Towards Closing the Performance Gap for Cryptographic Kernels Between CPUs and Specialized Hardware

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

Sophia Fu
Carnegie Mellon University
Pittsburgh, USA
syfu@andrew.cmu.edu

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

Abstract

Specialized hardware like application-specific integrated circuits (ASICs) remains the primary accelerator type for cryptographic kernels based on large integer arithmetic. Prior work has shown that commodity and server-class GPUs can achieve near-ASIC performance for these workloads. However, achieving comparable performance on CPUs remains an open challenge. This work investigates the following question: How can we narrow the performance gap between CPUs and specialized hardware for key cryptographic kernels like basic linear algebra subprograms (BLAS) operations and the number theoretic transform (NTT)?

To this end, we develop an optimized scalar implementation of these kernels for x86 CPUs at the per-core level. We utilize SIMD instructions—specifically AVX2 and AVX-512—to further improve performance, achieving an average speedup of 38 times and 62 times over state-of-the-art CPU baselines for NTTs and BLAS operations, respectively. To narrow the gap further, we propose a small AVX-512 extension, dubbed multi-word extension (MQX), which delivers substantial speedup with only three new instructions and minimal proposed hardware modifications. MQX cuts the slowdown relative to ASICs to as low as 35 times on a *single* CPU core. Finally, we perform a roofline analysis to evaluate the peak performance achievable with MQX when scaled across an entire multi-core CPU. Our results show that, with MQX, top-tier server-grade CPUs can approach the performance of state-of-the-art ASICs for cryptographic workloads.

CCS Concepts

• **Computer systems organization** → **Single instruction, multiple data**; • **Security and privacy** → **Cryptography**.

Keywords

Cryptography, large integer arithmetic, SIMD, ISA extension, performance modeling

ACM Reference Format:
Naifeng Zhang, Sophia Fu, and Franz Franchetti. 2025. Towards Closing the Performance Gap for Cryptographic Kernels Between CPUs and Specialized Hardware. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756120

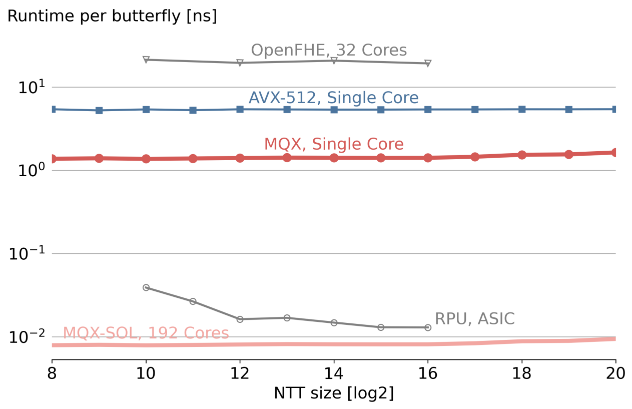


Figure 1: Performance comparison of NTT implementations on CPUs and an ASIC (lower is better). Our implementation of NTT using AVX-512 on a single core of AMD EPYC 9654 achieves a 3.8 times speedup compared to the state-of-the-art FHE library [1], which is benchmarked on AMD EPYC 7502 with 32 cores [42]. When scaled to 192 cores of AMD's server-grade CPU, the peak performance achievable with MQX suggests that CPU-based implementations could approach near-ASIC performance for NTTs.

1 Introduction

As today's computing landscape shifts towards distributed and cloud computing environments, advanced encryption techniques, such as fully homomorphic encryption (FHE), are gaining interest for privacy-preserving computations. FHE keeps the data encrypted as soon as it leaves the local party by allowing computations on the encrypted data on the external side. However, despite its strong privacy guarantees, FHE has not seen widespread adoption due to its prohibitive computational overhead, which primarily arises from polynomial arithmetic involving very large coefficients. To handle large integer arithmetic, prior works employ the residue number system (RNS) to decompose very large coefficients (e.g., over 1,000 bits) into smaller components (i.e., residues) that fit within machine words, such as 64 bits. Recently, several studies have explored the use of 128-bit residues [1, 31, 42, 51, 53]—going beyond common machine word sizes—to reduce the computational overhead associated with modulus raising and reduction [8], as well as to reduce the frequency of *bootstrapping*, a highly computationally intensive process in FHE [2]. These works have demonstrated efficient implementations of 128-bit arithmetic on both ASICs [31, 42, 53] and GPUs [51]. By natively supporting 128-bit operations in hardware, ASICs can achieve up to a 1,485 times speedup over the OpenFHE [1] CPU baseline, as shown by RPU [42]. On GPUs, MoMA [51] introduces *multi-word modular arithmetic*, which recursively decomposes large integer arithmetic into machine-word operations. This approach enables commodity GPUs such as NVIDIA GeForce RTX 4090 to achieve near-ASIC performance. On CPUs, OpenFHE [1] stands as one of the leading cryptographic libraries for FHE implementations. OpenFHE offers two options for large integer arithmetic: a built-in mathematical backend and the widely-used multi-precision library GMP [18]. However, both options deliver significantly lower performance compared to ASIC and GPU implementations as discussed above. Therefore, a competitive CPU-based implementation for FHE workloads remains an open challenge.

This paper aims to narrow this performance gap between CPUs and specialized hardware for two core cryptographic kernels in FHE: basic linear algebra subprograms (BLAS) operations and number theoretic transforms (NTTs). NTTs are among the most computationally intensive kernels in FHE, accounting for over 90% of runtime in many FHE-based workloads [13]. BLAS operations, likewise, serve as fundamental building blocks for most FHE schemes [1]. We begin by deriving highly optimized scalar implementations (i.e., standard C targeting x86-64 architectures) of NTT and BLAS operations and then explore vector parallelism enabled by single instruction multiple data (SIMD) instructions. As shown in Figure 1, using Advanced Vector Extension 512 (AVX-512) instructions on a *single core* of AMD EPYC 9654, we can achieve a 3.8 times speedup over OpenFHE that is benchmarked on a 32-core CPU.

To further enhance performance, we propose a small instruction set architecture (ISA) extension, named multi-word extension (MQX), that augments AVX-512 with just three new vector instructions. MQX is designed to minimize hardware engineering efforts when integrated with existing AVX intrinsics, drawing on parallels with x86 scalar operations and prior SIMD instructions from Intel's Knights Corner (KNC) intrinsics [10, 23] and Larrabee New Instructions (LRBni) [38]. We define MQX at the ISA level and model its performance using a novel technique, performance projection using proxy ISA (PISA). Through PISA, the measured projected performance of MQX delivers an additional 3.7 times speedup over our AVX-512 baseline for NTTs on AMD EPYC 9654. Finally, we conduct a roofline analysis to estimate the upper performance bound when MQX is optimally scaled across a high-end multi-core AMD CPU. Our analysis shows that the projected performance approaches that of ASIC implementations.

We believe that narrowing the performance gap between CPUs and specialized hardware for FHE has the potential to significantly broaden its applicability and make privacy-preserving applications more accessible. With this reduced gap, all FHE computations could remain on the CPU, eliminating the need for GPU- or ASIC-based acceleration. This would also remove the substantial overhead incurred by data transfers between separate memory spaces, a critical performance bottleneck for FHE at the application level [35]. Consequently, our approach gains a strong competitive edge, especially since many GPU and ASIC implementations reported in the literature do not account for data transfer times in their performance evaluations of cryptographic kernels.

Contributions. This paper makes the following contributions:

(1) Efficient implementations of cryptographic kernels on CPUs using scalar, AVX2, and AVX-512 instructions.
(2) A proposed ISA extension—multi-word extension (MQX)—that addresses the performance bottlenecks of AVX-512 for large integer arithmetic and further improves the performance of CPU-based cryptographic kernels.
(3) A demonstration that, on a single CPU core, AVX-512-based NTTs and BLAS operations achieve a 38 times and a 62 times speedup, respectively, over the state-of-the-art CPU baselines. The speedup is further compounded by MQX, resulting in a 77 times and 104 times speedup, respectively. Using a *single* CPU core, MQX narrows the performance gap to ASICs to as low as a 35 times slowdown for NTTs.
(4) A roofline analysis of MQX scaling across a multi-core CPU, which suggests that when all cores of a state-of-the-art server-grade CPU are optimally utilized, cryptographic kernels can achieve performance comparable to specialized hardware.

2 Background

In this section, we introduce modular arithmetic and double-word arithmetic, which, when combined, form the large integer arithmetic used in cryptographic kernels within the scope of this work. We then describe the cryptographic kernels of interest—NTT and BLAS operations—and their roles in polynomial arithmetic. Lastly, we discuss SIMD vectorization, a key technique for achieving high performance on CPUs. We follow some of the mathematical notations adopted in prior work on multi-word modular arithmetic [51].

2.1 Modular Arithmetic

Let $\mathbb{Z}_q$ be an integer ring modulo q. We can define the modular arithmetic mathematically as

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

where $a, b, c \in \mathbb{Z}_q$.

Given that on typical hardware the modulo operation is significantly more costly than basic operations such as addition and multiplication, we can take advantage of the fact that $0 \leq a < q$ and $0 \leq b < q$ for $a, b \in \mathbb{Z}_q$, and propose an algorithm to compute modular addition:

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

A similar strategy can be applied to modular subtraction:

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

However, for multiplication, the conditional check is no longer effective. Instead, we employ Barrett reduction [3], which uses a pre-computed value μ to replace the modulo operation with cheaper operations such as binary shift and multiplication. Formally:

$$c = ab - \lfloor ab\mu/2^k \rfloor q, \tag{4}$$

Table 1: Implementations of addition with carry in cryptographic settings: scalar, AVX-512, and MQX.

