NTT software optimization using an extended Harvey butterfly

Jonathan Bradbury[1], Nir Drucker[2] and Marius Hillenbrand[3]

[1] IBM Systems - Poughkeepsie
[2] IBM Research - Haifa
[3] IBM Research and Development Germany

Abstract. Software implementations of the number-theoretic transform (NTT) method often leverage Harvey's butterfly to gain speedups. This is the case in cryptographic libraries such as IBM's HElib, Microsoft's SEAL, and Intel's HEXL, which provide optimized implementations of fully homomorphic encryption schemes or their primitives.
We extend the Harvey butterfly to the radix-4 case for primes in the range $[2^{31}, 2^{52})$. This enables us to use the vector multiply sum logical (VMSL) instruction, which is available on recent IBM Z® platforms. On an IBM z14 system, our implementation performs more than 2.5× faster than the scalar implementation of SEAL we converted to native C. In addition, we implemented a mixed-radix implementation that uses AVX512-IFMA on Intel's Ice Lake processor, which happens to be ~ 1.1 times faster than the super-optimized implementation of Intel's HEXL. Finally, we compare the performance of our implementation using GCC versus Clang compilers and discuss the results.

Keywords: Number Theoretic Transform · Software optimization · IBM Z® platforms · Homomorphic Encryption · AVX512-IFMA

1 Introduction

Homomorphic encryption (HE) solutions and general privacy-preserving solutions are rapidly being adopted by large organizations. According to Gartner [Gar21]: "by 2025, 50% of large organizations will adopt privacy-enhancing computation for processing data in untrusted environments or multiparty data analytics use cases.". One of the barriers to adopting these solutions is the high cost of running private computation in terms of latency, memory usage, and bandwidth. Thus, improving the latency of HE solutions is of great interest. This is currently being done using dedicated hardware as in [DS16, SLPD20], optimized software algorithms as in [CHK+19, Har14], or by leveraging specific hardware capabilities in software such as in the homomorphic encryption for arithmetic of approximate numbers (HEAAN) library [JLK+20] and the Intel homomorphic encryption acceleration library (HEXL) [BKS+21].

The most time-consuming primitive required to evaluate a homomorphically encrypted circuit is the key-switching operation, which relies heavily on number-theoretic transform (NTT) computations for speeding up the required polynomial multiplications. For this reason, NTT is often the main target for optimization in research that focuses on FHE performance (e.g., [JLK+20, SLPD20]). In fact, the list of works that deal with NTT optimizations in hardware and software for different rings and moduli sizes is too long for us to include here. Thus, we refer the reader to [LN16, Har14, BKS+21], which will serve as our baseline for comparison.

The FHE cryptosystems we aim to optimize (e.g., CKKS [CHK+19]) operate over the ring of polynomials $\mathbb{F}_q[X]/(X^N+1)$, where N is a power of two. The performance of the NTT algorithm in this ring depends on the degree N, the modulo q, and the performance of the chosen NTT butterfly. The number of iterations in a radix-k NTT is $\log_k N$, where at every iteration the CPU consumes at least the N polynomial coefficients and N powers of ω, a primitive $2N$th root of unity. For example, when $2^{32} < q < 2^{64}$, every coefficient is often resides in an 8-byte word, and $2N$ coefficients reside in 512KiB. On modern processors with L2 caches that are smaller than 512KiB, this increases the number of slow load and store operations. In contrast, the L2d-cache of our IBM z14 is 4MiB large. In many cases, this allows all the NTT operations to be performed directly from the L2d cache. Here, the bottleneck of the algorithm becomes the serialization of the algorithm, particularly the performance of the butterfly algorithm.

It is possible to balance the register and memory pressure by choosing the correct NTT radix. For example, [BH21] uses radix-32 for IBM Z systems, and [JLK+20] uses radix-16 for x86-64 CPUs and radix-32 for GPUs. Even HEXL recently added a radix-4 scalar implementation that relies on radix-2 butterflies. However, other parameters may affect the algorithm run-time, including the number of multipliers, their operand sizes, and the instruction set architecture (ISA). For example, the AVX512-IFMA available on x86-64 CPUs allows implementors to receive the lower/higher part of a 104-bit product in a single instruction. Another example is the `VMSL` instruction available on IBM Z systems, which returns the sum of two 112-bit products in a single instruction. In this paper, we suggest new radix-4 butterflies and discuss the trade-offs they offer on different systems with different ISAs.

Our contribution. We present new radix-4 butterflies for the forward and inverse NTT, which allow us to leverage hardware capabilities such as the `VMSL` instruction on IBM Z platforms. We apply and measure these implementations on the IBM Z and x86-64 platforms. Our measurements show a speedup of more than 2.5× on IBM Z systems compared to the NTT code for SEAL [Lai17], which we converted from C++ to C, and ~ 1.1 times speedup on x86-64 platforms compared with the code for HEXL [BKS+21], which we also converted from C++ to C. Our measurements also show a 20% reduction in speed when we compiled the scalar code with GCC as compared to compiling it with Clang. For completeness, we analyze the compiled binaries and suggest an explanation for this slowdown. We made our code available as an open-source sample under the Apache-2.0 license and uploaded it to GitHub [1]

Organization. The document is organized as follows. Section 2 provides some background and describes our notation. Section 3 presents our extended Harvey butterflies. We discuss some optimizations that rely on mixed radices NTT algorithms in Section 4. In Sections 5 and 6, we describe specific optimizations for IBM Z and x86-64 systems, respectively. We report our experiments and results in Section 7. In Section 8, we analyze the GCC slowdown, and we conclude in Section 9.

2 Background and notation

Let $\mathbb{F}_q$ be a finite field of characteristic q with residue classes represented as elements from $Z \cap [0, q)$. The elements in the polynomial quotient ring $\mathcal{R}_q = \mathbb{F}_q[X]/(X^N+1)$ are polynomials of a degree at most $N-1$ with integer coefficients in $\mathbb{F}_q$, where $q \equiv 1 \pmod{2N}$ and N is a power of two. We may refer to a polynomial $a = \sum a_i x^i$ by its coefficients i.e., $a = (a_0, \ldots, a_{N-1})$. We denote coefficient-wise multiplication (hereafter

[1] https://github.com/IBM/optimized-number-theoretic-transform-implementations

element-wise multiplication) of two polynomials a, b by $a \odot b$. For a specific platform, we denote its word-size with β. For example, for typical CPUs $\beta = 2^{32}$ or $\beta = 2^{64}$. For two unsigned integers $0 < a, b < \beta$, the functions $l(ab) = ab \pmod{\beta}$ and $h(ab) = \left\lfloor \frac{ab}{\beta} \right\rfloor$ return the lower and higher parts of the product ab, respectively. The operator $\ll a$ is the left shift operator by a bits and by $a \mid b$ we mean that a divides b.

2.1 NTT

The NTT algorithm is a variant of the fast Fourier transform (FFT) algorithm over $\mathcal{R}_q$. It receives a polynomial $a = (a_0, \ldots, a_{N-1}) \in \mathcal{R}_q$ and a fixed N's primitive root of unity ω as inputs and it outputs $\tilde{a} = (\tilde{a}_0, \ldots, \tilde{a}_{N-1}) \in \mathcal{R}_q$, where $\tilde{a}_i = \sum_{j=0}^{N-1} a_j \omega^{ij}$. The inverse function $a = InvNTT_\omega(\tilde{a})$ is given by $a_i = \frac{1}{N} \sum_{j=0}^{N-1} \tilde{a}_j \omega^{-ij}$.

Polynomial multiplication in $\mathcal{R}_q$. For some primitive $2N$th root of unity $\psi \in \mathbb{F}_q$, $a, b \in \mathcal{R}_q$, $\bar{a} = (a_0, a_1\psi, \ldots, a_{N-1}\psi^{N-1})$ and $\bar{b} = (b_0, b_1\psi, \ldots, b_{N-1}\psi^{N-1})$ it holds that

$$ab = (1, \psi^{-1}, \ldots, \psi^{-(N-1)}) \odot InvNTT\left(NTT\left(\bar{a}\right) \odot NTT\left(\bar{b}\right)\right) \tag{1}$$

Equation (1) allows performing polynomial multiplication in $\mathcal{R}_q$ using $O(N \log N)$ modular multiplications over F_q instead of $O(N^2)$.

Shoup modular multiplication. The most time-consuming primitive in NTT algorithms is modular multiplication between the coefficients of a and the fixed (precomputed) powers of ω. Theorem 1 describes a modular multiplication method attributed to Victor Shoup [Har14] and implemented in NTL [2], a high-performance library for number theory operations.

Theorem 1 (Shoup reduction [Har14] part of Theorem 1)**.** *Let $w \in \mathbb{F}_q$, $t < \beta$, $\omega' = \left\lfloor \frac{\omega\beta}{q} \right\rfloor$, and $r = \left\lfloor \frac{\omega' t}{\beta} \right\rfloor$. When $q < \frac{\beta}{2}$ it follows that*

$$0 \leq \omega t - rq < 2q < \beta$$

Proof. [Har14][Theorem 1] □

We denote this multiplication by $ShoupModMul(t, \omega, \omega', q) = \omega t - q \left\lfloor \frac{\omega' t}{\beta} \right\rfloor$ for a fixed global parameter β.

