Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52

Fabian Boemer[1], Sejun Kim[1], Gelila Seifu[1], Fillipe D. M. de Souza[1], and Vinodh Gopal[1]

Intel Corporation
{fabian.boemer, sejun.kim, gelila.seifu, fillipe.souza, vinodh.gopal}@intel.com

Abstract. Modern implementations of homomorphic encryption (HE) rely heavily on polynomial arithmetic over a finite field. This is particularly true of the CKKS, BFV, and BGV HE schemes. Two of the biggest performance bottlenecks in HE primitives and applications are polynomial modular multiplication and the forward and inverse number-theoretic transform (NTT). Here, we introduce *Intel® Homomorphic Encryption Acceleration Library (Intel® HEXL)*, a C++ library which provides optimized implementations of polynomial arithmetic for Intel® processors. Intel HEXL takes advantage of the recent Intel® Advanced Vector Extensions 512 (Intel® AVX512) instruction set to provide state-of-the-art implementations of the NTT and modular multiplication. On the forward and inverse NTT, Intel HEXL provides up to 7.2x and 6.7x speedup, respectively, over a native C++ implementation. Intel HEXL also provides up to 6.0x speedup on the element-wise vector-vector modular multiplication, and 1.7x speedup on the element-wise vector-scalar modular multiplication. Intel HEXL is available open-source at https://github.com/intel/hexl under the Apache 2.0 license.

Keywords: privacy-preserving machine learning; Intel AVX512; homomorphic encryption

1 Introduction

Homomorphic encryption (HE) is a form of encryption which enables computation in the encrypted domain. Homomorphic encryption is useful in applications with sensitive data, particularly medical and financial settings [2, 3, 15]. However, HE currently suffers from enormous memory and runtime overheads of up to 30,000x [14].

Ciphertexts in many HE schemes are polynomials in finite fields, whose coefficients can be hundreds of bits and whose degree is typically a power in the range $[2^{10}, 2^{17}]$. Performing HE computations requires operating on these large polynomials. While recent years have seen tremendous improvement in HE performance due to algorithmic changes and optimized implementations, the performance overhead remains perhaps the biggest bottleneck to adoption of HE.

Here, we introduce *Intel® HEXL*, an open-source library which provides efficient implementations of integer arithmetic on finite fields. Intel HEXL targets polynomial operations with word-sized primes on 64-bit processors. For efficient implementation, Intel HEXL uses the Intel Advanced Vector Extensions 512 (Intel AVX512) instruction set to provide optimized implementations on Intel processors. In particular, the Intel® Advanced Vector Extensions 512 Integer Fused Multiply Add (Intel® AVX512-IFMA52) instructions introduced in the 3rd Gen Intel® Xeon® Scalable Processors provide significant speedup on primes below 50–52 bits.

We begin with a brief introduction of the mathematical concepts implemented in Intel HEXL (Section 2) and the Intel AVX512 instruction set (Section 2.3). Section 3 compares Intel HEXL to existing work, explaining Intel HEXL's unique contribution lies in the application of the Intel AVX512-IFMA52 instruction set to word-size finite field arithmetic, such as the number-theoretic transform (NTT). Next, we introduce the design (Section 4) and implementation (Section 4) of Intel HEXL. In particular, we provide detailed descriptions of the forward (Section 4.1) and inverse (Section 4.1) NTT and polynomial kernels (Section 4.2), including vector-vector modular multiplication (Section 4.2) and vector-scalar modular multiplication (Section 4.2). We demonstrate the performance of Intel HEXL in Section 5, showcasing up to 7.2x and 6.7x speedup over a native C++ implementation of the forward and inverse NTT, respectively, up to 6.0x on element-wise vector-vector modular multiplication and 1.7x speedup on the element-wise vector-scalar modular multiplication. Finally, we conclude in Section 6.

2 Background

We provide a brief background of the algorithms optimized used in Intel HEXL, which are common building blocks of lattice cryptography. Let $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$ be the polynomial quotient ring consisting of polynomials with degree at most $N-1$ and integer coefficients in the finite field $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$, where q is a word-sized prime satisfying $q \equiv 1 \mod 2N$. One way to represent a polynomial $a = \sum_{i=0}^{N-1} a_i x^i \in \mathcal{R}_q$ is via the coefficient embedding, i.e.

$$a = (a_0, a_1, a_2, \ldots, a_{N-1})$$

with $a_i \in \mathbb{Z}_q$.

Typical HE operations compute on polynomials in $\mathcal{R}_q$ as follows:

- *Element-wise addition.* Given $a, b \in \mathcal{R}_q$, compute $c = a + b$ such that $c_i = (a_i + b_i) \mod q$.
- *Element-wise negation.* Given $a \in \mathcal{R}_q$, compute $b = -a$ such that $b_i = q - a_i$.
- *Element-wise multiplication.* Given $a, b \in \mathcal{R}_q$, compute $c = a \odot b$ such that $c_i = (a_i \cdot b_i) \mod q$.
- *Element-wise vector-scalar multiplication.* Given $a \in \mathcal{R}_q, b \in \mathbb{Z}_q$, compute $c = a \cdot b$ such that $c_i = (a_i \cdot b) \mod q$.

– *Vector-vector multiplication.* Given $a, b \in \mathcal{R}_q$, compute $c = a * b$ such that

$$c_i = \sum_{j=0}^{i} a_j \cdot b_{i-j} - \sum_{j=i+1}^{N-1} a_j \cdot b_{N+i-j}.$$

Note,

$$X^N \equiv -1 \mod (X^N + 1),$$

which yields the negative coefficients.

In practice, element-wise addition and element-wise negation are typically much faster to compute than the types of multiplication. As such, we focus on the various multiplication functions in $\mathcal{R}_q$.

2.1 Barrett reduction

Scalar modular multiplication is a primary bottleneck in lattice cryptography. A simple implementation of scalar modular multiplication uses the 128-bit integer extension, supported by many modern compilers including gcc and clang. The modulus operator % is used for modular reduction. Listing 1.1 shows C/C++ source code for a simple implementation of scalar modular multiplication.

```
1 // Returns (a * b) mod q
2 uint64_t naive_modmul(uint64_t a, uint64_t b, uint64_t q) {
3     return uint64_t(uint128_t(a) * b) % q;
4 }
```

Listing 1.1: Native modular multiplication

Performance of this naive implementation is poor due to the modulus operator, which will perform integer division via, e.g., the extended Euclidean algorithm.

In typical HE applications, the modulus q is re-used for many modular multiplications. In this setting, *Barrett reduction* can be used to improve performance. Barrett reduction takes advantage of the fact that

$$x \mod q = x - \lfloor x/q \rfloor q$$

when x/q is computed exactly. If x/q is computed with sufficient accuracy, the result remains correct. Barrett reduction uses a pre-computed integer, k, based on the modulus q, to replace division with bit shifting. This approximation requires an extra conditional subtraction to guarantee correctness. Nevertheless, replacing integer division with bit shifting results in a speedup. Algorithm 1 shows Barrett's algorithm, as presented in [10].