Scalar	**AVX-512**	**MQX**
`// In: uint64_t a, b; _Bool ci`	`// In: __m512i a, b; __mmask8 ci`	`// In: __m512i a, b; __mmask8 ci`
`// Out: uint64_t c; _Bool co`	`// Out: __m512i c; __mmask8 co`	`// Out: __m512i c; __mmask8 co`
`uint64_t t0, t1; _Bool q0, q1;`	`__m512i t0, t1, one; __mmask8 q0, q1;`	`c = _mm512_adc_epi64(a, b, ci, &co);`
`t0 = a + b;`	`t0 = _mm512_add_epi64(a, b);`	
`t1 = t0 + ci;`	`one = _mm512_set1_epi64(1);`	
`q0 = (t1 < a);`	`t1 = _mm512_mask_add_epi64 (t0, ci, t0, one);`	
`q1 = (t1 < b);`	`q0 = _mm512_cmp_epu64_mask(t1, a, _MM_CMPINT_LT);`	
`co = q0 || q1;`	`q1 = _mm512_cmp_epu64_mask(t1, b, _MM_CMPINT_LT);`	
	`co = q0 | q1;`	

where $\mu = \lfloor 2^k/q \rfloor$, and $\lfloor . \rfloor$ denotes the floor operation. μ only needs to be computed once for the same modulus q. k is chosen to satisfy $2^{k/2} > q$, which guarantees sufficient precision when approximating division using μ. Barrett reduction is widely used within the cryptography community [26, 28, 33, 42, 46] as it works on general moduli rather than a specialized modulus chosen for application-specific optimizations (e.g, Goldilocks prime [20]). A key aspect of Barrett reduction is that if the target data bit-width in modular arithmetic is l (usually a power of two), the modulus q must have fewer than or equal to $l - 4$ bits. This ensures that μ stays within l bits. For example, for 128-bit integer arithmetic, q must be less than or equal to 124 bits.

2.2 Double-Word Arithmetic

In this work, we define a *word* (or *machine word*) as the largest integer data type that can fit into a general-purpose register. For example, a word has 64 bits on x86-64 architectures. In the context of this work, we refer to the bit-width of interest for FHE applications as a 128-bit *double-word.* A double-word can be mathematically represented using the following notation:

$$[x_0, x_1]_{2^{\omega_0}} = x_0 2^{\omega_0} + x_1 = x, \tag{5}$$

where ω_0 is the machine word width. In this work, $\omega_0 = 64$, with x_0 representing the higher 64 bits and x_1 the lower 64 bits of a 128-bit integer. The double-word arithmetic, however, is applicable to other machine word widths as well.

Now, we can formally introduce double-word integer arithmetic. Let $a = [a_0, a_1]_{2^{\omega_0}} = a_0 2^{\omega_0} + a_1$, $b = [b_0, b_1]_{2^{\omega_0}} = b_0 2^{\omega_0} + b_1$. Double-word addition can be written as

$$\begin{aligned} [\delta, c_2]_{2^{\omega_0}} &= a_1 + b_1, \\ [c_0, c_1]_{2^{\omega_0}} &= a_0 + b_0 + \delta, \end{aligned} \tag{6}$$

where $c = [c_0, c_1, c_2]_{2^{\omega_0}}$ and $\delta \in \{0, 1\}$. Table 1 presents three concrete implementations of $[c_0, c_1]_{2^{\omega_0}} = a_0 + b_0 + \delta$ in C, which performs addition with carry-in (δ) and carry-out (c_0): the scalar version (i.e., standard C targeting x86-64 architectures), the SIMD version using AVX-512, and the version utilizing our proposed ISA extension, MQX, where a single instruction computes addition with carry in a SIMD fashion. Note that both the scalar and AVX-512 implementations in Table 1 operate only on the higher bits of a 128-bit integer (a_0 and b_0). This approach leverages the fact that, under Barrett reduction, we work with 124-bit integers or smaller (as opposed to full 128-bit integers), as discussed in Section 2.1.

Double-word subtraction 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 $c = [c_0, c_1]_{2^{\omega_0}}$. The schoolbook double-word multiplication can be written as

$$c = (a_0 b_0) 2^{2\omega_0} + (a_0 b_1 + a_1 b_0) 2^{\omega_0} + a_1 b_1, \tag{8}$$

which involves four single-word multiplications. We also explore an alternative multiplication algorithm, the Karatsuba algorithm [25], which reduces the number of single-word multiplications to three and can be written as:

$$c = (a_0 b_0) 2^{2\omega_0} + \big((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1\big) 2^{\omega_0} + a_1 b_1. \tag{9}$$

2.3 Polynomial Operations and NTT

Polynomial operations with coefficients that reside in $\mathbb{Z}_q$ are the building blocks (which constitute the prohibitive overhead) in many advanced encryption schemes such as FHE. In this section, we introduce how point-wise polynomial operations can be captured as BLAS operations and how NTT accelerates polynomial multiplication.

Point-wise polynomial operations. Point-wise polynomial addition, subtraction, and multiplication are commonly utilized in FHE schemes [1]. Let f and g denote two polynomials of degree n, where $f = \sum_{i=0}^{n} a_i x^i$, $g = \sum_{j=0}^{n} b_j x^j$, and $a_i, b_j \in \mathbb{Z}_q$. Point-wise polynomial operation acts on the corresponding coefficients of two polynomials (a_i and b_j when $i = j$).

We can represent each polynomial using a vector; for example, f can be written in the vector form as $[a_0, a_1, \ldots, a_n]$. Then, we can use the BLAS abstraction [5] to capture the three aforementioned point-wise polynomial operations as vector addition, vector subtraction, and point-wise vector multiplication. Vector addition and subtraction can be interpreted as variants of a BLAS Level 1 operation named axpy, which is defined as $y = ax + y$, where x and y are vectors and a is a scalar. Point-wise vector multiplication can be interpreted as a special case of a BLAS Level 2 operation named gemv, which computes a general matrix-vector multiplication.

Polynomial multiplication using NTT. Using the same definition of f and g as above, we can define the schoolbook polynomial multiplication of f and g in $\mathbb{Z}_q$ as

$$f(x)g(x) = \sum_{j=0}^{2n} \left(\left(\sum_{i=0}^{j} a_i b_{j-i} \right) \bmod q \right) x^j, \tag{10}$$

where $a_i = b_i = 0$ for $i > n$. While the time complexity of the schoolbook polynomial multiplication is $O(n^2)$, NTT reduces the complexity to $O(n \log n)$. Similar to how the discrete Fourier transform converts a signal from the time domain to the frequency domain, NTT can convert a polynomial from its coefficient form (e.g., $f(x) = x^3 + 3x^2 + 2x + 1 \bmod 5$) to its evaluation form (e.g., $\{f(1), f(2), f(3), f(4)\}$). Formally, we define an n-point NTT as

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk} \bmod q, \quad 0 \le k \le n-1, \tag{11}$$

where x_j and y_k are input and output sequences, respectively, and ω_n is the n-th primitive root of unity. Even though NTT effectively reduces the time complexity of the schoolbook algorithm, polynomial multiplication using NTT still accounts for the majority of the computational overhead. Prior work has shown that NTT accounts for more than 90% of FHE-based application execution time in practice [13].

2.4 SIMD Vectorization

SIMD vector instructions are ISA extensions designed for parallel computation on short vectors of integers and floating-point numbers. These instructions operate on vector registers, typically 128 to 512 bits wide, and divide them into v-way vectors of smaller data types (usually ranging from 2-way to 16-way, where *way* refers to the number of elements or lanes in each vector). Popular families of SIMD instructions include MMX, Streaming SIMD Extensions (SSE), Advanced SIMD (Neon), and Advanced Vector Extensions (AVX).

In our work, we focus on two SIMD instruction sets in the AVX family: AVX2 and AVX-512. We utilize the largest integer data type supported by AVX, which is 64 bits, for 128-bit integer arithmetic. Introduced in 2013, AVX2 supports vector registers of 256 bits, which can be divided into 4 ways, with each element holding a 64-bit integer. AVX-512, released in 2016, extends AVX2 by supporting vector registers of 512 bits (e.g., 8-way 64-bit vectors) and mask registers. AVX-512 is a family of extensions that includes specialized instruction sets such as AVX-512 vector neural network instructions (AVX-512 VNNI) for neural networks and AVX-512 Galois field new instructions (AVX-512 GFNI) for Galois fields. While the core AVX-512 foundation (AVX-512F) extension is required across all AVX-512 implementations, the remaining extensions can be implemented independently of one another. Table 1 presents example AVX-512 instructions for 64-bit integers, including 8-way vector addition `_mm512_add_epi64` and 8-way vector comparison `_mm512_cmp_epu64_mask`.

3 SIMD-Vectorized Cryptographic Kernels

To narrow the performance gap between CPUs and specialized hardware, we first investigate the performance improvements achievable using existing technology on state-of-the-art CPUs for cryptographic kernels. Our initial step is to develop a high-performance

```
uint128_t addmod128(uint128_t a, uint128_t b, uint128_t m) {
    _Bool a31, a34, a35, a38, b1, c1, c2, i27, ...;
    uint64_t d1, d2, d3, t28, t29, t30, al, ah, bl, ...;
    al = LO64(a); ah = HI64(a); bl = LO64(b); bh = HI64(b);
    ml = LO64(m); mh = HI64(m);
    t30 = al + bl; q1 = (t30 < al); q2 = (t30 < bl);
    c1 = q1 || q2; t28 = ah + bh; t29 = t28 + c1;
    q3 = (t29 < ah); q4 = (t29 < bh); c2 = q3 || q4;
    a31 = (mh < t29); a35 = (mh == t29); a38 = (ml <= t30);
    a34 = (a35 && a38); i27 = (a31 || a34); i28 = c2 || i27;
    d1 = t30 - ml; b1 = !a38; d2 = t29 - mh;
    d3 = d2 - b1; ch = i28 ? d3 : t29; cl = i28 ? d1 : t30;
    return INT128(ch, cl);
}
```

Listing 1: Scalar double-word modular addition without using 128-bit data types for computation.

scalar implementation without utilizing any SIMD instructions. We then further enhance performance by leveraging SIMD instructions, specifically AVX2 and AVX-512.

3.1 Scalar Double-Word Modular Arithmetic

To efficiently implement the 128-bit arithmetic used in NTT and BLAS operations with 64-bit machine words, we combine modular arithmetic (Section 2.1) and double-word arithmetic (Section 2.2) to construct double-word modular arithmetic. We leverage the fact that $0 \le a < q$ and $0 \le b < q$ to design efficient modular addition and subtraction algorithms, replacing division by the modulus with a conditional check. For modular multiplication, we implement two algorithms: one based on the schoolbook multiplication and the other on the Karatsuba algorithm. Both employ Barrett reduction for efficient modulo operation, which replaces the division with less expensive multiplications and shifts. Most importantly, under the assumption that q is always less than or equal to 124 bits in our work (necessitated by Barrett reduction), much of the branching logic and conditional assignments can be eliminated.