NTT algorithms and implementations. Appendix A presents one variant of the NTT and inverse-NTT (InvNTT) algorithms as specified in [LN16]. These algorithms are implemented in different libraries that provide HE optimizations such as Microsoft SEAL [Lai17] and Intel HEXL [BKS+21]. We use these algorithms as our baseline.

The main bottleneck of the NTT and InvNTT algorithms is the Cooley-Tukey (CT) [CT65] and Gentleman-Sande (GS) [GS66] butterflies, respectively. These are implemented in SEAL and HEXL using Harvey's butterflies [Har14], which we present in Algorithm 1.

[2] https://libntl.org/

Algorithm 1 Harvey's Butterflies [Har14]

Global parameters: A word-size β, a modulus $q < \frac{\beta}{4}$; $\omega \in \mathbb{F}_q$; $\omega' = \left\lfloor \frac{\omega\beta}{q} \right\rfloor < \beta$

Input: $0 \leq x, y < 4q$
Output: $x = x + \omega y$, $y = x - \omega y \pmod{4q}$
1: **procedure** HarveyFwdButterfly($x, y, \omega, \omega', q, \beta$)
2: **if** $x \geq 2q$ **then** $x = x - 2q$
3: $t = ShoupModMul(y, \omega, \omega', q)$
4: **return** $(x + t,\ x - t + 2q)$

Input: $0 \leq x, y < 2q$
Output: $x = x + y$, $y = \omega(x - y) \pmod{2q}$
1: **procedure** HarveyInvButterfly($x, y, \omega, \omega', q, \beta$)
2: $x' = x + y$
3: **if** $x' \geq 2q$ **then** $x' = x' - 2q$
4: $t = x - y + 2q$
5: $y' = ShoupModMul(t, \omega, \omega', q)$
6: **return** x', y'

2.2 Dedicated CPU instructions

The optimizations in this paper target the IBM Z and x86-64 single instruction multiple data (SIMD) architectures. Specifically, they are designed for the IBM `VMSL` instruction and the Intel AVX512-IFMA instruction set.

VMSL. The IBM z14 and z15 systems come together with the vector facility (VF) and vector-enhancements facility (VEF) 1/2 ISA [IBM21]; these include the `VMSL` instruction that we illustrate in Algorithm 2. This instruction operates on wide 128-bit registers V_2, V_3, V_4, and a 4-bit scalar m_6. It first multiplies two 56-bit unsigned integers residing in two 64-bit words using a "redundant representation" in V_2, V_3. Subsequently, it adds the two products together with the 128-bit word of V_4 in the destination register V_1. Examples for software optimizations that leverage the `VMSL` instruction include [YZD+20, GADO17, BH21].

Algorithm 2 V1 = `VMSL`(V2, V3, V4, m6)

Inputs: V_2,V_3,V_4 (128-bit wide registers), m_6 a bitset elements of 4 bits.
Outputs: V_1 (a 128-bit wide register)
1: **procedure** VMSL(V_2, 3, 4, 6)
2: $T_1 = V_2[00:056] * V_3[00:056] \ll m6[0]$
3: $T_2 = V_2[64:120] * V_3[64:120] \ll m6[1]$
4: $V_1 = T_1 + T_2 + V_4$

AVX512-IFMA. The latest Intel processors, such as the third generation Intel® Xeon® Scalable Processors (code-named Ice Lake or Tiger Lake) include the new AVX512-IFMA ISA extension [int21a]. This extension presents two new instructions, `VPMADD52LUQ` and `VPMADD52HUQ`, which we illustrate in Algorithm 3. These instructions operate on wide 512-bit registers V_1, V_2, V_3. The instructions first multiply two 52-bit unsigned integers residing in two 64-bit words using a "redundant representation" in V_2, V_3. Subsequently, the instructions accumulate the low (`VPMADD52LUQ`) or high (`VPMADD52HUQ`) halves of the 104-bit products using 64-bit accumulators in the destination register DST. Some examples for using `AVX512-IFMA` include [DG19, GK16, DG17, KG19, BKS+21].

Algorithm 3 DST = VPMADD52(V_1, V_2, V_3)

Inputs: V_1, V_2, V_3 (512-bit wide registers)
Outputs: DST (a 512-bit wide register)

procedure VPMADD52LUQ(V_1, V_2, V_3)
  for $j = 0$ to 7 **do**
    $i = 64j$
    $T[127:0] = V_2[i+51:i] * V_3[i+51:i]$
    $\text{DST}[i+63:i] = V_1[i+63:i] + T[\mathbf{51}:\mathbf{0}]$

procedure VPMADD52HUQ(V_1, V_2, V_3)
  for $j = 0$ to 7 **do**
    $i = 64j$
    $T[127:0] = V_2[i+51:i] * V_3[i+51:i]$
    $\text{DST}[i+63:i] = V_1[i+63:i] + T[\mathbf{103}:\mathbf{52}]$

3 Extended Harvey's butterfly

Radix-2 NTT algorithms often involve loading and storing the polynomial coefficients $\log_2(N)$ times. When N is large, this may lead to many cache misses. To this end, some implementations prefer to use higher radix values, such as a power of two $k > 2$, that reduces the number of iterations to $\log_k(N)$. For example, the implementations reported in [JLK+20] use radix-32 NTT for GPUs and radix-2 through radix-16 for CPUs. Another example in [BH21] provides a radix-32 NTT implementation. These implementations load sets of k values and perform $\log_2(k)$ radix-2 butterflies on every set.

3.1 Expanded radix-4 butterflies

We aim to leverage the `VMSL` instruction that can handle two multiplications in parallel. For that, we first expand the radix-4 butterflies in Algorithm 4 and then present an optimized version of them (Algorithms 5, 6).

Algorithm 4 Radix-4 Butterflies

Input: $x, y, z, t \in \mathbb{F}_q$, $w_i \in \mathbb{F}_q$ for $0 < i < 5$
Output: $x, y, z, t \in \mathbb{F}_q$
Remark: All operations are performed in $\mathbb{F}_q$.

procedure CT RADIX-4 BUTTERFLY(x, y, z, t, ω_i)
  $x' = x + z\omega_1 + (y\omega_2 + t\omega_4)$
  $y' = x + z\omega_1 - (y\omega_2 + t\omega_4)$
  $z' = x - z\omega_1 + (y\omega_3 + t\omega_5)$
  $t' = x - z\omega_1 - (y\omega_3 + t\omega_5)$
  return (x', y', z', t')

procedure GS RADIX-4 BUTTERFLY(x, y, z, t, ω_i)
  $x' = x + y + z + t$
  $y' = (x + y - z - t)w_1$
  $z' = (x - y)w_2 + (z - t)w_3$
  $t' = (x - y)w_4 - (z - t)w_5$
  return (x', y', z', t')

Lemma 1. *Let $x, y, z, t \in \mathbb{F}_q$ be the inputs to the radix-4 butterfly and let $\omega_1, \omega_2, \omega_3 \in \mathbb{F}_q$ be the relevant three powers of the primitive $2N$th root of unity ω. Let $\omega_4 = \omega_1\omega_2$, $\omega_5 = -\omega_1\omega_3$, then the first procedure of Algorithm 4 is the expanded version of the radix-2 CT butterfly for the radix-4 case.*

Proof. Denote by $(a', b') = CT_2(a, b, \omega_i)$ the radix-2 CT butterfly function. Then the output of the first radix-2 iteration is

$$\begin{aligned}(a^1, b^1) &= CT_2(x, z, \omega_1) = (x + \omega_1 z, x - \omega_1 z)\\ (a^2, b^2) &= CT_2(y, t, \omega_1) = (y + \omega_1 t, y - \omega_1 t)\end{aligned}$$

and of the second radix-2 iteration is

$$\begin{aligned}(x', y') = CT_2(a^1, a^2, \omega_2) &= (a^1 + \omega_2 a^2, a^1 - \omega_2 a^2)\\ &= (x + z\omega_1 + y\omega_2 + t\omega_2\omega_1, x + z\omega_1 - y\omega_2 - t\omega_2\omega_1)\\ &= (x + z\omega_1 + y\omega_2 + t\omega_4, x + z\omega_1 - y\omega_2 + t\omega_4)\\ (z', t') = CT_2(b^1, b^2, \omega_3) &= (b^1 + \omega_3 b^2, b^1 - \omega_3 b^2)\\ &= (x - z\omega_1 + y\omega_3 - t\omega_3\omega_1, x - z\omega_1 - y\omega_3 + t\omega_3\omega_1)\\ &= (x - z\omega_1 + y\omega_3 + t\omega_5, x - z\omega_1 - y\omega_3 - t\omega_5)\end{aligned}$$

