Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52

Fabian Boemer
fabian.boemer@intel.com
Intel Corporation
Santa Clara, CA, USA

Sejun Kim
sejun.kim@intel.com
Intel Corporation
Hillsboro, Oregon, USA

Gelila Seifu
gelila.seifu@intel.com
Intel Corporation
Santa Clara, CA, USA

Fillipe D. M. de Souza
fillipe.souza@intel.com
Intel Corporation
Folsom, CA, USA

Vinodh Gopal
vinodh.gopal@intel.com
Intel Corporation
Hudson, MA, USA

ABSTRACT
Modern implementations of homomorphic encryption (HE) rely heavily on polynomial arithmetic over a finite field. This is particularly true of the BGV, BFV, and CKKS 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 single-threaded speedup, respectively, over a native C++ implementation. Intel HEXL also provides up to 6.0x single-threaded speedup on the element-wise vector-vector modular multiplication, and 1.7x single-threaded 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 and has been adopted by the Microsoft SEAL and PALISADE homomorphic encryption libraries.

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, 19]. However, HE currently suffers from enormous memory and runtime overheads of up to 30,000x [18].

Ciphertexts in many HE schemes, including BGV [4], BFV [11], and CKKS [5, 6], are polynomials in finite fields, whose coefficients can be hundreds of bits and whose degree is typically a power in the range [210, 217]. 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 4.0.1) and implementation (Section 4.0.2) of Intel HEXL. In particular, we provide detailed descriptions of the forward (Section 4.1.1) and inverse (Section 4.1.2) NTT and polynomial kernels (Section 4.2), including vector-vector modular multiplication (Section 4.2.1) and vector-scalar modular multiplication (Section 4.2.2). In Section 5, we describe the integration of Intel HEXL to two public HE libraries, Microsoft SEAL [23] and PALISADE [22]. We next demonstrate the performance of Intel HEXL in Section 6, 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 7.

2 BACKGROUND
We provide a brief background of the algorithms optimized used in Intel HEXL, which are common building blocks of lattice cryptography. Let \( R_q = \mathbb{Z}_q[X]/(X^N + 1) \) 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 $c = a \odot b$ such that $c_i = (a_i \cdot b) \mod q$.
- **Vector-vector multiplication.** Given $a, b \in \mathcal{R}_q$, compute $c = a \ast b$ such that

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

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 modulo operator % is used for modular reduction. Listing 1 shows C/C++ source code for a simple implementation of scalar modular multiplication.

```c
// Returns \((a \ast b) \mod q\)
uint64_t naive_modmul(uint64_t a, uint64_t b, uint64_t q) {
    return uint64_t((uint128_t(a) * b) % q);
}
```

### 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 are performed with respect to the modulus $q$. Let $\omega$ 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 $\tilde{a} = NTT(a)$, where

$$\tilde{a}_i = \sum_{j=1}^{N-1} a_{j \odot \omega^i} \mod q \text{ for } i = 0, 1, \ldots, N-1.$$

As described in [20], 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 $\psi$ be a primitive 2N'th root of unity in $\mathbb{Z}_q$. Let $a, b \in \mathbb{Z}_q^N$ and $\tilde{a} = (a_0, \psi a_1, \psi^2 a_2, \ldots, \psi^{N-1} a_{N-1}), \tilde{b} = (b_0, \psi b_1, \psi^2 b_2, \ldots, \psi^{N-1} b_{N-1})$. Then, the negacyclic NTT is defined as

$$c = (1, \psi^{-1}, \ldots, \psi^{-(N-1)}) \odot \text{InoNTT}(\text{NTT}(\tilde{a}) \odot \text{NTT}(\tilde{b}))),$$

which satisfies $c = a \ast b$ in $\mathcal{R}_q$. The NTT-based formulation reduces the runtime of polynomial-polynomial 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 [7]. 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 $O(N^2)$ subtraction to guarantee correctness. Nevertheless, replacing integer division with bit shifting results in a speedup. Algorithm 1 shows Barrett’s algorithm, as presented in [14].

