Code Generation for Cryptographic Kernels using Multi-word Modular Arithmetic on GPU

Naifeng Zhang
Carnegie Mellon University
Pittsburgh, USA
naifengz@cmu.edu

Franz Franchetti
Carnegie Mellon University
Pittsburgh, USA
franzf@andrew.cmu.edu

Abstract

Fully homomorphic encryption (FHE) and zero-knowledge proofs (ZKPs) are emerging as solutions for data security in distributed environments. However, the widespread adoption of these encryption techniques is hindered by their significant computational overhead, primarily resulting from core cryptographic operations that involve large integer arithmetic. This paper presents a formalization of multi-word modular arithmetic (MoMA), which breaks down large bit-width integer arithmetic into operations on machine words. We further develop a rewrite system that implements MoMA through recursive rewriting of data types, designed for compatibility with compiler infrastructures and code generators. We evaluate MoMA by generating cryptographic kernels, including basic linear algebra subprogram (BLAS) operations and the number theoretic transform (NTT), targeting various GPUs. Our MoMA-based BLAS operations outperform state-of-the-art multi-precision libraries by orders of magnitude, and MoMA-based NTTs achieve near-ASIC performance on commodity GPUs.

CCS Concepts: • **Theory of computation → Rewrite systems**; • **Security and privacy → Cryptography**; • **Computing methodologies → Parallel computing methodologies**.

Keywords: Multi-word modular arithmetic, code generation, rewrite system, BLAS, number theoretic transform, cryptography

ACM Reference Format:
Naifeng Zhang and Franz Franchetti. 2025. Code Generation for Cryptographic Kernels using Multi-word Modular Arithmetic on GPU. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3696443.3708948

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708948

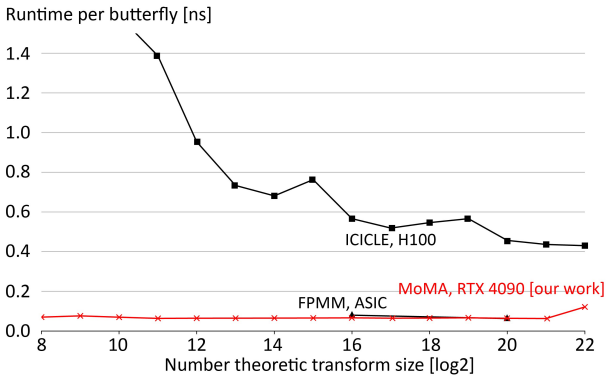


Figure 1. Performance of 256-bit NTT on GPUs and ASIC (lower is better). On NVIDIA GeForce RTX 4090, MoMA-based NTT outperforms state-of-the-art cryptographic acceleration library [29] running on NVIDIA H100 by an average of 14 times and achieves near-ASIC [63] performance.

1 Introduction

As data security becomes increasingly critical in distributed and cloud computing environments, advanced encryption schemes such as fully homomorphic encryption (FHE) and zero-knowledge proofs (ZKPs) are emerging as promising solutions for privacy preservation. However, these encryption techniques remain impractical for widespread use due to their computational overhead, much of which originates from core cryptographic operations involving large integer arithmetic. These operations are typically performed on large finite fields, where integer bit-width ranges from 256 to 768 bits for ZKP applications and from hundreds to over 1,000 bits for FHE-based systems.

To manage this computational cost, current ZKP implementations often rely on arbitrary-precision libraries such as the GNU Multiple Precision (GMP) library [22] or programming languages such as Rust and Python, which support large integer arithmetic natively. However, while these external libraries offer fundamental support for large integer operations, they come with limitations. For instance, many ZKP libraries are written in languages like Python that are not performance-oriented, or are constrained to CPU execution due to factors such as GMP's lack of support for GPUs. As integer bit-widths in popular FHE schemes extend to thousands of bits, the residue number system (RNS) is used to represent these large numbers via much smaller integers

(residues) that fit within machine words, typically 32 or 64 bits. However, using RNS with small residues introduces additional computational overhead for modulus raising and reduction [12] and requires frequent bootstrapping, which is a highly computationally intensive process in FHE [2].

To address these challenges, we formally define *multi-word modular arithmetic* (MoMA), in which large integers are represented as multiple machine words to exploit the native performance of single-word arithmetic. While similar ideas have been explored in the past, such as Intel's work in the 1980s [24], to the best of our knowledge, our work provides a novel formalization of multi-word modular arithmetic that *systematically* decomposes large integer arithmetic into machine word operations. This formalization directly enables many optimizations in symbolic space, including reducing redundant operations for inputs with non-power-of-two bit-widths as opposed to simply zero-padding the inputs. Moreover, we introduce a program transformation pass that implements MoMA as a set of rewrite rules in a term rewriting system. This pass, integrable with compiler frameworks and code generators, operates on data types and recursively transforms computations involving large data types into equivalent sequences of operations on smaller data types, continuing until all data types used in the kernel are natively supported by the machine.

In this paper, we focus on targeting GPUs due to their massive parallelism and high on-chip performance. We implement MoMA in SPIRAL [20], a code generator that produces highly efficient implementations for various hardware architectures and provides strong support for implementing mathematically formal rule systems. To evaluate our approach, we first implement several basic linear algebra subprograms (BLAS) operations on finite fields, which correspond to polynomial arithmetic operations that are fundamental to many advanced cryptographic schemes [1]. Next, we implement a more complex kernel, the number theoretic transform (NTT), which is a critical component in FHE and ZKPs. NTTs account for over 90% of the runtime in many FHE schemes and approximately 30% in ZKP applications [19, 60]. Our MoMA-based implementation of BLAS operations outperforms state-of-the-art multi-precision libraries by orders of magnitude. Additionally, our MoMA-based NTT implementation achieves near-ASIC performance on a commodity GPU, as shown in Figure 1, while maintaining the flexibility to support multiple input bit-widths across various NTT sizes.

By utilizing MoMA, we offer a more performant alternative to ZKP libraries that rely on GMP, Python or Rust for large integer arithmetic. In the context of FHE, transitioning from 64-bit to 128-bit (and higher bit-width) residues in the RNS representation using MoMA creates opportunities to reduce the frequency of costly operations, such as bootstrapping. In some FHE schemes, MoMA could potentially eliminate the need for RNS entirely if the original bit-width is in the hundreds and the cost of using MoMA to decompose the integer is less than the cost of employing RNS. Historically, extending integer bit-width beyond the machine word width has been viewed as highly expensive in the field of cryptography. MoMA's ability to reduce the computational cost of large integer arithmetic could potentially enable innovative cryptographic algorithms that are not constrained by the limitations of 64-bit word width.

Contributions. This paper makes the following contributions:

1. A formal definition of multi-word modular arithmetic (MoMA) that decomposes integer arithmetic on large bit-widths into native machine word operations.
2. A rewrite system that implements MoMA, compatible with compiler frameworks and code generators, that recursively rewrites computations of large data types into an equivalent sequence of operations that use smaller data types.
3. A demonstration of MoMA-based BLAS operations outperforming state-of-the-art multi-precision libraries, and MoMA-based NTTs achieving near-ASIC performance on commodity GPUs. By making large integer arithmetic efficient, our work may enable critical innovations in cryptographic algorithms.

2 Background

In this section, we begin by outlining the mathematical background of modular and multi-digit arithmetic. Next, we introduce polynomial operations built on modular arithmetic, which represent the primary computational bottlenecks in cryptographic applications like FHE and ZKPs.

2.1 Modular Arithmetic

When operating on an integer ring modulo q, $\mathbb{Z}_q$, additions, subtractions, and multiplications of two integers are defined as

$$\begin{aligned} c &= a + b \mod q, \\ c &= a - b \mod q, \\ c &= ab \mod q, \end{aligned} \tag{1}$$

where $a, b, c \in \mathbb{Z}_q$. However, modulo operation is significantly more costly than basic operations such as addition and multiplication on standard off-the-shelf hardware. To efficiently implement modular arithmetic, we aim to replace the modulo operations with cheaper operations instead. Using the definition that $0 \leq a < q$ and $0 \leq b < q$ for any $a, b \in \mathbb{Z}_q$, modular addition within the ring can be done by

$$c = \begin{cases} a + b - q, & \text{if } (a + b) > q, \\ a + b, & \text{otherwise.} \end{cases} \tag{2}$$

Modular subtraction can be done by

$$c = \begin{cases} a - b + q, & \text{if } a < b, \\ a - b, & \text{otherwise.} \end{cases} \tag{3}$$

For modular multiplication, we use Barrett reduction [4], a popular approach widely used within the cryptography community [1, 7], for general modulo (that is, not a specific modulus such as Goldilock prime [26]). Modular multiplication using Barrett reduction with the floor operation is defined as

$$c = ab - \lfloor ab/q \rfloor q. \tag{4}$$

Note that the result is exact. We discuss how to efficiently implement Equation 4 in Section 3.1.

2.2 Multi-Digit Arithmetic

Multi-digit arithmetic involves the execution of basic mathematical operations (i.e., addition, subtraction, multiplication, and division) on numbers with multiple digits. Formally, we define a function $[]_z : \mathbb{Z}^n \to \mathbb{Z}$, parameterized by base digit z, as

$$[x_0, x_1, \ldots, x_{n-1}]_z = x_0 z^{n-1} + x_1 z^{n-2} + \ldots + x_{n-1} = x. \tag{5}$$

For example, in the decimal system where $z = 10$, we can write $[8, 9]_{10} = 8 \cdot 10 + 9 = 89$. We can calculate n, given z and x, as $n = \lceil \log_z x \rceil$. It is important to note that, at this stage, we are discussing the mathematical concepts without any reference to implementation. This means that the value of z does not necessarily need to fit within a machine word and can be arbitrarily large or small. In Section 3, we will discuss how to efficiently implement multi-digit arithmetic, and how to combine it with modular arithmetic, when each digit is a machine word.

Here, we demonstrate multi-digit arithmetic on 2 digits. Let $a = [a_0, a_1]_z = a_0 z + a_1$ and $b = [b_0, b_1]_z = b_0 z + b_1$. The schoolbook multi-digit addition, $c = a + b$, can be written as

$$\begin{aligned} [\delta, c_2]_z &= a_1 + b_1, \\ [c_0, c_1]_z &= a_0 + b_0 + \delta, \end{aligned} \tag{6}$$

where $c = [c_0, c_1, c_2]_z$ and $\delta \in \{0, 1\}$.

The schoolbook multi-digit subtraction, $c = a - b$, can be written as

$$\begin{aligned} c_1 &= a_1 - b_1, \\ \delta &= \begin{cases} 1, & \text{if } a_1 < b_1, \\ 0, & \text{otherwise}, \end{cases} \\ c_0 &= a_0 - b_0 - \delta, \end{aligned} \tag{7}$$

where $c = [c_0, c_1]_z$.