□

Lemma 2. *Let $x, y, z, t \in \mathbb{F}_q$ be the inputs to the radix-4 butterfly and let $\omega_1, \omega_2, \omega_3 \in \mathbb{F}_q$ be the relevant three powers of the primitive $2N$th root of unity ω. Let $\omega_4 = \omega_1\omega_2$, $\omega_5 = -\omega_1\omega_3$, then the second procedure of Algorithm 4 is the expanded version of the radix-2 GS butterfly for the radix-4 case.*

Proof. Denote by $(a', b') = GS_2(a, b, \omega_i)$ the radix-2 GS butterfly function. Then the output of the first radix-2 iteration is

$$\begin{aligned}(a^1, b^1) &= GS_2(x, y, \omega_2) = (x + y, x\omega_2 - y\omega_2)\\ (a^2, b^2) &= GS_2(z, t, \omega_3) = (z + t, z\omega_3 - t\omega_3)\end{aligned}$$

and of the second radix-2 iteration is

$$\begin{aligned}(x', y') = GS_2(a^1, a^2, \omega_1) &= (a^1 + a^2, a^1\omega_1 - a^2\omega_1)\\ &= (x + y + z + t, x\omega_1 + y\omega_1 - z\omega_1 - t\omega_1)\\ &= \big(x + y + z + t, (x + y - z - t)\omega_1\big)\\ (z', t') = GS_2(b^1, b^2, \omega_1) &= (b^1 + b^2, b^1\omega_1 - b^2\omega_1)\\ &= (x\omega_2 - y\omega_2 + z\omega_3 + t\omega_3, x\omega_2\omega_1 - y\omega_2\omega_1 - z\omega_3\omega_1 + t\omega_3\omega_1)\\ &= ((x - y)\omega_2 + (z - t)\omega_3, (x - y)\omega_4 - (z - t)\omega_5)\end{aligned}$$

□

3.2 Optimized butterflies

We optimize the expanded butterflies by first extending Shoup's modular multiplication to receive two variables or more as input. Theorem 2 extends Theorem 1 for the sum of k parallel multiplications.

Theorem 2 (An extension of [Har14] Theorem 1)**.** *For some integers $k, \alpha > 0$ and an integer $0 < i < k$, let $w_i \in \mathbb{Z}_q$, $0 \le t_i < \alpha q$, $w_i' = \left\lfloor \frac{w_i \beta}{q} \right\rfloor$, and $r = \left\lfloor \frac{\sum_i^k w_i' t_i}{\beta} \right\rfloor$. When $q < \frac{\beta}{k\alpha}$ it follows that*

$$0 \le \left(\sum_i^k w_i t_i \right) - rq < 2q$$

Proof. From the definition of w'_i, and r we have

$$0 \leq \frac{w_i\beta}{q} - w'_i < 1, \qquad\qquad 0 \leq \frac{\sum_i^k w'_i t_i}{\beta} - r < 1$$

We multiply these equations by $\frac{qt_i}{\beta}$ and q, respectively, and get

$$0 \leq w_i t_i - \frac{w'_i t_i}{\beta} q < \frac{qt_i}{\beta}$$
$$0 \leq \frac{\sum_i^k w'_i t_i}{\beta} q - rq < q$$

Summing them together yields

$$0 \leq \left(\sum_i^k w_i t_i\right) - qr < \frac{q}{\beta}\left(\sum_i^k t_i\right) + q < \frac{k\alpha q^2}{\beta} + q < 2q < \beta$$

where the last equation holds when $\frac{k\alpha q^2}{\beta} < q$, i.e., $q < \frac{\beta}{k\alpha}$ □

Corollary 1. *For $0 \leq t_i < \beta$ it follows that $0 \leq w_1 t_1 + w_2 t_2 - rq < (k+1)q$*

We denote this multiplication for a fixed global parameter β, and the fixed values $\alpha = 8$ and $k = 2$ by

$$ExtendedModMul(t_1, t_2, \omega_1, \omega'_1, \omega_2, \omega'_2, q) = \omega_1 t_1 + \omega_2 t_2 - q\left\lfloor \frac{\omega'_1 t_1 + \omega'_2 t_2}{\beta} \right\rfloor.$$

Algorithms 5, 6 present an optimized version of the expanded radix-4 butterflies. In a sense, we followed Harvey's [Har14] method for butterfly optimization, but we do it for the radix-4 expanded cases.

Algorithm 5 CT radix-4 optimized butterfly

Input: $0 \leq x, y, z, t < 8q$, $w_i \in \mathbb{F}_q$, $w'_i = \left\lfloor \frac{w_i\beta}{q} \right\rfloor < \beta$ for $0 < i < 5$
Output: $0 \leq x', y', z', t' < 8q$
1: **procedure** CT RADIX-4 OPTIMIZED BUTTERFLY($x, y, z, t, \omega_i, \omega'_i$)
2: **if** $x > 4q$ **then** $x = x - 4q$
3: $z = ShoupModMul(z, \omega_1, \omega'_1, q)$
4: $a = ExtendedModMul(y, t, \omega_2, \omega'_2, \omega_4, \omega'_4, q)$
5: $b = ExtendedModMul(y, t, \omega_3, \omega'_3, \omega_5, \omega'_5, q)$
6: $x' = x + z + a$
7: $y' = 2q + x + z - a$
8: $z' = 2q + x - z + b$
9: $t' = 4q + x - z - b$
10: **return** (x', y', z', t')

Theorem 3. *For $q < \frac{\beta}{16}$, Algorithm 5 is correct and is equivalent to Algorithm 4 first procedure except that the inputs and outputs are reduced modulo $8q$ instead of q.*

Proof. It is easy to observe that Lines $6-8$ in Algorithm 5 are equivalent to lines 2-5 in the first procedure of Algorithm 4 up to some multiples of q. Therefore, it is only left to show that the outputs are reduced modulo $8q$. In Line 6 of Algorithm 5, $x < 4q$ by Line 2,

$z < 2q$ by Theorem 1, and $a, b < 2q$ by Theorem 2, where $y, t < 8q$ and $q < \frac{\beta}{16}$. It follows that

$$\begin{aligned} 0 &< x + z + a < 8q \\ 0 &< 2q + x + z - a < 8q \\ 0 &< 2q + x - z + b < 8q \\ 0 &< 4q + x - z - b < 8q \end{aligned}$$

□

Algorithm 6 GS radix-4 optimized butterfly

Input: $0 \leq x, y, z, t < 2q$, $w_i \in \mathbb{F}_q$, $w'_i = \left\lfloor \frac{w_i \beta}{q} \right\rfloor < \beta$ for $0 < i < 5$
Output: $0 \leq x', y', z', t' < 8q$

1: **procedure** GS RADIX-4 OPTIMIZED BUTTERFLY($x, y, z, t, \omega_i, \omega'_i$)
2: $a = z + t$
3: $b = x + y$
4: $z' = ShoupModMul(4q + b - a, \omega_1, \omega'_1, q)$
5: $d_1 = 2q + x - y$
6: $d_2 = 2q + z - t$
7: $y' = ExtendedModMul(d_1, d_2, \omega_2, \omega'_2, \omega_3, \omega'_3, q)$
8: $t' = ExtendedModMul(d_1, d_2, \omega_4, \omega'_4, \omega_5, \omega'_5, q)$
9: $x' = a + b$
10: **if** $x' > 4q$ **then** $x' = x' - 4q$
11: **if** $x' > 2q$ **then** $x' = x' - 2q$
12: **return** (x', y', z', t')

Theorem 4. *For $q < \frac{\beta}{8}$, Algorithm 6 is correct and is equivalent to the second procedure of Algorithm 4 except that the inputs and outputs are reduced modulo $2q$ instead of q.*

Proof. First, we show equivalence between the two algorithms. By Lines 2,3,9-11:

$$x' = a + b = z + t + x + y \pmod{2q}$$

By Lines 5-7:

$$\begin{aligned} y' &= d_1\omega_2 + d_2\omega_3 \pmod{2q} = \\ &= (2q + x - y)\omega_2 + (2q + z - t)\omega_3 \pmod{2q} \\ &\equiv (x - y)\omega_2 + (z - t)\omega_3 \pmod{2q} \end{aligned}$$

By Line 4:

$$z' = (2q + b - a)\omega_1 \pmod{2q} = (x + y - z - t)\omega_1 \pmod{2q}$$

Finally, by Lines 6-8

$$\begin{aligned} t' &= d_1\omega_4 + d_2\omega_5 \pmod{2q} \equiv \\ &= (2q + x - y)\omega_4 + (2q + z - t)\omega_5 \pmod{2q} \\ &= (x - y)\omega_4 + (z - t)\omega_5 \pmod{2q} \end{aligned}$$

It is left to show that the outputs are always reduced modulo $2q$ and that the inputs to the modular multiplication functions are of valid sizes. We know that $0 < x, y, z, t < 2q$. Then, at Line 4, the input to the $ShoupModMul$ function is $0 < 4q + b - a = 4q + x + y - z - t < 8q$, which is a valid input when $8q < \beta$. In addition, $0 < d1 = 2q + x - y < 4q$ and $0 < d2 = 2q + z - t < 4q$; therefore, according to Theorem 2, these are valid inputs to $ExtendedModMul$ when $q < \frac{\beta}{8}$. Finally, according to Theorems 1, 2 the outputs of the modular multiplications are always smaller than $2q$ and only x' requires additional reduction at Lines 10-11. □

3.3 Butterfly characteristics

Figures 1 and 2 provide schematic illustrations of a CT radix-4 butterfly using four radix-2 butterflies or using our proposed butterfly, respectively. For brevity, we analyze the forward butterfly, the inverse butterfly analysis follows. We highlight branches with green, and the lower $l()$ and upper $h()$ halves of modular multiplication products in yellow and orange, respectively; logical operations appear in white. These illustrations assume that the architecture only supports binary operations. Figure 1 (resp. 2) involves 4 (1) branches, 8 (8) calls to $l()$, 4 (5) calls to $h()$, and 16 (16) other binary operations. This means we reduce the number of branches by 75% at the cost of slightly more multiplications. One reason is that we designed our butterfly to receive inputs reduced by modulo $4q$ instead of $2q$ as before.