We implement two versions of scalar double-word modular arithmetic, one uses the natively supported double-word representation (i.e., `unsigned __int128` in C) to store the results of 64-bit arithmetic, while the other only uses the single-word data type (`uint64_t`) for computation. The former is used for benchmarking, as it allows the compiler to exploit specialized assembly instructions such as add with carry for efficient carry propagation. The second implementation is essential for SIMD-vectorized implementations, as it allows for a more natural translation to AVX2 and AVX-512 instructions, where the maximum data type supported for each vector element is 64 bits. In Listing 1, we demonstrate how scalar double-word modular addition is performed using only 64-bit integers for computation. In this code, `LO` is a macro that extracts the lower 64-bit part of a 128-bit integer, while `HI` extracts the higher 64-bit part. `INT128` combines two 64-bit integers together into a 128-bit integer, treating the first input as the higher part and the second as the lower part.

3.2 SIMD-Vectorized Double-Word Modular Arithmetic

To take advantage of the vector parallelism offered by CPUs, we employ SIMD instructions to vectorize the scalar implementation,

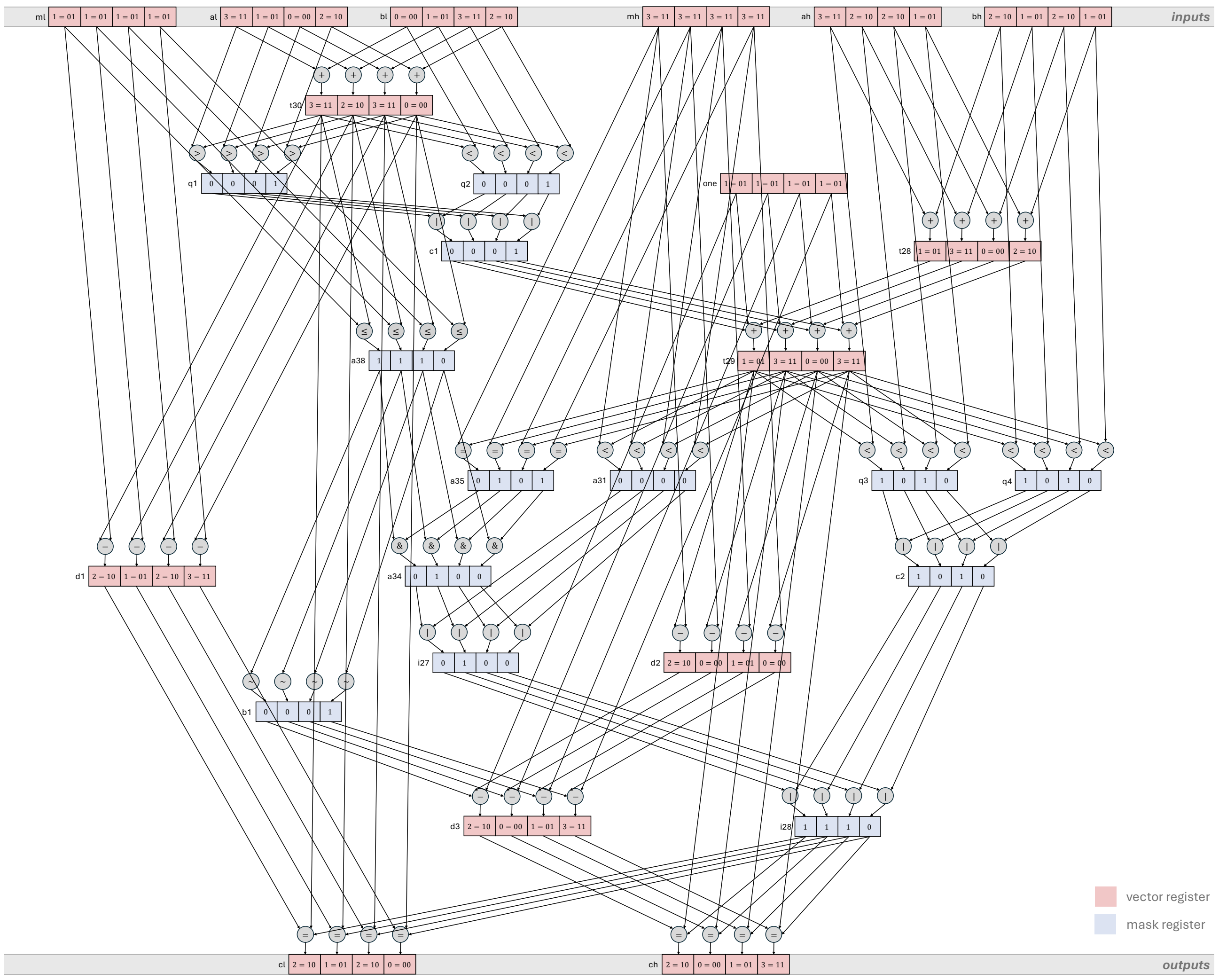


Figure 2: Illustration of double-word modular addition using 4-way vectors, where each element is a 2-bit integer.

beginning with AVX-512. Since the largest integer bit-width natively supported by AVX-512 is 64 bits, we divide the 128-bit input vector into two 64-bit vectors: one containing the high parts and the other containing the low parts. This is demonstrated in Figure 2, where input a is split into ah and al, for example. Consequently, unlike the scalar implementation, which processes one 128-bit integer per input (a, b, q in $c = a + b \bmod q$), the vectorized implementation passes in two 512-bit vectors per input, corresponding to eight 128-bit integers, and produces two 512-bit vectors as output. Figure 2 illustrates our strategy of implementing double-word modular addition using SIMD instructions, with a toy example where each vector contains four elements and each element is 2 bits. This example highlights the complexity of efficiently implementing double-word modular addition using SIMD instructions. The corresponding C implementation using AVX-512 is also provided in Listing 2, in which one is globally set as `_mm512_set1_epi64(1)`.

When translating from scalar implementation to AVX-512, most arithmetic operations map directly to corresponding AVX-512 instructions that operate on 512-bit vectors. Additionally, AVX-512 supports mask registers and associated operations, which allows us to perform conditional statements from the scalar code without branching. For instance, the vector compare function in AVX-512 outputs a packed 8-bit mask, with each bit corresponding to an element index. This mask can then be used in subsequent operations to control whether an operation is applied to a given index or lane within the vector. This capability allows us to use instructions such as `_mm512_mask_blend_epi64`, which takes two vectors and an 8-bit mask and produces a blended result with values based on which mask bits are set.

We also implement SIMD-vectorized modular arithmetic using AVX2. When transitioning from AVX-512 to AVX2, each SIMD instruction handles half as many elements as AVX-512 (four SIMD lanes instead of eight). Moreover, due to the absence of unsigned comparisons and mask registers in AVX2, the comparison operations for 128-bit modular arithmetic require more instructions and additional handling compared to AVX-512.

From arithmetic to cryptographic kernels. At this point, we have discussed how to build efficient scalar and SIMD-vectorized

```
void addmod128(__m512i* ch, __m512i* cl, __m512i ah, __m512i al,
               __m512i bh, __m512i bl, __m512i mh, __m512i ml) {
    __m512i t30, t28, t29, d1, d2, d3;
    __mmask8 q1, q2, c1, q3, q4, c2, a31, a35, ...;
    t30 = _mm512_add_epi64 (al, bl);
    q1 = _mm512_cmp_epu64_mask(t30, al, _MM_CMPINT_LT);
    q2 = _mm512_cmp_epu64_mask(t30, bl, _MM_CMPINT_LT);
    c1 = q1 | q2;
    t28 = _mm512_add_epi64 (ah, bh);
    t29 = _mm512_mask_add_epi64 (t28, c1, t28, one);
    q3 = _mm512_cmp_epu64_mask(t29, ah, _MM_CMPINT_LT);
    q4 = _mm512_cmp_epu64_mask(t29, bh, _MM_CMPINT_LT);
    c2 = q3 | q4;
    a31 = _mm512_cmp_epu64_mask(mh, t29, _MM_CMPINT_LT);
    a35 = _mm512_cmp_epu64_mask(mh, t29, _MM_CMPINT_EQ);
    a38 = _mm512_cmp_epu64_mask(ml, t30, _MM_CMPINT_LE);
    a34 = a35 & a38; i27 = a31 | a34; i28 = c2 | i27;
    d1 = _mm512_sub_epi64(t30, ml); b1 = ~a38;
    d2 = _mm512_sub_epi64(t29, mh);
    d3 = _mm512_mask_sub_epi64(d2, b1, d2, one);
    *ch = _mm512_mask_blend_epi64(i28, t29, d3);
    *cl = _mm512_mask_blend_epi64(i28, t30, d1);
}
```

Listing 2: Double-word modular addition using AVX-512.

double-word modular arithmetic. Extending modular arithmetic to BLAS operations is relatively straightforward, as BLAS operations are essentially vector-based modular arithmetic (e.g., a vector of 1,024 elements, not a SIMD vector). They can be implemented by looping over scalar or SIMD modular arithmetic. In cryptographic settings, the vector length used in BLAS operations is typically a power of two, and in this work, we assume it is a multiple of the SIMD lane size we use. Transitioning from double-word modular arithmetic to NTTs is more complicated. On the computation side, the core NTT building block is the *butterfly*, which consists of one modular addition, one modular subtraction, and one modular multiplication. An n-point NTT consists of $\log n$ stages, with each stage containing $n/2$ butterflies. The results from each butterfly in the previous stage need to be communicated to the next stage. We build upon prior work that implements AVX-512-based NTT [17], which employs the Pease algorithm [34] for the NTT dataflow. The data permutation stage requires the use of the unpack and permute instructions, such as `_mm512_unpacklo_epi64`, `_mm512_unpackhi_epi64`, and `_mm512_permutex2var_epi64` in AVX-512.