**Algorithm 1 Barrett Reduction**

**Require:** $q < 2^Q, d < 2^D, k = \lfloor \frac{q}{D} \rfloor$, with $Q \leq D \leq L$

**Ensure:** Returns $d \mod q$

1: function BARRETT REDUCTION(d, q, k, Q, L)
2:    c₁ ← d \mod (Q - 1)
3:    c₂ ← c₁k
4:    c₃ ← c₂ \mod (L - Q + 1)
5:    c₄ ← d - qc₃
6:    if c₄ ≥ q then
7:        c₄ ← c₄ - q
8:    end if
9:    return c₄
10: end function
runtime of a naive implementation. The choice of \(\min(N_1, N_2)\) determines the \textit{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

\[
i = 0b_{i_0}i_1 \ldots i_{\log_2(N)} \in \{0, 1\}^\log_2(N),
\]

the output of the Cooley-Tukey NTT at index \(i\)

\[
\text{NTT}(a)[0b_{i_0}i_1 \ldots i_{\log_2(N)} i_{\log_2(N)-1} \ldots i_{i_0}].
\]

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 [20]. Algorithm 3 shows the analogous radix-2 in-place Gentleman-Sande inverse NTT algorithm.

**Algorithm 2 Cooley-Tukey Radix-2 NTT**

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

**Ensure:** \(a \leftarrow \text{NTT}(a)\) in bit-reversed order.

1. function COOLEY-TUKEY RADIX-2 NTT(a, N, q, \(\psi_{\text{rev}}\))
   1. \(t \leftarrow n\)
   2. for \((m = 1; m < n; m = 2n)\) do
   3. \(t \leftarrow t/2\)
   4. for \((i = 0; i < m; i++)\) do
   5. \(j_i \leftarrow 2 \cdot i + t\)
   6. \(j_2 \leftarrow j_1 + t - 1\)
   7. \(W \leftarrow \psi_{\text{rev}}[m + i]\)
   8. for \((j = j_1; j \leq j_2; j++)\) do
   9. \(X_0 \leftarrow a_j\)
   10. \(X_1 \leftarrow a_{j+t}\)
   11. \(a_j \leftarrow X_0 + W \cdot X_1 \mod q\)
   12. \(a_{j+t} \leftarrow X_0 - W \cdot X_1 \mod q\)
   13. end for
   14. end for
   15. end for
   16. return a
   17. end function

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

\[
(X_0, X_1) \leftrightarrow (X_0 + WX_1 \mod q, X_0 - WX_1 \mod q).
\]

Harvey [16] 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 butterfly1. Using the Harvey butterfly in the Cooley-Tukey NTT yields outputs in \(\mathbb{Z}_q^N\) so an additional correction step is required to reduce the output to \(\mathbb{Z}_q^N\). Similar to the forward transform, Harvey [16] also provides an efficient butterfly

\[
(0, 2q).
\]

for the inverse NTT, using a redundant representation in \([0, 2q)\). Algorithm 5 shows the Harvey inverse NTT butterfly.

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

1Note. [16] presents Algorithm 2 as the inverse butterfly, whereas Intel HEXL 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 5 Harvey inverse NTT butterfly. $\beta$ is the word size, e.g. $\beta = 2^{64}$ on typical modern CPU platforms.

**Require:** $q < \beta / 4; 0 < W < q$
**Require:** $W' = \lceil W / \beta \rceil; 0 < W' < \beta$
**Require:** $0 \leq X_0, X_1 < 2^q$

**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:     $y_0 \leftarrow x_0 + x_1$
3:     if $y_0 \geq 2q$ then
4:         $y_0 \leftarrow y_0 - 2q$
5:     end if
6:     $T \leftarrow X_0 - X_1 + 2q$
7:     $Q \leftarrow \lfloor W'T / \beta \rfloor$
8:     $y_1 \leftarrow (WT - Qy) \mod \beta$
9:     return $y_0, y_1$
10: end function

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 \texttt{__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, \texttt{‘epi’} refers to extended packed integer and \texttt{‘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:

\texttt{__m512i__mm512_mullo_epi64 \quad (__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 \texttt{__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 [18].

The Intel AVX512-IFMA52 extension to Intel AVX512 [8] consists of several operations useful in lattice cryptography. In particular, Intel AVX512-IFMA52 introduces the following intrinsics:

\texttt{__m512i__mm512_madd52lo_epi64 \quad (__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.

\texttt{__m512i__mm512_madd52hi_epi64 \quad (__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-VMID2) instruction set:

\texttt{__m512i__mm512_shrdi_epi64 \quad (__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 [17] provides Intel AVX-accelerated implementations of modular integer arithmetic using a SIMD programming model. NFLib [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 NFLib consider Intel AVX512 implementations using the Intel AVX512-IFMA52 instruction set.