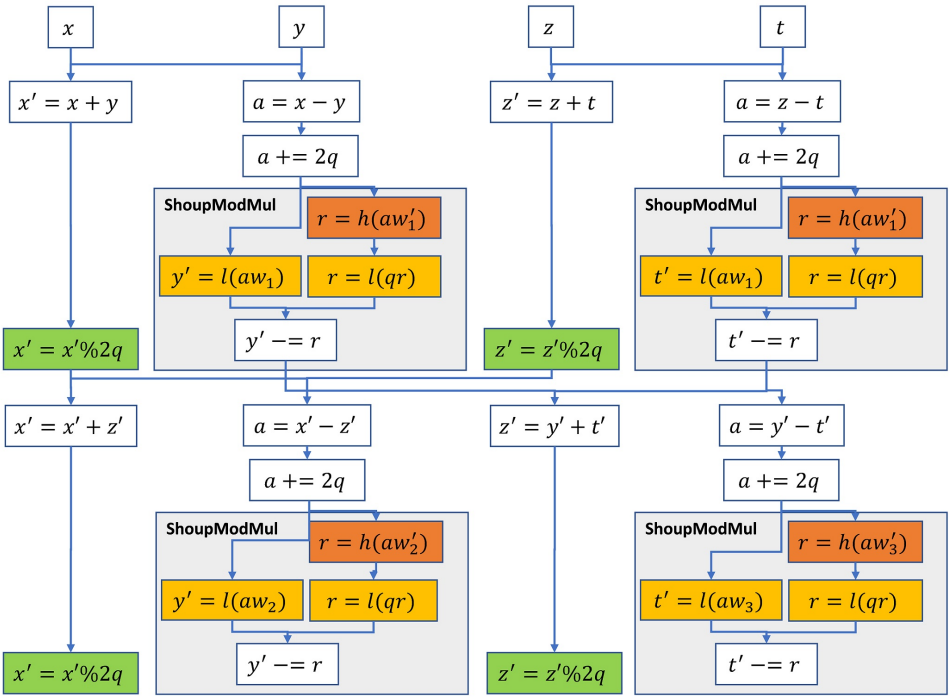


Figure 1: An illustration of a sequence of (binary) operations in a radix-4 butterfly that uses four radix-2 butterflies. The values $q, 2q, w_i, w_i'$ were loaded in advance.

The most important advantage of our butterfly is its 6-level depth. This is almost half the depth of the radix-2 based circuit, although radix-2 allows better pipelining when it is supported by the architecture. In contrast, the disadvantage of our butterfly is that it requires loading five powers of ω instead of only three in the radix-2 case. In practice, the above number doubles itself because we use Shoup's modular multiplication.

Consider two extreme cases: the first and last iteration of the NTT algorithm. The first iteration uses the same powers of ω for all butterflies, whereas the last iteration uses a different set of powers per butterfly. At the first iteration, we can fix 10 (resp. 6) registers to store the powers of ω and an extra 4 (resp. 3) to store $q, 2q, 4q$, and zero (resp $q, 2q$, and zero). The rest of the registers are available for the computations. In contrast, the last iteration needs to account for the load operations of the different powers of ω. For example, if an architecture includes 32 registers. Then, we can fix 14 (resp. 9) of them and use the other 18 (resp. 23) for the butterfly computations. In summary, our butterfly suggests a nontrivial trade-off between parallelization and register pressure.

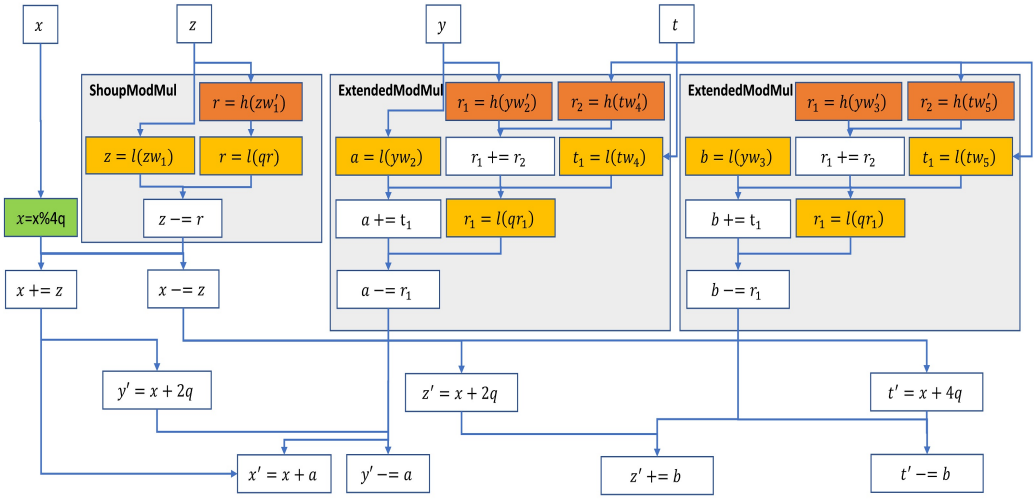


Figure 2: An illustration of a sequence of (binary) operations in our proposed radix-4 butterfly. The values $q, 2q, 4q, w_i, w_i'$ were loaded in advance.

Remark 1. When the architecture provides fused multiply-add (FMA) instructions, we can use them to reduce one depth level from every Shoup modular multiplication. We describe the transformation in Figure 3 and note that this transformation is already implemented in HEXL [BKS+21]. Applying this transformation results in a reduction of two levels (resp. one) in the depth of the schemes shown in Figure 1 and 2, respectively.

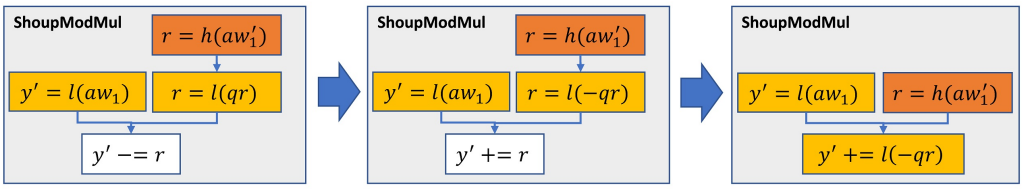


Figure 3: Shoup's modular multiplication using FMA instructions.

4 Mixed radix algorithms

Let $N = 2^m$, then a radix-k NTT algorithm assumes that $m \mid \frac{k}{2}$. When this is not the case, the algorithm must be a mixed-radix algorithm. Specifically in our case ($k = 4$), when m is odd, we need to perform one additional radix-2 iteration. We set this iteration as the last iteration in NTT and the first iteration in InvNTT. The reason is that these iterations consume the largest number of powers of ω, where the radix-2 variants may be preferred. Algorithms 7 and 8 present our mixed radix NTT and InvNTT implementations, respectively. We emphasize that before calling the Harvey butterflies, we must reduce the input values modulo $4q$.