The schoolbook multiplication, $c = ab$, can be written as

$$c = (a_0 b_0) z^2 + (a_0 b_1 + a_1 b_0) z + a_1 b_1, \tag{8}$$

where each addition can be further broken down using the aforementioned multi-digit addition. The Karatsuba algorithm [31] is a divide-and-conquer method for multiplying large numbers by recursively breaking down the multiplication of two n-digit numbers into three multiplications of $n/2$-digit numbers, along with some additions and subtractions. Formally,

$$c = (a_0 b_0) z^2 + \big((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1\big) z + a_1 b_1, \tag{9}$$

where each addition and subtraction can be further broken down using the aforementioned multi-digit addition and subtraction. In n-digit arithmetic, the addition of two n-digit integers produces a result with at most $n + 1$ digits, while subtraction yields a result with at most n digits, and multiplication can produce up to $2n$ digits.

2.3 Polynomial Operations and NTT

Polynomial operations and NTT are fundamental cryptographic kernels in advanced encryption schemes such as FHE and ZKPs. In this section, we provide a brief overview of these concepts.

Polynomial addition and subtraction. Polynomial operations, especially polynomial multiplications with coefficients reside in $\mathbb{Z}_q$, are the building blocks of advanced cryptographic schemes such as FHE and ZKPs [5, 47]. Let f and g denote two polynomials of degree n, where $f = \sum_{i=0}^{n} a_i x^i$ and $g = \sum_{j=0}^{n} b_j x^j$. The addition and subtraction of f and g are defined as the point-wise addition and subtraction of both polynomials' coefficients (a_i and b_j), respectively. Prior work [1] has shown that point-wise multiplication of both polynomials' coefficients is also commonly utilized in FHE schemes. Each of the above three operations can be efficiently implemented using vector operations, where a vector of length $n + 1$ represents the coefficients of the degree n polynomial.

We can then utilize the BLAS abstraction [6] to describe point-wise polynomial operations. Vector addition and subtraction can be interpreted as variants of a BLAS Level 1 operation known as axpy, which is formally defined as

$$y = ax + y, \tag{10}$$

where x and y are vectors and a is a scalar. Point-wise vector multiplication can be seen as a special case of gemv, a BLAS Level 2 operation that computes a general matrix-vector multiplication.

Polynomial multiplication. The multiplication of f and g is defined as

$$f(x)g(x) = \sum_{j=0}^{2n} \sum_{i=0}^{j} a_i b_{j-i} x^j, \tag{11}$$

which is an $O(n^2)$ operation. Similarly to how the Fourier transform converts a signal from the time domain to the frequency domain, NTT transforms a polynomial from its coefficient form (e.g., $f(x) = x^3 + 5x^2 + 2x + 1 \bmod 3$) to its evaluation form (e.g., $\{f(0), f(1), f(2), f(3), f(4)\}$), thereby reducing the time complexity of polynomial multiplication from $O(n^2)$ to $O(n \log n)$. Formally, an n-point NTT is defined as

$$y(k) = \sum_{j=0}^{n-1} x(j) \omega_n^{jk} \bmod p, \quad 0 \le k \le n-1, \tag{12}$$

where ω_n is the n-th primitive root of unity. As advanced encryption schemes heavily rely on polynomial arithmetic that comes with a prohibited computational overhead, NTT has been widely adopted to accelerate polynomial multiplications. Prior work has shown that NTT accounts for over 90% of FHE-based application execution time in practice [19] and around 30% of execution time for ZKP-based workloads [60]. Therefore, we focus on the previously mentioned BLAS operations and NTT as our primary cryptographic kernels, as they encompass a majority of practical cryptographic workloads [1].

3 Multi-word Modular Arithmetic

We define a machine word as the largest integer data type that is efficiently supported by the instruction set or hardware. Typically, a machine word is the largest data type that can fit into a single general-purpose register. For example, on x86-64 architectures, a machine word has 64 bits. To fully leverage the computing efficiency of natively supported machine words, we propose using these words to construct large integer arithmetic when the integer bit-width exceeds the native machine word width. We combine multi-digit arithmetic with modular arithmetic, treating each machine word as a digit, to develop a system called multi-word modular arithmetic (MoMA). Using our multi-digit definition from Equation 5, in MoMA, we now express an integer x as follows:

$$x = [x_0, x_1, \ldots, x_{k-1}]_{2^{\omega_0}}, \tag{13}$$

where $k = \lceil \log_{2^{\omega_0}} x \rceil$ and ω_0 is the machine word width. For simplicity, we denote this as

$$x = [x_0^{\omega_0}, x_1^{\omega_0}, \ldots, x_{k-1}^{\omega_0}], \tag{14}$$

where each $x_i^{\omega_0}$, for $0 \leq i < k$, is an integer of bit-width ω_0. This notation is used to build the formal rule system in Section 4.

As we map from multi-digit arithmetic to its implementation using machine words, we move from the mathematical level to the algorithm level where overflow must be explicitly managed when intermediate results exceed the machine word width. Addressing these challenges is the primary focus of MoMA and will be thoroughly discussed in this section. We begin with single-word modular arithmetic, where the input integer x fits within a single machine word. As discussed in Section 2.2, we have to handle results from addition and multiplication that exceed the machine word width. Next, we discuss double-word modular arithmetic, which builds upon single-word modular arithmetic. Double-word modular arithmetic is fundamental to MoMA, as it allows us to represent an arbitrarily large input integer as a double-word type (with an abstract machine word whose bit-width is half of the input bit-width) and decompose it into single-word operations. MoMA applies this decomposition recursively until the abstract machine word is reduced to an actual machine word that can be executed natively.

```
#define MBITS 60
typedef uint64_t i64;
typedef unsigned __int128 i128;

// addition
void _sadd(i128 *c, i64 a, i64 b) {
    *c = (i128) a + (i128) b; }

// modular addition
void _saddmod(i64 *c, i64 a, i64 b, i64 q) {
    i128 t; t = (i128) a + (i128) b;
    *c = t > q ? (i64) (t - (i128) q) : (i64) t; }

// subtraction
void _ssub(i64 *c, i64 a, i64 b) { *c = a - b; }

// modular subtraction
void _ssubmod(i64 *c, i64 a, i64 b, i64 q) {
    i64 t; t = a - b; *c = a < b ? t + q : t; }

// multiplication
void _smul(i128 *c, i64 a, i64 b) {
    *c = (i128) a * (i128) b; }

// modular multiplication using Barrett reduction
void _smulmod(i64 *c, i64 a, i64 b, i64 q, i64 mu) {
    i128 t, r; t = (i128) a * (i128) b; r = t;
    r >>= (MBITS - 2); r *= (i128) mu;
    r >>= (MBITS + 5); t -= (i128) r * (i128) q;
    // correct off-by-one approx. error
    *c = t > q ? (i64) (t - (i128) q) : (i64) t; }
```

Listing 1. Single-word modular arithmetic.

3.1 Single-Word Modular Arithmetic

We begin with single-word arithmetic, where all inputs fit entirely within a machine word. When the single-word data type matches the machine word, the single-word arithmetic is usually fully supported by the compiler, allowing us to directly implement the mathematical definitions from Section 2.1. Note that to fully support single-word arithmetic, the compiler must also provide a double-word representation. This is necessary to store results from single-word operations, even though full double-word arithmetic (which requires quad-word representation) is typically unavailable. For example, in C and CUDA, operations on `uint64_t` are fully supported, as the 128-bit `unsigned __int128` type exists for overflow handling. Utilizing the compiler-supported double-word data type allows compilers (e.g., `nvcc`) to leverage specialized instructions such as add-with-carry for carry propagation during compilation. In Listing 1, we illustrate both modular and non-modular arithmetic, using `uint64_t` in C as the single-word data type.

Single-word addition, subtraction, and multiplication are natively supported by the compiler. For modular addition and modular multiplication, we need to cast the input using a double-word datatype because the results might exceed the

bit-width of a single word. For modular addition, we implement the arithmetic according to Equation 2 and for modular subtraction, it is a direct translation from Equation 3. We show how to implement efficient modular multiplication using Barrett reduction according to Equation 4. Note that to implement what is shown in Equation 4 efficiently, we need to carefully choose the implementation of $\lfloor ab/q \rfloor$. The straightforward approach is to use integer division, which truncates the results toward zero in C. However, division is computationally expensive and can be vulnerable to timing attacks if it is not implemented as a constant-time operation on certain architectures. As a result, the cryptography community has developed methods to implement $\lfloor ab/q \rfloor$ using only multiplication and bit shifts, which are significantly more efficient than division. This approach relies on a precomputed value, μ, derived as follows. We want to approximate $1/q$ in $\lfloor ab/q \rfloor$ by

$$1/q = \mu/2^k, \tag{15}$$

so that we can compute $\lfloor ab/q \rfloor$ with $\lfloor ab\mu/2^k \rfloor$ using the right shifts. Therefore, we precompute μ using

$$\mu = \lfloor 2^k/q \rfloor, \tag{16}$$

so that μ is an integer. Now,

$$\mu/2^k = \lfloor 2^k/q \rfloor/2^k \leq 1/q. \tag{17}$$

We need to correct this off-by-one error using a conditional subtraction that corresponds to the last line of code in Listing 1. The entire Barrett reduction now becomes

$$ab - \lfloor ab\lfloor 2^k/q \rfloor/2^k \rfloor q. \tag{18}$$

We only need multiplications and shifts to compute Equation 4 rather than using divisions if we precompute $\lfloor 2^k/q \rfloor$ as μ using division for once for the same modulus q.

3.2 Double-Word Modular Arithmetic

We define double-word integers as integers with a bit-width of 2ω, where ω denotes the bit-width of a single word. While single-word modular arithmetic is relatively straightforward, as illustrated in Listing 1, double-word arithmetic is significantly more complex. This complexity arises because adding or multiplying two double-word integers can lead to quad-word (integers with a bit-width of 4ω) results, which further complicates arithmetic operations. We will start with double-word modular addition and subtraction.