Previous work [10, 15] using Intel AVX512-IFMA52 focuses on accelerating big integer multiplication without modular multiplication. Drucker and Gueron [9] 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

4.0.1 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 is typical of HE implementations on CPU. SEAL [23] for instance, bounds all coefficient moduli to 61 bits. GPUs implementations of HE, (e.g. [18, 21]) 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 [24] or NFLib [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 2 shows the application programming interface (API) for the NTT.

```java
class NTT {
  public:
    // Initializes an empty NTT object
```
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 \( f(x), g(x) \in R_q \) represented using the coefficient embedding, we can compute \( f \ast g \) by leaving the outputs of the forward NTT in the range \([0, 4g)\). This improves performance over the simplest choice of `input_mod_factor = 1, output_mod_factor = 1`.

### Listing 4: Use of `input_mod_factor` to optimize vector-vector modular multiplication

#### 4.0.2 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 moduli 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`. Intel HEXL also provides a reference native C++ implementation for each kernel, which has reduced performance but ensures Intel HEXL is compatible with non-AVX512 processors. The choice of implementation is determined at runtime based on the CPU feature availability. While currently Intel HEXL always prefers Intel AVX512-IFMA52 implementations over Intel AVX512-DQ implementations over native implementations, a future optimization may be to determine the best implementation dynamically via a small number of trials upon initializing the library. This would ensure performance is always at least on par with the native C++ implementation, which may be efficient enough for the compiler performing auto-vectorization.

Intel HEXL uses several general optimizations in the implementation. Loops are unrolled either manually or using a pre-processor directive, with a manually-tuned unrolling factor. Within manually-unrolled loops, instructions are reordered where possible for best pipelining. Some loops are hand-coded for common input vector lengths, e.g., \( N = 8192, 16384 \). As much as possible, cross-lane dependencies such as shuffles and permutations are avoided. Where applicable, memory is allocated to 64-byte boundaries to improve loads and stores. For best performance, the user input to Intel HEXL functions should also be aligned to 64-byte boundaries.

Intel HEXL 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 __m512i_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.

### Listing 3: Intel HEXL free function API

#### Listing 3:

```
Listing 3 shows the API for the element-wise operations.

```

```c
Listing 2: Intel HEXL NTT class API

Listing 2: Intel HEXL NTT class API

```c
Listing 3 shows the API for the element-wise operations.

```

```c
Listing 3: Intel HEXL free function API

Listing 3: Intel HEXL free function API

```c
```
Returns the low k-bit unsigned integer from the intermediate result. The implementation with \( k = 64 \) calls to \texttt{-mm512_mullo_epi64}. The implementation with \( k = 52 \) calls \texttt{-mm512_madd52lo_epi64} with the accumulator set to zero.

- \texttt{m512i _mm512_hexl_mullo_add_epi<k>(m512i x, m512i y, m512i z)}.

Multiples 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 added to the low k bits of \( x \). The implementation with \( k = 64 \) requires one call to \texttt{-mm512_mullo_epi64} and one call to \texttt{-mm512_add_epi64}. The implementation with \( k = 52 \) requires a single call to \texttt{-mm512_madd52lo_epi64}.

- \texttt{m512i _mm512_hexl_mulli_epi<k>(m512i x, m512i y)}.

Multiples 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 multiples, three right shift, four 64-bit additions and one packed logical and operation. The implementation with \( k = 52 \) calls \texttt{-mm512_madd52hi_epi64} with the accumulator set to zero.

- \texttt{m512i _mm512_hexl_small_mod_epi64<k>(m512i x, m512i q, m512i* q_times_2, m512i* 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 \( k \in \{2,4,8\} \), returns \( x \mod q \). The implementation for \( k = 2 \) uses the fact that for unsigned integers \( x < 2q \),

\[
 x \mod q = \begin{cases} 
 x - q & x \geq q \\
 0 & x < q 
\end{cases} = \min(x - q, 0),
\]

which calls \texttt{-mm512_sub_epi64} once and \texttt{-mm512_min_epi64} once. For \( k = 4 \) and \( k = 8 \), \( \log_2 k \) repeated calls are made to both \texttt{-mm512_sub_epi64} and \texttt{-mm512_min_epi64} which utilize the \texttt{q_times_2} \((k = 4,8)\) and \texttt{q_times_4} \((k = 8)\) inputs, which are required not to be nullptr in these cases. For instance, Listing 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 \([0,q)\).

