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, OR, 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, measuring up to 7.2x single-threaded speedup over a native C++ baseline. 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.

CCS CONCEPTS

• **Mathematics of computing → Mathematical software performance**.

KEYWORDS

privacy-preserving machine learning; Intel AVX512; homomorphic encryption; number-theoretic transform (NTT)

WAHC '21, November 15, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8656-2/21/11.
https://doi.org/10.1145/3474366.3486926

ACM Reference Format:
Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D. M. de Souza, and Vinodh Gopal. 2021. Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th Workshop on Encrypted Computing Applied Homomorphic Cryptography (WAHC '21), November 15, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3474366.3486926

1 INTRODUCTION

Homomorphic encryption (HE) is a form of encryption which enables computation in the encrypted domain. HE is useful in applications with sensitive data, particularly medical and financial settings [2, 3, 18]. However, HE currently suffers from enormous runtime overheads of up to 30,000x [17]. While algorithmic changes and optimized implementations have improved this overhead in recent years, it remains perhaps the biggest obstacle to HE adoption.

Here, we introduce *Intel® HEXL*, an open-source C++ library which provides efficient implementations of integer arithmetic on finite fields. Intel HEXL uses the Intel Advanced Vector Extensions 512 (Intel AVX512) instruction set to implement polynomial operations with word-sized primes on 64-bit Intel processors. In particular, Intel HEXL takes advantage of the Intel® Advanced Vector Extensions 512 Integer Fused Multiply Add (Intel® AVX512-IFMA52) [7] instructions introduced in the 3rd Gen Intel® Xeon® Scalable Processors to provide significant speedup on primes below 50 bits.

2 BACKGROUND

We provide a brief background of the algorithms optimized in Intel HEXL, which are common building blocks of lattice cryptography. Ciphertexts in many HE schemes, including BGV [4], BFV [10], and CKKS [5], consist of polynomials in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$, the 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 the *coefficient modulus* $q \equiv 1 \mod 2N$ is a word-sized prime. The *polynomial modulus degree* N is typically a power of two in the range $[2^{10}, 2^{17}]$. This setting is typical of efficient implementations using a residue number system (RNS).

Performing HE computations requires operating on these large polynomials. Three bottlenecks in current HE implementations are:

- *Element-wise vector-vector 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

$$c_i = \left(\sum_{j=0}^{i} a_j \cdot b_{i-j} - \sum_{j=i+1}^{N-1} a_j \cdot b_{N+i-j} \right) \mod q.$$

  Note, $X^N \equiv -1 \mod (X^N + 1)$, which yields the negative coefficients.

Element-wise vector-vector and vector-scalar multiplication are typically implemented via Barrett reduction [13]. Vector-vector multiplication takes advantage of the negacyclic number-theoretic transform (NTT) [6, 15, 19] to reduce the runtime complexity from $O(N^2)$ to $O(N \log N)$. Intel HEXL provides efficient Intel AVX512 implementations for element-wise vector-vector and element-wise vector-scalar multiplication, and the forward and inverse NTT.

2.1 Intel Advanced Vector Extensions

The Intel® Advanced Vector Extensions 512 (Intel® AVX512) is a set of single-instruction multiple data (SIMD) instructions for the x86 architecture which enable simultaneous computation on 512-bit data. Intel HEXL uses the Intel AVX512 `__m512i` 512-bit packed integer datatype, which we also denote `m512i`, to represent and compute on eight 64-bit integers simultaneously. To employ these SIMD instructions, Intel provides a set of C/C++-compatible intrinsics, which compile to the corresponding assembly instruction. 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 [17]. Intel AVX512-IFMA52 introduces:

- `m512i _mm512_madd52<lo|hi>_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/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 novelty in Intel HEXL is the use of Intel AVX512-IFMA52 to accelerate finite field arithmetic, in particular the NTT. Intel HEXL also utilizes several existing algorithms from previous work.

The Mathemagix library [16] and NFLlib [1] both provide Intel AVX2 implementations of modular integer arithmetic using a SIMD programming model. However, neither Mathemagix nor NFLlib provide Intel AVX512 implementations.

