omitted for conciseness.

Here, r denotes the number of RNS moduli (i.e., the ciphertext level), and N denotes the polynomial degree. Operations are executed in the NTT domain. Both parameters are fixed by the upstream CKKS approximation stage and materialized in ANT-ACE's CKKS IR. The pointer `modulus` iterates through the RNS modulus chain $(q_0, q_1, \ldots, q_{r-1})$. The library-provided `hw_*` primitives encapsulate N scalar modular operations and exploit SIMD vectorization across polynomial coefficients within each RNS channel.

```
1  //modulus pointer
2  MODULUS* modulus = Q_modulus();
3  //Number of RNS channels
4  r = Poly_level();
5  for (i = 0; i < r; i = i + 1) {
6    hw_modadd(Coeffs(&res1, i, N), Coeffs(&val1, i, N),
7              Coeffs(&val1, i, N), modulus, N);
8    hw_modadd(Coeffs(&res2, i, N), Coeffs(&val2, i, N),
9              Coeffs(&val2, i, N), modulus, N);
10   modulus = modulus + 1;
11 }
```

Listing 1.1. Original POLY IR of `CAdd` using SIMD-enabled `hw_*` primitives.

3.2 Loop-Carried Dependence Removal

Although the coefficient-wise arithmetic performed by the `hw_*` primitives is independent across RNS channels, the RNS loop in Listing 1.1 is not parallelizable as written. The reason is the loop-carried dependence induced by the shared `modulus` pointer, which is incremented in each iteration and therefore prevents the ANT-ACE compiler from safely applying parallelization.

This dependence is artificial and can be eliminated through variable privatization. Instead of mutating a shared `modulus` pointer, we derive a private modulus pointer for each RNS iteration directly from the loop index. This transformation removes the loop-carried dependence without affecting correctness.

3.3 Scalar Arithmetic Recovery

After variable privatization, one might attempt to parallelize the RNS loop directly while retaining the SIMD-enabled `hw_*` primitives. However, this approach yields little performance benefit in practice. The reason is that the number of

RNS channels r is often smaller than the available CPU thread count, especially when operating at low ciphertext levels.

To expose sufficient parallelism, we therefore recover the coefficient loop and revert the vectorized `hw_*` primitives to their scalar counterparts. ANT-ACE already provides scalar modular arithmetic routines, such as `Add_int64_with_mod`, which are pure, side-effect-free, and free of cross-coefficient or cross-RNS dependences. The resulting code is shown in Listing 1.2.

```
for (i = 0; i < r; i++) {
  MODULUS* local_modulus = modulus + i;
  for (j = 0; j < N; j++) {
    res1[i][j] = Add_int64_with_mod(val1[i][j], val1[i][j],
                          Get_mod_val(local_modulus));
    res2[i][j] = Add_int64_with_mod(val2[i][j], val2[i][j],
                          Get_mod_val(local_modulus));
  }
}
```

Listing 1.2. Privatized POLY IR for `CAdd` using scalar arithmetic.

3.4 Empirical Choice Between OpenMP and SIMD

Coefficient-level SIMD vectorization and RNS-level OpenMP parallelism are semantically composable. Different RNS channels are mathematically independent, while the polynomial coefficients within each channel can be processed using SIMD instructions. Assigning different RNS channels to OpenMP threads and retaining SIMD execution within each thread does not inherently violate data dependences or the cryptographic invariants of RNS-CKKS. CRISP's decision to replace the original SIMD primitives with scalar arithmetic before applying OpenMP is thus an empirical implementation choice for the target platform, runtime interface, and workloads, rather than a fundamental restriction on combining OpenMP and SIMD.

We replace the scalar arithmetic used by CRISP with the corresponding `hw_*` primitive calls while retaining OpenMP parallelization of the outer RNS loop. We refer to this execution mode as `Hybrid-RNS` hereafter. Although this design preserves coefficient-level SIMD parallelism, it introduces two performance limitations. First, reconstructing the `hw_*` calls gives up the flattened RNS–coefficient iteration space used by CRISP, which may thereby result in load imbalance.Second, the boundaries of the encapsulated `hw_*` primitives restrict data reuse across related polynomial computations. For example, ciphertext–plaintext multiplication applies the same plaintext polynomial to both ciphertext components. With separate `Hw_modmul` calls, the two components are processed in independent traversals. The shared plaintext operand cannot be retained in vector registers across the calls, and the compiler cannot fuse their internal computations. This restriction becomes more significant after function inlining and polynomial fusion, where intermediate multiplication results could otherwise be consumed immediately. In our implementation, these limitations caused the direct combination to perform worse than CRISP for several targeted operations.

To address both limitations, we further implemented a fused OpenMP–SIMD variant, referred to as `Hybrid-fused`. Within each chunk, the arithmetic is ex-

plicitly vectorized using AVX-512 instructions. This organization preserves a sufficiently large OpenMP iteration space while allowing related computations to share loaded operands.