4 Multi-word Extension (MQX)

Although SIMD vectorization enhances the performance of the scalar implementation, it still falls short of the levels achievable on ASICs based on our empirical results (discussed in Section 5). We compare the scalar implementation of double-word addition with carry against the AVX-512 implementation in Table 1 and observe that the scalar code can be captured in x86 assembly as a single ALU instruction, `ADC`, whereas the AVX-512 implementation requires six SIMD instructions to achieve the same functionality. Therefore, to address the performance bottlenecks of existing SIMD instructions in cryptographic settings, we propose multi-word extension (MQX), an ISA extension designed to support the large integer arithmetic used by cryptographic kernels while leveraging SIMD vector parallelism. The design philosophy behind MQX is to introduce minimal

```
void addmod128(__m512i* ch, __m512i* cl, __m512i ah, __m512i al,
               __m512i bh, __m512i bl, __m512i mh, __m512i ml) {
    __m512i el, eh, c1; __mmask8 elc, ehc, ehc1, ctrl, clc;
    el = _mm512_adc_epi64(al, bl, z_mask, &elc);
    eh = _mm512_adc_epi64(ah, bh, elc, &ehc);
    ehc1 = _mm512_cmp_epi64_mask(mh, eh, _MM_CMPINT_LT);
    ctrl = (ehc1 | ehc);
    c1 = _mm512_sbb_epi64(el, ml, z_mask, &clc);
    *cl = _mm512_mask_blend_epi64(ctrl, el, c1);
    c1 = _mm512_sbb_epi64(eh, mh, clc, &ehc);
    *ch = _mm512_mask_blend_epi64(ctrl, eh, c1);
}
```

Listing 3: Double-word modular addition using MQX.

changes to the existing ISA. We expect the required engineering effort to be manageable, given MQX's close similarity to prior vector instruction sets, and we introduce our performance modeling of MQX based on existing AVX-512 instructions. Table 2 presents the proposed MQX instruction set that extends AVX-512 (referred to as AVX-512 MQX). However, MQX is not limited to AVX-512; it can also be integrated with other SIMD instruction sets, such as AVX2 and Neon, since both the word size and the number of SIMD lanes are configurable in MQX. In this work, we focus on AVX-512 MQX and refer to it simply as MQX.

4.1 Design Philosophy

The design philosophy of MQX is to minimize the required engineering effort from the hardware side. To this end, we propose only three new instructions, all in SIMD form: i) widening multiplication, ii) addition with carry, and iii) subtraction with borrow. Each instruction is based on features already present in the x86 architecture. We also find strong similarities between the proposed MQX instructions and the SIMD counterparts implemented by Intel in the Larrabee architecture [38]. Introduced around 2008, Larrabee featured a new ISA known as Larrabee New Instructions (LRBni), which extended in-order x86 CPU cores with wide vector processing units and fixed-function logic blocks. Its goal was to achieve higher performance per watt and per unit area than out-of-order CPUs, particularly for highly parallel workloads. The Larrabee architecture later evolved into the Intel Many Integrated Core (MIC) architecture, with the first commercial product released as the Knights Corner (KNC) processor, whose intrinsics were documented in the Intel Intrinsics Guide [23] from versions 3.1 to 3.6.5. Built upon the foundations of both x86 and previously proposed SIMD extensions, each MQX instruction will be introduced in detail in the following paragraphs.

Widening multiplication. `void _mm512_mul_epi64(__m512i* ch, __m512i* cl, __m512i a, __m512i b)` is a widening multiplication. For each SIMD lane, it multiplies two 64-bit words and stores the high part of the result in one 64-bit word and the low part in another. This mirrors the behavior of the unsigned multiply `MUL` in x86, where the result is stored in a register pair containing both high and low parts. In AVX-512, there is support for variants of lower bit-width such as `__m512i _mm512_mul_epi32 (__m512i a, __m512i b)`, which multiplies the low signed 32-bit integers in each 64-bit SIMD lane and stores the 64-bit result. However, for 64-bit inputs in AVX-512, we only have multiply-low as `__m512i _mm512_mullo_epi64 (__m512i a, __m512i b)` that returns only the lower 64 bits of the result. In LRBni,

Table 2: AVX-512 multi-word extension (MQX).

Instruction	Emulation	Description
`void _mm512_mul_epi64(__m512i* ch, __m512i* cl, __m512i a, __m512i b)`	`for (i = 0; i < 8; i++) { *ch[i] = ((i128) a[i] * (i128) b[i]) >> 64; *cl[i] = a[i] * b[i]; }`	Multiply two words, and output the high and low parts of the result as two words.
`__m512i _mm512_adc_epi64(__m512i a, __m512i b, __mmask8 ci, __mmask8* co)`	`for (i = 0; i < 8; i++) { *co[i] = ((i128) a[i] + (i128) b[i] + (i128) ci[i]) >> 64; c[i] = a[i] + b[i] + ci[i]; return c; }`	Add two words and a carry bit, and output a word and a carry bit.
`__m512i _mm512_sbb_epi64(__m512i a, __m512i b, __mmask8 bi, __mmask8* bo)`	`for (i = 0; i < 8; i++) { *bo[i] = ((i128) a[i] - (i128) b[i] - (i128) bi[i]) >> 127; c[i] = a[i] - b[i] - bi[i]; return c; }`	Subtract two words and a borrow bit, and output a word and a borrow bit.

Intel introduced a vector multiply-high `vmulhpi` for 32-bit signed integers. Similarly, the KNC architecture includes `__m512i _mm512_mulhi_epi32 (__m512i a, __m512i b)`, as documented in earlier versions of the Intel Intrinsics Guide. In Section 5.5, we evaluate whether the hardware engineering cost of a widening multiply can be justified and analyze the performance trade-offs of implementing only multiply-high instead of a full widening multiplication.

Addition with carry. `__m512i _mm512_adc_epi64(__m512i a, __m512i b, __mmask8 ci, __mmask8* co)` performs a per-lane 64-bit addition with carry-in and outputs both the addition result and carry-out. This directly mirrors the behavior of `ADC` in x86, where the carry flag `CF` is used as input and is also updated as part of the result. In LRBni, `vadcpi` performs vector addition with carry-in and carry-out on 32-bit integers. There exists a 32-bit counterpart in KNC as well: `__m512i _mm512_adc_epi32 (__m512i v2, __mmask16 k2, __m512i v3, __mmask16* k2_res)`.

Subtraction with borrow. Similar to addition with carry, `__m512i _mm512_sbb_epi64(__m512i a, __m512i b, __mmask8 bi, __mmask8 bo)` implements a per-lane 64-bit subtraction with borrow-in and outputs the subtraction result with borrow-out. This instruction mirrors the behaviors of `SBB` in x86, where the carry flag `CF` (used for borrow) is taken as input and updated based on the result. In LRBni, the corresponding instruction is `vsbbpi`, which performs vector subtraction with borrow-in and borrow-out for 32-bit integers. Similarly, in KNC, we can find the 32-bit version of subtraction with borrow as `__m512i _mm512_sbb_epi32 (__m512i v2, __mmask16 k, __m512i v3, __mmask16* borrow)`.

The C code for double-word modular addition using the proposed MQX instructions is provided in Listing 3, where `z_mask` is a global 8-bit mask of zeros. In summary, MQX is designed with the goal of requiring relatively low hardware engineering effort. Each MQX instruction has a scalar equivalent and a closely related vector counterpart that has previously been proposed or implemented by Intel. The core idea behind MQX is to introduce wider multiplication units and first-class support for carry and borrow in vector operations, enabling more efficient handling of large integer arithmetic for cryptographic kernels.

4.2 Performance Projection using Proxy ISA (PISA)

After proposing the MQX instructions, we aim to evaluate the performance gains they offer and assess whether these gains justify implementing the instructions in hardware.

While open-source simulation tools such as gem5 [4] and zSim [37] offer detailed evaluations of microarchitectural designs, they fall short in capturing the proprietary designs developed by manufacturers like Intel and AMD. In particular, zSim was primarily designed for Haswell-era processors and has not been actively updated to support newer architectures: The majority of its codebase was committed over a decade ago, with the latest update occurring two years ago. The persistent lack of support for modern microarchitectures stems from the significant engineering effort required to build accurate simulators, compounded by the proprietary and undocumented nature of commercial CPU designs. Meanwhile, gem5 offers only limited support for native AVX intrinsics [27]. In principle, both baseline AVX and MQX could be implemented in gem5, but doing so would require substantial engineering effort—particularly given that the last officially published gem5 extension only partially implemented the SSE family that dates back over two decades. There have been two open-source, community-driven efforts to extend gem5 with AVX-512 support; however, both remain incomplete. Zhang's work [52] lacks masking support (which is essential for modular arithmetic), while Lee et al.'s work [27] omits memory alignment support.

To facilitate the performance modeling process, we propose a new performance modeling approach named performance projection using proxy ISA (PISA). PISA allows us to estimate the performance of MQX without trying to reproduce or reverse engineer the proprietary microarchitectures from Intel or AMD. Specifically, we designate AVX-512 as the proxy ISA and project the performance of MQX by mapping each MQX instruction to the most structurally similar AVX-512 instruction. In other words, if implemented efficiently, each MQX instruction could achieve performance comparable to its AVX-512 proxy. The specific proxy instructions in AVX-512 used in our modeling are shown in Table 3.

Table 3: Proxy instructions in AVX-512 for MQX performance projection.

MQX instruction	AVX-512 proxy instruction
`_mm512_mul_epi64`	`_mm512_mullo_epi64`
`_mm512_adc_epi64`	`_mm512_mask_add_epi64`
`_mm512_sbb_epi64`	`_mm512_mask_sub_epi64`

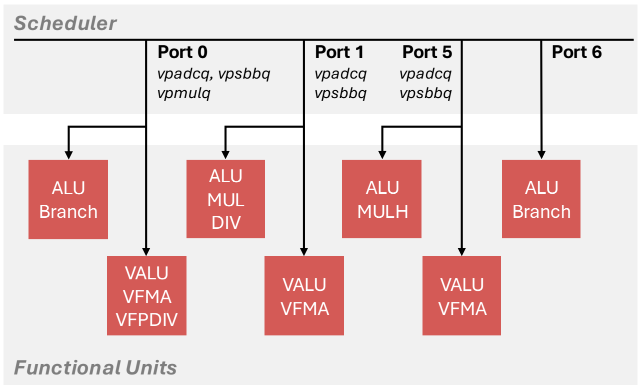


Figure 3: Simplified Sunny Cove microarchitecture used by Intel Xeon.