Previous work [9, 14] using Intel AVX512-IFMA52 focuses on accelerating big integer multiplication without modular multiplication. Drucker and Gueron [8] 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. 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, single-threaded C++11 library available under the Apache 2.0 license. Intel HEXL focuses on the case where $q < 2^{64}$, as implemented on a 64-bit word-sized CPU platform. As such, the Intel HEXL API uses unsigned 64-bit integers input vector types. 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, hiding their calculation from the end user.

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 bits, while the Intel AVX512-DQ implementations allow moduli up to 62 bits. 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.

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. Each function assumes each `m512i` unit stores 8 64-bit integers. Several functions take a template argument k, either 52 or 64, which is evaluated at compile time. These template parameters enable a shared, performant source code for primes less than and greater than 50 bits. Intel HEXL uses the following Intel AVX512 kernels:

- `m512i _mm512_hexl_mullo_epi<k>(m512i x, m512i y)`. Multiplies packed unsigned k-bit integers x and y to perform a $2k$-bit intermediate result. Returns the low k-bit unsigned integer from the intermediate result. The implementation calls `_mm512_madd52lo_epu64` with the accumulator set to zero ($k = 52$) or `_mm512_mullo_epi64` ($k = 64$).
- `m512i _mm512_hexl_mullo_add_epi<k>(m512i x, m512i y, m512i z)`. Multiplies packed unsigned k-bit integers 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 calls `_mm512_madd52lo_epu64` ($k = 52$) or `_mm512_mullo_epi64` and `_mm512_add_epi64` ($k = 64$).

- `m512i _mm512_hexl_mulhi_epi<k>(m512i x, m512i y)`. Multiplies packed unsigned k-bit integers 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 shifts, 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(m512i x,m512i q)`. Given each integer $0 \leq x_i < 2 \cdot q_i$, returns $x \mod q$. The implementation uses the fact that for unsigned integers $x < 2q$, $x \mod q = \min(x - q, x)$ and calls `_mm512_sub_epi64` and `_mm512_min_epu64`.

4.1 NTT

Intel HEXL provides optimized Intel AVX512 radix-2 implementations of the negacyclic NTT with bit-reversed outputs. The forward and inverse NTTs are implemented using the Cooley-Tukey and Gentleman-Sande butterflies, respectively, using Harvey's lazy modular reduction [15]. In each case, the butterfly is implemented across all 8 lanes of an Intel AVX512 input vector of 64-bit integers, as shown in Algorithm 1 and Algorithm 2. The Intel AVX512 NTT butterflies are used in every stage of the NTT. The forward NTT begins with $N/2$ subsequent butterfly calls in the first stage, then $N/4$ calls in the next stage, followed by successively halving the number of calls, until the final stage calls a single butterfly in sequence. Similarly, the inverse NTT begins with 1 subsequent butterfly call in the first stage, then 2 calls in the next stage, followed by successively doubling in the number of calls, until the final stage calls $N/2$ butterflies in sequence. As such, when the butterfly is called 8 or more times in succession, Algorithm 1 and Algorithm 2 are simple to apply (particularly since the number of loop iterations is divisible by 8). However, special consideration must be taken in the first and final three stages, denoted *InvT4, InvT2, InvT1, FwdT4, FwdT2, FwdT1* in Figure 1. In these cases, we permute the data within each `m512i` unit such that the butterfly is applied across all lanes.

We make a few remarks on the implementations:

- The template argument `InputLessThanMod` enables a compile-time optimization when the inputs $X, Y < q$ 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 the single-instruction `_mm512_hexl_mullo_add_epi<52>`.
- Lines 8–10 in the forward butterfly and Lines 14–16 in the inverse butterfly clear the high 12 bits from `T` / `Y`. This is required because the `_mm512_madd52lo_epu64` instruction uses a 64-bit, rather than 52-bit accumulator.
- The benefit of Intel AVX512-IFMA52 in the NTT is for primes less than 50 bits, in which case `_mm512_hexl_mulhi<52>` is a single assembly instruction, whereas large primes require `_mm512_hexl_mulhi<64>`, which is much more expensive (see Section 2.1).

Algorithm 1 Intel AVX512 Harvey NTT butterfly with word size $\beta \in \{2^{52}, 2^{64}\}$.