2.2 Number-theoretic transform (NTT)

The number-theoretic transform (NTT) is another performance bottleneck in typical lattice cryptography computations. The NTT is equivalent to the fast Fourier transform (FFT) in a finite field, i.e. all addition and multiplications

Algorithm 1 Barrett Reduction

Require: $q < 2^Q, d < 2^D, k = \lfloor \frac{2^L}{q} \rfloor$, with $Q \leq D \leq L$
Ensure: Returns $d \mod q$
1: **function** BARRETT REDUCTION(d, q, k, Q, L)
2: $c_1 \leftarrow d \gg (Q-1)$
3: $c_2 \leftarrow c_1 k$
4: $c_3 \leftarrow c_2 \gg (L-Q+1)$
5: $c_4 \leftarrow d - qc_3$
6: **if** $c_4 \geq q$ **then**
7: $c_4 \leftarrow c_4 - q$
8: **end if**
9: **return** c_4
10: **end function**

---

are performed with respect to the modulus q. Let ω be a primitive N'th root of unity in $\mathbb{Z}_q$ and $a = (a_0, \ldots, a_{N-1}) \in \mathbb{Z}_q^N$. Then, the forward cyclic NTT is defined as $\tilde{a} = \text{NTT}(a)$, where $\tilde{a}_i = \sum_{j=0}^{N-1} a_j \omega^{ij} \mod q$ for $i = 0, 1, \ldots, N-1$. The inverse cyclic NTT is given by $b = InvNTT(\tilde{a})$, where $b_i = \frac{1}{n} \sum_{j=0}^{N-1} \tilde{a}_j \omega^{-ij} \mod q$ for $i = 0, 1, \ldots, N-1$. Note, $InvNTT(NTT(a)) = a$.

The NTT can be used to speed up polynomial-polynomial multiplication in $\mathcal{R}_q$. However, using $\odot$ to indicate element-wise multiplication, the straightforward usage

$$InvNTT(NTT(a) \odot NTT(b))$$

corresponds to polynomial-polynomial multiplication in $\mathbb{Z}_q^N/(X^N - 1)$, whereas HE operates in $\mathcal{R}_q = \mathbb{Z}_q^N/(X^N + 1)$. As described in [16], a modification of the cyclic NTT, known as the *negacyclic NTT*, or *negative wrapped convolution*, can be used to perform polynomial multiplication in $\mathcal{R}_q$. Let ψ be a primitive $2N$'th root of unity in $\mathbb{Z}_q$. Let $a, b \in \mathbb{Z}_q^N$ and $\hat{a} = (a_0, \psi a_1, \psi^2 a_2, \ldots, \psi^{N-1} a_{N-1})$, $\hat{b} = (b_0, \psi b_1, \psi^2 b_2, \ldots, \psi^{N-1} b_{n-1})$. Then, the negacyclic NTT is defined as

$$c = (1, \psi^{-1}, \ldots, \psi^{-(N-1)}) \odot InvNTT(NTT(\hat{a}) \odot NTT(\hat{b})),$$

which satisfies $c = a * b$ in $\mathcal{R}_q$. The NTT-based formulation reduces the runtime of polynomial-polynomial modular multiplication from $O(N^2)$ to $O(N \log N)$.

Optimized implementation. The NTT inherits a rich history of optimizations from the FFT, in addition to several NTT-specific optimizations. Similar to the FFT, the NTT has a recursive formulation attributed to Cooley and Tukey [4]. Cooley-Tukey NTTs decompose an NTT of size $N = N_1 N_2$ as N_1 NTTs of size N_2 followed by N_2 NTTs of size N_1. This recursive formulation reduces the runtime of the NTT to $O(N \log N)$, improving upon the $O(N^2)$ runtime of a naive implementation. The choice of $\min(N_1, N_2)$ determines the *radix* of the implementation. One byproduct of the Cooley-Tukey forward NTT is that the output is in bit-reversed order. That is, given an index i in binary representation $i = 0bi_0 i_1 \ldots i_{\log_2(N)} \in \{0,1\}^{\log_2(N)}$, the output of the Cooley-Tukey

NTT at index i is $NTT(a)[0bi_{\log_2(N)}i_{\log_2(N)-1}\dots i_1 i_0]$. The inverse transform restores the standard bit ordering. Typically, any operations performed in the bit-reversed domain are performed element-wise, so the bit-reversal usually does not pose a problem. Furthermore, the Cooley-Tukey NTT may operate in-place, i.e. the output overwrites the input. Algorithm 2 shows a simple radix-2 in-place Cooley-Tukey NTT algorithm, taken from [16]. Algorithm 3 shows the analogous radix-2 in-place Gentleman-Sande inverse NTT algorithm.

Algorithm 2 Cooley-Tukey Radix-2 NTT

Require: $a = (a_0, a_1, \dots, a_{N-1}) \in \mathbb{Z}_q^N$ in standard ordering. N is a power of 2. q is a prime satisfying $q \equiv 1 \mod 2N$. $\psi_{\text{rev}} \in \mathbb{Z}_q^N$ stores the powers of ψ in bit-reversed order.

Ensure: $a \leftarrow NTT(a)$ in bit-reversed order.

```
 1: function Cooley-Tukey Radix-2 NTT(a, N, q, ψ_rev)
 2:     t ← n
 3:     for (m = 1; m < n; m = 2n) do
 4:         t ← t/2
 5:         for (i = 0; i < m; i++) do
 6:             j1 ← 2 · i · t
 7:             j2 ← j1 + t − 1
 8:             W ← ψ_rev[m + i]
 9:             for (j = j1; j ≤ j2; j++) do
10:                 X0 ← a_j
11:                 X1 ← a_{j+t}
12:                 a_j ← X0 + W · X1  mod q
13:                 a_{j+t} ← X0 − W · X1  mod q
14:             end for
15:         end for
16:     end for
17:     return a
18: end function
```

The *butterfly* refers to the radix-$r = \min(N_1, N_2)$ NTT. For instance, the butterfly for the radix-2 NTT in Algorithm 2 is given in lines 10–13, which compute

$$(X_0, X_1) \mapsto (X_0 + WX_1 \mod q, X_0 - WX_1 \mod q).$$

Harvey [12] provides an optimization to the butterfly using a redundant representation $X_0, X_1 \in \mathbb{Z}_{4q}$. Algorithm 4 shows the Harvey forward NTT butterfly[1]. Using the Harvey butterfly in the Cooley-Tukey NTT yields outputs in $\mathbb{Z}_{4q}^N$, so an additional correction step is required to reduce the output to $\mathbb{Z}_q^N$. Similar to

[1] Note, [12] presents Algorithm 2 as the inverse butterfly, whereas uses it for the forward NTT. This difference stems from the choice of 'decimation-in-time' vs. 'decimation-in-frequency.' The same applies for the inverse butterfly.

Algorithm 3 Gentleman-Sande (GS) Radix-2 InvNTT

Require: $a = (a_0, a_1, \ldots, a_{N-1}) \in \mathbb{Z}_q^N$ in bit-reversed ordering. N is a power of $2.q$ is a prime satisfying $q \equiv 1 \mod 2N$. $\psi_{\text{rev}}^{-1} \in \mathbb{Z}_q^N$ stores the powers of ψ^{-1} in bit-reversed order.

Ensure: $a \leftarrow InvNTT(a)$ in standard ordering.

```
 1: function Gentleman-Sande Radix-2 InvNTT(a, N, q, ψ_rev)
 2:     t ← 1
 3:     for (m = n; m > 1; m = m/2) do
 4:         j1 ← 0
 5:         h ← m/2
 6:         for (i = 0; i < h; i++) do
 7:             j2 ← j1 + t − 1
 8:             W ← ψ_rev^{-1}[h + i]
 9:             for (j = j1; j ≤ j2; j++) do
10:                 X0 ← a_j
11:                 X1 ← a_{j+t}
12:                 a_j ← X0 + X1  mod q
13:                 a_{j+t} ← (X0 − X1) · W  mod q
14:             end for
15:             j1 ← j1 + 2t
16:         end for
17:         t ← 2t
18:     end for
19:     for (j = 0; j < n; j++) do
20:         a[j] ← a[j] · n^{-1}  mod q
21:     end for
22:     return a
23: end function
```

Algorithm 4 Harvey NTT butterfly. β is the word size, e.g. $\beta = 2^{64}$ on typical modern CPU platforms.