PISA enables us to receive fast and early feedback in the software-hardware co-design cycle. By modeling many potential instructions via proxy mappings, PISA enables quick exploration of different ISA extensions. Moreover, rather than relying on simulation results, PISA allows us to report real measurements on actual hardware, grounding our projections in real-world performance. We view PISA and microarchitectural simulators as complementary: PISA offers a top-down, actionable alternative for performance modeling and helps assess whether an ISA extension is promising enough to warrant the substantial engineering effort required to implement it in cycle-level simulators or in other tools for detailed analysis, such as power and area estimation.

We are extremely careful in choosing proxy instructions. A key observation that grounds our performance projections is that, based on both Intel's optimization manual [22] and empirical data[1], the x86 instruction set shows no measurable latency difference between `ADD` and `ADC`, between `SUB` and `SBB`, or between 32-bit and 64-bit `MUL` on both Ice Lake and Zen 4 architectures, which we used to benchmark MQX's performance in Section 5. While we acknowledge that extending scalar performance behavior to SIMD is nontrivial, these observations provide a reasonable foundation for our performance projections. Furthermore, in Section 5.2, we present sanity checks that validate the proposed PISA methodology.

Machine code analysis. Using PISA, we are able to leverage machine code analysis tools such as LLVM-MCA to simulate how each MQX instruction would be scheduled on CPUs, assuming each instruction maps to the same execution port as its proxy ISA counterpart. We apply LLVM-MCA to analyze the behavior of modular addition, subtraction, and multiplication with AVX-512

[1]https://uops.info/table.html

```
AVX-512 - Resource pressure by instruction:
[0]    [1]    [2]    [3]    [4]    [5]    Instructions:
 -     1.00    -      -      -      -     vpaddq     %zmm2, %zmm3, %zmm3
 -      -      -      -      -     1.00   vpcmpltuq  %zmm2, %zmm3, %k1
 -     1.00    -      -      -      -     vpaddq     %zmm4, %zmm6, %zmm2
 -     1.00    -     1.00    -      -     vmovdqu64  one(%rip), %zmm5
1.00    -      -      -      -      -     vpaddq     %zmm5, %zmm2, %zmm2 {%k1}
 -      -      -      -      -     1.00   vpcmpnleuq %zmm0, %zmm2, %k0
 -      -      -      -      -     1.00   vpcmpeqq   %zmm0, %zmm2, %k1
 -      -      -      -      -     1.00   vpcmpltuq  %zmm1, %zmm3, %k2
 -      -      -      -      -     1.00   vpcmpnltuq %zmm1, %zmm3, %k1 {%k1}
 -      -      -      -      -     1.00   vpmaxuq    %zmm6, %zmm4, %zmm4
 -      -      -      -      -     1.00   vpcmpnleuq %zmm2, %zmm4, %k3
1.00    -      -      -      -      -     korb       %k0, %k3, %k0
1.00    -      -      -      -      -     korb       %k1, %k0, %k1
1.00    -      -      -      -      -     vmovdqa64  %zmm0, %zmm4
 -     1.00    -      -      -      -     vpaddq     %zmm5, %zmm0, %zmm4 {%k2}
 -     1.00    -      -      -      -     vpsubq     %zmm4, %zmm2, %zmm2 {%k1}
 -      -      -      -      -     1.00   vpsubq     %zmm1, %zmm3, %zmm3 {%k1}

MQX - Resource pressure by instruction:
[0]    [1]    [2]    [3]    [4]    [5]    Instructions:
 -     1.00    -      -      -      -     vpadcq %zmm7, %zmm3, %zmm3 {%k1} {%k3}
 -     1.00    -      -      -      -     vmovq  %xmm6, %xmm4
1.00    -      -      -      -      -     vpadcq %zmm5, %zmm4, %zmm4 {%k3} {%k0}
 -      -      -      -      -     1.00   vpcmpgtq %zmm0, %zmm4, %k2
1.00    -      -      -      -      -     korb   %k0, %k2, %k2
 -      -      -      -      -     1.00   vpsbbq %zmm1, %zmm3, %zmm3 {%k1} {%k3}
 -     1.00    -      -      -      -     vpsbbq %zmm2, %zmm4, %zmm4 {%k3} {%k0}
```

Listing 4: LLVM-MCA analysis (edited for brevity) of double-word modular addition with AVX-512 and MQX. The MQX instructions, shown in red, are included as a performance-oriented illustration. Square brackets [#] denote port numbers.

and MQX on Intel Xeon 8352Y, which is based on the Sunny Cove microarchitecture (a simplified version shown in Figure 3).

Listing 4 shows the LLVM-MCA analysis of double-word modular addition with AVX-512 and MQX. To construct the MQX part, we start from the LLVM-MCA analysis of the code used to estimate MQX performance with PISA and then manually edit the original AVX-512 assembly into its MQX counterpart (e.g., `vpsubq` is replaced by `vpsbbq`, and a mask register `k0` is prepended, which `vpsbbq` writes to). This modified listing is *not* semantically correct; rather, it serves as a performance-oriented demonstration that illustrates how MQX instructions would appear in assembly and how they would map to the same execution ports as their AVX-512 proxies.

Functional correctness. In our MQX implementation, we have a flag to check for functional correctness. When the flag is turned off, we execute the code using PISA with the expectation of not getting correct results. With that flag turned on, each MQX instruction is emulated by a scalar implementation as shown in Table 2. Hence, we can obtain functional correctness of MQX and verify that we are not missing any instructions. With PISA, we also carefully inspect the compiler-generated assembly code to make sure no instructions are incorrectly pruned.

5 Evaluation

In this section, we describe our experimental setup and discuss our results of scalar, SIMD-vectorized, and MQX-based cryptographic kernels. We also conduct two sets of sensitivity analyses: one on the MQX instructions and another on the choice between two multiplication algorithms.

5.1 Experimental Setup

We evaluated our proposed approaches on two state-of-the-art server-grade CPUs: Intel Xeon 8352Y, provided as part of the FASTER node at Texas A&M University, and AMD EPYC 9654, provided as part of the Launch node at the same institution. Detailed specifications for each CPU are shown in Table 4. For brevity, we refer to Intel Xeon 8352Y as Intel Xeon and AMD EPYC 9654 as AMD EPYC in the remainder of this paper.

Table 4: CPUs used for benchmarking.

Specification	Intel Xeon 8352Y	AMD EPYC 9654
Base Clock Rate	2.2 GHz	2.4 GHz
Max Clock Rate	3.4 GHz	3.7 GHz
Memory	256 GB DDR4	384 GB DDR5
L3 Cache	48 MB	384 MB

Measuring kernel runtime. We use the Intel oneAPI DPC++/C++ compiler (ICX, version 2024.2.0) to compile all kernels on Intel Xeon, and the AMD Optimizing C/C++ and Fortran Compiler (AOCC, version 4.1.0) to compile all kernels on AMD EPYC. For NTTs, we report the average runtime of the final 50 iterations out of 100 runs; for BLAS operations, we report the average runtime of the final 500 iterations out of 1,000 runs. This approach allows the cache to warm up and stabilize, and mitigates fluctuations in measured runtime (for BLAS operations, in particular). For all BLAS operations, the vector length is set to 1,024, as it aligns with typical polynomial sizes in FHE schemes. Although we have implemented both the Karatsuba and the schoolbook algorithm for modular multiplication, we only report the runtime of the schoolbook implementation, as it consistently outperforms Karatsuba (see Section 5.5 for further discussion). Our timing measurements include data transfer time.

5.2 Validating PISA

Before presenting the performance results for BLAS operations and NTTs, we first perform a sanity check on our PISA performance modeling methodology, which underpins our analysis of MQX. Specifically, to validate PISA, we applied the same modeling methodology used for MQX to *existing* SIMD instructions used in our NTT implementations so that we could establish ground truth and assess the accuracy of PISA's performance predictions.

As shown in Table 5, for multiplication, we use the existing AVX2 widening multiplication on 32-bit elements, `_mm256_mul_epu32`, as the ground truth and `_mm256_mullo_epi32` as the proxy instruction (since there is no `_mm256_mullo_epu32`). This closely mirrors how we model the widening multiplication instruction proposed in MQX, as shown in Table 3. Note that `_mm512_mul_epu32` in AVX-512 is not used as the ground truth, even though it more closely resembles our proposed instruction. The reason is that our AVX-512 NTT implementation operates on 64-bit aligned data and does not utilize 32-bit widening multiplication, and adapting 32-bit SIMD instructions would require re-implementing the entire NTT kernel. For addition and subtraction, we use regular SIMD addition to model masked addition in AVX-512. Consistent with our conservative modeling methodology for MQX, we insert an extra instruction and guard the output with `volatile` to preserve data dependencies on the mask register. This set of experiments is designed to demonstrate PISA's capability of handling cases where the proxy instruction does not take identical input arguments as the target instruction.

Table 5: Target and proxy instructions for validating PISA.

Target instruction	Proxy instruction
`_mm256_mul_epu32`	`_mm256_mullo_epi32`
`_mm512_mask_add_epi64`	`_mm512_add_epi64`
`_mm512_mask_sub_epi64`	`_mm512_sub_epi64`

To evaluate the accuracy of the projected performance, we define the relative error ε as

$$\varepsilon = \frac{t_{\text{target}} - t_{\text{proxy}}}{t_{\text{target}}} \cdot 100\%, \tag{12}$$

where t_{target} denotes the NTT runtime using the target instruction, and t_{proxy} denotes the runtime using the proxy instruction.

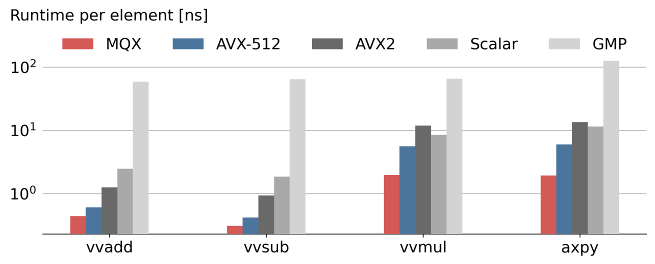
We validated PISA using an NTT size of 2^{14}, the average among the NTT sizes targeted in this paper. As shown in Table 6, the absolute relative error of using PISA is below 8% for all six test cases. Negative values indicate that PISA is overly conservative, projecting higher runtimes than observed. In all cases where PISA is overly optimistic, the relative error remains under 6%. The results suggest that PISA passes a sanity check when modeling both AVX2 and AVX-512 instructions on the Intel and AMD CPUs we tested.