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.

4.1.1 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.

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

**Require:** \( q < \beta /4, 0 < W < q \)

- \texttt{W = \[W/\beta\], 0 < W' < \beta}

- \texttt{0 \leq X, Y < 4q}

- \texttt{twice_modulus contains 2q in all 8 lanes}

- \texttt{neg_modulus contains \(-q\) in all 8 lanes}

- \texttt{Ensure: X \rightarrow X + W \mod q \leq Y < 4q}

- \texttt{Ensure: Y \rightarrow Y - W \mod q : 0 \leq Y < 4q}

- \texttt{InputLessThanMod then}

- \texttt{if InputLessThanMod then}

- \texttt{\( X \rightarrow \_mm512_hexl_small_mod_epi64(\leftarrow X, \_twice_modulus) \)}

- \texttt{end if}

- \texttt{end if}

- \texttt{if BitShift == 25 then}

- \texttt{end if}

- \texttt{end if}

- \texttt{end if}

- \texttt{end if}

- \texttt{end if}

- \texttt{end if}

- \texttt{end if}

- \texttt{end if}

- \texttt{end if}

1. \texttt{function HarveyNTTButterfly<\texttt{<BitShift, bool InputLessThanMod<\texttt{<W_precon, \_m512i neg_modulus, \_m512i twice_modulus>}}}}

2. \texttt{\_m512i Q = \_mm512_hexl_mullo_epi-BitShift(W_precon, Y);}

3. \texttt{\_m512i T = \_mm512_hexl_mullo_add_epi-BitShift(W, Y, Q, neg_modulus);}

4. \texttt{if BitShift == 25 then}

5. \texttt{end if}

6. \texttt{m512i twice_mod_minus_T = \_mm512_sub_epi64(twice_modulus, T);}

7. \texttt{\_m512i W_Y = \_mm512_hexl_mullo_epi-BitShift(W, Y, T);}

8. \texttt{\_m512i W_precon = \_mm512_hexl_mullo_epi-BitShift(W, Y, Q, neg_modulus);}

9. \texttt{\_m512i T = \_mm512_hexl_mullo_add_epi-BitShift(W, Y, Q, neg_modulus);}

10. \texttt{\_m512i W_precon = \_mm512_hexl_mullo_epi-BitShift(W, Y, T);}

11. \texttt{\_m512i W_Y = \_mm512_hexl_mullo_epi-BitShift(W, Y, T);}

12. \texttt{\_m512i T = \_mm512_hexl_mullo_add_epi-BitShift(W, Y, T);}

13. \texttt{\_m512i T = \_mm512_hexl_mullo_add_epi-BitShift(W, Y, T);}

14. \texttt{\_m512i T = \_mm512_hexl_mullo_add_epi-BitShift(W, Y, T);}

4.1.2 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 \texttt{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 \( q \), \texttt{InputLessThanMod} is true during the first pass through the data.

- The negated modulus \(-q\) is passed to the input of the forward and inverse butterflies. This enables the use of \texttt{-mm512_hexl_mullo_add_epi}, which is a single instruction when \texttt{BitShift} is 52. Note, the Intel AVX512-IFMA52 instruction set does not contain an \texttt{-mm512_msub52lo_epi64} 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

Listing 5: Small-input modular reduction

- \texttt{m512i _mm512_hexl_cmpge_epi64(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 \texttt{-mm512_maskz_bROADCASTq_epi64} and one call to \texttt{-mm512_cmpge_epi64}.
because the \texttt{_mm512_madd521lo_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 [16]), we compute $Y_{\text{min}} - 2q = Y - 2q$; $T = X - Y_{\text{min}} - 2q$. 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 $\text{FwdT4}, \text{FwdT2}, \text{FwdT1}$ 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 $\text{InvT1}, \text{InvT2}, \text{InvT4}$ 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-IMMA52 in the NTT on primes less than 50 bits, in which case \texttt{_mm512_hexl_mulhi<64>} via a single assembly call, whereas large primes require \texttt{_mm512_hexl_mulhi<64>}, which is much more expensive (see Section 2.3).