Algorithm 7 CT radix-4 NTT

Input: $a \in \mathcal{R}_q$, $N = 2^m$, q a prime satisfying $q \equiv 1 \pmod{2N}$, $\bar{\Omega}$ in Extended-Reverse order.
Output: $\tilde{a} = NTT_\omega(a)$ in bit-reversed order.

procedure CT RADIX-4 NTT($a, N, q, \bar{\Omega}$)
  $t = N$, $\tilde{a} = a$
  bound_r4 = m is even ? N : $\frac{N}{2}$
  for ($m = 1$; $m \leq bound\_r4$; $m = 4m$) **do**
    $t = t/4$
    for $i = 0$; $i < m$; i++ **do**
      $m_1 = m + i$
      $w = (\bar{\Omega}_{m_1}, \bar{\Omega}_{2m_1}, \bar{\Omega}_{2m_1+1}, \bar{\Omega}_{m_1}\bar{\Omega}_{2m_1}, -\bar{\Omega}_{m_1}\bar{\Omega}_{2m_1+1})$
      for ($j = 4it$; $j < (4i+1)t$; j++) **do**
        $(\tilde{a}_j, \tilde{a}_{j+t}, \tilde{a}_{j+2t}, \tilde{a}_{j+3t}) =$ CT-radix4butterfly$(\tilde{a}_j, \tilde{a}_{j+t}, \tilde{a}_{j+2t}, \tilde{a}_{j+3t}, w)$
  if m is odd **then**
    for ($i = 0$; $i < N$; $i+ = 2$) **do**
      $a_i = min(a_i, a_i - 4q)$
      $(\tilde{a}_i, \tilde{a}_{i+1}) =$ HarveyButtefly$(\tilde{a}_i, \tilde{a}_{i+1}, w_{N+i})$
  return $\tilde{a}$

Algorithm 8 GS radix-4 NTT

Input: $\tilde{a} \in \mathcal{R}_q$, $N = 2^m$, q a prime satisfying $q \equiv 1 \pmod{2N}$, $\bar{\Omega}$ in Extended-Reverse order.
Output: $a = InvNTT(\tilde{a})$ in bit-reversed order.

procedure GS RADIX-4 NTT(a, N, q, Ω)
  $t = 1$, $m = N$, $a = \tilde{a}$
  if m is odd **then**
    for ($i = 0$; $i < N$; $i+ = 2$) **do**
      $a_i = min(a_i, a_i - 4q)$
      $(a_i, a_{i+1}) =$ HarveyButtefly(a_i, a_{i+1}, w_{N+i})
    $m = \frac{m}{2}$, $t = 2t$
  else
    for ($i = 0$; $i < N$; $i+$) **do**
      $a_i = min(a_i, a_i - 4q)$
      $a_i = min(a_i, a_i - 2q)$
  for ($m = \frac{m}{4}$; $m > 0$; $m = \frac{m}{4}$) **do**
    for $j = 0$; $j < m$; j++ **do**
      $m_1 = m + i$
      $w = (\bar{\Omega}_{m_1}, \bar{\Omega}_{2m_1}, \bar{\Omega}_{2m_1+1}, \bar{\Omega}_{m_1}, \bar{\Omega}_{2m_1}, -\bar{\Omega}_{m_1}, \bar{\Omega}_{2m_1+1})$
      for ($i = 4jt$; $i < (4j+1)t$; i++) **do**
        $(a_j, a_{j+t}, a_{j+2t}, a_{j+3t}) =$ GS-radix4butterfly$(a_j, a_{j+t}, a_{j+2t}, a_{j+3t}, w)$
    $t = t * 4$
  return a

4.1 The order of the power of ω

The construction of [LN16] proposed storing in Ω the N powers of ω in a bit-reverse order. Algorithms 7 and 8 assume this reverse order; this means that at every iteration they read the element ω_1 from index i and two consecutive elements ω_2, ω_3 from index $2i$. Then, the algorithms compute ω_4 and ω_5 by performing a multiplication of the above operands. This multiplication can be pre-computed and cached. Caching ω_4, ω_5 requires storing an additional $2(N-1)/3 = \sum_{i=1}^{log_4 N} \left(\frac{2N}{4^i}\right)$ elements, two per the $\frac{N}{4^i}$ butterflies at the ith iteration. We propose two methods of storing the powers of ω that start from Ω and generate an array $\bar{\Omega}$.

- Extended-Reverse order - $\bar{\Omega}_0 = \Omega_0$, $\bar{\Omega}_2 = \Omega_1$, $\bar{\Omega}_1 = \bar{\Omega}_3 = 0$, and for $i \in [2, \ldots, N)$

$$\bar{\Omega}_{2i} = \Omega_i \qquad \bar{\Omega}_{2i+1} = \begin{cases} i \texttt{ is even} & \Omega_i * \bar{\Omega}_i \pmod q \\ \texttt{else} & q - (\Omega_{i-1} * \bar{\Omega}_i \pmod q) \end{cases}$$

- Compressed order - $\bar{\Omega}$ contains a consecutive list of sets of five elements $(\omega_1, \omega_2, \omega_3, \omega_4, \omega_5)$ in the exact order in which they are consumed by Algorithm 7.

The advantage of the Extended-Reverse method is that $\bar{\Omega}$ closely follows the structure of Ω from [LN16]. That said, the advantage of the Compressed method is that it only needs to store $\frac{5(N-1)}{3} < 2N$ elements.

5 The VMSL implementation

We followed work [BH21] that presents an optimized NTT implementation, which leverages the VMSL instruction for prime fields with characteristic $q < 2^{32}$. Specifically, [BH21] targets the cryptographic signature scheme Dilithium [DKL+17], which uses $q = 2^{23} - 2^{13} + 1$. As opposed to their method, we target primes in the range $[2^{32}, 2^{52})$, where the field elements do not fit in 32-bit containers and we cannot directly use 32-bit multipliers. We use an upper bound to limit the size of the primes to 2^{52} so they comply with the VMSL ability to handle $\beta = 56$-bit words and to meet the Theorem 3 requirement that $q = \frac{\beta}{16}$.

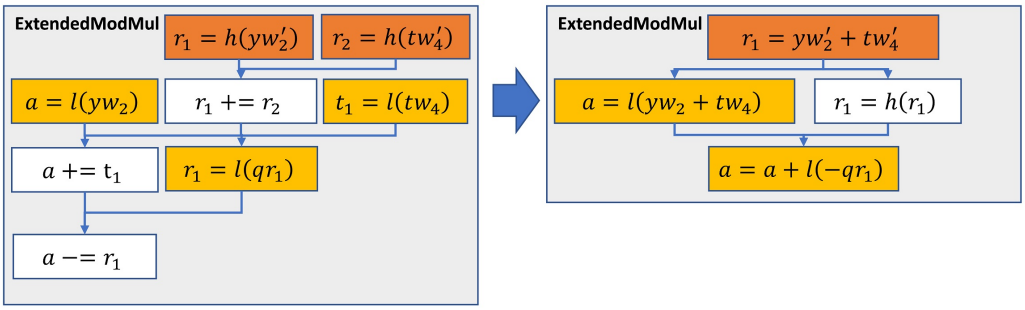


Figure 4: Implementing the ExtendedModMul algorithm using the VMSL instructions. Instructions description is available at [int21a].

Figure 4 demonstrates the advantage of using our proposed butterfly on architectures that provide VMSL-like instructions. Here, we compute the sum of the two products yw_2' and tw_4' by issuing the VMSL instruction. Then we take its lower and higher 56-bit halves in parallel and add the results to $l(-qr_1)$. We observed that using VMSL, we can reduce one circuit-depth level.

As explained above, the last NTT iteration is unique because it involves loading ten powers of ω per radix-4 butterfly. In contrast, the other iterations allow reusing these powers for two butterflies or more. Here, it is possible to perform two radix-4 butterflies in parallel using almost all the vector registers. We believe that this lack of generality between the iterations prevented the compiler from performing loop unrolling. Thus, we optimized the code by using two code paths: a single radix-4 butterfly for the last iteration and a double (parallel) radix-4 butterfly for the others.

6 The AVX512-IFMA implementation

Figure 4 shows how to perform our butterfly using AVX512-IFMA, which involves instructions that return the lower or higher half of a 104-bit product. Here, we target primes in the range $[2^{32}, 2^{48})$, where we use an upper bound to limit the size of the primes to 2^{48} so they comply with the AVX512-IFMA ability to handle $\beta = 52$-bit words and to meet the Theorem 3 requirement that $q = \frac{\beta}{16}$. Unlike the VMSL case, here the execution depth stayed the same, although the number of operations was reduced.

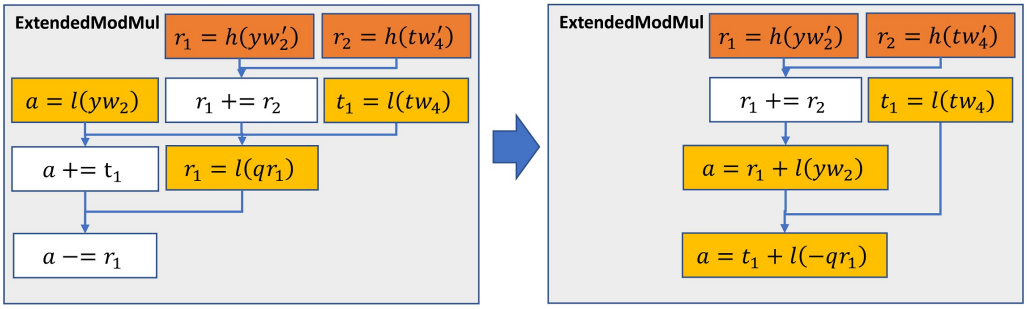


Figure 5: Implementing the ExtendedModMul algorithm using the AVX512-IFMA instructions.

We summarize the characteristics of the vector instructions that we used in our implementations in Table 1. We used the AVX512-IFMA instructions for multiplication, the `VPSUBQ` and `VPMINUQ` for modular reduction, and the memory/permute operations for arranging the data in registers. We used the instructions latencies to predict the overall latency of our implemented butterflies. A radix-2 butterfly involves five serial instructions: three additions and two multiplications, overall $3 * 4 + 2 = 11$ cycles. A radix-4 butterfly that relies on four radix-2 butterflies requires at least $11 * 2 = 22$ cycles, but we need to add four more cycles to account for the instructions' throughput. In contrast, our proposed butterfly involves six serial instructions: three multiplications, and three additions, overall $3 * 4 + 3 = 15$ cycles. Here, we also add four cycles to account for the instructions' throughput.