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

```
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
```

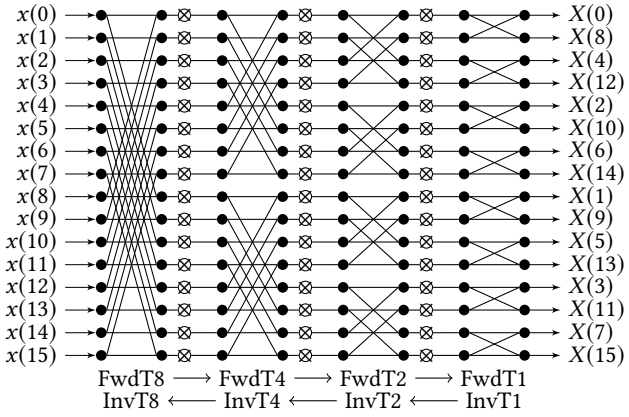

Figure 1: 16-point forward (left to right) and inverse (right to left) NTT data flows. *T1, T2, T4, T8* refer to a case when the for loop beginning in Line 8 of Algorithms 1 and 2 in [19] runs for 1/2/4/≥ 8 iterations. Modified from [11].

4.2 Polynomial Kernels

We present the core Intel AVX512 kernels used in the element-wise polynomial operations. In each case, the kernel is applied sequentially for each 512 bits of the input. Although typical HE applications use polynomials degree $N = 2^k \geq 1024$ a power of two, the Intel HEXL implementation includes logic that processes the $N \mod 8$ remaining loop iterations.

4.2.1 Element-wise Vector-Vector Multiplication. Intel HEXL provides two AVX512 implementations of element-wise vector-vector

Algorithm 2 Intel AVX512 Harvey Inverse NTT butterfly with word size $\beta \in \{2^{52}, 2^{64}\}$.

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

```
function HarveyInvNTTButterfly<int BitShift, bool InputLessThanMod>(__m512i* X, __m512i* Y, __m512i W_op, __m512i W_precon, __m512i neg_modulus, __m512i twice_modulus)
    __m512i Y_minus_2q = _mm512_sub_epi64(*Y, twice_modulus);
    __m512i T = _mm512_sub_epi64(*X, Y_minus_2q);
    if InputLessThanMod then
        *X = _mm512_add_epi64(*X, *Y);
    else
        *X = _mm512_add_epi64(*X, Y_minus_2q);
        __mmask8 sign_bits = _mm512_movepi64_mask(*X);
        *X = _mm512_mask_add_epi64(*X, sign_bits, *X, twice_modulus);
    end if
    __m512i Q = _mm512_hexl_mulhi_epi<BitShift>(W_precon, T);
    *Y = _mm512_hexl_mullo_add_epi<BitShift>(Q_p, W_op, T);
    __m512i Q_p = _mm512_hexl_mullo_epi<BitShift>(Q, neg_modulus);
    if BitShift == 52 then
        *Y = _mm512_and_epi64(*Y, _mm512_set1_epi64((1UL « 52) - 1));
    end if
end function
```

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. Both implementations use a variant of Barrett reduction.

Intel AVX512-DQ integer implementation

The Intel AVX512-DQ integer implementation uses Algorithm 1 from [13]. We choose $Q = \lceil \log_2 q \rceil$ 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, i.e. the low 64 bits of c_2 do not need to be computed. Algorithm 3 shows the pseudocode for the Intel AVX512-DQ integer modular multiplication implementation.

Algorithm 3 VectorVectorModMulAVX512Int

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

```
function EltwiseMultModAVX512Int<int L>(__m512i X, __m512i Y, __m512i barr_lo, __m512i q)
    __m512i prod_hi = _mm512_hexl_mulhi_epi<64>(X, Y);
    __m512i prod_lo = _mm512_hexl_mullo_epi<64>(X, Y);
    __m512i c1 = _mm512_hexl_shrdi_epi64<L - 1>(prod_lo, prod_hi);
    __m512i c3 = _mm512_hexl_mulhi_epi<64>(c1, barr_lo);
    __m512i c4 = _mm512_hexl_mullo_epi<64>(c3, q);
    c4 = _mm512_sub_epi64(prod_lo, c4);
    return _mm512_hexl_small_mod_epu64(c4, q);
end function
```

Intel AVX512-DQ floating-point implementation

For $q < 2^{50}$, Intel HEXL adapts Function 3.10 from Mathemagix [16] to Intel AVX512, in a similar manner as Fortin et al. [12]. Algorithm 4 shows the Intel AVX512 floating-point kernel.