Table 6: Relative error (ε) of PISA-projected runtime on Intel and AMD CPUs.

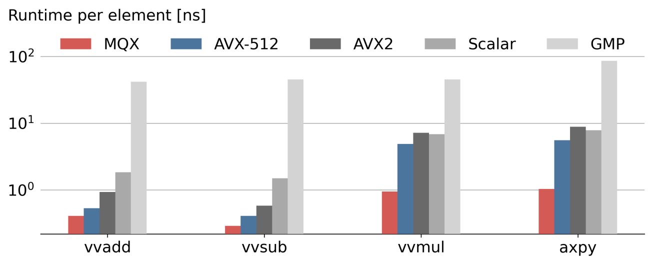
Target instruction	Intel Xeon	AMD EPYC
`_mm256_mul_epu32`	3.23%	2.64%
`_mm512_mask_add_epi64`	−7.68%	5.25%
`_mm512_mask_sub_epi64`	−4.30%	1.27%

5.3 BLAS Operation Results

We evaluated the performance of our BLAS operation implementations using scalar, AVX2, AVX-512, and MQX, and compared them against a baseline implementation using the state-of-the-art arbitrary-precision library GMP. The GMP library was configured to perform exact integer arithmetic, ensuring bitwise-identical results with both our implementation and other baselines. Figure 4a shows the results on Intel Xeon. On Intel Xeon, the scalar implementation of NTT outperforms AVX2 in vector multiplication and axpy, but falls behind in vector addition and subtraction. AVX-512 delivers an average speedup of 2.2 times over AVX2 across four BLAS operations. MQX further improves performance, achieving a 2.2 times speedup over AVX-512 and outperforming all other baselines. GMP exhibits a slowdown of 18.4 times compared to the slower of the AVX2 and scalar implementations.

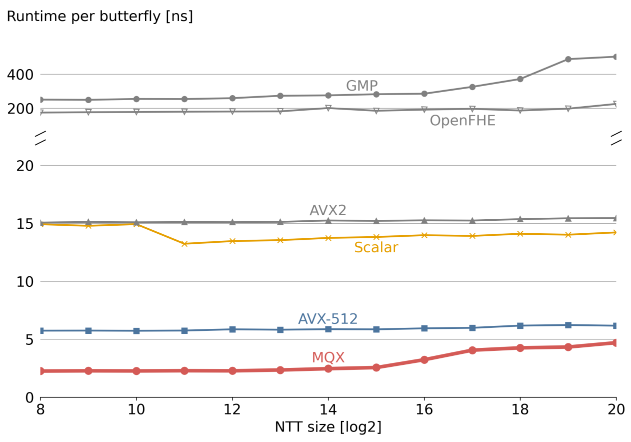


(a) Intel Xeon.

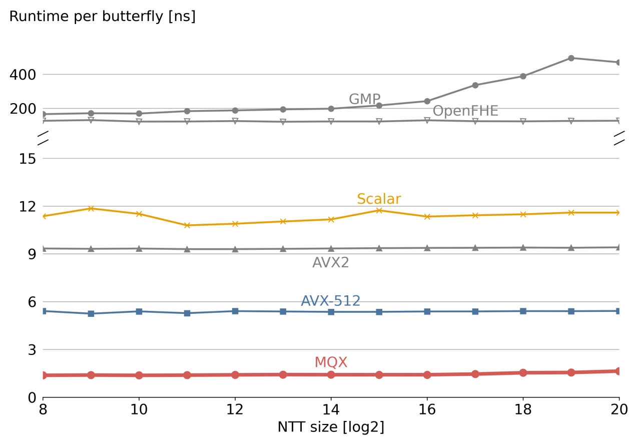


(b) AMD EPYC.

Figure 4: Performance of BLAS operations on a single CPU core.



(a) Intel Xeon.



(b) AMD EPYC.

Figure 5: Performance of NTT on a single CPU core.

As shown in Figure 4b, we observe a similar performance trend on AMD EPYC as on Intel Xeon: MQX performs best, followed by AVX-512, AVX2, scalar, and GMP. AVX2 performs slightly worse than the scalar implementation in vector multiplication and axpy, but achieves a 1.6 times overall speedup across all four BLAS operations. On AMD EPYC, the performance gap between AVX2 and AVX-512 is smaller than on Intel Xeon, with AVX2 falling behind AVX-512 by 1.6 times on average. MQX delivers a 3.2 times speedup over AVX-512, while GMP shows a 17.3 times slowdown compared to the slowest of our implementations.

In summary, by leveraging natively supported SIMD instructions (i.e., AVX-512), we achieve an average speedup of 62 times over the state-of-the-art arbitrary-precision library across the two CPUs evaluated. With the proposed MQX ISA extension, we gain an additional average speedup of 2.7 times over AVX-512.

5.4 NTT Results

For the 128-bit bit-width targeted in this work, we compared against two state-of-the-art libraries. The first one is GMP [18], a general-purpose arbitrary-precision library. The second baseline is the open-source fully homomorphic encryption library (OpenFHE) [1], one of the most widely used open-source FHE libraries that offers efficient implementations of all major FHE schemes. In our experiments, we used its default mathematical backend for large integer arithmetic. We evaluated the performance of our proposed NTT implementations against these two baselines, with all experiments conducted on a single CPU core.

Intel Xeon. As shown in Figure 5a, on Intel Xeon, our scalar implementation outperforms the state-of-the-art FHE library OpenFHE [1] by 13.5 times on average across all NTT sizes. Notably, AVX2 and scalar implementations achieve comparable performance, with the scalar implementation being slightly faster. Using AVX-512 yields a 2.4 times speedup over the scalar implementation, increasing the performance gap against the best baseline to 31.9 times on average. The significant gain achieved even with the scalar implementation highlights the efficiency of our approach.

Using our proposed ISA extension, MQX, we achieve an additional 2.1 times speedup over the AVX-512 implementation, resulting in a 66.9 times speedup over the best baseline. We observe that MQX performance begins to degrade at the NTT size of 2^{16}. We hypothesize that this is the point at which the memory required for NTT exceeds the capacity of the L2 cache. As MQX accelerates computation, the kernel becomes memory-bound, and spilling beyond L2 leads to the observed slowdown. This hypothesis is supported by the calculation that, for an NTT size of 2^{15}, each stage of NTT must hold about 1 MB of 128-bit integers; for a 2^{16}-point NTT, this requirement doubles to 2 MB, exceeding the 1.28 MB per-core L2 cache on Intel Xeon. In contrast, the performance of AVX-512

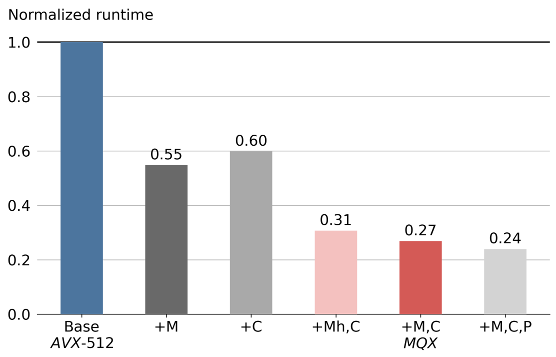


Figure 6: Sensitivity analysis of NTT runtime with respect to MQX instructions.

remains relatively flat across all NTT sizes, as it continues to be compute-bound.

AMD EPYC. As shown in Figure 5b, both GMP and OpenFHE exhibit performance trends similar to those on Intel Xeon. On AMD EPYC, our scalar implementation achieves an average 11 times speedup over OpenFHE across all NTT sizes. Unlike on Intel Xeon, AVX2 outperforms the scalar implementation across all NTT sizes by an average of 1.2 times. AVX-512 delivers a further 1.7 times speedup over AVX2, resulting in a 23.2 times improvement over the state-of-the-art baseline, OpenFHE. The gains from MQX on AMD EPYC are even more evident than on Intel Xeon: With MQX, we achieve another 3.7 times speedup over AVX-512, yielding an overall 86.5 times improvement over OpenFHE.

5.5 Sensitivity Analysis

MQX components. We quantify how each component of MQX contributes to the overall performance gain over the best-performing existing SIMD instruction set, AVX-512. In Figure 6, we report the average runtime per butterfly across all tested NTT sizes, normalized to the AVX-512 runtime (Base). All experiments were conducted on AMD EPYC. In the figure, +M refers to augmenting AVX-512 with only widening multiplication, while +C adds only carry-flag support (including both addition with carry and subtraction with borrow). +M,C represents the full MQX extension, incorporating both features. The results show that widening multiplication contributes slightly more to performance improvements than carry-flag support when each is added individually.

Next, we investigate whether the hardware requirements can be reduced by replacing full widening multiplication with a multiply-high operation, denoted as +Mh,C. Under our PISA model, we project the performance of widening multiplication using a single multiply-low instruction. In this analysis, we also model multiply-high with the same latency as multiply-low, decomposing the widening multiplication into two separate instructions. Our results show that this substitution leads to only a minor performance degradation. This suggests that hardware vendors could implement a multiply-high instruction as a lower-cost alternative to full widening multiplication, while still retaining most of the performance benefits.

Predicated execution. When designing MQX, we also explored whether adding predicated execution support (denoted as +M,C,P) to the existing MQX instructions would further improve performance. To this end, we propose predicated addition with carry and predicated subtraction with borrow. Predicated addition with carry returns either the summation of two words and a carry-in as a single word (without setting the output carry flag), or simply returns the first input word. Predicated subtraction with borrow is defined similarly. Predication was also explored in LRBni, where each vector instruction can be predicated using a mask register. Similar concepts have been explored in the Intel Itanium architecture through the use of 1-bit predicate registers. However, as shown in Figure 6, the performance improvement from predicated execution is modest—only a 1.1 times speedup over standard MQX. As a result, we chose not to include predicated executions in MQX in order to minimize additional hardware engineering effort.