Addition and subtraction. Listing 2 shows the C implementation of double-word modular addition and subtraction. As a quad-word cannot be natively represented, we break it down to four single words $a = [a_0^{64}, a_1^{64}, a_2^{64}, a_3^{64}]$ as defined in Equation 14. In the implementation of `_dadd`, carry extraction and propagation must be handled explicitly in the code, as this cannot be managed automatically by the compiler. Similarly, `_dsub` requires explicit management of the borrow. For the modulo operation, the only missing part is the comparison in the conditional assignments within `_saddmod` and `_ssubmod`. Thus, we implement `_dlt` to perform comparisons between two double words. In this specific example, we assume that the single-word data type corresponds to the machine word (i.e., `uint64_t`). This implies that for `_dsub` and `_dlt`, native operations can be utilized if we combine $[a_0^{64}, a_1^{64}]$ and $[b_0^{64}, b_1^{64}]$ into a 128-bit representation and use the natively supported 128-bit subtraction and comparison. However, we present an implementation that does not rely on double-word support to demonstrate how operations are handled when the abstract single word is not the machine word. In Section 4, we can then observe a clear mapping from these double-word operations to formal rewrite rules within MoMA.

```
1  // addition: quad = double + double
2  void _dadd(i64 *c0, i64 *c1, i64 *c2, i64 *c3,
3             i64 a0, i64 a1, i64 b0, i64 b1) {
4      i128 s; int cr; s = (i128) a1 + (i128) b1;
5      *c3 = (i64) s; cr = s >> 64;
6      s = (i128) a0 + (i128) b0 + (i128) cr;
7      *c2 = (i64) s; *c1 = s >> 64; *c0 = 0; }
8
9  // subtraction
10 void _dsub(i64 *c0, i64 *c1, i64 a0, i64 a1,
11            i64 b0, i64 b1) {
12     int br; *c1 = a1 - b1; br = a1 < b1;
13     *c0 = a0 - b0 - br; }
14
15 // less than
16 void _dlt(int *c, i64 a0, i64 a1, i64 b0, i64 b1) {
17     int i0, i1, i2, i3; i0 = (a0 < b0);
18     i1 = (a0 == b0); i2 = (a1 < b1);
19     i3 = i1 && i2; *c = i0 || i3; }
20
21 // modular addition
22 void _daddmod(i64 *c0, i64 *c1, i64 a0, i64 a1,
23               i64 b0, i64 b1, i64 q0, i64 q1) {
24     i64 t0, t1, t2, t3, t4, t5; int i;
25     _dadd(&t0, &t1, &t2, &t3, a0, a1, b0, b1);
26     _dlt(&i, q0, q1, t2, t3);
27     _dsub(&t4, &t5, t2, t3, q0, q1);
28     *c0 = i ? t4 : t2; *c1 = i ? t5 : t3; }
29
30 // modular subtraction
31 void _dsubmod(i64 *c0, i64 *c1, i64 a0, i64 a1,
32               i64 b0, i64 b1, i64 q0, i64 q1) {
33     i64 t0, t1, t2, t3, t4, t5; int i;
34     _dsub(&t0, &t1, a0, a1, b0, b1);
35     _dadd(&t2, &t3, &t4, &t5, t0, t1, q0, q1);
36     _dlt(&i, a0, a1, b0, b1);
37     *c0 = i ? t4 : t0; *c1 = i ? t5 : t1; }
```

Listing 2. Double-word modular addition and subtraction.

Schoolbook multiplication. In Listing 3, we present the implementation of schoolbook multiplication, denoted as `_dmuls`, as defined by Equation 8. In MoMA, representing large integers as multiple words simplifies operations such as shifting by one or multiples of word width. For example, shifting $a = [a_0^{64}, a_1^{64}, a_2^{64}, a_3^{64}]$ left by one word width

```
// addition: quad = quad + quad
void _qadd(i64 *c0, i64 *c1, i64 *c2, i64 *c3,
           i64 a0, i64 a1, i64 a2, i64 a3,
           i64 b0, i64 b1, i64 b2, i64 b3) {
    i128 s; i64 t0, t1; int cr;
    _dadd(&t0, &t1, c2, c3, a2, a3, b2, b3);
    // t1 is either 0 or 1
    s = (i128) a1 + (i128) b1 + (i128) t1;
    *c1 = (i64) s; cr = s >> 64;
    s = (i128) a0 + (i128) b0 + (i128) cr;
    *c0 = (i64) s; }

// schoolbook multiplication
void _dmuls(i64 *c0, i64 *c1, i64 *c2, i64 *c3,
            i64 a0, i64 a1, i64 b0, i64 b1) {
    i64 t0, t1, t2, t3, t4, t5, t6, t7,
        t8, t9, t10, t11; i128 s;
    s = (i128) a1 * (i128) b1;
    t0 = s >> 64; t1 = (i64) s;
    s = (i128) a0 * (i128) b0;
    t2 = s >> 64; t3 = (i64) s;
    s = (i128) a0 * (i128) b1;
    t4 = s >> 64; t5 = (i64) s;
    s = (i128) a1 * (i128) b0;
    t6 = s >> 64; t7 = (i64) s;
    // a0b1 + a1b0
    _dadd(&t8, &t9, &t10, &t11, t4, t5, t6, t7);
    // a0b0z^2 + (a0b1 + a1b0)z + a1b1
    _qadd(c0, c1, c2, c3, t2, t3, t0, t1,
          t9, t10, t11, 0); }
```

Listing 3. Double-word schoolbook multiplication.

results in $[a_1^{64}, a_2^{64}, a_3^{64}, 0]$. To complete Equation 8, we need to support the addition of two quad-words. This is achieved through _qadd, which extends _dadd by applying the same multi-word addition strategy. We have omitted the example code for Karatsuba multiplication due to space constraints; however, it can be derived using Equation 9.

Modular multiplication. The most complex operation for double-word arithmetic is modular multiplication. As illustrated in _dmulmod in Listing 4, it involves three multiplications, two right shifts by non-multiples of the word width, and a final conditional subtraction. _dmulmod is built upon _smulmod from Listing 1. To handle shifting by MBITS − 2, we use _qshr, which shifts a quad-word by k bits, where k ranges from one single-word width to two single-word width. Shifting right by MBITS + 5 is more straightforward, as it involves discarding the lower part of the quad-word. Note that in the second multiplication, the lower part of the result is discarded due to the subsequent shift operation.

To implement the double-word version of `t -= r * q` in _smulmod, we need to perform a subtraction between two quad-words. According to Barrett reduction, t can only be either c or $c + q$, where $c = ab \bmod q$. Since c is less than q and q has a bit-width smaller than the word width, both c and $c + q$ are less than the word width. Consequently, we only need to subtract the lower part of rq from the lower part of t, because the final result will fit within a double word.

```
#define MBITS 124

// right shift: double = quad >> k, where k in [64, 128]
void _qshr(i64 *c0, i64 *c1, i64 a0, i64 a1,
           i64 a2, i64 a3, int b) {
    i64 t0, t1, t2, t2, t4, t5, t6, t7; int i0, i1;
    i0 = b - 64; i1 = 128 - b; t0 = a2 >> i0;
    /* a mask of 1s */
    t1 = (i64) 1; t2 = t1 << i0; t3 = t2 - 1;
    t4 = a0 & t3; t5 = t4 << i1; t6 = a1 >> i0;
    *c0 = t5 | t6; t7 = a1 << i1; *c1 = t7 || t0; }

// modular multiplication using Barrett reduction
void _dmulmod(i64 *c0, i64 *c1, i64 a0, i64 a1,
              i64 b0, i64 b1, i64 q0, i64 q1,
              i64 mu0, i64 mu1) {
    i64 t0, t1, t2, t3, t4, t5, t6, t7,
        t8, t9, t10, t11, t12, t13, t14, t15,
        t16, t17, t18, t19, t20, t21; int i;
    _dmuls(&t0, &t1, &t2, &t3, a0, a1, b0, b1);
    _qshr(&t4, &t5, t0, t1, t2, t3, MBITS-2)
    // t8, t9 will not be used
    _dmuls(&t6, &t7, &t8, &t9, t4, t5, mu0, mu1);
    // [t10, t13] = [t6, t7, t8, t9] >> MBITS+5
    t10 = t6 >> 1; t11 = t6 << 63;
    t12 = t7 >> 1; t13 = t11 | t12;
    // t14, t15 will not be used
    _dmuls(&t14, &t15, &t16, &t17, t10, t13, q0, q1);
    // optimization given that the first half matches
    _dsub(&t18, &t19, t2, t3, t16, t17);
    _dsub(&t20, &t21, t18, t19, q0, q1);
    _dlt(&i, t18, t19, q0, q1)
    *c0 = i ? t18 : t20; *c1 = i ? t19 : t21; }
```

Listing 4. Double-word modular multiplication.

Therefore, _dsub is used, and the higher part of the third multiplication is not needed.

Multi-word modular arithmetic via recursion. Given our formal definition of double-word modular arithmetic, we can now define MoMA through recursion. Let the bit-width of the input integer be λ. We start by applying double-word modular arithmetic to break it down into equivalent computations using data types with bit-width $\lambda/2$. This process is recursively applied to the resulting $\lambda/2$ bit-width data types, continuing until $(\lambda/2^k) \leq \omega_0$, where ω_0 is the machine word width and k is the number of recursion steps. For example, if the input integer a is 512 bits and the machine word width is 64 bits, three recursion steps are required. The data type breakdown would proceed as follows:

$$a = [a_0^{256}, a_1^{256}] = [a_0^{128}, a_1^{128}, a_2^{128}, a_3^{128}] = [a_0^{64}, a_1^{64}, \ldots, a_7^{64}].$$

The complexity of the associated computations will increase significantly as we recursively break down the data type.

4 Code Generation: Rewriting on Data Types

To implement MoMA, we introduce a rule system composed of numerous rewrite rules that operate on integer data types. This system recursively decomposes operations involving

Table 1. Multi-word modular arithmetic core rewrite rules.

$$a^{2\omega} \rightarrow [a_0^\omega, a_1^\omega] \tag{19}$$

$$c_0^\omega = \lfloor [a_0^\omega, a_1^\omega]/2^\omega \rfloor \rightarrow c_0^\omega = a_0^\omega \tag{20}$$

$$c_0^\omega = [a_0^\omega, a_1^\omega] \bmod 2^\omega \rightarrow c_0^\omega = a_1^\omega \tag{21}$$

$$[c_0^1, c_1^\omega, c_2^\omega] = [a_0^\omega, a_1^\omega] + [b_0^\omega, b_1^\omega] \rightarrow [\delta_0^1, c_2^\omega] = a_1^\omega + b_1^\omega,\ [c_0^1, c_1^\omega] = \delta_0^1 + a_0^\omega + b_0^\omega \tag{22}$$

$$[c_0^1, c_1^\omega] = a_1^\omega + b_1^\omega \rightarrow c_0^1 = \lfloor (a_1^\omega + b_1^\omega)/2^\omega \rfloor,\ c_1^\omega = (a_1^\omega + b_1^\omega) \bmod 2^\omega \tag{23}$$

$$[c_0^\omega, c_1^\omega] = [a_0^1, a_1^\omega, a_2^\omega] \bmod [q_0^\omega, q_1^\omega] \rightarrow \begin{aligned} &\delta_0^1 = [q_0^\omega, q_1^\omega] < [a_1^\omega, a_2^\omega], \\ &\delta_1^1 = (0 < a_0^1) \vee \left((a_0^1 =_? 0) \wedge \delta_0^1\right), \\ &[b_0^\omega, b_1^\omega] = [a_1^\omega, a_2^\omega] - [q_0^\omega, q_1^\omega], \\ &[c_0^\omega, c_1^\omega] = \begin{cases} [b_0^\omega, b_1^\omega], & \text{if } \delta_1^1 =_? 1, \\ [a_1^\omega, a_2^\omega], & \text{otherwise} \end{cases} \end{aligned} \tag{24}$$

$$[c_0^\omega, c_1^\omega] = [a_0^\omega, a_1^\omega] - [b_0^\omega, b_1^\omega] \rightarrow c_1^\omega = a_1^\omega - b_1^\omega,\ \delta_0^1 = a_1^\omega < b_1^\omega,\ c_0^\omega = a_0^\omega - b_0^\omega - \delta_0^1 \tag{25}$$

$$\delta_0^1 = [a_0^\omega, a_1^\omega] < [b_0^\omega, b_1^\omega] \rightarrow \delta_0^1 = (a_0^\omega < b_0^\omega) \vee \left((a_0^\omega =_? b_0^\omega) \wedge (a_1^\omega < b_1^\omega)\right) \tag{26}$$