Algorithm 4 VectorVectorModMulAVX512Float

Require: $0 < X, Y < q < 2^{52}$
Require: $u \ge 1/q$ stores 1/(double)q in each lane
Ensure: Returns $X \cdot Y \mod q$

```
function EltwiseMultModAVX512Float(__m512i X, __m512i Y, __m512d u, __m512d q)
    const int rounding = _MM_FROUND_TO_POS_INF|_MM_FROUND_NO_EXC;
    __m512d xi = _mm512_cvt_roundepu64_pd( X, rounding);
    __m512d yi = _mm512_cvt_roundepu64_pd( Y, rounding);
    __m512d h = _mm512_mul_pd(xi, yi);
    __m512d l = _mm512_fmsub_pd(xi, yi, h);   ▷ rounding error; h + l == x * y
    __m512d b = _mm512_mul_pd(h, u);                          ▷ ~(x * y) / q
    __m512d c = _mm512_floor_pd(b);                           ▷ ~floor(x * y / q)
    __m512d d = _mm512_fnmadd_pd(c, q, h);
    __m512d g = _mm512_add_pd(d, l);
    __mmask8 m = _mm512_cmp_pd_mask(g, _mm512_setzero_pd(), _CMP_LT_OQ);
    g = _mm512_mask_add_pd(g, m, g, q);
    return _mm512_cvt_roundpd_epu64(g, rounding);
end function
```

4.2.2 Element-wise Vector-Scalar Multiplication. The `EltwiseFMAMod` function implements vector-scalar modular multiplication, with an additional modular vector addition. The `BitShift` template parameter is used to distinguish between the Intel AVX512-DQ (`BitShift = 64`) and Intel AVX512-IFMA52 (`BitShift = 52`) implementations. Algorithm 5 shows the algorithm for element-wise vector-scalar multiplication, which is similar to the element-wise vector-vector multiplication algorithm(3) with additional pre-computation using the scalar multiplicand.

Algorithm 5 EltwiseFMAModAVX512

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

```
function EltwiseFMAModAVX512<int BitShift>(__m512i X, __m512i Y, __m512i Y_barr, __m512i Z, __m512i q)
    __m512i XY = _mm512_hexl_mullo_epi<64>(X, Y);
    __m512i R = _mm512_hexl_mulhi_epi<BitShift>(X, Y_barr);
    __m512i Rq = _mm512_mullo_epi64(R, q);
    R = _mm512_sub_epi64(XY, Rq);
    R = _mm512_hexl_small_mod_epu64(R, q);            ▷ Barrett subtraction
    R = _mm512_add_epi64(vq, Z);
    return _mm512_hexl_small_mod_epu64(R, q);
end function
```

5 INTEGRATION WITH HE LIBRARIES

Intel HEXL is designed for easy integration into existing HE libraries, and has been integrated with Microsoft SEAL [21] since version 3.6.4 and PALISADE [20] since version v1.11.3. Existing HE libraries including Microsoft SEAL [21] and PALISADE [20] implement an HE cryptography layer by calls to a lower-level polynomial arithmetic layer.

Intel HEXL is designed to intercept HE libraries at this polynomial layer, with polynomials in RNS form. The majority of each HE operation's runtime lies at this polynomial layer, so speedup at the polynomial level will propagate to higher-level HE operations. Additionally, 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 visualizes how Intel HEXL integrates to standard HE libraries. For best

Table 1: Single-threaded, single-core Intel HEXL kernel runtimes in microseconds on a 50-bit modulus.

(a) Forward NTT

Implementation	N / Speedup					
	1024		4096		16384	
Native C++	9.08	1.0x	38.8	1.0x	177	1.0x
NFLlib[1]	4.82	1.8x	21.1	1.8x	97.8	1.8x
Intel AVX512-DQ	3.26	2.7x	13.4	2.8x	62.3	2.8x
NTL[22]	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

(b) Inverse NTT

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[22]	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