Require: $q < \beta/4$; $0 < W < q$

Require: $W' = \lfloor W\beta/q \rfloor$, $0 < W' < \beta$

Require: $0 \le X_0, X_1 < 4q$

Ensure: $Y_0 \leftarrow X_0 + WX_1 \mod q; 0 \le Y_0 < 4q$

Ensure: $Y_1 \leftarrow X_0 - WX_1 \mod q; 0 \le Y_1 < 4q$

```
 1: function HarveyNTTButterfly(X0, X1, W, W', q, β)
 2:     if X0 ≥ 2q then
 3:         X0 ← X0 − 2q
 4:     end if
 5:     Q ← ⌊W'X1/β⌋
 6:     T ← (WX1 − Qq)  mod β
 7:     Y0 ← X0 + T
 8:     Y1 ← X0 − T + 2q
 9:     return Y0, Y1
10: end function
```

the forward transform, Harvey [12] also provides an efficient butterfly for the inverse NTT, using a redundant representation in $[0, 2q)$. Algorithm 5 shows the Harvey inverse NTT butterfly.

Algorithm 5 Harvey inverse NTT butterfly. β is the word size, e.g. $\beta = 2^{64}$ on typical modern CPU platforms.

Require: $q < \beta/4$; $0 < W < q$
Require: $W' = \lfloor W\beta/q \rfloor$; $0 < W' < \beta$
Require: $0 \leq X_0, X_1 < 2q$
Ensure: $Y_0 \leftarrow X_0 + X_1 \mod q; 0 \leq Y_0 < 2q$.
Ensure: $Y_1 \leftarrow W(X_0 - X_1) \mod q; 0 \leq Y_1 < 2q$.
1: **function** HarveyInvNTTButterfly($X_0, X_1, W, W', q, \beta$)
2: $\quad Y_0 \leftarrow X_0 + X_1$
3: $\quad$ **if** $Y_0 \geq 2q$ **then**
4: $\quad\quad Y_0 \leftarrow Y_0 - 2q$
5: $\quad$ **end if**
6: $\quad T \leftarrow X_0 - X_1 + 2q$
7: $\quad Q \leftarrow \lfloor W'T/\beta \rfloor$
8: $\quad Y_1 \leftarrow (WT - Qq) \mod \beta$
9: $\quad$ **return** Y_0, Y_1
10: **end function**

2.3 Intel Advanced Vector Extensions

The Intel® Advanced Vector Extensions (Intel® AVX) is a set of single-instruction multiple data (SIMD) instructions for the x86 architecture. Intel AVX instructions enable simultaneous computation on chunks of data larger than typical word-sized chunks. For instance, the legacy Intel® Streaming SIMD Extensions (Intel® SSE) operates on 128-bit data chunks. The AVX2 instruction set expanded the SIMD capability to 256-bit data chunks. In recent years, the Intel AVX512 instruction set further expanded the SIMD capability to 512-bit data chunks. Each SIMD instruction set can use the data chunks to represent multiple smaller-width inputs. For instance, Intel AVX512 intrinsics use the `__m512i` datatype, which represents a packed 512-bit integer, which may represent eight 64-bit integers.

To employ these SIMD instructions, users may call the desired assembly function. Additionally, for easier use, Intel provides a set of C/C++-compatible intrinsics, which compile to the relevant assembly instruction. To understand the naming of Intel intrinsics, 'epi' refers to *extended packed integer* and 'epu' refers to *extended packed unsigned integer* and the last number indicates the number of bits.

For instance, the Intel® AVX512 Doubleword and Quadword (Intel® AVX512-DQ) extension contains the following intrinsic:

– `__m512i _mm512_mullo_epi64 (__m512i a, __m512i b)`. Given packed 64-bit integers in a and b, return the low 64 bits of the 128-bit product $a \cdot b$.

However, there is no matching `_mm512_mulhi_epi64` instruction. Instead, it may be emulated with, e.g. four Intel AVX512 32-bit multiplies, five Intel AVX512 64-bit adds and five Intel AVX512 64-bit shift instructions [14].

The Intel AVX512-IFMA52 extension to Intel AVX512 [5] consists of several operations useful in lattice cryptography. In particular, Intel AVX512-IFMA52 introduces the following intrinsics:

– `__m512i _mm512_madd52lo_epu64 (__m512i a, __m512i b, __m512i c)`. Given packed unsigned 52-bit integers in each 64-bit element of b and c, compute the 104-bit product $b \cdot c$. Add the low 52 bits of the product to the packed unsigned 64-bit integers in a and return the result.
– `__m512i _mm512_madd52hi_epu64 (__m512i a, __m512i b, __m512i c)`. Given packed unsigned 52-bit integers in each 64-bit element of b and c, compute the 104-bit product $b \cdot c$. Add the high 52 bits of the product to the packed unsigned 64-bit integers in a and return the result.

Intel HEXL also utilizes one intrinsic from the Intel AVX512 Vector Bit Manipulation Version 2 (Intel AVX512-VBMI2) instruction set:

– `__m512i _mm512_shrdi_epi64 (__m512i a, __m512i b, int imm8)`. Given packed 64-bit integers in b and c, concatenate them to a 128-bit intermediate result. Shift the result right by *imm8* bits and return the lower 64 bits of the result.

3 Previous Work

The key differentiating factor between Intel HEXL and previous work is the use of the Intel AVX512-IFMA52 instruction set to accelerate finite field arithmetic, in particular the number-theoretic transform. Apart from this contribution, Intel HEXL utilizes several existing algorithms from previous work.

The Mathemagix library [13] provides Intel AVX-accelerated implementations of modular integer arithmetic using a SIMD programming model. NFLlib [1] provides similar acceleration of primitives common to the ring $\mathbb{Z}_q/(X^N + 1)$ using Intel SSE and Intel AVX2 instructions. However, neither Mathemagix nor NFLlib consider Intel AVX512 implementations using the Intel AVX512-IFMA52 instruction set.

Previous work [7, 11] using Intel AVX512-IFMA52 focuses on accelerating big integer multiplication without modular multiplication. Drucker and Gueron [6] use Intel AVX512-IFMA52 to accelerate large integer modular squaring via Montgomery multiplication. In contrast, our work uses Barrett reduction and focuses on word-sized modular multiplication. Furthermore, to our knowledge, Intel HEXL is the first work to accelerate the NTT using Intel AVX512-IFMA52.

4 Intel HEXL

Design Intel HEXL is an open-source C++11 library available under the Apache 2.0 license. Intel HEXL focuses on the case where $q < \beta = 2^{64}$, as implemented on a 64-bit word-sized CPU platform. This restriction of is typical of HE implementations on CPU. SEAL [18] for instance, bounds all coefficient moduli to 61 bits. GPUs implementations of HE, (e.g. [14, 17]) however, often restrict $q < 2^{32}$ since 64-bit support for integers is often restricted or emulated, yielding lower performance. As such, the Intel HEXL API uses unsigned 64-bit integers input vector types. Unlike other libraries, however, Intel HEXL does not currently provide a BigNum type for multi-precision arithmetic, such as is found in NTL [19] or NFLlib [1].

