CS 598 APE HW 2: Homomorphic Encryption

Ryan To
University of Illinois, Urbana-Champaign
Champaign-Urbana, Illinois, USA
rto4@illinois.edu

Kaushik Varadharajan
University of Illinois, Urbana-Champaign
Champaign-Urbana, Illinois, USA
kv22@illinois.edu

Abstract

This paper enhances a baseline homomorphic encryption implementation through algorithmic refinements, compiler optimizations, and comprehensive refactoring. The enhanced implementation delivered 200-fold to 700-fold performance gains across three benchmarks while preserving correctness of encrypted computations.

1 Introduction

Homomorphic encryption enables computation on encrypted data without decryption, but suffers from significant computational overhead. We were provided a C implementation of the Fan-Vercauteren scheme and tasked with optimizing it. We modified the implementation through algorithmic optimizations, compiler techniques, and code cleanliness. Performance was evaluated across three benchmarks: encrypted matrix multiplication, B+W image conversion, and Sobel edge detection on encrypted images. The optimized implementation achieved 200-fold to 700-fold speedup over the baseline while maintaining correctness. Most times were in the range of 500-fold speedups. Results demonstrate the effectiveness of different optimization approaches for improving homomorphic encryption performance. All code can be found here: Github

2 Optimizations

We implement a variety of optimizations to decrease runtime of the homomorphic encryption.

2.1 Compiler

The biggest compiler optimization was actually swapping compilers. Initially, the code was being compiled with gcc. We swapped to g++ to be able to take advantage of many C++ optimizations.

We add many compiler flags to this program, contribute significantly to speeding up the code.

1. `-O3`: The catch all that enables many higher level optimizations, including inlining, vectorization, and loop transformations.
2. `-funroll-loops`: Unrolls loops where possible
3. `-ffast-math`: Enables potentially rule breaking math optimizations for floats that are significantly faster.

We also tried a variety of other compiler flags that did not end up improving performance. Some of these actually slowed down the runtime. The following two were surprisingly ineffective.

1. `-march=native`: Allows for optimizations specific to the machine's CPU.
2. `-flto`: Enables Link Time Optimization, increasing speed across files

2.2 Algorithmic Changes

The most impactful optimizations target the core polynomial arithmetic operations that dominate execution time. Specifically, *poly_mul* and *poly_divmod* were expensive.

2.2.1 *poly_divmod*. The original implementation used a general polynomial division algorithm *poly_divmod* for computing remainders modulo the polynomial $x^n + 1$. Since the modulus polynomial in the FV scheme is always of the form $x^n + 1$, we implemented a specialized *poly_rem* function that exploits this structure. Division was not required, as the actual division was not being used, only the remainder. Thus, only finding the remainders is required.

The original algorithm performed full polynomial long division with $O(n^2)$ complexity, iteratively computing quotient coefficients and subtracting scaled divisor terms. Only finding the remainders would already be an optimization. However, our optimized version recognizes that for modulus $x^n + 1$, any coefficient of degree greater than or equal to n can be directly reduced by subtracting it from the coefficient at position $degree - n$, since $x^n \equiv -1 (mod x^n + 1)$. This transforms the operation from iterative division to a simple linear scan, reducing complexity to $O(n)$. This optimization directly impacts all ring operations, which form the computational bottleneck in homomorphic encryption.

2.2.2 *poly_mul*. We attempted to implement a Fast Fourier Transform multiply to replace *poly_mul*, as a $O(nlog(n))$ would be faster in theory than a $O(n^2)$ algorithm. However, this proved to be significantly slower than not including it. The FFT multiply has significant overhead that is extremely detrimental to the running time.

2.2.3 *he_decrypt* and *he_eval*. Minor optimizations were done to *he_decrypt* and *he_eval*. Decrypt was converted from a $O(n)$ execution to $O(1)$. Minor calculation optimization was applied to eval. Both of these are minor improvements to time.

2.2.4 Ring Optimizations. We streamlined the ring arithmetic functions by removing redundant modular reduction operations. The original code applied $coeff\_mod$ multiple times unnecessarily, while our version applies it only when required, reducing computational overhead in critical paths like $ring\_mul\_mod$ and $ring\_add\_mod$.

2.3 Code Cleanliness and Memory Management

2.3.1 Dynamic Memory Allocation. The most significant structural change was replacing fixed-size arrays with dynamic vectors. The original implementation allocated fixed arrays of size $MAX\_POLY\_DEGREE = 10000$ for every polynomial, regardless of actual usage. Our optimized version uses std::vector<double> that grows only as needed. This change provides massive memory savings since most polynomials in the benchmarks use only a small fraction of the maximum degree.

2.3.2 Const and Reference Variables. We converted the code base from C to C++, introducing const-correctness and pass-by-reference semantics. Functions now take const Poly & parameters instead of copying entire polynomial structures, eliminating unnecessary memory allocations and copies in hot paths. This is significantly faster, as passing variables by reference is better than passing by pointers.