$$\delta_0^1 = [a_0^\omega, a_1^\omega] =_? [b_0^\omega, b_1^\omega] \rightarrow (a_0^\omega =_? b_0^\omega) \wedge (a_1^\omega =_? b_1^\omega) \tag{27}$$

$$[c_0^\omega, c_1^\omega, c_2^\omega, c_3^\omega] = [a_0^\omega, a_1^\omega] \cdot [b_0^\omega, b_1^\omega] \rightarrow \begin{aligned} &[d_0^\omega, d_1^\omega] = a_1^\omega \cdot b_1^\omega,\ [e_0^\omega, e_1^\omega] = a_0^\omega \cdot b_0^\omega, \\ &[f_0^\omega, f_1^\omega] = a_0^\omega \cdot b_1^\omega,\ [g_0^\omega, g_1^\omega] = a_1^\omega \cdot b_0^\omega, \\ &[h_0^1, h_1^\omega, h_2^\omega] = [f_0^\omega, f_1^\omega] + [g_0^\omega, g_1^\omega], \\ &[c_0^\omega, c_1^\omega, c_2^\omega, c_3^\omega] = [e_0^\omega, e_1^\omega, d_0^\omega, d_1^\omega] + [h_0^1, h_1^\omega, h_2^\omega, 0] \end{aligned} \tag{28}$$

$$[c_0^\omega, c_1^\omega, c_2^\omega, c_3^\omega] = [a_{0\text{-}3}^\omega] + [b_{0\text{-}3}^\omega] \rightarrow \begin{aligned} &[\delta_0^1, c_3^\omega] = a_3^\omega + b_3^\omega,\ [\delta_1^1, c_2^\omega] = a_2^\omega + b_2^\omega + \delta_0^1, \\ &[\delta_2^1, c_1^\omega] = a_1^\omega + b_1^\omega + \delta_1^1,\ [0, c_0^\omega] = a_0^\omega + b_0^\omega + \delta_2^1 \end{aligned} \tag{29}$$

We use the symbol of equality (=) to denote assignment and the symbol of equality with a question mark ($=_?$) to indicate a comparison of equality. We assume that any comparison evaluates to 1 if true and 0 if false. For brevity, we use $[a_{0\text{-}n}^\omega]$ to represent the sequence $[a_0^\omega, \ldots, a_n^\omega]$.

large data types into equivalent operations using smaller data types until all operations are conducted with data types natively supported by the machine. In this section, we formally define the rule system, which can be integrated into compiler infrastructures and code generators. At each recursive step, we treat the current maximal integer data type as a double word, as discussed in Section 3, and decompose the computations into equivalent operations with single words. This process continues until the single word at that step matches the machine word width. Therefore, to apply the rule system as a program transformation pass, we assume that the input bit-width for application kernels is known at compile/code generation time.

In each recursion step, let the single word width be denoted by ω and the double word width is 2ω. We use x^ω to represent a single-word integer of bit-width ω. For instance, when $\omega = 256$, a double-word integer can be defined as a^{512} and decomposed into two single-word integers using the definition from Equation 14: $a^{512} = [a_0^{256}, a_1^{256}]$. This data type breakdown process is formally defined as rewrite rule (19). For more complex rules, we use the symbol of equality (=) to denote assignment. For example, $c^{2\omega} = a^\omega + b^\omega$ indicates that the result of $a^\omega + b^\omega$ is assigned to $c^{2\omega}$. To denote the comparison of equality, we use the equality symbol with a question mark ($=_?$). We use δ^1 to represent the result of a comparison or a carry/borrow bit, with the assumption that any comparison evaluates to 1 when true and 0 when false.

The core rewrite rules used to implement MoMA are presented in Table 1. Since each rule reduces the integer bit-width required during computation, the largest data type incurred from the right-hand side computations is always smaller than the largest data type incurred from the left-hand side operation. On the right-hand side, the sequence of computations is crucial, as each operation must be executed from left to right and top to bottom.

Example: rewriting modular addition. We now show a concrete example of applying the rewrite rules shown in Table 1 to break down a double-word modular addition:

$$c^{2\omega} = (a^{2\omega} + b^{2\omega}) \bmod q^{2\omega}, \tag{30}$$

where q is the modulus. When $\omega = 64$, we can implement a^{ω} using the `uint64_t` data type and $a^{2\omega}$ using the `unsigned __int128` data type in C. The above operation can then be directly mapped to the implementation of `_daddmod` in Listing 2:

```
1  void _daddmod(i64 *c0, i64 *c1, i64 a0, i64 a1,
2                i64 b0, i64 b1, i64 q0, i64 q1)
```

For illustration purposes, in this example we consider ω to be the bit-width of the abstract single word at a recursion step that is not the final recursion step, that is, $\omega > \omega_0$, where ω_0 is the machine word width.

To begin with, by (19), we can decompose $c^{2\omega}$ into $[c_0^{\omega}, c_1^{\omega}]$ by performing floor division and modulo operation with 2^{ω} to extract the higher and lower parts, respectively. These operations are formally represented by (20) and (21). Note that, when dealing with an abstract single word during intermediate recursion steps, we do not need to explicitly perform floor division and modulo operations. Instead, we can conceptually transform a single variable $c^{2\omega}$ into an array of two variables, $[c_0^{\omega}, c_1^{\omega}]$. It is only at the final recursion step, when the abstract representation must be concretized into actual code, that we need to implement these operations explicitly. Specifically, we use a right shift by ω_0 to extract the higher part and typecasting to T_{ω_0} to extract the lower part, where T_{ω_0} represents the integer data type corresponding to the machine word width ω_0. For instance, in C, if $\omega = 64$, then T_{ω_0} is `uint64_t`. We also apply (19) to $a^{2\omega}, b^{2\omega}, q^{2\omega}$ and obtain

$$[c_0^{\omega}, c_1^{\omega}] = ([a_0^{\omega}, a_1^{\omega}] + [b_0^{\omega}, b_1^{\omega}]) \bmod [q_0^{\omega}, q_1^{\omega}]. \tag{31}$$

Then, to compute a double-word addition $[a_0^{\omega}, a_1^{\omega}] + [b_0^{\omega}, b_1^{\omega}]$, we apply (22) which in turn uses (23) to break down double-word addition to single-word additions. Thus, we have rewrited (30) into

$$\begin{aligned} [\delta_0^1, d_2^{\omega}] &= a_1^{\omega} + b_1^{\omega}, \\ [d_0^1, d_1^{\omega}] &= \delta_0^1 + a_0^{\omega} + b_0^{\omega}, \\ [c_0^{\omega}, c_1^{\omega}] &= [d_0^1, d_1^{\omega}, d_2^{\omega}] \bmod [q_0^{\omega}, q_1^{\omega}]. \end{aligned} \tag{32}$$

We then need to rewrite $[d_0^1, d_1^{\omega}, d_2^{\omega}] \bmod [q_0^{\omega}, q_1^{\omega}]$ as native modulo support is not available for an integer with a bit-width of $2\omega + 1$. As we introduced earlier in Section 2.1, modulo after addition can be computed by a comparison, a subtraction, and a conditional assignment. Therefore, by (24), we have

$$\begin{aligned} [\delta_0^1, d_2^{\omega}] &= a_1^{\omega} + b_1^{\omega}, \\ [d_0^1, d_1^{\omega}] &= \delta_0^1 + a_0^{\omega} + b_0^{\omega}, \\ \delta_0^1 &= [q_0^{\omega}, q_1^{\omega}] < [d_1^{\omega}, d_2^{\omega}], \\ \delta_1^1 &= (0 < d_0^1) \vee \left((d_0^1 =_? 0) \wedge \delta_0^1\right), \\ [f_0^{\omega}, f_1^{\omega}] &= [d_1^{\omega}, d_2^{\omega}] - [q_0^{\omega}, q_1^{\omega}], \\ [c_0^{\omega}, c_1^{\omega}] &= \begin{cases} [f_0^{\omega}, f_1^{\omega}], & \text{if } \delta_1^1 =_? 1, \\ [d_1^{\omega}, d_2^{\omega}], & \text{otherwise.} \end{cases} \end{aligned} \tag{33}$$

Lastly, there are two double-word computations that need to be broken down: i) $\delta_0^1 = [q_0^{\omega}, q_1^{\omega}] < [d_1^{\omega}, d_2^{\omega}]$ and ii) $[f_0^{\omega}, f_1^{\omega}] = [d_1^{\omega}, d_2^{\omega}] - [q_0^{\omega}, q_1^{\omega}]$. Using (25) and (26), we obtain

$$\begin{aligned} [\delta_0^1, d_2^{\omega}] &= a_1^{\omega} + b_1^{\omega}, \\ [d_0^1, d_1^{\omega}] &= \delta_0^1 + a_0^{\omega} + b_0^{\omega}, \\ \delta_0^1 &= (q_0^{\omega} < d_1^{\omega}) \vee \left((q_0^{\omega} =_? d_1^{\omega}) \wedge (q_1^{\omega} < d_2^{\omega})\right), \\ \delta_1^1 &= (0 < d_0^1) \vee \left((d_0^1 =_? 0) \wedge \delta_0^1\right), \\ f_1^{\omega} &= d_2^{\omega} - q_2^{\omega}, \delta_0^1 = d_2^{\omega} < q_2^{\omega}, f_0^{\omega} = d_1^{\omega} - q_1^{\omega} - \delta_0^1, \\ [c_0^{\omega}, c_1^{\omega}] &= \begin{cases} [f_0^{\omega}, f_1^{\omega}], & \text{if } \delta_1^1 =_? 1, \\ [d_1^{\omega}, d_2^{\omega}], & \text{otherwise.} \end{cases} \end{aligned} \tag{34}$$

Note that double-word assignments, such as $[c_0^{\omega}, c_1^{\omega}] = [a_0^{\omega}, b_1^{\omega}]$, can be trivially implemented by individually assigning the corresponding single words. The same applies to conditional assignments. Consequently, we do not explicitly list these transformations as rewrite rules.

The above sequence of computations directly maps to `_daddmod` in Listing 2 which consists of `_add`, `_dlt`, `_dsub` and a conditional assignment at the end. In other words, when $\omega = 64$, Equation 34 can be implemented exactly as shown for `_daddmod`. However, if ω exceeds the machine word width and further recursion is required, an additional rule (27) is needed to break down the equality comparison in (24) and (26).

In summary, we use the rules from (19) to (27) to decompose double-word modular addition. The same set of rules is sufficient to break down double-word modular subtraction. For double-word multiplication (without modulo), two additional rules, (28) and (29), are necessary. Due to space constraints, the remaining rules developed for MoMA are omitted.

Non-power-of-two input bit-widths. The proposed mathematical formalism directly enables optimizations for inputs with non-power-of-two bit-widths, which is common in ZKP applications where the input bit-widths are often 381 bits (e.g., for the BLS12-381 elliptic curve) or 753 bits (e.g., for the MNT4753 elliptic curve). Let the non-power-of-two input bit-width be denoted by λ, and let ω be the closest power-of-two less than λ, i.e., $\omega < \lambda < 2\omega$. In this case, while using the data type $\mathsf{T}^{2\omega}$ to represent the input with bit-width λ allows

us to apply MoMA rewrite rules, many operations will involve zeros and/or evaluate to zero at runtime. Additionally, the number of these redundant operations cascades as the recursion depth required to reduce the bit-width to machine words grows, particularly when $(2\omega - \lambda) > (\lambda - \omega)$. For example, when $\lambda = 576$ and $\omega = 512$, 448 bits per input are zero at runtime, offering room for optimization by pruning no-ops during code generation.