Intel HEXL consists of a class for the NTT functionality in addition to several free functions implementing element-wise modular arithmetic on word-sized primes. The NTT class performs pre-computation for the roots of unity and their pre-computed factors during initialization. The element-wise functions perform any pre-computations outside the critical loop, rendering it unnecessary for the end user to perform any pre-computation. Intel HEXL is single-threaded and thread safe. Listing 1.2 shows the application programming interface (API) for the NTT.

```
1   class NTT {
2     public:
3       /// Initializes an empty NTT object
4       NTT();
5
6       /// Destructs the NTT object
7       ~NTT();
8
9       /// Initializes an NTT object with degree degree and modulus q.
10      /// @param[in] degree a.k.a. N. Size of the NTT transform. Must be a power of
          2
11      /// @param[in] q Prime modulus. Must satisfy q == 1 mod 2N
12      /// @brief Performs pre-computation necessary for forward and inverse
13      /// transforms
14      NTT(uint64_t degree, uint64_t q);
15
16      /// @brief Initializes an NTT object with degree degree and modulus q
17      /// @param[in] degree a.k.a. N. Size of the NTT transform. Must be a power of
          2
18      /// @param[in] q Prime modulus. Must satisfy q == 1 mod 2N
19      /// @param[in] root_of_unity 2N'th root of unity in Z_q
20      /// @details  Performs pre-computation necessary for forward and inverse
21      /// transforms
22      NTT(uint64_t degree, uint64_t q, uint64_t root_of_unity);
23
24      /// @brief Compute forward NTT. Results are bit-reversed.
25      /// @param[out] result Stores the result
26      /// @param[in] operand Data on which to compute the NTT
27      /// @param[in] input_mod_factor Assume input operand are in [0,
28      /// input_mod_factor * q). Must be 1, 2 or 4.
29      /// @param[in] output_mod_factor Returns output operand in [0,
30      /// output_mod_factor * q). Must be 1 or 4.
31      void ComputeForward(uint64_t* result, const uint64_t* operand,
32                          uint64_t input_mod_factor, uint64_t output_mod_factor);
33
34      /// Compute inverse NTT. Results are bit-reversed.
35      /// @param[out] result Stores the result
36      /// @param[in] operand Data on which to compute the NTT
37      /// @param[in] input_mod_factor Assume input operand are in [0,
38      /// input_mod_factor * q). Must be 1 or 2.
39      /// @param[in] output_mod_factor Returns output operand in [0,
40      /// output_mod_factor * q). Must be 1 or 2.
41      void ComputeInverse(uint64_t* result, const uint64_t* operand,
42                          uint64_t input_mod_factor, uint64_t output_mod_factor);
43  }
```

Listing 1.2: Intel HEXL NTT class API

Listing 1.3 shows the API for the element-wise operations.

```
/// @brief Adds two vectors element-wise with modular reduction
/// @param[out] result Stores result
/// @param[in] operand1 Vector of elements to add. Each element must be less
/// than the modulus
/// @param[in] operand2 Vector of elements to add. Each element must be less
/// than the modulus
/// @param[in] n Number of elements in each vector
/// @param[in] modulus Modulus with which to perform modular reduction. Must be
/// in the range [2, 2^{63} - 1].
/// @details Computes operand1[i] = (operand1[i] + operand2[i]) mod modulus
/// for i=0, ..., n-1.
void EltwiseAddMod(uint64_t* result, const uint64_t* operand1,
                   const uint64_t* operand2, uint64_t n, uint64_t modulus);

/// @brief Computes fused multiply-add (arg1 * arg2 + arg3) mod modulus element-wise,
    broadcasting scalars to vectors.
/// @param[out] result Stores the result
/// @param[in] arg1 Vector to multiply
/// @param[in] arg2 Scalar to multiply
/// @param[in] arg3 Vector to add. Will not add if arg3 == nullptr
/// @param[in] n Number of elements in each vector
/// @param[in] modulus Modulus with which to perform modular reduction. Must be
/// in the range [2, 2^{61} - 1]
/// @param[in] input_mod_factor Assumes input elements are in [0,
/// input_mod_factor * q). Must be 1, 2, 4, or 8.
void EltwiseFMAMod(uint64_t* result, const uint64_t* arg1, uint64_t arg2,
                   const uint64_t* arg3, uint64_t n, uint64_t modulus,
                   uint64_t input_mod_factor);

/// @brief Multiplies two vectors element-wise with modular reduction
/// @param[in] result Result of element-wise multiplication
/// @param[in] operand1 Vector of elements to multiply. Each element must be
/// less than the modulus.
/// @param[in] operand2 Vector of elements to multiply. Each element must be
/// less than the modulus.
/// @param[in] n Number of elements in each vector
/// @param[in] modulus Modulus with which to perform modular reduction
/// @param[in] input_mod_factor Assumes input elements are in [0,
/// input_mod_factor * q) Must be 1, 2 or 4.
/// @details Computes result[i] = (operand1[i] * operand2[i]) mod modulus for i=0,
    ..., n - 1
void EltwiseMultMod(uint64_t* result, const uint64_t* operand1,
                    const uint64_t* operand2, uint64_t n, uint64_t modulus,
                    uint64_t input_mod_factor);
```

Listing 1.3: Intel HEXL free function API

Several of the functions have input arguments `input_mod_factor` or `output_mod_factor`. These allows for optimized implementations via lazy reduction. For instance, given two polynomials $f(x), g(x) \in \mathcal{R}_q$ represented using the coefficient embedding, we can compute $f * g$ by leaving the outputs of the forward NTT in the range $[0, 4q)$. This improves performance over the simplest choice of `input_mod_factor = 1, output_mod_factor = 1`.

```
/// Compute f(x) * g(x)
/// @param[in] N Size of input vectors
/// @param[in] p Modulus
VectorVectorMultMod(uint64_t* out, uint64_t* f, uint64_t* g, uint64_t N, uint64_t q)
    {
  NTT(N, p).ComputeForward(f, f, 1, 4);
  NTT(N, p).ComputeForward(g, g, 1, 4);
  EltwiseMultMod(out, f, g, N, q, 4);
  NTT(N, p).ComputeInverse(out, out, 1, 1);
}
```

Listing 1.4: Use of `input_mod_factor` to optimize vector-vector modular multiplication

Implementation The primary functionality of Intel HEXL is to provide optimized Intel AVX512-DQ and Intel AVX512-IFMA52 implementations for the forward and inverse NTT, element-wise vector-vector multiplication and element-wise vector-scalar multiplication. The Intel AVX512-IFMA52 implementations

are valid on prime numbers less than 50–52 bits, while the Intel AVX512-DQ implementations allow moduli up to ∼62 bits, where the exact conditions depend also on the input_mod_factor. The choice of which implementation is used is determined at runtime based on the CPU feature availability. While the current implementation always prefers Intel AVX512-IFMA52 implementations over Intel AVX512-DQ implementations (where the input conditions are met) over native implementation, a future optimization may be to determine the best implementation dynamically via a small number of trials upon initializing the library.

The implementation uses several Intel AVX512 helper functions, which are inlined for best performance. Several functions take a template argument, either 52 or 64, which is evaluated at compile time. These template parameters enable a unified implementation between primes less than 50–52 bits and primes larger than 52 bits, with no performance degradation. The following Intel AVX512 kernels are used in Intel HEXL, where we use m512i to refer to the __m512i datatype:

- m512i _mm512_hexl_mullo_epi<k>(m512i x, m512i y).
  Multiplies packed unsigned k-bit integers in each 64-bit element of x and y to perform a $2k$-bit intermediate result. Returns the low k-bit unsigned integer from the intermediate result. The implementation with $k = 64$ calls to _mm512_mullo_epi64. The implementation with $k = 52$ calls _mm512_madd52lo_epu64 with the accumulator set to zero.
- m512i _mm512_hexl_mullo_add_epi<k>(m512i x, m512i y, m512i z).
  Multiplies packed unsigned k-bit integers in each 64-bit element of y and z to perform a $2k$-bit intermediate result. Returns the low k-bit unsigned integer from the intermediate result added to the low k bits of x. The implementation with $k = 64$ requires one call to _mm512_mullo_epi64 and one call to _mm512_add_epi64. The implementation with $k = 52$ requires a single call to _mm512_madd52lo_epu64.
- m512i _mm512_hexl_mulhi_epi<k>(m512i x, m512i y).
  Multiplies packed unsigned k-bit integers in each 64-bit element of x and y to perform a $2k$-bit intermediate result. Returns the high k-bit unsigned integer from the intermediate result. The implementation with $k = 64$ requires two 32-bit shuffles, four 32-bit multiplies, three right shift, four 64-bit additions and one packed logical and operation. The implementation with $k = 52$ calls _mm512_madd52hi_epu64 with the accumulator set to zero.
- m512i _mm512_hexl_small_mod_epi64<k>(m512i x, m512i q, m512i* q_times_2, mm512i* q_times_4).
  Given packed unsigned 64-bit integers in x and q, with each integer $0 \leq x_i < k \cdot q_i$, where $k \in \{1, 2, 4, 8\}$, returns $x \mod q$. The implementation for $k = 2$ uses the fact that for unsigned integers $x < 2q$,

$$x \mod q = \begin{cases} x - q & x \geq q \\ x \end{cases} = \min(x - q, x),$$

which calls `_mm512_sub_epi64` once and `_mm512_min_epu64` once. For $k = 4$ and $k = 8$, $\log_2 k$ repeated calls are made to both `_mm512_sub_epi64` and `_mm512_min_epu64` which utilize the `q_times_2` ($k = 4, 8$) and `q_times_4` ($k = 8$)inputs, which are required not to be nullptr in these cases. For instance, Listing 1.5 shows the implementation when $k = 8$. The three statements map the input from the range $[0, 8q)$ to $[0, 4q)$, then to $[0, 2q)$, and finally to $[0, q)$.

```
1  // Fast computation of x mod q for x < 8q
2  __m512i _mm512_hexl_small_mod_epu64<8>(__m512i x, __m512i q, __m512i*
       q_times_2, __m512i* q_times_4) {
3    x = _mm512_min_epu64(x, _mm512_sub_epi64(x, *q_times_4));
4    x = _mm512_min_epu64(x, _mm512_sub_epi64(x, *q_times_2));
5    return _mm512_min_epu64(x, _mm512_sub_epi64(x, q));
6  }
```

Listing 1.5: Small-input modular reduction

– `m512i _mm512_hexl_cmpge_epu64(m512i x,m512i y, uint64_t v)`.
  Given packed unsigned 64-bit integers in x, y, returns a packed 64-bit integer with value v in each element for which $x > y$ and 0 otherwise. The implementation makes one call to `_mm512_maskz_broadcastq_epi64` and one call to `_mm512_cmpge_epu64_mask`.

4.1 NTT

Intel HEXL provides optimized Intel AVX512 implementations of the negacyclic number-theoretic transform (NTT) with bit-reversed outputs. At a high level, the implementation follows the radix-2 implementation from Cooley-Tukey and Gentleman-Sande, using the Harvey butterflies (see Section 2.2). In each case, the butterfly is implemented across all 8 lanes of an Intel AVX512 input vector of 64-bit integers.

Forward NTT The forward NTT is implemented using the Cooley-Tukey radix-2 transform in Algorithm 2. The key acceleration using Intel AVX512 is the loop in lines 9 to 14. Algorithm 6 shows the Harvey forward NTT butterfly implemented in Intel AVX512.

Inverse NTT The inverse NTT is implemented using the Gentleman-Sande radix-2 implementation from Algorithm 3. The key acceleration using Intel AVX512 is the loop in lines 9 to 14. Algorithm 7 shows the inverse Harvey NTT butterfly implemented in Intel AVX512. We make a few remarks on the implementations:

– The template argument `InputLessThanMod` enables a compile-time optimization in when the inputs X, Y are known to be less than q. For instance, when the input polynomial to the forward or inverse NTT has all coefficients less than q, `InputLessThanMod` is true during the first pass through the data.

Algorithm 6 Intel AVX512 Harvey NTT butterfly. β is the word size, with either $\beta = 2^{52}$ or $\beta = 2^{64}$.

Require: $q < \beta/4$; $0 < W < q$
Require: $W' = \lfloor W\beta/q \rfloor$, $0 < W' < \beta$
Require: $0 \le X, Y < 4q$
Require: twice_modulus contains $2q$ in all 8 lanes
Require: neg_modulus contains $-q$ in all 8 lanes
Ensure: $X \leftarrow X + WY \mod q; 0 \le Y < 4q$
Ensure: $Y \leftarrow X - WY \mod q; 0 \le Y < 4q$

1: **function** HarveyNTTButterfly<int BitShift, bool InputLessThanMod>(__m512i* X, __m512i* Y, __m512i W_op, __m512i W_precon, __m512i neg_modulus, __m512i twice_modulus)
2: **if** !InputLessThanMod **then**
3: *X = _mm512_hexl_small_mod_epu64(*X, twice_modulus);
4: **end if**
5: __m512i Q = _mm512_hexl_mulhi_epi<BitShift>(W_precon, *Y);
6: __m512i W_Y = _mm512_hexl_mullo_epi<BitShift>(W_op, *Y);
7: __m512i T = _mm512_hexl_mullo_add_epi<BitShift>(W_Y, Q, neg_modulus);
8: **if** BitShift == 52 **then**
9: T = _mm512_and_epi64(T, _mm512_set1_epi64((1UL « 52) - 1));
10: **end if**
11: __m512i twice_mod_minus_T = _mm512_sub_epi64(twice_modulus, T);
12: *Y = _mm512_add_epi64(*X, twice_mod_minus_T);
13: *X = _mm512_add_epi64(*X, T);
14: **end function**

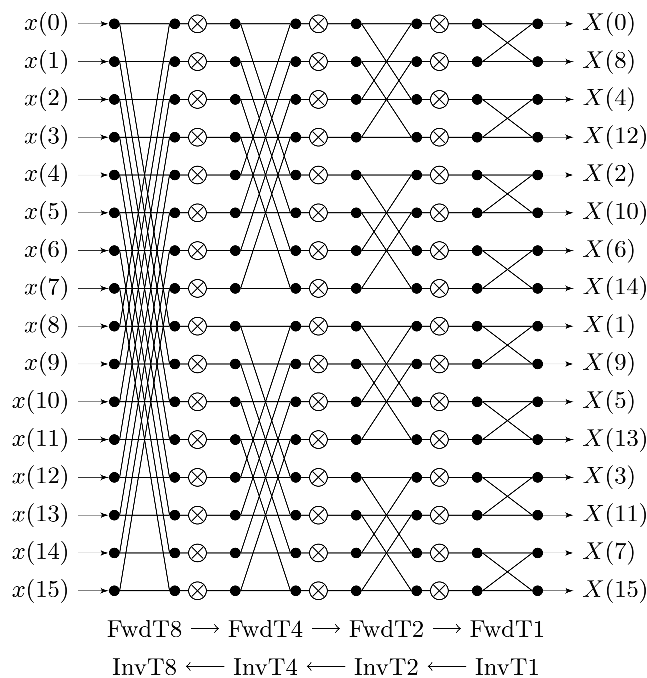

Fig. 1: NTT dataflow with root of unity factors omitted for clarity. The forward transform dataflow is from left to right. *FwdT8* refers to a case when the `for` loop in Lines 9–14 of Algorithm 2 runs for more than 8 iterations. Similarly, the *FwdT4* corresponds to a loop with 4 iterations, *FwdT2* corresponds to a loop with 2 iterations and *FwdT1* corresponds to a loop with 1 iteration. The inverse forward transform dataflow is from right to left, with *InvT8, InvT4, InvT2, InvT1* named analogously. Modified from [8].