Listing 1.3 illustrates the implementation using ciphertext–plaintext multiplication as an example. The plaintext vector is loaded once and immediately reused for both ciphertext components.

```
#pragma omp parallel for schedule(static)
for (chunk = 0; chunk < num_chunks; chunk++) {
  // chunk_size = 4096, num_chunks = r * N / chunk_size
  for (j = 0; j < chunk_size; j += 8) {
    vp  = AVX512_Load(plain[rns] + offset + j);
    vc0 = AVX512_Load(input.c0[rns] + offset + j);
    vc1 = AVX512_Load(input.c1[rns] + offset + j);
    vr0 = AVX512_BarrettMul(vc0, vp, q);
    vr1 = AVX512_BarrettMul(vc1, vp, q);
    AVX512_Store(output.c0[rns] + offset + j, vr0);
    AVX512_Store(output.c1[rns] + offset + j, vr1);
  }
}
```

Listing 1.3. SIMD implementation of Hybrid-fused ciphertext–plaintext multiplication.

Overall, `Hybrid-fused` improves the targeted polynomial kernels, confirming that coefficient-level SIMD vectorization can be combined with CRISP's thread-level parallelism. However, its kernel-level improvement does not translate into a stable end-to-end benefit, where the performance difference falls within the observed measurement variation. CRISP therefore adopts the flattened OpenMP implementation as its default strategy, while treating OpenMP–SIMD composition as a valid but workload- and implementation-dependent alternative. A detailed analysis is presented in §6.2.

3.5 Loop Flattening

Because the polynomial degree N is typically large, the coefficient loop provides sufficient parallelism to saturate modern multi-core CPUs. A straightforward strategy would be to parallelize the inner coefficient loop. However, doing so retains the sequential outer RNS loop and can introduce unnecessary synchronization overhead. Instead, we flatten the nested RNS and coefficient loops into a single one-dimensional iteration space of size $N \times r$. This transformation exposes a larger and more uniform parallel workload and enables efficient static scheduling. The flattened version is shown in Listing 1.4.

```
MODULUS* modulus = Q_modulus();
r = Poly_level();
#pragma omp parallel for schedule(static)
for (i = 0; i < N * r; i++) {
  MODULUS* local_modulus = modulus + i / N;
  res1[i] = Add_int64_with_mod(val1[i], val1[i],
                        Get_mod_val(local_modulus));
  res2[i] = Add_int64_with_mod(val2[i], val2[i],
                        Get_mod_val(local_modulus));
}
```

Listing 1.4. Flattened polynomial-level code with OpenMP parallelization.

Loop flattening is valid because the original RNS and coefficient loops are first transformed into a perfectly nested form. We introduce an additional compiler pass that sinks the privatized modulus computation into the innermost loop (similar to the approach used in [35]) and rewrites its update using integer division on the flattened loop index. This transformation preserves semantic correctness while maximizing available parallelism.

3.6 OpenMP Annotation Insertion

As shown in Listing 1.4, we annotate the flattened loop with an OpenMP `parallel for` directive using static scheduling. Static scheduling is well suited here because each iteration performs a constant amount of work and the mapping from iteration index to RNS modulus is deterministic. Importantly, static scheduling prevents unpredictable modulus jumps across threads, which is essential for maintaining the correctness and locality properties of homomorphic polynomial arithmetic.

In summary, CRISP systematically trades coefficient-level SIMD vectorization for coarse-grained OpenMP parallelism over a combined RNS–coefficient space. This design is motivated by the target platform, the boundaries of the existing runtime primitives, and our empirical evaluation, it is an implementation choice rather than a semantic restriction on combining OpenMP and SIMD. This compiler-guided transformation enables `PAdd`, `PMul`, and `CAdd` to fully exploit multi-core parallelism without relying on hard-coded runtime FHE libraries. As we show in §6.1, this trade-off is both safe and performance-beneficial.

3.7 Limitations of General-Purpose Automatic Parallelization

At first glance, the above transformations resemble classical compiler optimizations. However, applying automatic parallelization to polynomial programs generated by FHE compilers is non-trivial for two reasons.

First, polynomial loops operate over ciphertexts in the RNS-CKKS representation, where each polynomial is decomposed into multiple modulus components and coefficients that must preserve strict domain invariants, including modulus alignment and coefficient ordering across compilation stages. General-purpose compilers lack knowledge of these cryptographic invariants and therefore cannot safely reason about loop independence, causing conservative dependence analysis to prevent parallelization. Second, naive loop parallelization may violate assumptions made by other compilation stages. Several polynomial operators are fused or reused during CKKS-level optimizations, and parallelizing loops without this context may introduce redundant memory traffic, break fusion opportunities, or disrupt the ciphertext layouts expected by runtime primitives.

CRISP addresses these challenges because it has explicit knowledge of polynomial representations, RNS decomposition, and inter-stage constraints, enabling safe parallelization impossible in general-purpose parallelizing compilers.

4 OpenMP–SIMD Integration for Fused Polynomial Ops

Fused polynomial operators impose additional constraints on parallelization due to their internal structure and cryptographic side effects. In this section, we study `CMul`, the fused polynomial operator that combines multiplication, rescaling, and relinearization. While we again trade SIMD vectorization for OpenMP parallelism in parts of this operator, its fused nature prevents a uniform application of this strategy without risking correctness regressions.