Choice of multiplication algorithms. We conduct another sensitivity analysis to evaluate the performance impact of two multiplication algorithms, schoolbook and Karatsuba, across various NTT implementations. Our results show that schoolbook outperforms Karatsuba in almost all NTT variants (i.e., scalar, AVX2, AVX-512, and MQX) on both CPUs, with the only exception that their performance is nearly identical for the scalar implementation on AMD EPYC. When the schoolbook multiplication achieves better performance, it provides an average 1.1 times speedup over Karatsuba. These results suggest that, for NTT workloads on CPUs, replacing multiplication with more additions using Karatsuba does not lead to significant performance gains. This contrasts with prior findings on GPUs [51], where the Karatsuba algorithm achieved a 2.1 times speedup over the schoolbook multiplication for 128-bit inputs on NVIDIA GeForce RTX 4090.

6 Roofline Analysis

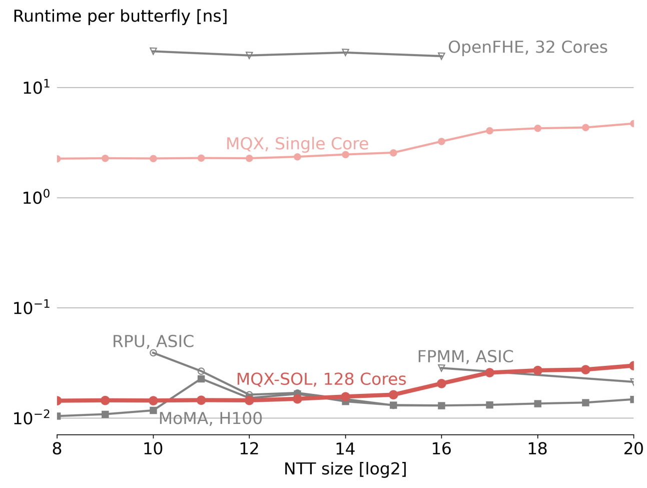
In this section, we present a roofline-like [48] speed-of-light (SOL) performance model to estimate the upper bound achievable on top-tier server-class CPUs, assuming an optimal mapping of our current single-core implementation across the entire CPU. Our analysis compares projected CPU performance against two ASIC implementations, RPU [42] and Zhou et al.'s work [53], one GPU implementation, MoMA [51], and the multi-core performance of OpenFHE, as reported in the RPU paper. For brevity, we henceforth refer to Zhou et al.'s work as FPMM.

For our SOL performance estimates, we select the highest-end server-grade Intel and AMD CPUs that support AVX-512. We normalize the performance by clock frequency and scale it by the number of cores. Formally, let c_1 and c_2 denote the number of cores on the CPU used for measurement and the target CPU, respectively. Let f_m be the measured operating frequency and f_max the all-core boost frequency of the target CPU. Given the measured runtime t_m, we define the SOL runtime t_sol as

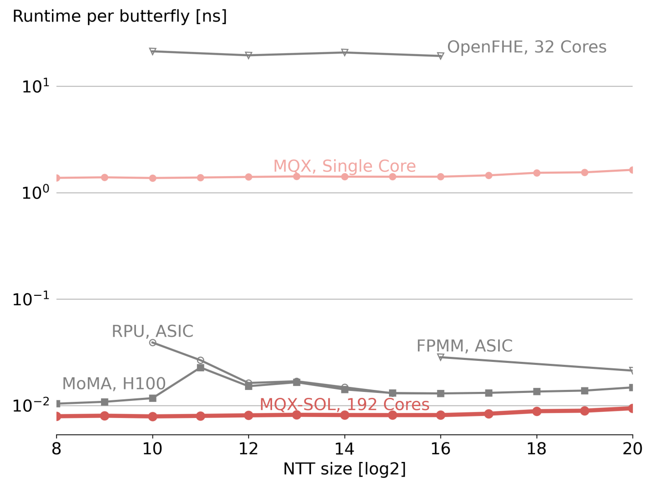
$$t_\text{sol} = t_\text{m} \cdot \frac{c_1}{c_2} \cdot \frac{f_\text{m}}{f_\text{max}}. \tag{13}$$

Since all our measurements are taken on a single core, the equation above simplifies to $t_\text{sol} = t_\text{m} \cdot f_\text{m}/(c_2 \cdot f_\text{max})$.

Intel Xeon. The highest-end Intel CPU we choose within the Xeon family is Intel Xeon 6980P, which has 128 cores, 504 MB L3 cache, with an all-core boost frequency of 3.2 GHz. As shown in Figure 7a, MQX-SOL is on average 1.3 times *faster* than RPU across



(a) Intel Xeon family.



(b) AMD EPYC family.

Figure 7: Speed-of-light performance of NTT on multi-core CPUs.

the NTT sizes supported by RPU, outperforming it at sizes from 1,024 to 8,192. We also include another ASIC baseline, FPMM [53]; averaging across the two NTT sizes that FPMM supports, MQX-SOL delivers approximately the same performance as FPMM. Compared to the GPU baseline, MoMA, MQX-SOL is 1.4 times slower on average.

AMD EPYC. We choose AMD EPYC 9965S as the highest-end CPU for our speed-of-light analysis. AMD EPYC 9965S has 192 cores with an all-core boost frequency of 3.35 GHz, and 384 MB of L3 cache. Modeled on AMD EPYC 9965S, MQX-SOL achieves a 2.5 times *speedup* over RPU on average across all RPU-supported NTT sizes and 2.9 times *speedup* over FPMM. Compared to the GPU baseline MoMA, MQX-SOL achieves 1.7 times speedup on average.

Towards realizing SOL performance. We acknowledge that the MQX-SOL runtimes shown in the figures represent idealized projections and may not be fully achievable in practice, as near-linear scaling across all cores requires significant effort, including advanced thread management and low-level system optimization. The SOL estimates intend to provide an upper bound when scaling our single-core implementation to an entire top-tier server-class CPU. Although optimistic, this model helps illustrate what could be possible under ideal conditions with careful system design and sufficient engineering effort. In practice, we can leverage the fact that real FHE workloads often batch NTTs and BLAS operations without data dependencies [1, 51], enabling substantial parallelism.

Nonetheless, the SOL analysis offers valuable insight into the performance potential of MQX. For example, our projected SOL performance on AMD EPYC 9965S outperforms RPU by 2.5 times. This suggests that, if we achieve a 77 times speedup by mapping our single-core implementation to batch NTTs on a 192-core machine, our performance would be on par with ASICs. Under a more conservative assumption of a 48 times speedup when mapping to a 192-core machine, our approach would be only about 1.6 times slower than RPU, demonstrating that near-ASIC performance is attainable on general-purpose CPUs.

It is also worth noting that ASIC and GPU measurements typically exclude data transfer time between the host CPU and the accelerator, whereas our CPU-based measurements include all data movement. This further strengthens our case: If the entire computation is hosted on a single CPU, the overhead of moving data between the host and the accelerator can be eliminated altogether.

7 Discussion

Automatic mapping to multi-core CPUs. In this work, we develop highly optimized implementations of cryptographic kernels for a single CPU core. One practical next step is to integrate the techniques proposed in this work with a compiler or code generation framework to automate the process of hardware-specific optimization and mapping to multiple CPU cores. This could be achieved by abstracting our hand-written implementation into an intermediate representation and integrating with existing compilation passes. For example, the high-performance code generator SPIRAL [14] has been used to generate optimized SIMD implementations of the fast Fourier transform, targeting instruction sets such as LRBni and AVX [15, 16, 30]. Moreover, SPIRAL already supports code generation for NTT and BLAS operations [49–51], making it a strong candidate for completing the last mile toward achieving SOL performance.

Generalizing to larger bit-widths. Another advantage of integrating this work with a compiler or code generation framework is the ability to generalize our optimized hand-written code to support higher bit-widths. The GPU baseline we compare against, MoMA, introduces a mathematical formalization of multi-word modular arithmetic and implements a rewrite system that can be combined with our approach. Leveraging MoMA, we can further extend our work to support higher input bit-widths, enabling its use in other cryptographic applications such as zero-knowledge proofs (ZKPs). This approach would offer a more performant alternative to general-purpose arbitrary-precision libraries like GMP for many cryptographic workloads on CPUs.

Implementing MQX. In this work, we estimate the potential performance of MQX and suggest possible implementation strategies (e.g., widening multiplication as a single instruction or as a sequence of multiply-low and multiply-high), leaving the implementation details open for future exploration. That said, we have shown that very similar instructions were proposed and implemented by Intel in the past, and our design follows Intel's established methodology for introducing ISA sub-families (e.g., AVX-512 VNNI for neural networks and AVX-512 GFNI for Galois fields). By identifying a minimal subset of the existing and legacy instructions and adapting them to a new, compelling domain—cryptography for privacy-preserving machine learning and artificial intelligence—we illustrate the performance potential that could be unlocked if a hardware vendor decides to implement MQX (or a simplified version of MQX, as demonstrated in Section 5.5).

8 Related Work

In this section, we first review prior work that utilizes specialized hardware and GPU for NTT acceleration, followed by a review of current solutions on CPUs.

Specialized hardware and GPU for NTT acceleration. Due to the importance of NTT in FHE-based workloads, many specialized hardware accelerators have been developed for NTTs [11, 24, 35, 36, 42, 45, 53]. However, few of these works target 128-bit NTTs. Among the designs that support 128-bit integer arithmetic, the performance gap between the CPU baseline and ASIC is substantial. For example, RPU [42] achieves a speedup of 545 to 1,485 times compared to the CPU baseline implemented using OpenFHE on a 32-core machine. Our work, even without MQX, reduces this gap to as low as 138 times on a single CPU core. This makes our work a stronger CPU baseline for future ASIC implementations.

There is also a significant body of work on GPU acceleration for NTTs [12, 26, 28, 29, 32, 33, 40, 44, 46, 47, 51]. These works primarily focus on bit-widths that are natively supported by GPUs, due to the lack of performant multi-precision libraries for GPUs. A recent work, MoMA [51], proposes and implements a rewrite system that can take arbitrary input bit-widths and decompose them into machine-supported arithmetic, achieving near-ASIC performance on commodity GPUs. One potential future direction is to integrate our CPU optimizations into MoMA and incorporate them into compilers or code generators to target higher input bit-widths. Similar to our MQX proposal, Shivdikar et al. [39] propose coarse-grained modular instructions for accelerating FHE on AMD compute DNA (CDNA) GPUs, embedding full modular operations (e.g., modular multiplication and reduction) into single high-level vector instructions. In contrast, MQX introduces fine-grained SIMD extensions for x86 CPUs, with each instruction closely resembling existing or legacy SIMD patterns to support multi-word modular arithmetic.