\section{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.

\subsection{4.2.1 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.

\section{Intel AVX512-DQ integer implementation}

The Intel AVX512-DQ integer implementation uses Algorithm 1. We choose $Q = \lceil \log_2 q \rceil + 1 = 63 + Q$. This choice has a few benefits. Firstly, this ensures $q > 2^{Q-1}$, which implies the Barrett factor $k = 2^L / q < 2^L / 2^{Q-1} = 2^{64}$, i.e. it fits in a single 64-bit integer. Secondly, this choice ensures $L - Q + 1 = 64$, which implies $c_1$ 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 - $q$
Require: $\text{InputModFactor} \cdot q < 2^{63}$
Require: twice_{q} stores 2q across all 8 lanes
Require: barr_lo stores $\lfloor 2^{62}/q \rfloor$ across all 8 lanes

Ensure: Returns $X \cdot Y \mod q$

1: function EltwiseMultiplyModAVX512Int<int BitShift, int InputModFactor>::(_m512 X, _m512 Y, _m512 barr_lo, _m512i q, _m512i twice_{q})
2: $X = _\text{mm512 hexl small_mod_epu64}\cdot$InputModFactor$= (X \cdot q \cdot \text{twice}_{q})$
3: $Y = _\text{mm512 hexl small_mod_epu64}\cdot$InputModFactor$= (Y \cdot q \cdot \text{twice}_{q})$
4: $\text{m512 prod hi} = _\text{mm512 hexl multl_epi64}(X, Y)$
5: $\text{m512 prod lo} = _\text{mm512 hexl multl_epi64}(X, Y)$
6: $\text{c4} = _\text{mm512 hexl small_mod_epu64}(\text{ InputModFactor}, q, twice_{q})$
7: $c4 = _\text{mm512 sub_epi64}(\text{prod}_{\text{lo}}, c4)$
8: \text{ result} = _\text{mm512 hexl small_mod_epu64}(c4, q);  
9: \text{return result}; 

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 [17] to Intel AVX512, in a similar manner as Fortin et al. [13]. Algorithm 9 shows the implementation for the Intel AVX512-floating point kernel.

Algorithm 9 VectorVectorModMulAVX512Float

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

1: function EltwiseFloatModAVX512Float<int BitShift, int InputModFactor>::(_m512 X, _m512 Y, _m512 u)
2: $\text{const int rounding} = _\text{MM\_FROUND\_POS, INFINITY, MM\_FROUND\_NO\_EXC}$
3: $\text{m512d x} = _\text{mm512 cvt_roundpd4 pdl}(X, \text{rounding})$
4: $\text{m512d y} = _\text{mm512 cvt_roundpd4 pdl}(Y, \text{rounding})$
5: $\text{m512d h} = _\text{mm512 mul pdl(x, y)}$
6: $\text{m512d l} = _\text{mm512 fsub pdl}(), y, h)$
7: $\text{m512d b} = _\text{mm512 mod pdl(h)}$
8: $\text{m512d c} = _\text{mm512 floor pdl(b)}$
9: $\text{m512d d} = _\text{mm512 fmod pdl(c, q, h)}$
10: $g = _\text{mm512 add pdl(d)}$
11: $\text{mmask} = m = _\text{mm512 cmp pdl(g, _\text{mm512 setzero pdl})}$
12: $g = _\text{mm512 mask add pdl, g, m, p)$
13: $\text{c512 result} = _\text{mm512 cvt_roundpd epus64}(g, \text{rounding})$
14: $\text{return result}$
15: end function

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

- 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.

4.2.2 Element-wise Vector-Scalar Multiplication
The EltwiseFMAMod function implements vector-scalar modular multiplication, with an additional optional scalar modulus addition. The Intel AVX512-DQ and Intel AVX512-IFMA52 implementations use the same underlying kernel, with the Intel AVX512-DQ kernel using BitShift_t = 64 and the Intel AVX512-IFMA52 kernel using BitShift_t = 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 - $q$
Require: $0 < Y < q$
Require: $V_{\text{rem}} = \text{BitShift} \cdot q$
Require: $\text{InputModFactor} \cdot q < 2^{s}$
Ensure: stores the modulus across all 8 lanes