To address this challenge, we selectively apply OpenMP parallelization to phases of `CMul` that satisfy both data-dependence and domain-invariance requirements, while retaining SIMD vectorization in others. This selective strategy results in an OpenMP–SIMD hybrid execution model. The resulting POLY IR is shown in Listing 1.5. For clarity, we do not inline `Rescale` and `Relinearize`, which are discussed separately. All compiler transformations and OpenMP annotation insertions follow the same methodology as described in §3.

```
MODULUS* modulus = Q_modulus();
r = Poly_level();
// Phase I: Ciphertext--ciphertext multiplication
#pragma omp parallel for schedule(static)
for (i = 0; i < N * r; i++) {
  MODULUS* local_modulus = modulus + i / N;
  res1[i] = Mul_int64_mod_barret(val1[i], val2[i], local_modulus);
  res2[i] = Mul_int64_mod_barret(val2[i], val1[i], local_modulus);
  res3[i] = Mul_int64_mod_barret(val1[i], val1[i], local_modulus);
  res4[i] = Add_int64_with_mod(res1[i], res2[i],
                               Get_mod_val(local_modulus));
  res5[i] = Mul_int64_mod_barret(val2[i], val2[i], local_modulus);
}
// Phase II: Rescaling
Init_ciph3_down_scale(&tmp_1, &output);
Rescale(&tmp_1._c0_poly, &output._c0_poly);
Rescale(&tmp_1._c1_poly, &output._c1_poly);
Rescale(&tmp_1._c2_poly, &output._c2_poly);
// Phase III: Relinearization
result = Relinearize(tmp_1);
```

Listing 1.5. Parallel POLY IR code for `CMul`.

4.1 Full Parallelization of Ciphertext–Ciphertext Multiplication

As shown in Phase I of Listing 1.5, the core arithmetic of ciphertext–ciphertext multiplication is parallelized by flattening the RNS and coefficient loops and applying OpenMP to the resulting iteration space. This phase replaces the original coefficient-level SIMD vectorization with coarse-grained OpenMP parallelism.

In addition to `Add_int64_with_mod` that has been discussed in §3.3, Phase I also invokes `Mul_int64_mod_barret`, a modular multiplication routine that performs Barrett modular reduction [2] to avoid expensive division. Both functions are pure, side-effect-free, and operate independently across RNS channels and coefficients. Reapplying our compiler data-dependence analysis confirms the absence of cross-RNS and cross-coefficient dependences, making the OpenMP annotation safe and correct.

4.2 Selective Parallelization within Auxiliary Operations

In contrast to Phase I, Phase II and III introduce extra structural and cryptographic constraints that prevent uniform OpenMP parallelization. Without loss of generality, Listing 1.6 illustrates the internal structure of `Relinearize`, with OpenMP annotations already inserted where permitted. Irrelevant initialization and bookkeeping code is omitted for brevity.

```
for (i = 0; i < Num_decomp(); i++) {
  // Omitted setup code
  ...
  modulus = Q_modulus();
  r = Poly_level();
  // Inner loop: retain SIMD vectorization
  for (j = 0; j < r; j++) {
    Hw_modmul(Coeffs(&tmp, 0, N), Coeffs(&key0, j, N),
              Coeffs(&_pgen_ext_4, j, N), modulus, N);
    ...
  }
  modulus = P_modulus();
  // Inner loop: retain SIMD vectorization
  for (j = 0; j < P; j++) {
    Hw_modmul(Coeffs(&tmp, 0, N), Coeffs(&key0, key_p_idx, N),
              Coeffs(&_pgen_ext_4, p_idx, N), modulus, N);
    ...
  }
}
r = Poly_level();
// Outermost loop: OpenMP-parallelized
#pragma omp parallel for schedule(static)
for (i = 0; i < N * r; i++) {
  ...
}
```

Listing 1.6. Parallel POLY IR code for `Relinearize` (simplified).

Unlike Phase I, we selectively parallelize only the final flattened loop in `Relinearize`. This loop operates on fully materialized polynomial data and mirrors the structure of the flattened multiplication loop in Listing 1.5. The two inner loops shown above continue to use SIMD-accelerated `hw_*` primitives.

Although these inner loops appear to be candidates for OpenMP parallelization, we avoid doing so. Their common outer loop is bounded by `Num_decomp()`, which depends on modulus decomposition. The decomposition procedure involves domain transitions between coefficient and NTT representations. Flattening and parallelizing these loops with OpenMP may result in threads operating on polynomials in inconsistent domains, thereby violating cryptographic domain invariants and producing incorrect results. In addition, simply inserting OpenMP annotation above these two loops without flattening with the common outer loop may incur heavy synchronization overhead across CPU threads.

By selectively parallelizing only domain-stable loops and preserving SIMD vectorization where domain transitions occur, our approach ensures correctness while maximizing available parallelism. As we show in §6.1, this phase-selective execution is also safe and performance-beneficial. Since this selectively parallelized `Relinearize` phase is inlined into `CMul`, the resulting fused operator exhibits an OpenMP–SIMD hybrid execution model.

4.3 Runtime SIMD Vectorization vs. Compiler-Guided Parallelism

Although both SIMD vectorization and OpenMP parallelization exploit hardware parallelism, their roles in current FHE systems differ significantly. Existing FHE libraries implement SIMD vectorization through specialized runtime primitives (e.g., `hw_*` functions) that execute fixed-width vector instructions over polynomial coefficients. These fine-grained primitives are manually engineered in the runtime library. Thus, the compiler only observes calls to these primitives and has limited visibility into their internal vectorized computation.

In contrast, our approach exposes the polynomial loops directly in the compiler IR and applies OpenMP parallelization to the flattened RNS–coefficient loop. This shifts parallelization decisions from library implementations to the compiler itself, enabling optimization at a coarser granularity and allowing the compiler to reason about surrounding loop transformations such as fusion and flattening. Consequently, the proposed approach complements existing SIMD optimizations while providing a compiler-driven mechanism for exploiting thread-level parallelism that is difficult to implement inside runtime libraries alone.