Table 1: The characteristics of the vector instructions used in our implementations. The data for the Ice Lake architecture is taken from [Int21b].

Purpose	Instructions	Latency	Throughput
Multiplication	`VPMADD52HUQ`, `VPMADD52LUQ`	4	0.5
Math & Logic	`VPADDQ`, `VPANDQ`, `VPXOR`	1	0.5
Math	`VPSUBQ`	1	0.3
Math	`VPMINUQ`	3	1
Memory	`VMOVDQ64`	8	0.5
Memory	`VPBROADCASTQ`	3	1
Memory	`VPGATHERQQ`	26	5
Memory	`VPSCATTERQQ`	11	2
Memory	`VPUNPCKLQDQ`, `VPUNPCKHQDQ`	1	1
Permute	`VSHUFI64X2`, `VPERMT2Q`, `VINSERTI64X4`	3	1
Permute	`VPBLENDMQ`	1	0.33

Similar to what we did for our IBM Z code, we provide here a special treatment for the final iterations, where we switched back from our proposed radix-4 butterfly to perform several radix-2 butterflies. Specifically, as noted in [BKS+21], the last three iterations of a radix-2 butterfly are unique because loading and storing the involved operands into eight-wide 64-bit word registers must be done in a special order. To this end, the code of [BKS+21] uses a serial combination of `VMOVDQU`, `VPERMT2Q`, and `VPBLENDMQ` per iterations with a latency of $3 + 1 + 8 = 14$. We observed that when combining the last four radix-2 iterations into one big radix-16 iteration, we can avoid some loads and stores. In addition, we can replace the serial `VPERMT2Q`, and `VPBLENDMQ`, with only one `VSHUFI64X2` instruction with a latency of 3. This also provides code simplification. Figure 6 shows the data arrangement in wide registers and the instructions we use.

Remark 2. When there is no inter-operable requirement between different HE libraries, it is possible to have the coefficients of $\tilde{a} = NTT(a)$ in any order as long as it can be understood by the $InvNTT(\tilde{a})$ implementation. This was already observed in [LN16] where Algorithm 9 leaves the output in bit-reverse order. We can use this observation to avoid the final `VPUNPCKLQDQ` in Figure 6.

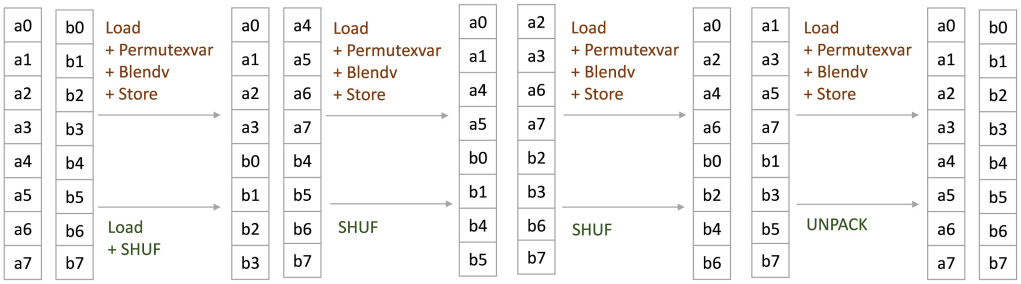


Figure 6: A comparison between the permutation instructions for the last 4 radix-2 iterations used in [BKS+21] (upper red arrows) and our radix-16 implementation (lower green arrows).

6.1 The order $\bar{\Omega}$

Our x86-64 implementation starts from a compressed order of $\bar{\Omega}$ and uses some order techniques from [BKS+21]. For example, to avoid extra shuffling during the NTT implementation, we duplicate some powers of ω in the last three iterations. Specifically, 4 and 2 times for the third to last and second to last iterations. In addition, we shuffle the powers of the last iterations to meet the order as presented in Figure 9c. Consequently, our radix-4 implementation requires $\frac{5N}{2^5} + \frac{10N}{2^4} + \frac{5N}{2^2} = \frac{65}{32}N < 3N$ and our combined radix-4 with radix-16 implementation requires $\frac{5N}{2^4} + \frac{4N}{2^3} + \frac{2N}{2^2} + \frac{N}{2} = \frac{29}{16}N < 2N$.

7 Experiments methodology and results

We start by describing our experimentation platforms and methodology for measurements.

7.1 Experiments setup

We carried out the experiments on two platforms:

- **IBM Z:** A z14 platform. This platform has 15 GB RAM, 128KiB L1d and L1i cache, 4MiB L2d cache, and 128MiB L3 cache. Running through a KVM hypervisor.
- **x86-64:** Dell XPS 13 7390 2-in-1 with the 10^{th} Intel®Core^{TM} Generation (Micro architecture Codename "Ice Lake"[ICL]) Intel®Core^{TM} i7-1065G7 CPU 1.30GHz. This platform has 16 GB RAM, 48K L1d and 32K L1i cache, 512K L2 cache, and 8MiB L3 cache, where we turned off the Intel® Turbo Boost Technology.

The code. The code is written in C using IBM Z and x86-64 C intrinsics. The implementation uses the VEF (vector) instructions and the x86-64 uses the `AVX512` and `AVX512-IFMA` instructions. The code was compiled with GCC (version 10) in 64-bit mode, using the "O3" optimization level, and run on a Linux (Ubuntu 20.04.2 LTS) OS.

We compared several NTT implementations.

- rad2-ref - A reference (scalar) implementation based on [LN16].
- rad2-seal - A radix-2 scalar NTT code that we extracted from SEAL [Lai17] and converted to C.
- rad4 - A radix-4 scalar implementation based on Algorithms 7, 8.
- rad4x4 - A radix-16 scalar implementation based on four radix-4 butterflies.
- rad2-dbl - Two rad2-ref scalar implementations that run in parallel.

For IBM z14 we also compared

- rad4-vmsl - A radix-4 vectorized implementation based on Algorithms 7, 8 and the `VMSL` instruction.

For x86-64 we also compared

- rad2-hexl - A radix-2 NTT code that we extracted from HEXL [BKS+21] and converted to C. This code uses the AVX512-IFMA ISA.
- rad2-ifma - A radix-4 scalar NTT code that uses AVX512-IFMA instructions.
- rad2-ifma2 - A radix-4 scalar NTT code that uses AVX512-IFMA instructions but returns the results in a different order then the reversed-binary order.
- r4r2-ifma - A radix-4 NTT implementation that uses AVX512-IFMA and performs the last 4 iterations in radix-2.
- r216-ifma - A radix-16 NTT implementation that uses AVX512-IFMA and radix-2 butterflies.

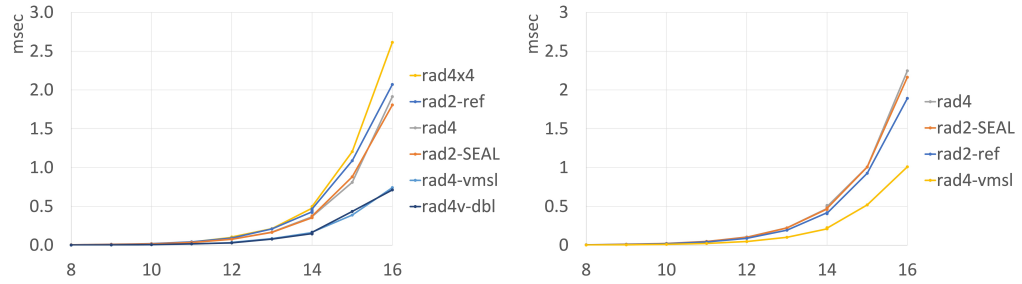
Third party code. For a fair comparison we converted the SEAL and HEXL implementations from C++ to C using the following procedures. We removed templates and made some functions `static inline` functions. We replaced `reinterpret_cast` and `static_cast` with C-style casting. Specifically, for HEXL we fixed the `BitShift` parameter to 52 and removed code paths (and branches) that deal with other values. Set the `InputLessThanMod` parameter as an input parameter to the relevant functions. Converted the `HEXL_LOOP_UNROLL_N` macros to `LOOP_UNROLL_N` macros and Defined the `HEXL_CHECK` and `HEXL_VLOG` macros as empty macros.

7.2 Measurements methodology

The performance measurements reported hereafter were measured in nanoseconds using the `clock_gettime` C function (per single core), where a lower count is better. We used the following measurement methodology in our experiments. Each measured function was isolated, run 10 times (warm-up), followed by 200 iterations that were measured and averaged. To minimize the effect of background tasks running on the system, we repeated every experiment 10 times and recorded the minimal result.