1: function EltwiseFMAModAVX512<int BitShift, int InputModFactor>::(_m512 X, _m512 Y, _m512 barr, _m512i Z, _m512i q)
2: $X = _\text{mm512 hexl small_mod_epu64}\cdot$InputModFactor$= (X \cdot q)$
3: $Z = _\text{mm512 hexl small_mod_epu64}\cdot$InputModFactor$= (Z \cdot q)$
4: $\text{m512i XY} = _\text{mm512 hexl multl_epi64}(X, Y)$
5: $\text{m512i R} = _\text{mm512 hexl multl_epi64}(X, Y)\text{ barr}$
6: $\text{m512i Rq} = _\text{mm512 multl_epi64(R, q)}$
7: $R = _\text{mm512 sub_epi64(X, Y, Rq)}$
8: $\text{if} R < _\text{mm512 hexl small_mod_epu64}(R, q)$
9: $\text{Conditional Barrett subtraction}$
10: $R = _\text{mm512 add_epi64}(R, q)$
11: $\text{return R}$
12: end function

5 INTEGRATION WITH HE LIBRARIES
Existing homomorphic encryption libraries such as Microsoft SEAL [23] and PALISADE [22] typically provide a public API at the level of the HE scheme implemented, e.g. encryption, homomorphic multiplication and addition, and decryption. This HE cryptography layer is typically implemented by calls to a lower-level polynomial arithmetic layer. This polynomial arithmetic layer may implement $R_{q}$ for a multi-word coefficient modulus $q \in \mathbb{Z}_{q}$. A common optimization is to represent a multi-word integer $x \cdot mod \cdot q$ as a vector of pairwise coprime word-sized moduli $x \cdot mod \cdot q \mapsto (x \cdot mod \cdot q_{1}, x \cdot mod \cdot q_{2}, \ldots, x \cdot mod \cdot q_{t})$. This form is known as the residue number system (RNS) and has correctness guaranteed by the Chinese remainder theorem. The benefit of the RNS form is that two numbers in RNS form can be multiplied in $\mathbb{Z}_{q}$ using the element-wise residues, yielding substantial speedup over a multi-word multiplication algorithm.

Intel HXEL is designed to intercept HE libraries at the polynomial layer, with polynomials in RNS form. We chose this as the target integration layer for a few reasons. Firstly, the majority of the runtime in each HE operation lies at or below this polynomial layer, with minimal overhead from the HE layer to the polynomial layer. Therefore, any speedup at the polynomial level will propagate to higher-level HE operations. Secondly, the polynomial layer
We provide two levels of benchmarking. First, we benchmark the low-level kernel implemented in Intel HEXL. We also integrate Intel HEXL with two popular open-source libraries: Microsoft SEAL [23] and PALISADE [22] and benchmark the higher-level HE operations within each library. 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.

6 RESULTS

We provide two levels of benchmarking. First, we benchmark the low-level kernel implemented in Intel HEXL. We also integrate Intel HEXL with two popular open-source libraries: Microsoft SEAL [23] and PALISADE [22] and benchmark the higher-level HE operations within each library. 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 the clang-10 compiler with the '-march=native -O3' optimization flags. Each benchmark runs single-threaded on a single core.

6.1 Intel HEXL Kernels

6.1.1 NTT. We benchmark the performance of the Intel HEXL 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 [24] and NTL [1], two open-source libraries implementing the NTT. NTL is compiled with clang-10 using the NTL_ENABLE_AVX_FFT flag, which enables an experimental Intel AVX512 implementation using floating-point arithmetic.NFLlib is compiled with clang-10 using the NFL_Optimized=ON flag, which enables an Intel AVX256 implementation using 32-bit limbs.

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. NFLlib’s use of AVX256 yields performance between that of the native C++ and AVX512 implementations. 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 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. Similarly, NFLlib’s use of AVX256 yields performance between that of the native C++ and AVX512 implementations. 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.

6.1.2 Polynomial Kernels. 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++ integer 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 1: Single-threaded, single-core runtime in microseconds of the forward NTT on a 50-bit modulus with input_mod_factor = output_mod_factor = 1.

| Implementation                  | N   | Speedup |
|---------------------------------|-----|---------|
| Native C++                      | 1024| 1.0x    |
| NFLlib [1]                      | 4.82| 1.8x    |
| Intel AVX512-DQ                 | 3.26| 2.7x    |
| NTL [24]                        | 2.44| 3.7x    |
| Intel AVX512-IFMA52             | 1.25| 7.2x    |

Figure 2: Architecture diagram of typical HE libraries showing Intel HEXL integrates at the polynomial layer.

is usually common to different HE schemes, allowing Intel HEXL to accelerate multiple HE schemes with a minimal integration surface to the HE library. Figure 2 shows a high-level visualization of where Intel HEXL integrates to the different layers in a standard HE library.