Algorithm 7 Intel AVX512 Harvey Inverse NTT butterfly. β is the word size, with either $\beta = 2^{52}$ or $\beta = 2^{64}$.

Require: $q < \beta/4$; $0 < W < q$
Require: $W' = \lfloor W\beta/q \rfloor$, $0 < W' < \beta$
Require: $0 \leq X, Y < 2q$
Require: twice_modulus contains $2q$ in all 8 lanes
Require: neg_modulus contains $-q$ in all 8 lanes
Ensure: $X \leftarrow X + Y \mod q; 0 \leq Y < 2q$
Ensure: $Y \leftarrow W(X - Y) \mod q; 0 \leq Y < 2q$.

```
 1: function HarveyInvNTTButterfly<int BitShift, bool InputLessThanMod>(__m512i* X, __m512i* Y, __m512i W_op, __m512i W_precon, __m512i neg_modulus, __m512i twice_modulus)
 2:     __m512i Y_minus_2q = _mm512_sub_epi64(*Y, twice_modulus);
 3:     __m512i T = _mm512_sub_epi64(*X, Y_minus_2q);
 4:     if InputLessThanMod then
 5:         *X = _mm512_add_epi64(*X, *Y);
 6:     else
 7:         *X = _mm512_add_epi64(*X, Y_minus_2q);
 8:         __mmask8 sign_bits = _mm512_movepi64_mask(*X);
 9:         *X = _mm512_mask_add_epi64(*X, sign_bits, *X, twice_modulus);
10:     end if
11:     __m512i Q = _mm512_hexl_mulhi_epi<BitShift>(W_precon, T);
12:     __m512i Q_p = _mm512_hexl_mullo_epi<BitShift>(Q, neg_modulus);
13:     *Y = _mm512_hexl_mullo_add_epi<BitShift>(Q_p, W_op, T);
14:     if BitShift == 52 then
15:         T = _mm512_and_epi64(T, _mm512_set1_epi64((1UL « 52) - 1));
16:     end if
17: end function
```

- The negated modulus $-q$ is passed to the input of the forward and inverse butterflies. This enables the use of `_mm512_hexl_mullo_add_epi`, which is a single instruction when `BitShift` is 52. Note, the Intel AVX512-IFMA52 instruction set does not contain an `_mm512_msub52lo_epu64` instruction, which would enable using q instead of $-q$.
- Lines 8–10 in the forward butterfly and Lines 14–16 in the inverse butterfly clear the high 12 bits from `T`. This is required because the `_mm512_madd52lo_epu64` instruction uses a 64-bit accumulator, whereas our algorithm requires a 52-bit accumulator.
- In the inverse butterfly, rather than computing $Y_0 = X_0 + X_1; T = X_0 - X_1 + 2q$ (as presented in Algorithm 5 and [12]), we compute `Y_minus_2q = Y - 2q; T = X - Y_minus_2q`. This saves two scalar additions at the cost of one extra scalar subtraction.

The Intel AVX512 NTT butterflies are used in every stage of the NTT. The forward NTT `for` loop in Lines 9 - 14 of Algorithm 2 begins with $N/2$ iterations in the first stage, then $N/4$ iterations in the next stage, followed by successive divisions by 2 until the final loop runs for a single iteration in the final stage. As such, when the loop runs for 8 or more iterations, the use of Algorithm 6 is simple to apply (particularly since the number of loop iterations is divisible by 8). However, special consideration must be taken in the final three stages, denoted *FwdT4, FwdT2, FwdT1* in Figure 1 when the loop runs for 4 iterations, 2 iterations, and 1 iteration, respectively. In these cases, we permute the data within each Intel AVX512 unit such that the butterfly is still applied across all lanes.

Similarly, the inverse NTT `for` loop in Lines 9–14 of Algorithm 3 begins with 1 iteration in the first stage, 2 iterations in the next stage, followed by successive multiplications by 2 until the final loop runs for $N/2$ iterations in the final stage. Analogous to the forward NTT, for the first three stages, denoted *InvT1, InvT2, InvT4* in Figure 1, we permute the data within each Intel AVX512 unit such that the butterfly is still applied across all lanes. We note the primary benefit of Intel AVX512-IFMA52 is in the NTT on primes less than 50 bits, in which case `_mm512_hexl_mulhi<52>` via a single assembly call, whereas large primes require `_mm512_hexl_mulhi<64>`, which is much more expensive (see Section 2.3).

4.2 Polynomial kernels

For simplicity, in our presentation, we assume the degree of the input polynomials is divisible by 8. This is typically the case for our HE applications, in which the polynomials are of degree $N = 2^k \geq 1024$ a power of two. Nevertheless, the Intel HEXL implementation includes logic that processes the $N \mod 8$ remaining loop iterations.

Element-wise vector-vector multiplication Intel HEXL provides two AVX512 implementations of element-wise vector-vector multiplication: 1) an Intel AVX512-DQ implementation using integer logic; 2) an Intel AVX512-DQ implementation

using floating-point logic. For each implementation, we use a pre-processor directive to tune the loop unrolling factor for best performance. Each Intel AVX512 kernel implements SIMD modular multiplication across all 8 lanes of the Intel AVX512 data and is sequentially applied to each 512-bit chunk of the input data.

Intel AVX512-DQ integer implementation

The Intel AVX512-DQ integer implementation uses Algorithm 1. We choose $Q = \lfloor \log_2 q \rfloor + 1$ and $L = 63 + Q$. This choice has a few benefits. Firstly, this ensures $q > 2^{Q-1}$, which implies the Barrett factor $k = \lfloor 2^L/q \rfloor < 2^L/2^{Q-1} = 2^{64}$, i.e. it fits in a single 64-bit integer. Secondly, this choice ensures $L - Q + 1 = 64$, which implies c_3 is simply the high 64 bits of c_2 (see lines 3–4), i.e. the low 64 bits of c_2 do not need to be computed.

Note, since the input may be larger than q (when the `input_mod_factor` is larger than 1), the shifted product $d \gg (Q-1)$ may be larger than 2^{64}. To prevent overflow in this case (which may happen when q exceeds 59 bits), the inputs are first reduced to the range $[0, q)$ via a sequence of fixed-time conditional subtractions. Algorithm 8 shows the pseudocode for the Intel AVX512-DQ integer modular multiplication implementation.

Algorithm 8 VectorVectorModMulAVX512Int

Require: $q < 2^{62}$ stores the modulus in all 8 lanes
Require: $0 < X, Y <$ `InputModFactor` $\cdot\, q$
Require: InputModFactor $\cdot\, q < 2^{63}$
Require: `twice_q` stores $2q$ across all 8 lanes
Require: `barr_lo` stores $\lfloor 2^L/q \rfloor$ across all 8 lanes
Ensure: Returns $X \cdot Y \mod q$

```
 1: function EltwiseMultModAVX512Int<int BitShift, int InputModFactor>(__m512i X, __m512i Y, __m512i barr_lo, __m512i q, __m512i twice_q)
 2:     X = _mm512_hexl_small_mod_epu64<InputModFactor>(X, q, twice_q);
 3:     Y = _mm512_hexl_small_mod_epu64<InputModFactor>(Y, q, twice_q);
 4:     __m512i prod_hi = _mm512_hexl_mulhi_epi<64>(X, Y);
 5:     __m512i prod_lo = _mm512_hexl_mullo_epi<64>(X, Y);
 6:     __m512i c1 = _mm512_hexl_shrdi_epi64<BitShift - 1>(prod_lo, prod_hi);
 7:     __m512i c3 = _mm512_hexl_mulhi_epi<64>(c1, barr_lo);
 8:     __m512i c4 = _mm512_hexl_mullo_epi<64>(c3, q);
 9:     c4 = _mm512_sub_epi64(prod_lo, c4);
10:     __m512i result = _mm512_hexl_small_mod_epu64(c4, q);
11:     return result;
12: end function
```

Intel AVX512-DQ Floating-point implementation

For $q < 2^{50}$, Intel HEXL uses a floating-point Intel AVX512-DQ implementation. This implementation adapts Function 3.10 from Mathemagix [13] to Intel AVX512, in a similar manner as Fortin et al. [9]. Algorithm 9 shows the implementation for the Intel AVX512 floating-point kernel.