5 Preserving Polynomial-Level Fusion

With the parallelization strategy introduced in the previous sections, ANT-ACE can achieve higher performance than relying solely on SIMD vectorization. However, in the original ANT-ACE design, SIMD vectorization is further combined with two complementary fusion strategies: (1) compile-time RNS-level loop fusion and (2) runtime coefficient-level polynomial fusion. These fusion mechanisms are critical for reducing memory traffic and improving locality. An illustrative example is shown in Listing 1.7. In this example, two `hw_modmul` invocations appearing within the same RNS loop are fused via runtime polynomial fusion into a single custom primitive, `hw_modmuladd`. In contrast, a `hw_modmul` invocation appearing in a separate loop is fused via compile-time loop fusion.

```
// Before fusion
for (i = 0; i < r; i++) {
  hw_modadd(Coeffs(&res1, i, N), Coeffs(&val1, i, N),
            Coeffs(&val1, i, N), modulus, N);
  hw_modmul(Coeffs(&res1, i, N), Coeffs(&val1, i, N),
            Coeffs(&val1, i, N), modulus, N);
  ...
}
for (i = 0; i < r; i++) {
  hw_modmul(Coeffs(&res1, i, N), Coeffs(&val1, i, N),
            Coeffs(&val1, i, N), modulus, N);
  ...
}
// After fusion
for (i = 0; i < r; i++) {
  // Polynomial fusion with the hw_modmul in the first loop
  hw_modmuladd(Coeffs(&res1, i, N), Coeffs(&val1, i, N),
               Coeffs(&val1, i, N), modulus, N);
  // Loop fusion with the hw_modmul in the second loop
  hw_modmul(Coeffs(&res1, i, N), Coeffs(&val1, i, N),
            Coeffs(&val1, i, N), modulus, N);
  ...
}
```

Listing 1.7. An example of polynomial fusion in ANT-ACE.

Our scalar arithmetic recovery step (§ 3.3) eliminates the `hw_*` primitives, while the loop flattening step (§3.5) embeds RNS-level loop fusion by collapsing nested RNS and coefficient loops into a single iteration space. Thus, the original fusion mechanisms cannot be directly reused. Nevertheless, we show that both forms of fusion can be preserved through alternative compiler-guided strategies.

For polynomial fusion, instead of introducing new runtime primitives, we perform compile-time function inlining. Scalar routines such as `Add_int64_with_mod` and `Mul_int64_mod_barret` are inlined and combined into a single computation, with intermediate results stored in temporary variables. This achieves an effect equivalent to `hw_modmuladd`, while remaining entirely within the compiler and avoiding reliance on specialized runtime primitives.

For loop fusion, we retain the same fusion logic as in ANT-ACE. Loop flattening changes only the iteration space representation, not the static trip count of the flattened loop. Consequently, two flattened loops can still be legally fused as long as they have identical iteration bounds. This preserves the benefits of RNS-level loop fusion even after OpenMP parallelization is applied.

In summary, CRISP trades SIMD vectorization for OpenMP parallelization while preserving both loop fusion and polynomial fusion opportunities. As a result, the proposed parallelization does not invalidate existing fusion optimizations and remains compatible with ANT-ACE's original optimization pipeline.

6 Experimental Results

CRISP is implemented within ANT-ACE [26]. The implementation will be made publicly available upon acceptance. All experiments are conducted on an Intel Xeon Platinum 8538C multi-core CPU @ 4.0 GHz, a dual-socket NUMA platform running Ubuntu 22.04.

Unless otherwise specified, we evaluate the generated parallel code using four CPU threads compared to the 512-bit vectorization width on this platform. We also demonstrate parallelism up to 16 CPU threads. For correctness validation, all encrypted outputs are decrypted and compared against both the original ANT-ACE results and plaintext ground truth; no incorrect results are observed.

6.1 Polynomial-Level Performance

We first evaluate the impact of CRISP on the four polynomial operators described in §3. Fig.3 reports execution-time speedups normalized to the sequential version generated by ANT-ACE. For reference, the raw execution time of the default SIMD implementation of ANT-ACE is annotated within each bar.

To evaluate inter-stage safety, we consider two ciphertext level settings, $r = 15$ and $r = 2$, which are determined by the upstream CKKS approximation stage. Under the large-level setting $r = 15$, CRISP achieves a mean speedup of 2.65× across all four polynomial operations. `PMul` achieves the largest speedup of 7.17×, benefiting most from coarse-grained OpenMP parallelism due to its higher computational intensity compared to addition. In contrast, `CMul` exhibits

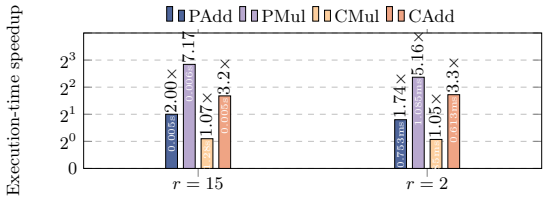


Fig. 3. Execution-time speedups of polynomial operations normalized to ANT-ACE.

the smallest improvement, with a minimum speedup of 1.07×, as parallelism is selectively applied in its fused implementation (§4).

Under $r = 2$, the observed speedups are comparable to those at $r = 15$, demonstrating that CRISP provides consistent performance improvements across varying ciphertext levels. Note that the absolute execution times under this configuration are significantly smaller than those in the large-level setting.