Since we assume the input bit-width is known at compile/code generation time, it is straightforward to optimize operations on redundant bits using MoMA and the associated rule system. Let ω_0 denote the machine word width, which is typically a power of two. For any bit-width λ such that $\omega < \lambda < 2\omega$, we represent λ using $k\omega_0$, where $(k-1)\omega_0 < \lambda < k\omega_0$. Using the multi-word representation defined in Equation 14 and let x be the input with bit-width λ, we have

$$x = [0, \ldots, 0, x_0^{\omega_0}, \ldots, x_{k-1}^{\omega_0}]. \tag{35}$$

For example, when $\lambda = 753$ and $\omega_0 = 64$, x can be written as

$$[0,\ 0,\ 0,\ 0,\ x_0^{64}, \ldots, x_{11}^{64}], \tag{36}$$

or equivalently, $[0,\ 0,\ x_0^{128}, \ldots, x_5^{128}]$ and $[0,\ x_0^{256}, x_1^{256}, x_2^{256}]$. This implies that, during the recursive application of MoMA rewrite rules, we can set many single words (with bit-width ω) to zero at certain recursion levels, thereby pruning many operations at compile time. We employed this optimization when evaluating NTTs for 384-bit and 768-bit inputs.

Implementation. We implement MoMA using the rewrite rules detailed in this section through a high-performance code generator, SPIRAL [20]. SPIRAL converts high-level mathematical specifications into highly optimized code for various architectures. It provides a declarative and platform-agnostic mathematical language that offers excellent support for implementing mathematically formal rule systems. Specifically, we implement the MoMA rewrite rules as a recursive code generation pass that operates at the abstract code level in SPIRAL. Given the input bit-width and the machine word width, this pass generates equivalent abstract code where each variable's data type is natively supported by the machine. We build on the SPIRAL NTTX package [58] to map the final abstract code to efficient CUDA implementations on GPUs.

5 Evaluation

In this section, we describe the experimental setup and discuss the results of applying the MoMA rule system to implement BLAS operations and NTT for inputs with large bit-widths.

5.1 Experimental Setup

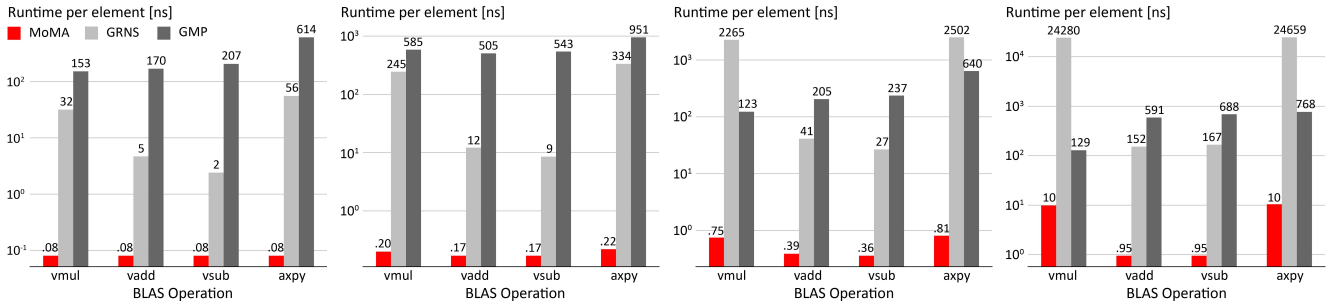
Prior work [59] introduced a code generation pass in SPIRAL targeting NVIDIA GPUs for NTTs (encapsulated within the SPIRAL NTTX package [58]). However, this work was limited to NTTs and only supported input bit-widths twice the machine word width (e.g., 128 bits on NVIDIA GPUs), relying on manually implemented double-word modular arithmetic functions.

We extended this work by implementing the MoMA rule system as a recursive program transformation pass, enabling the NTTX package to handle much larger bit-widths. Additionally, we expanded the SPIRAL NTTX package to generate efficient implementations for BLAS operations, including vector addition, subtraction, multiplication, and axpy (vector-scalar product followed by vector addition). This extension significantly broadens the NTTX package's capabilities, allowing it to support a wide range of cryptographic workloads through efficient polynomial arithmetic.

Parallelization techniques. MoMA, built on top of the SPIRAL NTTX package [58, 59], leverages the parallelization techniques provided by the framework. For NTTs, each CUDA thread processes one or more butterfly operations in each stage of the NTT, as there are no data dependencies between butterfly operations within the same stage. This parallelism is limited to $\min(n/2, 1024)$-way for NTTs of size n, as each stage consists of $n/2$ butterflies and each thread block can support up to 1,024 threads. We refer readers to prior works [58, 59] on the NTTX package for further details. For BLAS operations, each CUDA thread handles the computation for one element of the input vector. Therefore, we can perfectly parallelize up to 1,024-way vector-vector BLAS operations. As both ZKPs and FHE require multiple NTTs and BLAS operations to run concurrently [1, 32, 40, 60], we employ batch processing on the GPU to harness additional levels of parallelism. This allows us to comprehensively evaluate the full computational capabilities of a single GPU for executing cryptographic kernels.

Measuring kernel runtime. We calculate the runtime for a single NTT and BLAS operation as $t_{\text{single}} = t_{\text{all}}/k$, where k is the batch size. We report the steady-state runtime, defined as the minimum t_{single} achievable across all batch sizes, from one to the maximum batch size that can be compiled and run. Empirically, for NTTs, close-to-minimal steady-state runtime is achieved with a batch size greater than 8 for input bit-widths ranging from 128 to 384 bits, and greater than 64 for 768-bit inputs. For BLAS operations, close-to-minimal steady-state runtime is achieved with a batch size greater than 128.

To profile kernel runtime (i.e., t_{all}), we use the `nsys nvprof` profiling tool provided by the NVIDIA HPC SDK from the command line. BLAS operations and smaller NTTs (less than 1,024-point) are compiled without any additional flags. For larger NTTs, we compile with `-Xcompiler -mcmodel=medium -Xcompiler \"-Wl,--no-relax\"` to handle very large array sizes. In line with previous work on NTT acceleration, we exclude data transfer time between the CPU and GPU from the performance measurements.



(a) 128-bit **(b)** 256-bit **(c)** 512-bit **(d)** 1,024-bit

Figure 2. Performance of BLAS operations with various input bit-widths on CPU and GPU.

Hardware configuration. We benchmarked the performance of SPIRAL-generated cryptographic kernels across three types of GPUs from different hardware generations and price points. The NVIDIA H100 Tensor Core (H100) and NVIDIA Tesla V100 Tensor Core (V100, provided by Bridges-2 at Pittsburgh Supercomputing Center [9]) represent two generations of server-class GPUs, released in 2023 and 2017, respectively. The NVIDIA GeForce RTX 4090 (RTX 4090), launched in 2022, is a consumer-grade GPU primarily designed for gaming and video editing. Detailed specifications for each GPU are provided in Table 2.

Table 2. NVIDIA GPUs used for benchmarking.

Model	H100	RTX 4090	V100
#Cores	16896	16384	5120
Max Freq.	1980 MHz	2595 MHz	1530 MHz
RAM Size	80 GB	24 GB	32 GB
Bus Type	HBM3	GDDR6X	HBM2
Toolkit	12.2	12.0	11.7

5.2 BLAS Operation Results

We first evaluated MoMA's performance on BLAS kernels that are commonly used in encryption schemes [1], namely vector multiplication, vector addition, vector subtraction, and axpy, with four different input bit-widths: 128 bits, 256 bits, 512 bits, and 1,024 bits. In cryptographic settings, the bit-width of the modulus is usually slightly less than a power of two or a multiple of a power of two. For example, in ZKPs, prior works [29, 37] use moduli of 377 bits and 753 bits, based on the underlying elliptic curves, while for FHE, researchers [52] employ a 116-bit modulus. For consistency and ease of comparison, we categorize the bit-widths of prior works into the nearest (multiple of a) power-of-two category relative to their actual modulus bit-width when presenting the results for both BLAS operations and NTTs. In many prior approaches that use Barrett reduction for cryptographic kernels [32, 35, 41, 52, 55], the modulus q is less than or equal to $k-4$ bits to ensure that μ remains within k bits, where k represents the nearest (multiple of a) power-of-two bit-width in context. Similarly, our work employs a modulus with a bit-width of $k-4$ (e.g., 124 bits in the context of 128-bit results). It is noteworthy that the infrastructure we developed in SPIRAL using MoMA also supports a modulus of full bit-width, employing Montgomery multiplication [38].

We compared MoMA against the state-of-the-art integer multi-precision libraries on both CPU and GPU. The GNU Multiple Precision Arithmetic Library (GMP) [22] is a widely recognized library for multi-precision arithmetic on CPUs, supporting both integer and floating-point arithmetic. GRNS [30], which relies on GMP for initialization, supports basic arithmetic by using RNS to decompose very large integers into natively supported integers and employs floating point processing units on GPUs. We benchmarked MoMA-based implementations and GRNS (version 1.1.4) on V100, and GMP (version 6.1.2) on the Intel Xeon Gold 6248 @ 2.50GHz. For each BLAS operation with each input bit-width, we executed the vector operation in batch and measured the runtime for processing 2^{20} integers in total. We report the steady-state runtime per element, defined as the total runtime t_{all} divided by 2^{20}. For the GMP-based implementation, we utilized OpenMP for parallelization on CPU, with the OpenMP directive `#pragma omp parallel for`.