We make a few notes about the floating-point implementation:

Algorithm 9 VectorVectorModMulAVX512Float

Require: $0 < X, Y <$ InputModFactor $\cdot q$
Require: InputModFactor $\cdot q < 2^{52}$
Require: u stores 1/(double)q in each lane, rounding toward infinity, so that $u \geq 1/q$
Ensure: Returns $X \cdot Y \mod q$

```
1: function EltwiseMultModAVX512Float<int BitShift, int InputModFactor>(__m512i X, __m512i Y, __m512d u)
2:     const int rounding = _MM_FROUND_TO_POS_INF|_MM_FROUND_NO_EXC;
3:     __m512d xi = _mm512_cvt_roundepu64_pd( X, rounding);
4:     __m512d yi = _mm512_cvt_roundepu64_pd( Y, rounding);
5:     __m512d h = _mm512_mul_pd(xi, yi);
6:     __m512d l = _mm512_fmsub_pd(xi, yi, h);   ▷ rounding error; h + l == x * y
7:     __m512d b = _mm512_mul_pd(h, u);          ▷ ∼(x * y) / q
8:     __m512d c = _mm512_floor_pd(b);           ▷ ∼floor(x * y / q)
9:     __m512d d = _mm512_fnmadd_pd(c, q, h);
10:    __m512d g = _mm512_add_pd(d, l);
11:    __mmask8 m = _mm512_cmp_pd_mask(g, _mm512_setzero_pd(), _CMP_LT_OQ);
12:    g = _mm512_mask_add_pd(g, m, g, p);
13:    __m512i result = _mm512_cvt_roundpd_epu64(g, rounding);
14:    return result
15: end function
```

- The implementation is valid as long as InputModFactor $\cdot q < 2^{52}$. As such, there is no explicit modulus reduction step required for InputModFactor $\in \{1, 2, 4\}$.
- We experimented with several Intel AVX512-IFMA52 implementations. However, we found this Intel AVX512 floating-point implementation yields best performance.

Element-wise vector-scalar multiplication The `EltwiseFMAMod` function implements vector-scalar modular multiplication, with an additional optional scalar modular addition. The Intel AVX512-DQ and Intel AVX512-IFMA52 implementations use the same underlying kernel, with the Intel AX512-DQ kernel using `BitShift = 64` and the Intel AVX512-IFMA52 kernel using `BitShift = 52`. The Intel AVX512-IFMA52 kernel is valid for InputModFactor $\cdot q < 2^{52}$, while the Intel AVX512-DQ kernel is valid for `InputModFactor` $\cdot q < 2^{62}$. Algorithm 10 shows the Algorithm for vector-scalar multiplication. Compared to the vector-vector multiplication algorithm, the vector-scalar multiplication algorithm performs additional pre-computation using the scalar factor. Where required, Lines 2 and 3 perform conditional subtractions to reduce the input to the range $[0, q)$.

Algorithm 10 EltwiseFMAModAVX512

Require: $0 < X, Z <$ InputModFactor $\cdot\, q$
Require: $0 < Y < q$
Require: $Y_{\text{barr}} = \lfloor y \ll \text{BitShift}/q \rfloor$
Require: InputModFactor $\cdot\, q < 2^{\text{BitShift}}$
Require: q stores the modulus across all 8 lanes
Ensure: Returns $X \cdot Y + Z \mod q$

```
 1: function EltwiseFMAModAVX512<int BitShift, int InputModFactor>(__m512i X, __m512i Y, __m512i Y_barr, __m512i Z, __m512i q)
 2:     X = _mm512_hexl_small_mod_epu64<InputModFactor>(X, q);
 3:     Z = _mm512_hexl_small_mod_epu64<InputModFactor>(Z, q);
 4:     __m512i XY = _mm512_hexl_mullo_epi<64>(X, Y);
 5:     __m512i R = _mm512_hexl_mulhi_epi<BitShift>(X, Y_barr);
 6:     __m512i Rq = _mm512_mullo_epi64(R, q);
 7:     R = _mm512_sub_epi64(XY, Rq);
 8:     R = _mm512_hexl_small_mod_epu64(R, q);        ▷ Conditional Barrett subtraction
 9:     R = _mm512_add_epi64(vq, Z);
10:     R = _mm512_hexl_small_mod_epu64(R, q);
11:     return R
12: end function
```

5 Results

We benchmark each kernel on a 3rd Gen Intel Xeon® Scalable Processors Platinum 8360Y 2.4GHz processor with 64GB of RAM and 72 cores, running the Ubuntu 20.04 operating system. The code is compiled using clang-10 with the '-march=native -O3' optimization flags. We run each kernel for 10 seconds and report the average execution time.

NTT We benchmark the performance of the forward and inverse NTT in three settings: 1) the native C++ implementation; 2) the Intel AVX512-DQ implementation; 3) the Intel AVX512-IFMA52 implementation. The default implementation uses the radix-2 Cooley-Tukey (forward NTT) and Gentleman-Sande (inverse NTT) formulations, using the Harvey butterfly (see Section 2.2). Additionally, we compare against NTL v11.4.3 [19], an open-source number theory library. NTL is compiled with clang-10 using the *NTL_ENABLE_AVX_FFT* flag, which enables an experimental Intel AVX512 implementation using floating-point arithmetic.

We measure the performance on three different input sizes: $N = 1024, 4096, 16384$. Table 1 shows the runtimes for the forward transform. The Intel AVX512-DQ implementation provides a significant 2.7x–2.8x speedup over the native implementation, with the Intel AVX512-IFMA52 increasing this speedup to 7.2x for the smallest size. The AVX512-IFMA52 implementation speedup decreases to 5.3x for the larger $N = 16384$ case as L1 cache misses bottleneck the memory access. NTL's implementation uses floating-point arithmetic for integer computation,

and is therefore correct only for primes up to 50 bits. As such, while NTL may provide best performance on systems without Intel AVX512-IFMA52, no additional speedup is expected on NTL on systems with the Intel AVX512-IFMA52 instruction set.

Table 1: Single-threaded, single-core runtime in microseconds of the forward NTT with `input_mod_factor` = `output_mod_factor` = 1.

Implementation	N / Speedup					
	1024		4096		16384	
Native C++	9.08	1.0x	38.8	1.0x	177	1.0x
Intel AVX512-DQ	3.26	2.7x	13.4	2.8x	62.3	2.8x
NTL	2.44	3.7x	8.48	4.5x	40.2	4.3x
Intel AVX512-IFMA52	1.25	7.2x	5.81	6.6x	33.1	5.3x

Table 2 shows the runtimes for the inverse NTT. We see the Intel AVX512-DQ implementation provides a similar speedup of 2.5x–2.6x over the native implementation. The Intel AVX512-IFMA52 implementation improves this speedup to 6.7x on the smaller transforms, which diminishes to 5.3x on the largest transform. As with the forward NTT, the speedup on the larger transforms is diminished due to L1 cache misses. As with the forward transform, while NTL may provide best performance on systems without Intel AVX512-IFMA52, no additional speedup is expected on NTL on systems with the Intel AVX512-IFMA52 instruction set.