One downside to this layer of integration is the runtime is typically too small to effectively offload the computation to an accelerator such as a field-programmable gate array (FPGA) or graphic processing unit (GPU). Another downside is that the optimization surface within Intel HEXL is relatively small. As such, for best performance, the HE library must still take care to optimize the sequence of calls to the polynomial layer, taking particular care to avoid many pitfalls in HE implementations, such as unnecessary NTT conversions and modular reductions, and cache misses when computing on multiple large polynomials. Additionally, the HE library should align any input data to 64-byte boundaries aligned for best performance.

Intel HEXL is publicly integrated with Microsoft SEAL [23] since version 3.6.4 and PALISADE [22] since version v1.11.3.
Table 2: Single-threaded, single-core runtime in microseconds of the inverse NTT on a 50-bit modulus 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 |
| NFLlib[1]     | 6.07 | 1.3x | 26.7 | 1.4x | 124 | 1.4x |
| Intel AVX512-DQ | 3.16 | 2.6x | 14.6 | 2.5x | 68.2 | 2.5x |
| NTL[24]       | 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 |

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++ Int | 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 |

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 |

Table 5: Speedup in microseconds of single-threaded, single-core Microsoft SEAL [23] benchmarks with polynomial modulus degree $N = 8192$ and $L = 3$ 50-bit coefficient moduli (plus one auxiliary prime used for relinearization only).

| Benchmark               | Speedup |
|-------------------------|---------|
| Forward NTT             | 4.70x   |
| Inverse NTT             | 5.46x   |
| BFV Encrypt             | 1.54x   |
| BFV Decrypt             | 2.48x   |
| BFV Multiply            | 1.43x   |
| BFV Multiply Relinearize| 1.56x   |
| BFV Rotate 1            | 2.38x   |
| CKKS Encode             | 1.68x   |
| CKKS Decode             | 1.23x   |
| CKKS Encrypt            | 1.87x   |
| CKKS Decrypt            | 2.80x   |
| CKKS Multiply           | 2.66x   |
| CKKS Multiply Relinearize| 2.22x  |
| CKKS Rescale            | 3.26x   |
| CKKS Rotate 1           | 2.08x   |

6.2 HE Kernels

We benchmark the performance of two HE libraries that have adopted Intel HEXL, Microsoft SEAL [23] and PALISADE [22]. We benchmark Microsoft SEAL [23] version 3.6.5 and PALISADE [22] version 1.11.3. For each library, we compile with and without Intel HEXL support and compare the throughput. We compile PALISADE using WITH_OPENMP=OFF and WITH_NATIVEOPT=ON for best single-threaded performance.

Table 5 and Table 6 show the runtime speedup in Microsoft SEAL [23] and PALISADE [22] HE kernels due to Intel HEXL. The amount of speedup for each kernel depends on several factors, including which Intel HEXL functions the implementation calls for the underlying computation and how efficiently the HE library implements each kernel natively. We report only the speedup rather than the execution time of each kernel to avoid any performance comparison between Microsoft SEAL [23] and PALISADE [22]. There are several implementation details that differ between the libraries that make such a comparison tricky beyond the scope of this work. Rather, the main point here is that Intel HEXL has a flexible enough API to support different HE libraries, and that the integration at the polynomial level is effective in improving overall performance of the library.

7 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 single-threaded 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 single-threaded speedup over the native C++ implementation, while the Intel AVX512-IFMA52 implementation of element-wise vector-scalar modular multiplication yields 1.7x single-threaded 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. Intel HEXL has been adopted by the Microsoft SEAL [23] and PALISADE [22] homomorphic encryption libraries.

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 [18]. We also plan to integrate Intel HEXL with additional open-source homomorphic encryption libraries, as well as expand the API to encompass a larger variety of applications. Adding a programming model which enables compiling several Intel HEXL kernels in sequence may enable cross-kernel compiler optimizations such as loop fusion for additional performance improvement.