6.2 Comparison of OpenMP-flat and Hybrid-fused

Fig. 4 compares the absolute execution times of the optimized kernel regions in CRISP's OpenMP-flat implementation and the `Hybrid-fused` implementation, excluding the portions unaffected by the optimization. The results show that `Hybrid-fused` improves the execution performance of the targeted FHE kernels. However, a complete polynomial operation also includes serial and non-vectorizable work, and the optimized element-wise computation accounts for only part of its total execution time. Consequently, the kernel-level improvement provided by `Hybrid-fused` translates into only a small absolute latency reduction for the complete operation.

This dilution becomes more significant when `Hybrid-fused` is integrated into end-to-end encrypted inference. In addition to the execution stages outside the optimized kernels, the large evaluation keys used during FHE computation compete with ciphertext operands and intermediate data for shared cache and memory resources, further obscuring the small kernel-level benefit. As a result, the end-to-end performance difference between `Hybrid-fused` and CRISP is small and falls within the observed measurement variation, making it difficult to determine whether the difference originates from `Hybrid-fused` or from runtime fluctuation.

6.3 Impact on Encrypted ML Operators

This section evaluates the impact of CRISP on the full set of ML operators in ResNet-20 [19], including convolution, matrix multiplication, ReLU, and global average pooling (GAP). The ciphertext level is set to $r = 15$, and operator execution times range from 0.51 s (ReLU8) to 167.5 s (ReLU9). We report execution-time reduction percentages relative to the SIMD version of ANT-ACE in Fig. 5.

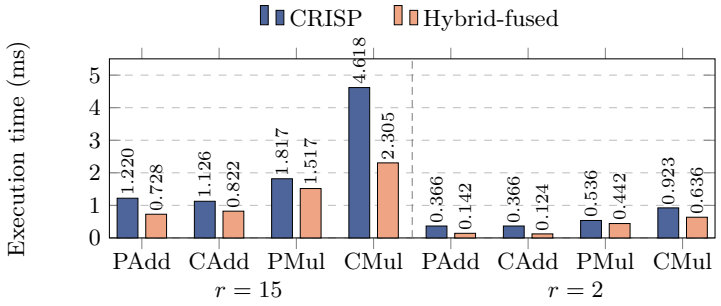


Fig. 4. Absolute execution times of the optimized kernel regions using CRISP and Hybrid-fused at ciphertext levels $r = 15$ and $r = 2$. The portions unaffected by the optimization are excluded.

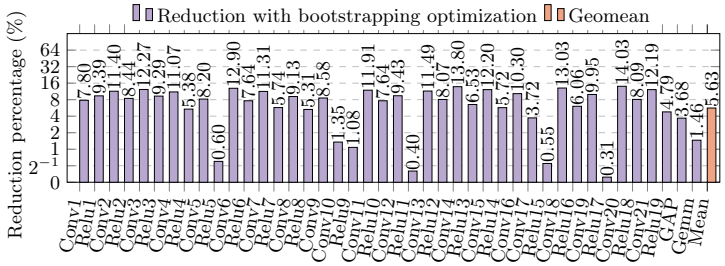


Fig. 5. Execution-time reduction percentages for ML operators relative to the SIMD-accelerated ANT-ACE baseline, with parallel bootstrapping fully optimized.

To demonstrate inter-stage safety and effectiveness, we evaluate CRISP with the parallelized bootstrapping placement determined by the CKKS approximation stage. Bootstrapping placement is performance-critical in FHE-encrypted ML inference, and ANT-ACE optimizes it using ReSBM [27]. After placement, ANT-ACE inserts OpenMP annotations for bootstrapping, an optimization orthogonal and complementary to our approach.

As shown in Fig.5, CRISP achieves a mean execution-time reduction of 5.63% (corresponding to an average reduction of 2.68 s) when combined with CKKS-level bootstrapping optimization. These results demonstrate that ML operators can be effectively approximated using our parallelized polynomial operations.

Smaller reductions are observed for ReLU5, ReLU9, ReLU11, ReLU15, and ReLU17. Unlike convolution and matrix multiplication, ReLU is a nonlinear operator that cannot be directly expressed using homomorphic addition and multiplication. Following [16], ReLU is approximated using a low-degree polynomial and is integrated with bootstrapping to control noise growth. Since bootstrapping is computation-heavy and already parallelized in ANT-ACE, the relative benefit of CRISP is reduced for these operators.

6.4 End-to-End Encrypted Inference Latency

We next evaluate end-to-end FHE-encrypted ML inference workloads. We select six representative models, including CryptoNet [16], LeNet [25], AlexNet [23], ResNet-20 [19], MobileNet [20], and SqueezeNet [21]. Fig. 6 reports the results. Bars denote execution-time reduction percentages (left y-axis), while curves represent raw execution times in seconds (right y-axis). We evaluate configurations with parallelized bootstrapping. The SIMD-accelerated ANT-ACE baseline latencies range from 7.06 s (CryptoNet) to 6355.43 s (SqueezeNet). AlexNet and MobileNet encounter out-of-memory failures, thus missing data in this experiment.