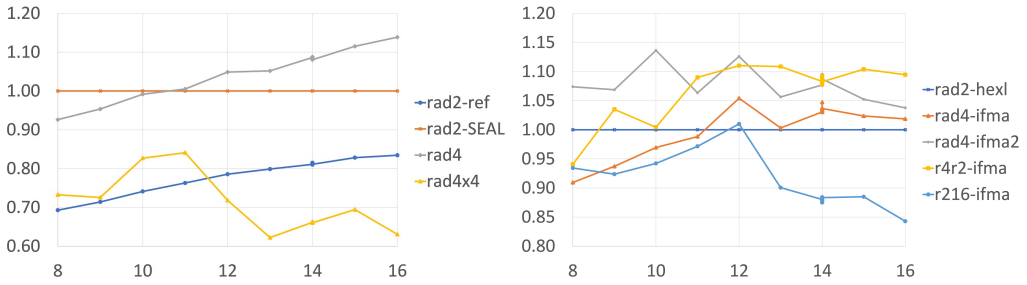
7.3 Results

Figure 7 compares our different implementations on IBM Z for the forward and inverse NTT. We start by observing that the implementation of SEAL is faster than our naïve reference implementation. The main difference lies in the manual loop unrolling that is done in the SEAL code. Our scalar radix-4 (without special loop unrolling) implementation



(a) NTT (b) InvNTT

Figure 7: Performance comparison of different NTT implementations on z14. The results are reported in milliseconds where lower is better.



(a) Scalar implementations (b) AVX512-IFMA implementations

Figure 8: A comparison of different NTT implementations on x86-64 platform. Panel (a) and Panel (b) use the SEAL and HEXL code that we converted to C, respectively, as a baseline. The rest of the implementations were normalized accordingly, where higher is better.

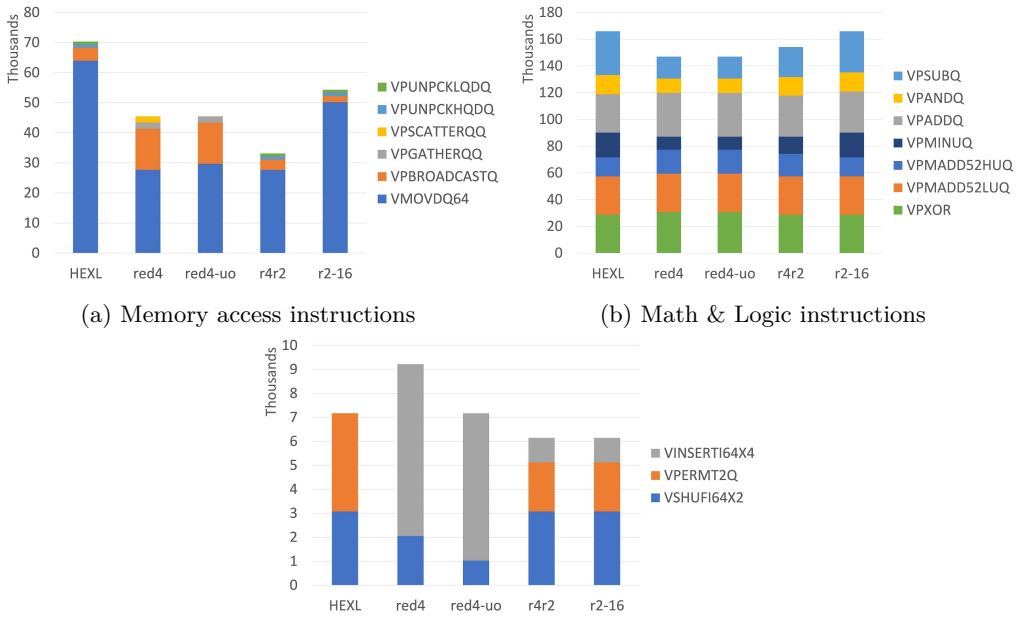
has similar performance to the radix-2 implementation of SEAL. Next, we observed that our radix-4 vector implementation is 2.5× faster than the rad2-seal implementation.

Our attempt to perform two NTT evaluations in parallel, in order to reuse the values of $\bar{\Omega}$ did not show a significant performance improvement. We saw similar performance for the InvNTT. The InvNTT is slower than the NTT because of the final multiplication by N^{-1} and the fact that we need to first reduce the input modulo $2q$.

HEXL provides a super optimized implementation that already uses the AVX512-IFMA instruction set. Thus, in Figure 8 Panel b we set the implementation of HEXL as the baseline and normalized the performance of the other vectorized implementations accordingly. We did the same for the scalar implementations in Panel a, where we set the SEAL code as our baseline.

For the scalar implementations, we saw a performance advantage for our radix-4 butterflies with larger values of N. In contrast, we did not observe any advantage to using a radix-16 implementation with our radix-4 butterflies. For the vectorized implementations, we observed that our radix-4 butterflies achieved similar performance to the HEXL radix-2 implementation. However, we did see a $\sim$ 1.1 times speedup when using the radix-4 butterflies for the first few iterations and then using our optimized radix-16 butterfly for the last iteration. We also noted some performance improvement when we left the NTT output unordered.

To complete the analysis on the x86-64 platform, we used Intel software developer emulator (SDE) [Int17] to count the number of instructions executed during each of the



(a) Memory access instructions

(b) Math & Logic instructions

(c) Permute instructions

Figure 9: Instruction number comparison between different NTT implementations on x86-64 that leverage the AVX512-IFMA instructions.

tested functions. We wrapped the boundaries of each function with the "SSC marks" 1 and 2, respectively. This was done by executing "`movl ssc_mark, %ebx; .byte 0x64, 0x67, 0x90`" and invoking the SDE with the flags "`-start_ssc_mark 1 -stop_ssc_mark 2 -mix`". The results are presented in Figure 9, where we organized the instructions into three groups: memory operations, math and logic operations, and permutation operations. A full description of the instructions is available at [int21a]. We see that our r4r2 implementation uses the least amount of memory instructions. As expected, the number of `VPXORQ` (xor operation) and `VPMADD52LUQ` stayed the same in all implementations while the number of `VPMADD52HUQ` was slightly higher for implementations that involved the radix-4 butterfly. The reduction in the number of `VPSUBQ` and `VPMINUQ` instructions (subtraction and minimization instructions, resp.) for radix-4 based implementations is due to the lower number of modulo reductions that these implementations perform.

8 GCC slowdown

We evaluated the code using both GCC and Clang. Surprisingly, we saw a large gap of 20% between the resulting binaries. Figure 10 provides this comparison.

Analysis on x86-64. We invoked the following instruction to collect system information about the rad2, rad2-seal and rad4 implementations when compiled with GCC or Clang on the x86-64 platform.

> sudo perf stat -B -e cache-references, cache-misses, cycles, instructions, branches, faults, mem_load_retired.l1_miss, branch-misses, mem-loads, mem-stores, mem_load_retired.l2_miss, mem_load_retired.l3_miss ntt-variants-bench x

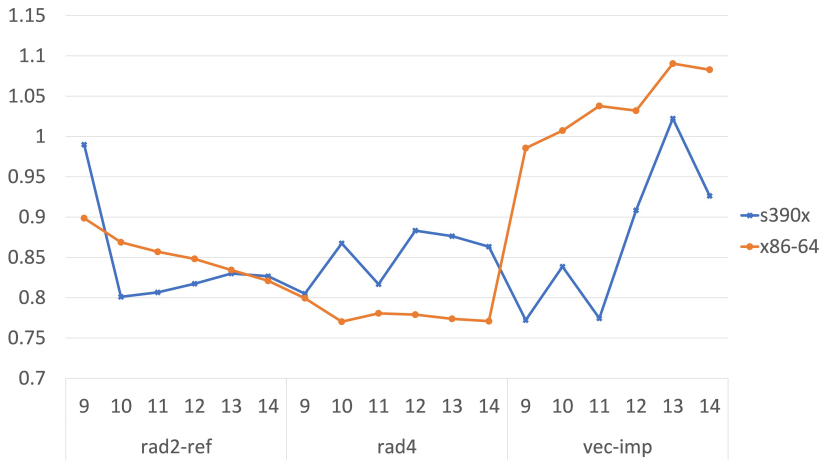


Figure 10: Slowdown comparison of different NTT implementations for $N \in \{9, \ldots, 14\}$ when compiled with Clang versus GCC. Lower values indicate a bigger gap between GCC and Clang. The code was compiled on the IBM Z and x86-64 platforms.