The results presented in Figure 2 demonstrate that MoMA-based implementations outperform both GMP and GRNS across all four operations with bit-widths ranging from 128 bits to 1,024 bits, achieving speedups of at least 13 times. For multiplication-based kernels, namely, vector multiplication and axpy, MoMA's speedup increases relative to GRNS but diminishes compared to GMP (although still maintaining a speedup of over 10 times for 1,024-bit inputs). This behavior is expected, as GMP utilizes fast Fourier transform (FFT)-based algorithms for large bit-width multiplications, which is reflected in the fact that GMP's runtime for 512-bit and 1,024-bit inputs is lower than for 128-bit and 256-bit inputs. For addition and subtraction operations, GRNS outperforms GMP across all bit-widths, with GMP narrowing the gap as bit-width increases. MoMA achieves at least 31 times

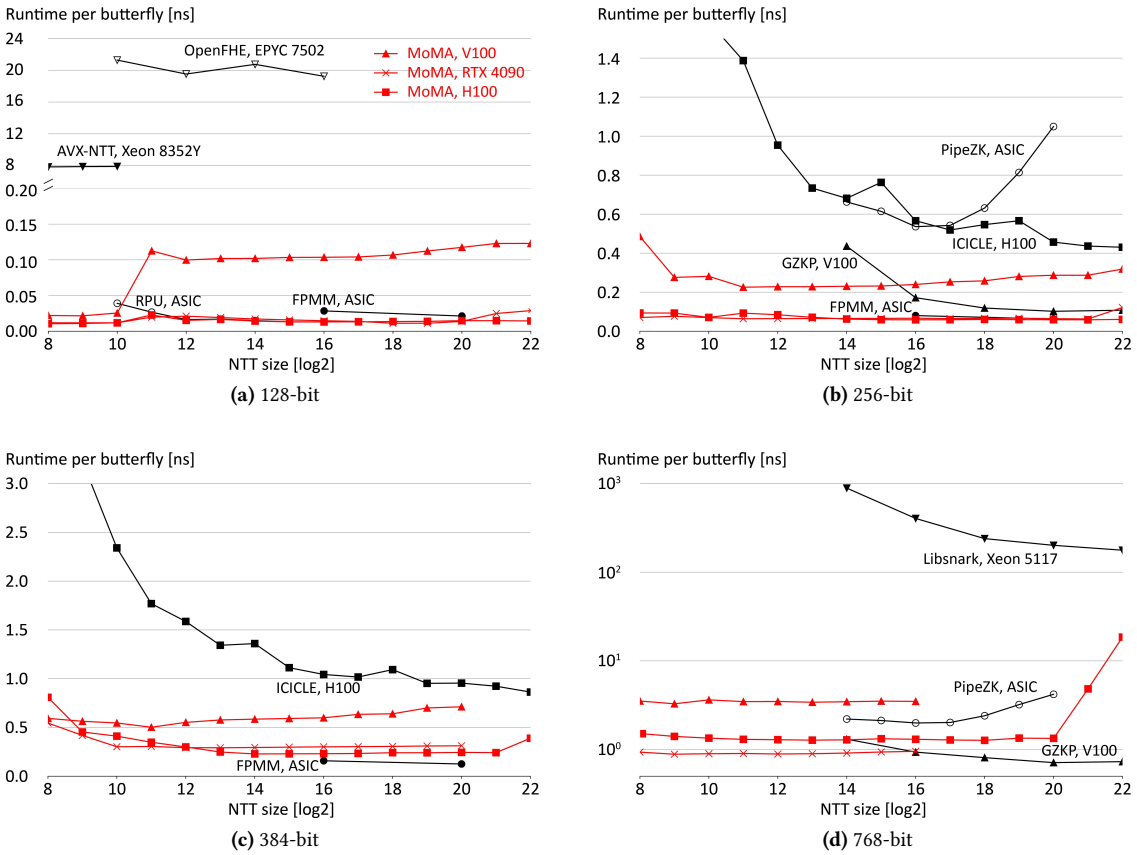


Figure 3. Performance of NTT with various input bit-widths on CPUs, GPUs, and ASICs.

speedup over GRNS and at least 527 times speedup over GMP for addition and subtraction operations.

5.3 NTT Results

Following the assessment of BLAS operations, we evaluated MoMA's performance on a more complex cryptographic kernel that serves as a core component in both FHE and ZKP workloads, NTT. For NTT evaluations, we did not employ any specialized primes (e.g., Goldilocks primes [26] or Montgomery-friendly primes [3]) for performance gain, thereby ensuring the rule system's general-purpose applicability. The code generator SPIRAL equipped with MoMA proved to be highly versatile and performant, allowing us to compare it against eight baseline implementations, each typically optimized for specific input bit-widths and NTT sizes. In terms of *generalizability*, the closest comparable library is ICICLE [29], which supports NTTs for two input bit-widths (256 bits and 768 bits) and is applicable across all tested NTT sizes (ranging from 2^8 to 2^{22}). Despite this, MoMA demonstrates a significant performance advantage, achieving a 13 times speedup over ICICLE for 256-bit inputs and a 4.8 times speedup for 384-bit inputs. In terms of *performance*, MoMA-based NTT running on RTX 4090, a $2,000 consumer-grade GPU, outperforms NTTs on two state-of-the-art ASICs (RPU [52] and Zhou et al.'s work [63]), for 128-bit and 256-bit inputs. In the following texts, we refer to Zhou et al.'s work as FPMM and our MoMA-based NTT as MoMA.

In the context of NTT, a butterfly operation includes one modular addition, one modular subtraction, and one modular multiplication. In Figure 3, we present the runtime per butterfly for our approach and all baselines on the y-axis. This metric is defined as $2t_{\text{single}}/(n \log_2 n)$, where n is the NTT size, $(n \log_2 n)/2$ is the number of butterflies in an n-point NTT, and t_{single} is the runtime for a single n-point NTT.

128-bit inputs. In Figure 3a, for NTT sizes up to 2^{10}, the entire NTT fits within the GPU's shared memory, which accounts for the significant slowdown observed on V100 for size 2^{11} and larger. On H100 and RTX 4090, going out of the shared memory results in a 1.5 times slowdown. We compare our results against two CPU baselines, OpenFHE [1] (based on the benchmarking results reported by RPU [52]) and AVX-NTT [21], as well as two ASIC baselines, RPU and FPMM [63]. On H100, MoMA outperforms RPU, an accelerator specifically designed for FHE, by 1.4 times on average and FPMM by 1.8 times on average. Notably, on RTX 4090, a consumer-grade GPU, MoMA achieves an average speedup of 1.3 times over RPU and 1.7 times over FPMM.

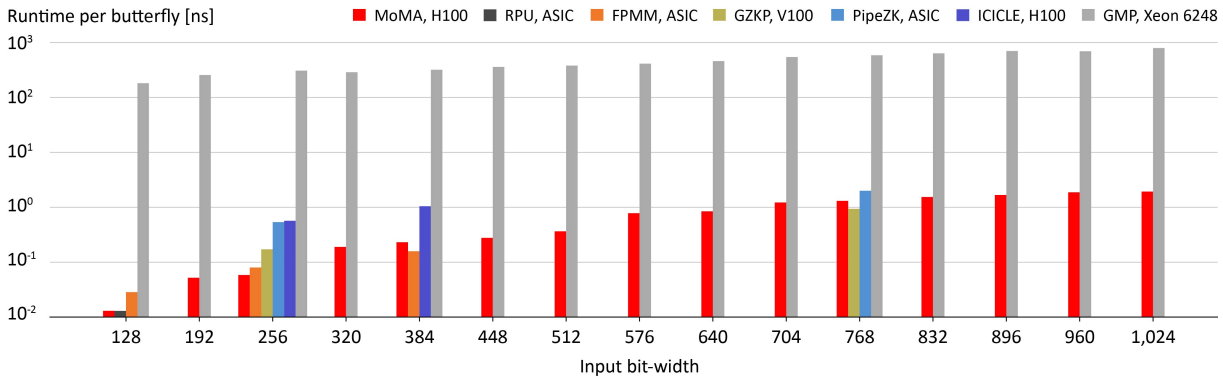


Figure 4. Performance of 2^{16}-point NTT with input bit-widths ranging from 128 to 1,024 on CPUs, GPUs, and ASICs.

256-bit inputs. Figure 3b compares MoMA with two GPU-based approaches, GZKP [37] and ICICLE [29], and two ASIC-based approaches, PipeZK [60] and FPMM. On H100, our approach shows a 13 times average speedup across all tested NTT sizes over ICICLE. On all three tested GPUs, MoMA outperforms PipeZK, an ASIC designed to accelerate ZKPs using a pipelined architecture. On the H100 and RTX 4090, MoMA also outperforms FPMM, an ASIC with reported results for two NTT sizes. On V100, MoMA is outperformed by GZKP for large NTT sizes due to the fact that GZKP exploits all the floating-point processing units on GPU. However, MoMA outperforms GZKP on smaller sizes, even when using only integer processing units.

384-bit inputs. For 384-bit inputs, as shown in Figure 3c, we compare MoMA with ICICLE on H100 and FPMM, an ASIC. MoMA on H100 achieves an average speedup of 4.8 times across all tested NTT sizes against ICICLE. Moreover, on V100, MoMA also outperforms ICICLE on H100 by 3 times on average. MoMA-based NTT runs into segmentation faults at size 2^{21} on both RTX 4090 and V100, with an error message indicating running out of the stack space during compilation. Although MoMA outperforms FPMM for 128-bit and 256-bit inputs, FPMM achieves a 1.7 times speedup over our approach at 384-bit inputs.

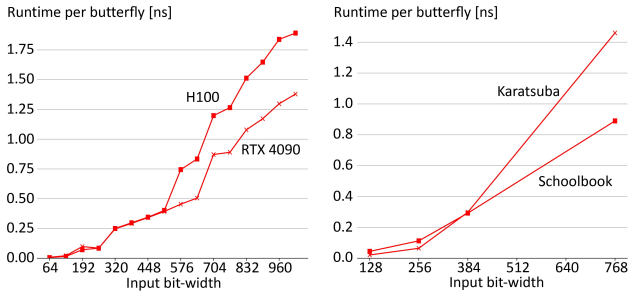
768-bit inputs. In Figure 3d, we compare MoMA with PipeZK, GZKP, and Libsnark [34]. Libsnark is a CPU-based ZKP library that implements 768-bit NTTs, and we plot the benchmarking results of Libsnark as reported by GZKP. For 768-bit inputs, RTX 4090 outperforms H100 until the NTT code runs into a segmentation fault at size 2^{16} (which also occurs for V100). The speedup of RTX 4090 over H100 could be attributed to its higher clock speed. As the bit-width increases, each butterfly operation becomes significantly more computationally intensive. In such scenarios, a GPU with a higher clock speed, such as RTX 4090, may be more favorable. Both H100 and RTX 4090 outperform PipeZK, with H100 achieving a 2 times speedup over PipeZK for sizes ranging from 2^{14} to 2^{20}. However, the performance of H100 degrades significantly beyond size 2^{20}, suggesting that the hardware or compiler limits are being approached. For 768-bit inputs, from size 2^{16} onwards, MoMA is outperformed by GZKP. This comparison highlights that utilizing all floating-point units on the GPU is a clear advantage for accelerating NTT computations, especially with large input bit-widths. Since MoMA rewrite rules are well-defined and composable, future efforts could leverage GPU floating-point processing units and utilize specialized primes, such as Montgomery-friendly primes for certain ZKP applications, to further improve performance.

Comparison among multiple input bit-widths. As detailed in Section 4, MoMA's formalism enables optimizations for inputs with non-power-of-two bit-widths. This allows MoMA to optimally support a wide range of fine-grained bit-widths, rather than resorting to zero-padding the inputs to the nearest power-of-two. Figure 4 illustrates this flexibility, where we compare the performance of MoMA with NTTs implemented using the state-of-the-art multi-precision library GMP, along with results from other works discussed earlier, plotted for the relevant bit-widths. This plot can be interpreted as a cross-cut of the four subplots in Figure 3, where we use an NTT size of 2^{16} and assemble the subplots accordingly. We choose 2^{16} as the NTT size because for this size there exists the most comparable work across multiple input bit-widths. The key takeaway from this plot is that MoMA demonstrates: i) significant performance improvements over libraries supporting general input bit-widths, and ii) comparable performance to approaches that utilize specialized hardware and/or are optimized for a limited set of input bit-widths.