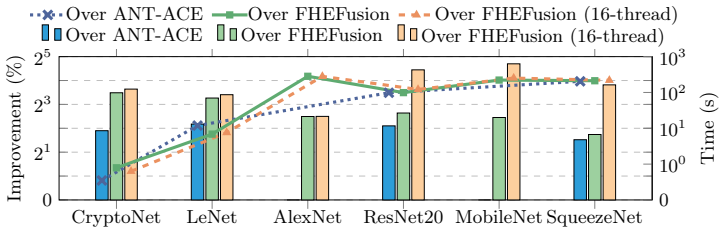


Fig. 6. End-to-end inference latency reductions relative to the ANT-ACE and FHEFusion. Bars show percentage reductions (left y-axis), and curves show absolute execution times (right y-axis). Comparison against ANT-ACE is colored in blue, while comparison against FHEFusion is shown in green (4 threads) and orange (16 threads), respectively.

CRISP achieves mean latency reductions of 6.41% over ANT-ACE when parallelized bootstrapping operation is enabled. This reduction percentage number is not minor, because it corresponds to average latency reductions of 137.24 seconds, respectively. This is meaningful for encrypted ML inference workloads, demonstrating practical benefits for real-world encrypted inference workloads.

It is important to note that the reported results characterize performance at different evaluation levels and should not be interpreted proportionally. The 2.65× result is the geometric mean of the speedups of four independent polynomial primitives, rather than a value weighted according to their contributions to the overall application execution time. It reflects the local optimization capability of CRISP for the targeted polynomial kernels, rather than predicting the speedup of the entire polynomial layer or application. The 5.63% and 6.41% results are calculated from the measured execution times of complete ML operators and end-to-end inference, respectively. Since these executions include other serial computations outside the scope of the current CRISP pass, the reported improvements already reflect the dilution of the local performance gains.

6.5 Compatibility with Graph-Level Fusion

CRISP is also compatible with graph-level optimizations. To validate this, we integrate CRISP with FHEFusion [31], the graph-level fusion optimizer in ANT-ACE. The end-to-end latency reductions are also shown in Fig. 6. Compared with FHEFusion, CRISP achieves a mean latency reduction of 3.82% with parallelized bootstrapping enabled, corresponding to an average reduction 101.81 s. As graph-level fusion reduces memory consumption by avoiding the use of some intermediate ciphertexts, AlexNet and MobileNet are successfully executed.

We further evaluate the effectiveness of preserving loop fusion and polynomial fusion as described in §5. The results show performance comparable to the original fusion mechanisms in ANT-ACE, with only minor fluctuations attributable to system noise. Due to space limitations, we omit these detailed results.

6.6 Performance Sensitivity to Thread Count

This subsection further evaluates the sensitivity of CRISP to increased thread counts. In particular, we compare the end-to-end inference latency achieved by ANT-ACE and CRISP when the number of threads is increased to 16.

This configuration is chosen because the polynomial operations targeted in this work expose coarse-grained parallelism over the flattened RNS–coefficient loop. The benefits of OpenMP parallelization become apparent when multiple CPU cores are available, while further scaling may be constrained by memory bandwidth, shared-resource contention, and synchronization overheads. Evaluating configurations with up to 16 threads therefore allows us to characterize how increasing concurrent demand on the memory system and contention for shared resources affect the scalability of the proposed parallelization.

One source of CRISP's performance improvement is its ability to exploit the aggregate private cache and execution resources provided by multiple CPU cores, thereby increasing memory-level parallelism. This means that the same total volume of data is transferred through more concurrent memory requests during execution, thereby increasing the instantaneous bandwidth demand. FHE polynomial operations process large ciphertexts across multiple RNS channels, and their working sets generally cannot remain entirely within the private caches. Consequently, execution continuously accesses the shared last-level cache and main memory. Increasing the thread count improves aggregate data throughput only while sufficient shared-memory bandwidth remains available. Once the shared cache and memory subsystem approach their sustainable bandwidth, additional threads mainly contend for the same resources and can no longer supply data to the cores proportionally faster, causing performance to saturate.

Under this setting, CRISP reduces the inference latency by 7.11% on average, corresponding to an average latency reduction of 146.57 seconds when the parallelized bootstrapping optimization is enabled. As shown in Fig.6, CRISP retains an end-to-end performance benefit at 16 threads, but the modest improvement over the four-thread configuration indicates diminishing returns as the shared cache and memory subsystem approach bandwidth saturation.

6.7 Compilation Overhead

Finally, we report compilation overhead before and after integrating CRISP into ANT-ACE in Tab.1. The reported times include the entire compilation process, from input computation graph to generated polynomial program. While the original compilation time ranges from 0.577 s to 16.706 s, CRISP increases compilation overhead by only 6.81% on average. This modest overhead demonstrates that our compiler-guided parallelization incurs negligible compilation cost.

Table 1. Compilation overhead of ANT-ACE before and after integrating CRISP.

Model	before (s)	after (s)	Increased percentage
CryptoNet	0.577	0.613	5.87%
LeNet	1.143	1.199	4.67%
AlexNet	16.706	18.860	11.42%
ResNet20	6.329	7.204	12.15%
MobileNet	6.815	6.855	0.58%
SqueezeNet	9.191	9.797	6.19%
Mean	–	–	**6.81%**

7 Related Work

FHE remained an open problem until Craig Gentry proposed the first lattice-based construction in 2009 [13, 15]. While establishing the theoretical feasibility of FHE, Gentry's scheme relies on the costly bootstrapping operation [14] to control noise growth during homomorphic computation. In ANT-ACE [26], the CKKS approximation stage optimizes the placement of this auxiliary operation and parallelizes it using OpenMP. This optimization is orthogonal to our work, which focuses on parallelizing homomorphic addition and multiplication. Our experiments evaluate combinations with parallelized bootstrapping at both the ML-operator and end-to-end levels, demonstrating the compatibility of our work.