2.3.3 Redundant Operation Elimination. We identified and removed several redundant operations:

1. Eliminated duplicate $coeff\_mod$ calls in ring operations
2. Removed unnecessary zeroing of coefficients that were immediately overwritten
3. Simplified polynomial creation by removing redundant initialization loops

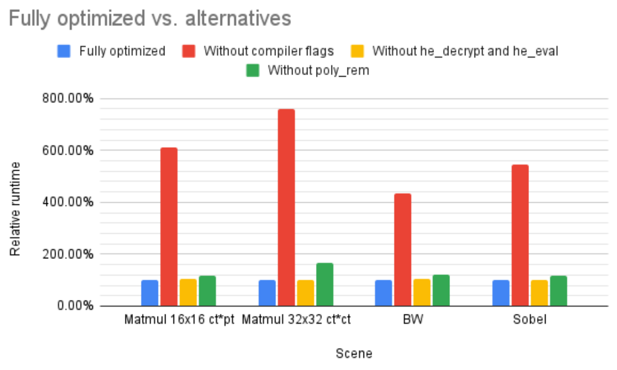

Figure 1

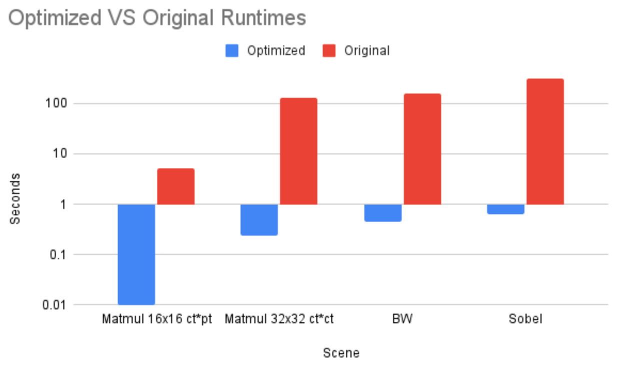

Figure 2

3 Results

Figure 1 shows the running times of our code without some optimizations relative to our final optimized code. Anything not shown is either such a great optimization it would dwarf the ones shown in scale, or are too difficult to unweave from the current code and isolate in change. Some of these changes were codebase wide refactors, which makes it difficult to revert to show. For instance, excluding the $poly\_divmod$ optimization is a 5300% increase, as an estimate.

Figure 2 shows the log_{10} runtimes of the optimized code vs the original code. It's on a log scale because the increases are of such a large magnitude.

4 Introspection

Our optimization approach focused on finding bottlenecks, and either removing or fixing them. The memory optimization through dynamic allocation proved unexpectedly impactful. The original fixed-array approach wasted enormous amounts of memory and likely caused cache misses that degraded performance.

The polynomial remainder operation emerged as the most critical hotspot, consuming the majority of execution time across all benchmarks. By specializing this operation for the FV scheme's specific polynomial modulus structure, we achieved substantial speedups. There also is potential for more to be done here to improve it further.

Some optimizations had diminishing returns. While compiler flags provided meaningful improvements, they were less impactful than the algorithmic changes. We also explored other optimizations like loop unrolling and SIMD operations, but profiling showed these provided minimal benefit, or sometimes even slowed down the program.

One disappointing implement that did not pan out was using FFTs to improve the polynomial multiplication. In hindsight, the scale of problem we are solving is too small for an FFT to actually be faster. However, if this code was to be used for significantly larger problems, a FFT multiplication would likely make this code better.

One area we didn't fully explore was parallelization. The polynomial operations have inherent parallelism, but the added complexity of thread management and synchronization may not provide benefits for the relatively small polynomial degrees used in our benchmarks. This could be valuable for future work with larger parameter sets.

Regarding fidelity of results, the first 3 tests maintain all accuracy. The results are perfect matches with the results of the original code runs. The only test that may have some variance is the edge detector. As seen below in figure 3, the edge detector reuslts are similar on an eyeball test. There may be some differences, but that is expected, as described in the writeup. The main structures of the edges exist.

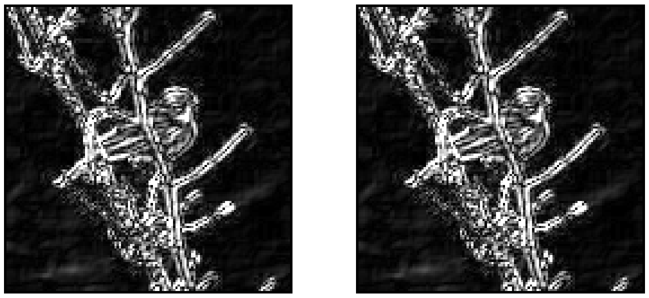

Figure 3. Comparison of originally ran sobel edge detector and the optimized one.