Current solutions on CPU. Current solutions on CPUs are primarily FHE libraries such as OpenFHE [1], SEAL [7], HEXL [6], HElib [19], and TFHE [9]. Most of these libraries are highly generalizable, supporting multiple FHE schemes. Notably, Intel's HEXL [6] utilizes AVX-512 to accelerate FHE-based workloads but only for 64-bit arithmetic. A closely related work by van der Hoeven and Lecerf [43] also uses SIMD instructions such as AVX2 and AVX-512 to accelerate NTT on CPUs, but their approach relies on modular floating-point arithmetic and a specialized prime to accelerate computations, while our method works with general primes.

Similar to most GPU-based approaches, the majority of CPU-based solutions support only 32-bit or 64-bit arithmetic and rely on RNS to break down large integers into these natively supported bit-widths. OpenFHE provides a built-in mathematical library for 128-bit integer arithmetic; however, its implementation is 32 times slower than our AVX-512 implementation on Intel Xeon. Another popular approach to implement NTT with large input bit-widths is to use arbitrary-precision libraries like GMP [18]. Libraries such as a library for doing number theory (NTL) [41] and fast library for number theory (FLINT) [21] also support arbitrary-precision arithmetic, but rely on GMP as the backend. In Section 5.4, we show that our AVX-512-based NTT outperforms the GMP baseline by 53 times on Intel Xeon.

9 Conclusion

To answer the question of whether near-specialized hardware performance can be achieved on CPUs for cryptographic kernels, our work begins by using scalar and SIMD instructions to accelerate NTTs and BLAS operations, resulting in 38 times and 62 times speedup over state-of-the-art CPU baselines, respectively. We further enhance performance by proposing MQX, an ISA extension that addresses the performance bottlenecks of AVX-512 for large-integer arithmetic. MQX introduces only three new SIMD instructions, each grounded in similar instructions that were previously explored and implemented by Intel. Using a roofline analysis, we demonstrate that by optimally scaling the single-core performance of MQX across all cores of a server-grade CPU, we can achieve near-ASIC performance for all tested NTT sizes. This finding suggests that, with continued software optimizations and minimal changes to the hardware ISA, CPUs have the potential to match the performance of GPUs and ASICs, making them a viable alternative for cryptographic workloads.

Acknowledgments

The authors thank Eric Tang for many helpful discussions throughout the course of this work. 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 Numbers DE-FOA-0002460 and DE-SC0025645, 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 FASTER and Launch at Texas A&M University through allocation CIS230287 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 includes the source code for our highly optimized implementation of BLAS operations and NTTs, targeting CPUs with AVX2 and AVX-512 support. Compilation on Intel CPUs uses the Intel oneAPI DPC++/C++ Compiler (ICX), while compilation on AMD CPUs uses the AMD Optimizing C/C++ and Fortran Compiler (AOCC), which employs Clang as its frontend.

A.2 Artifact Checklist (Meta-Information)

- **Compilation:** ICX ≥ 2024.2.0, AOCC ≥ 4.1.0, and Clang ≥ 16.0.3.
- **Dataset:** The required dataset is included with this artifact.
- **Hardware:** Intel and AMD CPUs as detailed in Table 4.
- **Runtime state:** Users should ensure that no other processes are competing for system resources during benchmarking.
- **Execution:** We provide a detailed README file within the artifact, and users only need to run the provided Bash scripts to perform benchmarking.
- **Metrics:** Execution time.
- **Output:** Performance measurements will be displayed in the terminal window.
- **How much disk space is required (approximately)?:** Around 6 MB.
- **How much time is needed to complete experiments (approximately)?:** Less than 1 hour.

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 repository on GitHub[2]. The artifact requires approximately 6 MB of disk space.

Hardware dependencies. To reproduce the exact results, Intel Xeon 8352Y and AMD EPYC 9654 are required. Otherwise, the hardware should at least support AVX2 and AVX-512 instructions.

Software dependencies. For Intel CPUs, the artifact requires ICX version 2024.2.0 or later for compilation. For AMD CPUs, it requires AOCC version 4.1.0 or later, with AMD Clang version 16.0.3 or later.

A.4 Installation

Instructions for installing dependencies are provided in the README file.

A.5 Experiment Workflow

A detailed README file is provided in the root directory of the artifact. For example, to reproduce Figure 4b, run the following command on an AMD CPU:

```
$ bash ./benchmark/blas_aocc.sh
```

To reproduce Figure 5a, run the following command on an Intel CPU:

```
$ bash ./benchmark/ntt_icx.sh
```

[2]https://github.com/naifeng/benchntt

A.6 Evaluation and Expected Results

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

A.7 Experiment Customization

Users can customize the parameters in Equation 13 to match their specific CPUs for more accurate roofline analysis. Detailed instructions are provided in the README file.

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. https://doi.org/10.1145/3560827.3563379

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

[3] 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. https://doi.org/10.1007/3-540-47721-7_24

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[5] 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. https://doi.org/10.1145/567806.567807

[6] 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. https://doi.org/10.1145/3474366.3486926

[7] 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. https://doi.org/10.1007/978-3-319-70278-0_1

[8] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International conference on the theory and application of cryptology and information security*. Springer, 409–437.

[9] 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. https://doi.org/10.1007/s00145-019-09319-x

[10] George Chrysos. 2014. Intel® xeon phi™ coprocessor-the architecture. *Intel Whitepaper* 176, 2014 (2014), 43–50.

[11] Alhad Daftardar, Brandon Reagen, and Siddharth Garg. 2024. SZKP: A scalable accelerator architecture for zero-knowledge proofs. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*. 271–283.

[12] Sultan Durrani, Muhammad Saad Chughtai, Mert Hidayetoglu, Rashid Tahir, Abdul Dakkak, Lawrence Rauchwerger, Fareed Zaffar, and Wen-mei 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. https://doi.org/10.1109/PACT52795.2021.00032

[13] 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. https://doi.org/10.1109/HPCA56546.2023.10071017

[14] 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. https://doi.org/10.1109/JPROC.2018.2873289

[15] Franz Franchetti, Markus Puschel, Yevgen Voronenko, Srinivas Chellappa, and José MF Moura. 2009. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine* 26, 6 (2009), 90–102.

[16] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. 2006. FFT program generation for shared memory: SMP and multicore. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 115–es.

[17] 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)*.

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

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

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

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

[22] Intel. 2023. Intel® 64 and ia-32 architectures optimization reference manual. 1 (2023), 1–857.

[23] Intel Corporation. 2013. *Intel Intrinsics Guide v3.1.* https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html Version 3.1.

[24] Siddharth Jayashankar, Edward Chen, Tom Tang, Wenting Zheng, and Dimitrios Skarlatos. 2024. Cinnamon: A Framework For Scale-Out Encrypted AI. (2024).

[25] 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.

[26] 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. https://doi.org/10.1109/IISWC50251.2020.00033

[27] Seungmin Lee, Youngsok Kim, Dukyun Nam, and Jong Kim. 2024. Gem5-AVX: Extension of the Gem5 Simulator to Support AVX Instruction Sets. *IEEE Access* 12 (2024), 20767–20778.

[28] 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. https://doi.org/10.1109/MM.2023.3253052

[29] 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. https://doi.org/10.1007/978-3-319-48965-0_8

[30] Daniel S McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. 2011. Automatic SIMD vectorization of fast Fourier transforms for the Larrabee and AVX instruction sets. In *Proceedings of the international conference on Supercomputing*. 265–274.

[31] Mohammed Nabeel, Deepraj Soni, Mohammed Ashraf, Mizan Abraha Gebremichael, Homer Gamil, Eduardo Chielle, Ramesh Karri, Mihai Sanduleanu, and Michail Maniatakos. 2023. CoFHEE: A co-processor for fully homomorphic encryption execution. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–2.

[32] Ali Şah Özcan, Can Ayduman, Enes Recep Türkoğlu, and Erkay Savaş. 2023. Homomorphic encryption on GPU. *IEEE Access* 11 (2023), 84168–84186. https://doi.org/10.1109/ACCESS.2023.3265583

[33] 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. https://doi.org/10.1007/s11227-021-03980-5

[34] Marshall C Pease. 1968. An adaptation of the fast Fourier transform for parallel processing. *Journal of the ACM (JACM)* 15, 2 (1968), 252–264.

[35] 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. https://doi.org/10.1145/3466752.3480070

[36] 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. https://doi.org/10.1145/3470496.3527393

[37] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news* 41, 3 (2013), 475–486.

[38] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics (TOG)* 27, 3 (2008), 1–15.

[39] Kaustubh Shivdikar, Yuhui Bao, Rashmi Agrawal, Michael Shen, Gilbert Jonatan, Evelio Mora, Alexander Ingare, Neal Livesay, José L Abellán, John Kim, et al. 2023. Gme: Gpu-based microarchitectural extensions to accelerate homomorphic encryption. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 670–684.

[40] 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. https://doi.org/10.1109/SEED55351.2022.00013

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

[42] 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. https://doi.org/10.1109/ISPASS57527.2023.00034

[43] Joris van der Hoeven and Grégoire Lecerf. 2024. Implementing number theoretic transforms. (2024).

[44] 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. https://doi.org/10.1007/978-3-031-17143-7_25

[45] 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. https://doi.org/10.1109/ICCAD57390.2023.10323744

[46] 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. https://doi.org/10.1109/TPDS.2022.3228628

[47] 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. https://doi.org/10.1109/ICCD58817.2023.00061

[48] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[49] 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. https://doi.org/10.1109/HPEC58863.2023.10363559

[50] 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)*.

[51] 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*. 476–492.

[52] Sean Zhang. 2022. Extending gem5 to Support AVX Instructions. https://seanzw.github.io/posts/gem5-avx/.

[53] 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). https://doi.org/10.1109/TCAD.2024.3410847