Later work improved lattice-based FHE via learning with errors (LWE) [29] and Ring-LWE [28]–based schemes, which support bounded homomorphic computations across predefined levels and reduce or eliminate the need for bootstrapping. These schemes primarily exploit SIMD vectorization for parallelism, a design adopted by widely used schemes such as CKKS [6] and its RNS variant [5]. Implementations of these schemes appear in multiple FHE libraries [9, 12, 18, 34], each typically supporting a limited set of schemes due to substantial engineering complexity. In contrast, CRISP replaces manually implemented SIMD vectorization in runtime libraries with compiler-driven OpenMP parallelism, offering a portable alternative for accelerating homomorphic operations.

These FHE schemes underpin domain-specific compilation frameworks for FHE-encrypted ML inference workloads. Despite differences in implementation,

existing systems share a staged lowering strategy from ML graphs to polynomial-level programs, as summarized in Fig. 1. To our knowledge, this abstraction has not been explicitly characterized in prior work. We implement our approach in ANT-ACE [26], which automates all lowering stages and allows us to focus on compiler optimizations. ANT-ACE also integrates a runtime inspired by SEAL [30] and OpenFHE [1], and employs ReSBM [27] to optimize bootstrapping placement for efficient ML inference.

Several compilers for FHE computation rely primarily on SIMD parallelism. For instance, nGraph-HE [3] exploits SIMD vectorization and could benefit from our coarse-grained OpenMP parallelization. In contrast, CHET [11], EVA [10], and Fhelipe [22] implement staged lowering using directed acyclic graphs, which complicates exploration of polynomial-level parallelism. More recent systems such as HECO [33] and Dacapo [7] adopt MLIR [24] as their internal representation, where integrating our approach as an MLIR dialect would require modest engineering effort. Finally, the E3 compiler [8] lowers encrypted ML workloads to Boolean circuits, an approach orthogonal to our work that targets high-level polynomial arithmetic.

8 Conclusion

This work identifies and formalizes a three-stage compilation structure common to modern FHE compilers and implements a compiler-driven parallelization strategy in ANT-ACE to accelerate polynomial operations. Our approach replaces fine-grained runtime-managed SIMD vectorization with coarse-grained OpenMP parallelism applied at compile time. By shifting performance optimization from manually engineered runtime libraries to the compiler, the approach simplifies the development of FHE-encrypted ML inference workloads while preserving existing optimization opportunities such as polynomial-level fusion.

Techniques such as OpenMP parallelization and SIMD vectorization are well established in parallel programming and compiler optimization. The contribution of CRISP does not lie in inventing these mechanisms, but in enabling their safe and composable application at the POLY IR level of an FHE compiler. By incorporating FHE-specific knowledge of RNS representations, cryptographic domain invariants, and inter-stage optimization constraints, CRISP enables transformations that general-purpose compilers cannot safely derive from opaque runtime primitives, while preserving correctness and compatibility with optimizations across compilation stages.

Extensive evaluation with correctness validation demonstrates consistent performance improvements across polynomial primitives, ML operators, and end-to-end encrypted inference workloads. Our experiments also confirm inter-stage safety and compatibility with CKKS approximation settings, polynomial-level fusion, and graph-level transformations. These results show that compiler-guided OpenMP parallelism can effectively complement existing SIMD-based optimizations without compromising correctness or composability.

Several opportunities remain for future optimization. For example, the flattened loop structure is well suited for GPU thread-block mapping. Since ANT-ACE currently lacks CUDA or OpenCL backends, exploring GPU support and heterogeneous execution is a promising direction for future work.

References

1. Al Badawi, A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Quah, I., Polyakov, Y., R.V., S., Rohloff, K., Saylor, J., Suponitsky, D., Triplett, M., Vaikuntanathan, V., Zucca, V.: Openfhe: Open-source fully homomorphic encryption library. In: Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography. p. 53–63. WAHC'22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3560827.3563379, https://doi.org/10.1145/3560827.3563379
2. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) Advances in Cryptology — CRYPTO' 86. pp. 311–323. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)
3. Boemer, F., Lao, Y., Cammarota, R., Wierzynski, C.: ngraph-he: A graph compiler for deep learning on homomorphically encrypted data (2019), https://arxiv.org/abs/1810.10121
4. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., Krishnamurthy, A.: Tvm: an automated end-to-end optimizing compiler for deep learning. In: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation. p. 579–594. OSDI'18, USENIX Association, USA (2018)
5. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full rns variant of approximate homomorphic encryption. In: Cid, C., Jacobson Jr., M.J. (eds.) Selected Areas in Cryptography – SAC 2018. pp. 347–368. Springer International Publishing, Cham (2019)
6. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017. pp. 409–437. Springer International Publishing, Cham (2017)
7. Cheon, S., Lee, Y., Kim, D., Lee, J.M., Jung, S., Kim, T., Lee, D., Kim, H.: DaCapo: Automatic bootstrapping management for efficient fully homomorphic encryption. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 6993–7010. USENIX Association, Philadelphia, PA (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/cheon
8. Chielle, E., Mazonka, O., Gamil, H., Tsoutsos, N.G., Maniatakos, M.: E3: A framework for compiling c++ programs with encrypted operands. Cryptology ePrint Archive, Paper 2018/1013 (2018), https://eprint.iacr.org/2018/1013
9. CryptoLab: Heaan library (2022), https://www.cryptolab.co.kr/en/products-en/heaan-he/
10. Dathathri, R., Kostova, B., Saarikivi, O., Dai, W., Laine, K., Musuvathi, M.: Eva: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 546–561. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3386023, https://doi.org/10.1145/3385412.3386023