Table 2: Single-threaded, single-core runtime in microseconds of the inverse NTT with `input_mod_factor` = `output_mod_factor` = 1.

Implementation	N / Speedup					
	1024		4096		16384	
Native C++	8.25	1.0x	37.8	1.0x	174	1.0x
Intel AVX512-DQ	3.16	2.6x	14.6	2.5x	68.2	2.5x
NTL	2.12	3.8x	9.05	4.1x	42.4	4.1x
Intel AVX512-IFMA52	1.23	6.7x	5.72	6.6x	32.4	5.3x

Polynomial We benchmark the performance of the element-wise vector-vector and vector-scalar modular multiplication kernels. For element-wise vector-vector modular multiplication, we compare three implementations: 1) the native C++ implementation; 2) the Intel AVX512-DQ integer implementation; 3) the Intel

AVX512-DQ floating-point implementation. As with the NTT, we consider three input sizes: $N = 1024, 4096, 16384$.

Table 3 shows the runtimes for the element-wise vector-vector modular multiplication. The Intel AVX512-DQ integer implementation provides a 1.5x–1.9x speedup over the native implementation, which increases to 5.1x–6.0x with the Intel AVX512-DQ floating-point implementation.

Table 3: Single-threaded, single-core runtime in microseconds of element-wise vector-vector modular multiplication with `input_mod_factor` = 1.

Implementation	N / Speedup					
	1024		4096		16384	
Native C++	1.51	1.0x	5.71	1.0x	23.6	1.0x
Intel AVX512-DQ Int	0.982	1.5x	3.43	1.6x	12.3	1.9x
Intel AVX512-DQ Float	0.251	6.0x	1.08	5.2x	4.58	5.1x

For the element-wise vector-scalar modular multiplication kernel, we compare three implementations: 1) the native C++ implementation; 2) the Intel AVX512-DQ implementation; 3) the Intel AVX512-IFMA52 implementation.

Table 4 shows the runtimes for the element-wise vector-scalar modular multiplication with scalar addition. The Intel AVX512-DQ implementation no significant speedup over the native implementation, as the compiler's auto-vectorizer does a sufficient job using Intel AVX instructions. The Intel AVX512-IFMA52 implementation provides a moderate 1.7x speedup over the native implementation.

Table 4: Single-threaded, single-core runtime in microseconds of element-wise vector-scalar modular multiplication with scalar addition and `input_mod_factor` = 1.

Implementation	N / Speedup					
	1024		4096		16384	
Native C++	0.53	1.0x	2.11	1.0x	9.01	1.0x
Intel AVX512-DQ	0.53	1.0x	2.11	1.0x	9.01	1.0x
Intel AVX512-IFMA52	0.302	1.7x	1.20	1.7x	5.08	1.7x

6 Conclusion

Here, we introduced Intel HEXL, a C++ library using the Intel AVX512 instruction set to accelerate key primitives in lattice cryptography. Intel HEXL provides optimized implementations of the number-theoretic transform (NTT) and polynomial operations, including element-wise vector-vector modular multiplication and element-wise vector-scalar modular multiplication. The Intel AVX512-DQ instruction set is used to accelerate the operations for a wide range of word-sized primes, up to 62 bits. The recent Intel AVX512-IFMA52 extension to the Intel AVX512 instruction set further improves performance for primes less than 50–52 bits. In particular, the Intel AVX512-IFMA52 instructions yield up to 7.2x and 6.7x speedup over a native C++ implementation of the forward and inverse NTT, respectively. The Intel AVX512-DQ floating-point implementation of element-wise modular multiplication yields up to 6.0x speedup over the native C++ implementation, while the Intel AVX512-IFMA52 implementation of element-wise vector-scalar modular multiplication yields 1.7x speedup over the native C++ implementation. The Intel HEXL library is available open-source at https://github.com/intel/hexl under the Apache 2.0 license.

Future work improving Intel HEXL includes exploring additional NTT implementations, such as higher-radix implementations. In particular, higher-radix NTT implementations may reduce the memory pressure, which currently bottlenecks the larger-size NTT performance, as observed in [14]. We also plan to integrate Intel HEXL with open-source homomorphic encryption libraries, as well as expand the API to encompass a larger variety of applications. In particular, adding a programming model which enables compiling several Intel HEXL kernels may enable compiler optimizations such as loop fusion, which may improve performance and usability.

Acknowledgement

We would like to thank Ilya Albrekht for guidance on the AVX512 implementation.

References

1. Aguilar-Melchor, C., Barrier, J., Guelton, S., Guinet, A., Killijian, M.O., Lepoint, T.: Nfllib: Ntt-based fast lattice library. In: Cryptographers' Track at the RSA Conference. pp. 341–356. Springer (2016)
2. Bergamaschi, F., Halevi, S., Halevi, T.T., Hunt, H.: Homomorphic training of 30,000 logistic regression models. In: International Conference on Applied Cryptography and Network Security. pp. 592–611. Springer (2019)
3. Blatt, M., Gusev, A., Polyakov, Y., Goldwasser, S.: Secure large-scale genome-wide association studies using homomorphic encryption. Proceedings of the National Academy of Sciences **117**(21), 11608–11613 (2020)
4. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Mathematics of computation **19**(90), 297–301 (1965)
5. Corporation, I.: Intel intrinsics guide, https://software.intel.com/sites/landingpage/IntrinsicsGuide/#avx512techs=AVX512IFMA52
6. Drucker, N., Gueron, S.: Fast modular squaring with avx512ifma. In: 16th International Conference on Information Technology-New Generations (ITNG 2019). pp. 3–8. Springer (2019)
7. Edamatsu, T., Takahashi, D.: Accelerating large integer multiplication using intel avx-512ifma. In: International Conference on Algorithms and Architectures for Parallel Processing. pp. 60–74. Springer (2019)
8. Fauske, K.M.: Texample.net, https://texample.net/tikz/examples/radix2fft/
9. Fortin, P., Fleury, A., Lemaire, F., Monagan, M.: High performance simd modular arithmetic for polynomial evaluation. arXiv preprint arXiv:2004.11571 (2020)
10. Géraud, R., Maimuţ, D., Naccache, D.: Double-speed barrett moduli. In: The New Codebreakers, pp. 148–158. Springer (2016)
11. Gueron, S., Krasnov, V.: Accelerating big integer arithmetic using intel ifma extensions. In: 2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH). pp. 32–38. IEEE (2016)
12. Harvey, D.: Faster arithmetic for number-theoretic transforms. Journal of Symbolic Computation **60**, 113–119 (2014)
13. Hoeven, J.V.D., Lecerf, G., Quintin, G.: Modular simd arithmetic in mathemagix. ACM Transactions on Mathematical Software (TOMS) **43**(1), 1–37 (2016)
14. Jung, W., Lee, E., Kim, S., Lee, K., Kim, N., Min, C., Cheon, J.H., Ahn, J.H.: Heaan demystified: Accelerating fully homomorphic encryption through architecture-centric analysis and optimization. arXiv preprint arXiv:2003.04510 (2020)
15. Kocabas, O., Soyata, T.: Towards privacy-preserving medical cloud computing using homomorphic encryption. In: Virtual and Mobile Healthcare: Breakthroughs in Research and Practice, pp. 93–125. IGI Global (2020)
16. Longa, P., Naehrig, M.: Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In: International Conference on Cryptology and Network Security. pp. 124–139. Springer (2016)
17. Morshed, T., Al Aziz, M.M., Mohammed, N.: Cpu and gpu accelerated fully homomorphic encryption. In: 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 142–153. IEEE (2020)
18. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL (Nov 2020), microsoft Research, Redmond, WA.
19. Shoup, V., et al.: Ntl: A library for doing number theory (2001)