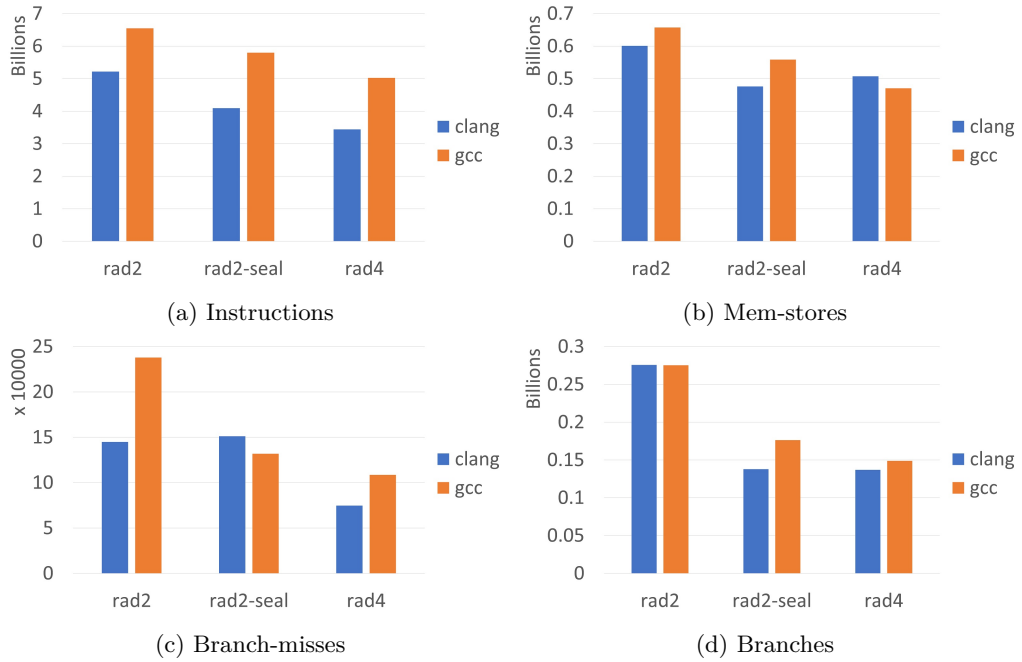
We observed no difference in the following parameters: mem_load_retired.l2_miss, cache-misses, faults, mem_load_retired.l3_miss. Figure 11 shows some parameters where we observed a major difference. First, we noticed that the number of branches and branch-misses when using our radix-4 butterflies is reduced compared to a radix-2 implementation. While the number of branches was only slightly higher in the GCC output binary, we still observed a higher number of branch misses. In addition, the number of executed instructions is $25 - 46\%$ higher in GCC compared to the Clang output binary. Finally, we saw that at least for the radix-2 case, the number of memory stores is higher in the GCC case, which may suggest that Clang does a better job of managing the registers.

9 Conclusions

Having a performant NTT implementation is critical for accelerating HE computations. We presented new radix-4 butterflies and analyzed their characteristics. We then explored and measured several different mixed-radix implementations. We demonstrated the advantage of our butterflies on two different architectures: IBM Z and x86-64. We believe that other architectures may also find it useful, especially future hardware designs that target NTT.

A NTT algorithms

Algorithms 9 and 10 are the forward and inverse NTT algorithms from [LN16], respectively.



(a) Instructions (b) Mem-stores

(c) Branch-misses (d) Branches

Figure 11: A comparison of several parameters of the rad2, rad2-seal, and rad4 implementations when compiled with GCC or Clang on an x86-64 platform.

Algorithm 9 CT radix-2 NTT [LN16]

Input: $a \in \mathcal{R}_q$, N a power of 2, q a prime satisfying $q \equiv 1 \pmod{2N}$, ψ_{rev}, which holds the powers of ψ in bit-reversed order.
Output: $\tilde{a} = NTT_{\psi}(a)$ in bit-reversed order.

procedure CT RADIX-2 NTT(a, N, q, ψ_{rev})
  $t = N$, $\tilde{a} = a$
  for ($m = 0$; $m < N$; $m = 2m$) **do**
    $t = t/2$
    for $i = 0$; $i < m$; i++ **do**
      $w = \psi_{rev}[m + i]$
      for ($j = 2it$; $j < (2i + 1)t$; j++) **do**
        $(X, Y) = (\tilde{a}_j, \tilde{a}_{j+t}w)$
        $(\tilde{a}_j, \tilde{a}_{j+t}) = (X + Y, X - Y) \pmod{q}$
  return $\tilde{a}$

Algorithm 10 Gentleman-Sande (GS) Radix-2 InvNTT [LN16]

Input: $\tilde{a} \in \mathcal{R}_q$, N a power of 2, q a prime satisfying $q \equiv 1 \pmod{2N}$, ψ_{rev}^{-1}, which holds the powers of ψ^{-1} in bit-reversed order.
Output: $a = InvNTT(\tilde{a})$ in bit-reversed order.

procedure GENTLEMAN-SANDE (GS) RADIX-2 INVNTT($\tilde{a}, N, q, \psi_{rev}^{-1}$)
  $t = 1$, $a = \tilde{a}$
  for ($m = N/2$; $m > 0$; $m \gg= 1$) **do**
    for ($i = 0$; $i < m$; i++) **do**
      $w = \psi_{rev}^{-1}[m + i]$
      for $j = 2it$; $j < (2i + 1)t$; j++) **do**
        $(X, Y) = (a_j, a_{j+t})$
        $(a_j, a_{j+t}) = (X + Y, w(X - Y)) \pmod{q}$
    $t = 2t$
  for ($j = 0$; $j < N$; j++) **do**
    $a_j = a_j \cdot N^{-1}$
  return a

References

[BH21] Jonathan Bradbury and Basil Hess. Fast Quantum-Safe Cryptography on IBM Z. Technical report, 2021. URL: https://csrc.nist.gov/CSRC/media/Events/third-pqc-standardization-conference/documents/accepted-papers/hess-fast-quantum-safe-pqc2021.pdf.

[BKS+21] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D.M. de Souza, and Vinodh Gopal. Intel HEXL : Accelerating Homomorphic Encryption with Intel AVX512-IFMA52. Technical report, 2021. URL: https://eprint.iacr.org/2021/420.

[CHK+19] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A Full RNS Variant of Approximate Homomorphic Encryption. In Carlos Cid and Michael J Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, pages 347–368, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-10970-7_16.

[CT65] James W Cooley and John W Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965. doi:10.2307/2003354.

[DG17] Nir Drucker and Shay Gueron. Paillier-encrypted databases with fast aggregated queries. In *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 848–853. IEEE, IEEE, Jan 2017. doi:10.1109/CCNC.2017.7983244.

[DG19] Nir Drucker and Shay Gueron. Fast Modular Squaring with AVX512IFMA. In Shahram Latifi, editor, *16th International Conference on Information Technology-New Generations (ITNG 2019)*, pages 3–8, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-14070-0_1.

[DKL+17] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation. 2017. URL: https://pq-crystals.org/dilithium/data/dilithium-specification.pdf.

[DS16] Wei Dai and Berk Sunar. cuHE: A Homomorphic Encryption Accelerator Library. In Enes Pasalic and Lars R Knudsen, editors, *Cryptography and Information Security in the Balkans*, pages 169–186, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-29172-7_11.

[GADO17] Umme Salma Gadriwala, Christopher Kumar Anand, Curtis D'Alves, and Bill O'Farrell. Accelerating Poly1305 Cryptographic Message Authentication on the Z14. In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, CASCON '17, pages 48–54, Riverton, NJ, USA, 2017. IBM Corp. doi:10.5555/3172795.3172802.

[Gar21] Gartner. Gartner identifies top security and risk management trends for 2021. Technical report, March 2021. URL: https://www.gartner.com/en/newsroom/press-releases/2021-03-23-gartner-identifies-top-security-and-risk-management-t.

[GK16] Shay Gueron and Vlad Krasnov. Accelerating big integer arithmetic using intel ifma extensions. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 32–38, 2016. doi:10.1109/ARITH.2016.22.

[GS66] W. M. Gentleman and G. Sande. Fast fourier transforms—For fun and profit. *AFIPS Conference Proceedings - 1966 Fall Joint Computer Conference, AFIPS 1966*, pages 563–578, 1966. doi:10.1145/1464291.1464352.

[Har14] David Harvey. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation*, 60:113–119, 2014. doi:10.1016/j.jsc.2013.09.002.

[IBM21] IBM. z/Architecture Principles of Operation SA22-7832-12, April 2021. URL: https://www.ibm.com/support/pages/zarchitecture-principles-operation.

[Int17] Intel. Intel® Software Development Emulator, version 8.12.0. https://software.intel.com/en-us/articles/intel-software-development-emulator, January 2017.

[int21a] intel. Intel® 64 and IA-32 architectures software developer's manual, June 2021. URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[Int21b] Intel. Intel® intrinsics guide, Oct 2021. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[JLK+20] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Keewoo Lee, Namhoon Kim, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. HEAAN Demystified: Accelerating Fully Homomorphic Encryption Through Architecture-centric Analysis and Optimization, 2020. arXiv:2003.04510.

[KG19] Dusan Kostic and Shay Gueron. Using the new vpmadd instructions for the new post quantum key encapsulation mechanism sike. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 215–218, 2019. doi:10.1109/ARITH.2019.00050.

[Lai17] Kim Laine. Simple Encrypted Arithmetic Library 2.3.1. Technical report, Microsoft, WA, USA, 2017. URL: https://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security*, pages 124–139, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-48965-0_8.

[SLPD20] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: An architecture for computing on encrypted data. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 1295–1309, 2020. doi:10.1145/3373376.3378523.

[YZD+20] James You, Qi Zhang, Curtis D'Alves, Bill O'Farrell, and Christopher K. Anand. Using z14 Fused-Multiply-Add Instructions to Accelerate Elliptic Curve Cryptography. (Report 2020/481):284–291, 2020. URL: https://eprint.iacr.org/2020/481.