11. Dathathri, R., Saarikivi, O., Chen, H., Laine, K., Lauter, K., Maleki, S., Musuvathi, M., Mytkowicz, T.: Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 142–156. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314628, https://doi.org/10.1145/3314221.3314628
12. EPFL-LDS, T.I.S.: Lattigo v6 (Aug 2024), https://github.com/tuneinsight/lattigo
13. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford, CA, USA (2009), aAI3382729
14. Gentry, C., Halevi, S.: Implementing gentry's fully-homomorphic encryption scheme. In: Paterson, K.G. (ed.) Advances in Cryptology – EUROCRYPT 2011. pp. 129–148. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
15. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the aes circuit. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology – CRYPTO 2012. pp. 850–867. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
16. Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In: Balcan, M.F., Weinberger, K.Q. (eds.) Proceedings of The 33rd International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 48, pp. 201–210. PMLR, New York, New York, USA (20–22 Jun 2016), https://proceedings.mlr.press/v48/gilad-bachrach16.html
17. Google: Xla: Optimizing compiler for machine learning (2017), https://www.tensorflow.org/xla
18. Halevi, S., Shoup, V.: Design and implementation of helib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481 (2020), https://eprint.iacr.org/2020/1481
19. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 770–778 (June 2016). https://doi.org/10.1109/CVPR.2016.90, https://www.computer.org/10.1109/CVPR.2016.90
20. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications (2017), https://arxiv.org/abs/1704.04861
21. Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J., Keutzer, K.: Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size (2016), https://arxiv.org/abs/1602.07360
22. Krastev, A., Samardzic, N., Langowski, S., Devadas, S., Sanchez, D.: A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption. Proc. ACM Program. Lang. **8**(PLDI) (Jun 2024). https://doi.org/10.1145/3656382, https://doi.org/10.1145/3656382
23. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C., Bottou, L., Weinberger, K. (eds.) Advances in Neural Information Processing Systems. vol. 25. Curran Associates, Inc. (2012), https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
24. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: Mlir: Scaling compiler infrastructure for domain specific computation. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 2–14 (2021). https://doi.org/10.1109/CGO51591.2021.9370308

25. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998). https://doi.org/10.1109/5.726791
26. Li, L., Lai, J., Yuan, P., Sui, T., Liu, Y., Zhu, Q., Zhang, X., Xiao, L., Chen, W., Xue, J.: Ant-ace: An fhe compiler framework for automating neural network inference. In: Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization. pp. 193–208. CGO '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3696443.3708924, https://doi.org/10.1145/3696443.3708924
27. Liu, Y., Lai, J., Li, L., Sui, T., Xiao, L., Yuan, P., Zhang, X., Zhu, Q., Chen, W., Xue, J.: Resbm: Region-based scale and minimal-level bootstrapping management for fhe via min-cut. In: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. p. 924–939. ASPLOS '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3669940.3707276, https://doi.org/10.1145/3669940.3707276
28. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) Advances in Cryptology – EUROCRYPT 2010. pp. 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
29. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J. ACM **56**(6) (Sep 2009). https://doi.org/10.1145/1568318.1568324, https://doi.org/10.1145/1568318.1568324
30. Research, M.: Microsoft seal (release 3.6) (2020), https://github.com/Microsoft/SEAL
31. Sui, T., Lai, J., Li, L., Yuan, P., Liu, Y., Zhu, Q., Zhang, x., Xiao, L., Zhang, M., Xue, J.: Fhefusion: Enabling operator fusion in fhe compilers for depth-efficient dnn inference. In: Proceedings of the 24th ACM/IEEE International Symposium on Code Generation and Optimization. p. Just Accepted. CGO '26, Association for Computing Machinery, New York, NY, USA (2026)
32. Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., Devito, Z., Moses, W.S., Verdoolaege, S., Adams, A., Cohen, A.: The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically. ACM Trans. Archit. Code Optim. **16**(4) (Oct 2019). https://doi.org/10.1145/3355606, https://doi.org/10.1145/3355606
33. Viand, A., Jattke, P., Haller, M., Hithnawi, A.: Heco: Fully homomorphic encryption compiler (2023), https://arxiv.org/abs/2202.01649
34. Zama: Concrete: Tfhe compiler that converts python programs into fhe equivalent (2022), https://github.com/zama-ai/concrete
35. Zhao, J., Kruse, M., Cohen, A.: A polyhedral compilation framework for loops with dynamic data-dependent bounds. In: Proceedings of the 27th International Conference on Compiler Construction. pp. 14–24. CC 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3178372.3179509, https://doi.org/10.1145/3178372.3179509
36. Zhao, J., Li, B., Nie, W., Geng, Z., Zhang, R., Gao, X., Cheng, B., Wu, C., Cheng, Y., Li, Z., Di, P., Zhang, K., Jin, X.: Akg: Automatic kernel generation for neural processing units using polyhedral transformations. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 1233–1248. PLDI'21, ACM, New York, NY, USA (2021), https://doi.org/10.1145/3453483.3454106