5.4 Sensitivity Analysis

We now conduct two sensitivity analyses on MoMA's performance, examining the impact of input bit-width and the choice of multiplication algorithm on NTT runtime. Both analyses are performed with a fixed NTT size of 4,096.

Impact of input bit-width on runtime. We investigate how the performance of MoMA-based NTT scales with increasing input bit-width. In Figure 5a, we benchmark MoMA-based NTTs for input bit-widths ranging from 64 to 1,024 bits



(a) Runtime against input bit-width on H100 and RTX 4090.

(b) Comparison between the Karatsuba algorithm and the Schoolbook algorithm on RTX 4090.

Figure 5. Sensitivity analyses on NTT runtime.

on both H100 and RTX 4090. We observe that both GPUs exhibit similar scaling trends with increasing bit-width. Specifically, from 320 bits to 512 bits, both RTX 4090 and H100 show a linear increase in runtime. However, at 576 bits, the runtime on H100 starts to increase non-linearly, while on RTX 4090 it maintains a linear increase up to 640 bits. Beyond 512 bits, the performance gap between RTX 4090 and H100 remains relatively constant. On H100, we observe a 2.9 times slowdown when increasing from 64 bits to 128 bits, a 5.6 times slowdown from 128 bits to 256 bits, a 4.8 times slowdown from 256 bits to 512 bits, and a 4.7 times slowdown from 512 bits to 1,024 bits. The corresponding slowdowns on RTX 4090 are 2.7, 4, 4.6, and 3.5 times, respectively.

Choices of multiplication algorithm. As discussed in Section 2.2, MoMA offers two implementation choices for multiplication: i) the Schoolbook algorithm and ii) the Karatsuba algorithm. The Schoolbook algorithm for double-word multiplication requires 4 single-word multiplications and 6 single-word additions (excluding carry propagation). In contrast, the Karatsuba algorithm uses 3 single-word multiplications, 12 single-word additions/subtractions (excluding carry propagation), and several single-word comparisons. The Karatsuba algorithm reduces the number of multiplications, which are typically the most computationally intensive operations, by replacing one multiplication with several relatively cheaper additions. In MoMA, both algorithms are included and, when implemented in the SPIRAL code generator, users can select the preferred multiplication algorithm to optimize performance on the target hardware. Figure 5b compares the performance of the Schoolbook algorithm and the Karatsuba algorithm for NTT computations on RTX 4090 across various bit-widths. For 128-bit and 256-bit inputs, the Karatsuba algorithm outperforms the Schoolbook algorithm by 2.1 times and 1.7 times, respectively. Both algorithms exhibit similar performance for 384-bit inputs, while the Schoolbook algorithm begins to outperform the Karatsuba algorithm for 768-bit inputs, with a 1.6 times speedup.

6 Related Work

We review the state-of-the-art approaches for multi-precision integer arithmetic and prior work on NTT acceleration.

Multi-precision integer arithmetic. There are many prior work that efficiently implements multi-precision arithmetic. However, many focus on a single bit-width or a limited range of bit-widths [16, 18, 39, 42, 43, 59, 61, 62], focus on one arithmetic operation (usually multiplication) [14, 17, 23, 33, 49], or is limited to a specific CPU/GPU architecture or a specific ASIC design [10, 44, 52, 63]. There are also libraries designed for generalizability, which implement arbitrary precision arithmetic. One of the most well-known and well-maintained libraries is GMP [22]. Other libraries such as a library for doing number theory (NTL) [51] and fast library for number theory (FLINT) [27] also support arbitrary precision arithmetic but use GMP behind the scenes. As GMP is written in C, both GMP and libraries that depend on GMP run on CPU only. On the other hand, many programming languages, such as Python and Rust, offer native support for computations with arbitrarily large integers. However, as prior studies have shown [28], these languages are generally outperformed by GMP in terms of performance. GRNS [30] is a GPU-based arbitrary precision integer library that relies on GMP for initialization. GRNS uses RNS to break down very large integers into natively supported integers and utilizes floating-point processing units on GPU for RNS-based integer arithmetic. MoMA outperforms both GMP and GRNS on all four common input bit-width for popular encryption schemes by orders of magnitude, as shown in Figure 2 and 4.

NTT acceleration. Numerous works have been proposed to accelerate NTT due to its significance in terms of runtime in FHE and ZKP workloads. Most of the prior work focuses on designing specific accelerators [45, 46, 52, 54, 63] to address the prohibitive computational overhead introduced by FHE and ZKPs. GPU-based approaches for accelerating NTT primarily focus on input bit-widths that is natively supported by the machine (i.e., 32 bits and 64 bits) [15, 32, 35, 36, 40, 41, 50, 53, 55, 56]. ICICLE [29] is a very relevant work that offers great generalizability as a high-performance cryptographic acceleration library. As shown in Figure 3, ICICLE is the only work that can compile and run on all NTT sizes that we tested on two input bit-widths. GZKP [37] is another GPU-based approach that accelerates NTT of sizes larger than machine word width. As mentioned at the end of Section 5.3, GZKP leverages GPU floating-point processing units, a feature that can be incorporated into MoMA given the generalizability of the rewrite rules; however, this is beyond the scope of this work. There are many CPU-based libraries for FHE and ZKP workloads [1, 7, 8, 11, 13, 25, 34]. Most of these libraries prioritize generalizability (e.g., supporting various encryption

schemes) and demonstrate suboptimal performance on NTTs compared to GPU- and AISC-based approaches [52].

7 Discussion

As shown in Figures 2, 3 and 4, MoMA-based kernels demonstrate strong performance for bit-widths between 128 and 1,024 bits. However, the performance gap with the state-of-the-art multi-precision library narrows as the bit-width increases. We anticipate that, for example, GMP's FFT-based multiplication will outperform MoMA for very large bit-widths (e.g., 8,192 bits). Nonetheless, as a rule system, MoMA can incorporate algorithms tailored to specific scenarios due to its composable and well-defined formalism. We believe that MoMA's exceptional performance on bit-widths relevant to FHE and ZKPs opens the door to exploring new encryption schemes involving large bit-widths, as the cost of exceeding machine word arithmetic has been significantly reduced.

Driven by artificial intelligence (AI) and machine learning (ML) applications, customized hardware continues trending toward smaller machine word widths (e.g., 16-bit unsigned integer on the Cerebras Wafer Scale Engine [48]). However, FHE, an enabling encryption scheme for privacy-preserving AI/ML, requires support for large integer arithmetic. FHE-specific hardware is therefore designed to natively support large bit-width arithmetic operations (e.g., 128-bit modular arithmetic supported by RPU [52]). As future work, we plan to investigate whether MoMA can work effectively on AI/ML-specialized hardware to keep the runtime of FHE-based workloads manageable with native small integer arithmetic. In other words, we aim to explore whether MoMA can bridge the gap between these two conflicting hardware trends at the software level. Moreover, we plan to integrate MoMA into various compilers, ranging from generic frameworks like LLVM to specialized compilers such as the CSL compiler for Cerebras accelerators, further expanding the impact of our approach.

8 Conclusion

To address the critical need for efficient large integer arithmetic in cryptographic applications, our work formally defines MoMA, which decomposes large bit-width integer arithmetic into operations based on machine words. We developed a mathematically formal rewrite system that implements MoMA and can be seamlessly integrated into compilers and code generators. For evaluation, we implemented the MoMA rule system in SPIRAL and generated cryptographic kernels, including BLAS operations and NTTs, across three types of GPUs. Our results show that the generated BLAS operations outperform state-of-the-art multi-precision libraries by several orders of magnitude, while MoMA-based NTT achieves near-ASIC performance on commodity GPUs.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1127353, the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-FOA-0002460, and PRISM, a center in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by the Defense Advanced Research Projects Agency (DARPA). This work used Bridges-2 at Pittsburgh Supercomputing Center through allocation CIE160039 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Department of Energy, and DARPA. Franz Franchetti was partially supported as the Kavčić-Moura Professor of Electrical and Computer Engineering.

A Artifact Appendix

A.1 Abstract

Our artifact [57] includes the source code for MoMA, requiring NVIDIA GPUs along with `nvcc` and `nsys nvprof` for compilation and performance profiling. While installing the SPIRAL code generation system is highly recommended for full reproducibility and customization, it is not required, as we provide pre-generated NTT and BLAS code from SPIRAL to reproduce key results.

A.2 Artifact Checklist (Meta-information)

- **Program:** MoMA as part of the SPIRAL NTTX package. The artifact includes benchmarking data for NTT and BLAS operations.
- **Compilation:** The artifact requires `nvcc` >= 11.7 to compile the generated CUDA code and `nsys` >= 2022.4.2 for performance measurement.
- **Transformations:** SPIRAL >= 8.5.0 is required for code generation; however, pre-generated code is provided for users who prefer not to install SPIRAL or spend time on code generation.
- **Hardware:** NVIDIA GPUs as detailed in Table 2.
- **Execution:** We provide a Bash script to benchmark the pre-generated code directly or to invoke SPIRAL for code generation followed by benchmarking.
- **Metrics:** Execution time.
- **Output:** Performance measurements will be displayed in the terminal window.
- **Experiments:** A detailed `README` file in the `nttx` directory explains how to reproduce key results, with or without SPIRAL installation. Expected outputs for specific cases are also provided in the `README` file.
- **How much disk space required (approximately)?:** Around 50 MB.

- **How much time is needed to prepare workflow (approximately)?:** Using pre-generated code, the user can immediately execute the workflow via an automated benchmarking script. Installing SPIRAL typically takes less than an hour.
- **How much time is needed to complete experiments (approximately)?:** Using pre-generated code, benchmarking 1,024-point NTT with 256-bit inputs takes approximately five minutes. Starting from code generation, this process takes approximately six minutes. Completing all experiments on all three GPUs from end to end may take days, as the code generation time increases exponentially with the input bit-width.
- **Publicly available?:** Yes.
- **Archived (provide DOI)?:** 10.5281/zenodo.14564393.

A.3 Description

Here we provide a short description of how the artifact is delivered and its dependencies.

How delivered. The artifact is provided as a zip file available on Zenodo[1] and as a repository on GitHub[2]. The artifact requires approximately 50 MB of disk space.

Hardware dependencies. NVIDIA GPUs. The key results in this paper were obtained using NVIDIA H100, V100, and RTX 4090 GPUs, with details provided in Table 2.

Software dependencies. `nvcc` >= 11.7 and `nsys` >= 2022.4.2. SPIRAL >= 8.5.0 is highly recommended but not required.

A.4 Installation

SPIRAL can be installed following its installation guide[3]. To link the artifact with SPIRAL, place the artifact into the `spiral-software/namespaces/packages/` subdirectory of the SPIRAL installation tree.

A.5 Experiment Workflow

A detailed `README` file is provided at `nttx/README.md`. For example, with SPIRAL installed, users can reproduce our 128-bit NTT results on H100 by running the following commands:

```
$ cd nttx/cuda/cuda-test
$ bash ./benchmark.sh -d 128 -p h100
```

The option `-d` specifies the input bit-width (128, 256, 384, or 768), while `-p` enables performance tuning for the target platform (currently supported platforms are H100, V100, and RTX 4090). For other platforms, the `-p` option can be omitted, which defaults to `-p general`.