ACKNOWLEDGEMENTS

We would like to thank Ilya Albrecht for guidance on the AVX512 implementation. We would also like to thank Kim Laine and Wei Dai for their support integrating Intel HEXL to Microsoft SEAL [23] as well as Kurt Rohloff and Yuriy Polyakov for their support integrating Intel HEXL to PALISADE [22].
REFERENCES

[1] Carlos Aguilar-Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrede Lepoint. 2016. NFLlib: NTT-based fast lattice library. In Cryptographers’ Track at the RSA Conference. Springer, 341–356.

[2] Flavio Bergamaschi, Shai Halevi, Trapora T Halevi, and Hamish Hunt. 2019. Homomorphic Training of 30,000 Logistic Regression Models. In International Conference on Applied Cryptography and Network Security. Springer, 592–611.

[3] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, and Shafi Goldwasser. 2020. Secure large-scale genome-wide association studies using homomorphic encryption. Proceedings of the National Academy of Sciences 117, 21 (2020), 11608–11613.

[4] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) 6, 3 (2014), 1–36.

[5] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. A full RNS variant of approximate homomorphic encryption. In International Conference on Selected Areas in Cryptography. Springer, 347–368.

[6] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In International Conference on the Theory and Application of Cryptology and Information Security. Springer, 409–437.

[7] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. Mathematics of computation 19, 90 (1965), 297–301.

[8] Intel Corporation. 2021. Intel Intrinsic Guide. (2021). Retrieved 2021-06-11 from https://software.intel.com/sites/landingpage/IntrinsicsGuide/. (2021-06-11).

[9] Nir Drucker and Shay Gueron. 2019. Fast modular squaring with AVX512IFMA. In 16th International Conference on Information Technology-New Generations (ITNG 2019). Springer, 3–8.

[10] Takuya Edamatsu and Daisuke Takahashi. 2019. Accelerating Large Integer Multiplication Using Intel AVX-512IFMA. In International Conference on Algorithms and Architectures for Parallel Processing. Springer, 60–74.

[11] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. (2012).

[12] Kjell Magne Fauske. 2006. Texample.net. (2006). Retrieved 2021-06-11 from https://texample.net/tikz/examples/radix2fft/.

[13] Pierre Fortin, Ambroise Fleury, François Lemaire, and Michael Monagan. 2020. High performance SIMD modular arithmetic for polynomial evaluation. arXiv preprint arXiv:2004.11571 (2020).

[14] Rémi Géraud, Diana Maimut, and David Naccache. 2016. Double-speed barrett moduli. In The New Codebreakers. Springer, 148–158.

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

[16] David Harvey. 2014. Faster arithmetic for number-theoretic transforms. Journal of Symbolic Computation 60 (2014), 113–119.

[17] Joris Van Der Hoeven, Grégouere Lecerf, and Guillaume Quintin. 2016. Modular SIMD arithmetic in Mathemagix. ACM Transactions on Mathematical Software (TOMS) 43, 1 (2016), 1–37.

[18] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Keewoo Lee, Namhoon Kim, Choshong Min, Jung Hee Cheon, and Jung Ho Ahn. 2020. HEAAN Demystified: Accelerating Fully Homomorphic Encryption Through Architecture-centric Analysis and Optimization. arXiv preprint arXiv:2003.04510 (2020).

[19] Ovunc Kocabas and Tolga Soyata. 2020. Towards privacy-preserving medical cloud computing using homomorphic encryption. In Virtual and Mobile Healthcare: Breakthroughs in Research and Practice. IGI Global, 93–125.

[20] Patrick Longa and Michael Naehrig. 2016. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In International Conference on Cryptology and Network Security. Springer, 124–139.

[21] Toufique Morshed, Md Monim Al Aziz, and Noman Mohammed. 2020. CPU and GPU Accelerated Fully Homomorphic Encryption. In 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 142–153.

[22] Kurt Rohloff. 2018. The PALISADE Lattice Cryptography Library. (2018). Retrieved 2021-06-11 from https://palisade-crypto.org/software-library/.

[23] SEAL 2020. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL. (Nov. 2020). Microsoft Research, Redmond, WA.

[24] Victor Shoup et al. 2001. NTL: A library for doing number theory. (2001). https://eprint.iacr.org/2012/144.