A.6 Evaluation and Expected Result

After completing the experiments, the terminal will display the results as follows: for NTT, the runtime per butterfly and the runtime per NTT for each NTT size; for BLAS operation, the runtime per element and the runtime per vector operation. All runtimes are reported in nanoseconds.

[1] https://doi.org/10.5281/zenodo.14564393
[2] https://github.com/naifeng/moma
[3] https://github.com/spiral-software/spiral-software

References

[1] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, et al. 2022. OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 53–63.

[2] Ahmad Al Badawi and Yuriy Polyakov. 2023. Demystifying bootstrapping in fully homomorphic encryption. *Cryptology ePrint Archive* (2023).

[3] Jean Claude Bajard and Sylvain Duquesne. 2021. Montgomery-friendly primes and applications to cryptography. *Journal of Cryptographic Engineering* 11, 4 (2021), 399–415.

[4] Paul Barrett. 1986. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 311–323.

[5] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct {Non-Interactive} zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*. 781–796.

[6] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[7] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, and Vinodh Gopal. 2021. Intel HEXL: accelerating homomorphic encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 57–62.

[8] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 947–964.

[9] Shawn T Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A Nystrom. 2021. Bridges-2: a platform for rapidly-evolving and data intensive research. In *Practice and Experience in Advanced Research Computing*. 1–4.

[10] Boon-Chiao Chang, Bok-Min Goi, Raphael C-W Phan, and Wai-Kong Lee. 2018. Multiplying very large integer in GPU with pascal architecture. In *2018 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*. IEEE, 401–405.

[11] Hao Chen, Kim Laine, and Rachel Player. 2017. Simple encrypted arithmetic library-SEAL v2. 1. In *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21*. Springer, 3–18.

[12] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2019. A full RNS variant of approximate homomorphic encryption. In *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*. Springer, 347–368.

[13] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[14] Adrian P Dieguez, Margarita Amor, Ramón Doallo, Akira Nukada, and Satoshi Matsuoka. 2022. Efficient high-precision integer multiplication on the GPU. *The International Journal of High Performance Computing Applications* 36, 3 (2022), 356–369.

[15] Sultan Durrani, Muhammad Saad Chughtai, Mert Hidayetoglu, Rashid Tahir, Abdul Dakkak, Lawrence Rauchwerger, Fareed Zaffar, and Wenmei Hwu. 2021. Accelerating fourier and number theoretic transforms using tensor cores and warp shuffles. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 345–355.

[16] Takuya Edamatsu and Daisuke Takahashi. 2020. Accelerating large integer multiplication using Intel AVX-512IFMA. In *Algorithms and Architectures for Parallel Processing: 19th International Conference, ICA3PP 2019, Melbourne, VIC, Australia, December 9–11, 2019, Proceedings, Part I 19*. Springer, 60–74.

[17] Niall Emmart and Charles Weems. 2011. High precision integer multiplication with a GPU. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 1781–1787.

[18] Timothée Ewart, Andreas Hehn, and Matthias Troyer. 2013. VLI–A Library for High Precision Integer and Polynomial Arithmetic. In *Supercomputing: 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings 28*. Springer, 267–278.

[19] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. Tensorfhe: Achieving practical computation on encrypted data using gpgpu. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 922–934.

[20] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and José MF Moura. 2018. SPIRAL: Extreme performance portability. *Proc. IEEE* 106, 11 (2018), 1935–1968.

[21] Sophia Fu, Naifeng Zhang, and Franz Franchetti. 2024. Accelerating High-Precision Number Theoretic Transforms using Intel AVX-512. In *2024 33th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[22] Torbjörn Granlund. 1996. Gnu mp. *The GNU Multiple Precision Arithmetic Library* 2, 2 (1996).

[23] Shay Gueron and Vlad Krasnov. 2016. Accelerating big integer arithmetic using intel ifma extensions. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. IEEE, 32–38.

[24] Part Guide. 2011. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* 2, 11 (2011), 0–40.

[25] Shai Halevi. 2021. HElib (version 2.1.0). https://github.com/homenc/HElib

[26] Mike Hamburg. 2015. Ed448-Goldilocks, a new elliptic curve. *Cryptology ePrint Archive* (2015).

[27] William B Hart. 2013. Flint: Fast library for number theory. *Computeralgebra Rundbrief* (2013).

[28] Wilfred Hughes. 2014. The Fastest BigInt In The West. https://www.wilfred.me.uk/blog/2014/10/20/the-fastest-bigint-in-the-west/

[29] Karthik Inbasekar, Yuval Shekel, and Michael Asa. 2024. ICICLE v2: Polynomial API for Coding ZK Provers to Run on Specialized Hardware. *Cryptology ePrint Archive* (2024).

[30] Konstantin Isupov. 2021. High-performance computation in residue number system using floating-point arithmetic. *Computation* 9, 2 (2021), 9.

[31] Anatolii Alekseevich Karatsuba and Yu P Ofman. 1962. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, Vol. 145. Russian Academy of Sciences, 293–294.

[32] Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. 2020. Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 264–275.

[33] Koji Kitano and Noriyuki Fujimoto. 2014. Multiple precision integer multiplication on GPUs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer ..., 1.

[34] SCIPR LAB. 2022. libsnark: a C++ library for zkSNARK proofs. https://github.com/scipr-lab/libsnark

[35] Neal Livesay, Gilbert Jonatan, Evelio Mora, Kaustubh Shivdikar, Rashmi Agrawal, Ajay Joshi, José L Abellán, John Kim, and David Kaeli. 2023. Accelerating finite field arithmetic for homomorphic encryption on GPUs. *IEEE Micro* 43, 5 (2023), 55–63.

[36] Patrick Longa and Michael Naehrig. 2016. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *Cryptology and Network Security: 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings 15*. Springer, 124–139.

[37] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 2023. Gzkp: A gpu accelerated zero-knowledge proof system. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 340–353.

[38] Peter L Montgomery. 1985. Modular multiplication without trial division. *Mathematics of computation* 44, 170 (1985), 519–521.

[39] Eduardo Ochoa-Jiménez, Luis Rivera-Zamarripa, Nareli Cruz-Cortés, and Francisco Rodríguez-Henríquez. 2020. Implementation of RSA signatures on GPU and CPU architectures. *IEEE Access* 8 (2020), 9928–9941.

[40] Ali Şah Özcan, Can Ayduman, Enes Recep Türkoğlu, and Erkay Savaş. 2023. Homomorphic encryption on GPU. *IEEE Access* 11 (2023), 84168–84186.

[41] Özgün Özerk, Can Elgezen, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2022. Efficient number theoretic transform implementation on GPU for homomorphic encryption. *The Journal of Supercomputing* 78, 2 (2022), 2840–2872.

[42] Dong-won Park, Seokhie Hong, Nam Su Chang, and Sung Min Cho. 2021. Efficient implementation of modular multiplication over 192-bit NIST prime for 8-bit AVR-based sensor node. *The Journal of Supercomputing* 77, 5 (2021), 4852–4870.

[43] Thomas Plantard. 2021. Efficient word size modular arithmetic. *IEEE Transactions on Emerging Topics in Computing* 9, 3 (2021), 1506–1518.

[44] Kamil Rudnicki, Tomasz P Stefański, and Wojciech Żebrowski. 2020. Open-source coprocessor for integer multiple precision arithmetic. *Electronics* 9, 7 (2020), 1141.

[45] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 238–252.

[46] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 173–187.

[47] Ardianto Satriawan, Infall Syafalni, Rella Mareta, Isa Anshori, Wervyan Shalannanda, and Aleams Barra. 2023. Conceptual review on number theoretic transform and comprehensive review on its implementations. *IEEE Access* (2023).

[48] Justin Selig. 2022. The cerebras software development kit: A technical overview. *Technical Report. Cerebras* (2022).

[49] Ming-Der Shieh and Wen-Ching Lin. 2010. Word-based Montgomery modular multiplication algorithm for low-latency scalable architectures. *IEEE transactions on computers* 59, 8 (2010), 1145–1151.

[50] Kaustubh Shivdikar, Gilbert Jonatan, Evelio Mora, Neal Livesay, Rashmi Agrawal, Ajay Joshi, José L Abellán, John Kim, and David Kaeli. 2022. Accelerating polynomial multiplication for homomorphic encryption on GPUs. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 61–72.

[51] Victor Shoup et al. 2001. NTL: A library for doing number theory. (2001).

[52] Deepraj Soni, Negar Neda, Naifeng Zhang, Benedict Reynwar, Homer Gamil, Benjamin Heyman, Mohammed Nabeel, Ahmad Al Badawi, Yuriy Polyakov, Kellie Canida, et al. 2023. Rpu: The ring processing unit. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 272–282.

[53] Lipeng Wan, Fangyu Zheng, Guang Fan, Rong Wei, Lili Gao, Yuewu Wang, Jingqiang Lin, and Jiankuo Dong. 2022. A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator. In *European Symposium on Research in Computer Security*. Springer, 514–534.

[54] Cheng Wang and Mingyu Gao. 2023. SAM: A Scalable Accelerator for Number Theoretic Transform Using Multi-Dimensional Decomposition. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[55] Zhiwei Wang, Peinan Li, Rui Hou, Zhihao Li, Jiangfeng Cao, XiaoFeng Wang, and Dan Meng. 2023. HE-Booster: an efficient polynomial arithmetic acceleration on GPUs for fully homomorphic encryption. *IEEE Transactions on Parallel and Distributed Systems* 34, 4 (2023), 1067–1081.

[56] Zhiwei Wang, Peinan Li, Rui Hou, and Dan Meng. 2023. NTTFusion: Efficient Number Theoretic Transform Acceleration on GPUs. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 357–365.

[57] Naifeng Zhang. 2024. Artifact for 'Code Generation for Cryptographic Kernels using Multi-word Modular Arithmetic on GPU'. https://doi.org/10.5281/zenodo.14564393

[58] Naifeng Zhang, Austin Ebel, Negar Neda, Patrick Brinich, Benedict Reynwar, Andrew G Schmidt, Mike Franusich, Jeremy Johnson, Brandon Reagen, and Franz Franchetti. 2023. Generating High-Performance Number Theoretic Transform Implementations for Vector Architectures. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[59] Naifeng Zhang and Franz Franchetti. 2023. Generating number theoretic transforms for multi-word integer data types. In *2023 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[60] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. 2021. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 416–428.

[61] Kaiyong Zhao. 2009. Implementation of multiple-precision modular multiplication on gpu. In *GPU Technology Conference*.

[62] Kaiyong Zhao and Xiaowen Chu. 2010. GPUMP: A multiple-precision integer library for GPUs. In *2010 10th IEEE International Conference on Computer and Information Technology*. IEEE, 1164–1168.

[63] Hao Zhou, Changxu Liu, Lan Yang, Li Shang, and Fan Yang. 2024. A Fully Pipelined Reconfigurable Montgomery Modular Multiplier Supporting Variable Bit-Widths. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

Received 2024-09-12; accepted 2024-11-04