(c) Element-wise vector-vector modular multiplication

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

(d) Element-wise vector-scalar-vector modular multiply-add

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

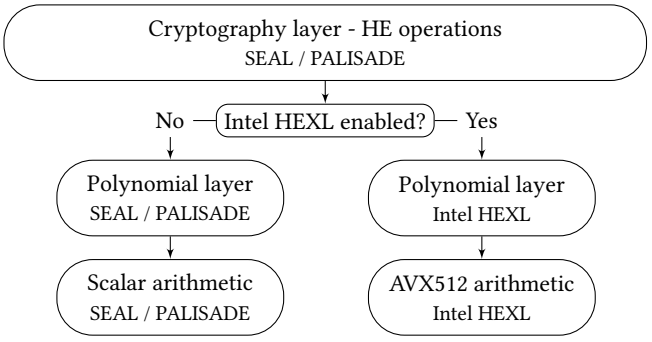

Figure 2: Architecture diagram of typical HE libraries showing Intel HEXL integrates at the polynomial layer.

performance, the HE library should align any input data to 64-byte boundaries.

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 graphics processing unit (GPU).

6 RESULTS

We provide two levels of benchmarking. First, we benchmark the low-level kernels implemented in Intel HEXL. We also benchmark the higher-level HE operations using Microsoft SEAL [21] and PALISADE [20] with Intel HEXL integration. 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

We compare the Intel AVX512 implementations of each Intel HEXL kernel against a native C++ baseline. We measure the performance with a 50-bit modulus on input sizes $N = 1024, 4096, 16384$.

6.1.1 NTT. The default implementation uses the radix-2 Cooley-Tukey (forward NTT) and Gentleman-Sande (inverse NTT) formulations, using the Harvey butterfly [15]. Additionally, we compare against NTL v11.4.3 [22][1] and NFLlib [1][2], two open-source libraries implementing the NTT. NTL's implementation uses floating-point arithmetic for integer computation, and is therefore correct for primes up to 50 bits, as is the Intel AVX512-IFMA52 implementation. Table 1a and Table 1b shows the runtimes for the forward and inverse NTT, respectively. NFLlib's use of Intel AVX256 yields 1.3x–1.8x speedup over the native C++ implementations. Intel HEXL's Intel AVX512-DQ implementation improves this speedup to 2.5x–2.8x. Intel AVX512-IFMA52 provides additional speedup, due to the fast high and low 52-bit multiplies. We note the Intel AVX512-IFMA52 implementation speedup decreases to 5.3x for the largest $N = 16384$ case as L1 cache misses bottleneck the memory access, which may be improved by a more cache-friendly algorithm.

6.1.2 Polynomial Kernels. Table 1c 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 C++ implementation, which the Intel AVX512-DQ floating-point implementation improves to 5.1x–6.0x.

Table 1d shows the runtimes for the element-wise vector-scalar modular multiplication with vector addition. The Intel AVX512-DQ implementation yields no speedup over the native C++ 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.

[1] with the *NTL_ENABLE_AVX_FFT* flag, enabling an Intel AVX512 floating-point NTT
[2] with the *NFL_OPTIMIZED* flag, enabling an Intel AVX256 32-bit implementation

Table 2: Speedup of single-threaded, single-core Microsoft SEAL [21] and PALISADE [20] benchmarks with polynomial modulus degree $N = 8192$ and 3 50-bit coefficient moduli.

Library	Benchmark	Intel HEXL Speedup
SEAL	Forward NTT	4.70x
SEAL	Inverse NTT	5.46x
SEAL	BFV Encrypt	1.54x
SEAL	BFV Decrypt	2.48x
SEAL	BFV Multiply	1.43x
SEAL	BFV Multiply Relinearize	1.56x
SEAL	BFV Rotate 1	2.38x
SEAL	CKKS Encode	1.68x
SEAL	CKKS Decode	1.23x
SEAL	CKKS Encrypt	1.87x
SEAL	CKKS Decrypt	2.80x
SEAL	CKKS Multiply	2.66x
SEAL	CKKS Multiply Relinearize	2.22x
SEAL	CKKS Rescale	3.26x
SEAL	CKKS Rotate 1	2.08x
PALISADE	Forward NTT	6.26x
PALISADE	Inverse NTT	4.83x
PALISADE	BFV Encode	2.84x
PALISADE	BFV Decode	1.72x
PALISADE	BFV Encrypt	1.23x
PALISADE	BFV Decrypt	1.91x
PALISADE	BFV Multiply	1.50x
PALISADE	BFV Rotate 1	2.12x
PALISADE	CKKS Encode	1.69x
PALISADE	CKKS Encrypt	1.19x
PALISADE	CKKS Multiply	2.59x
PALISADE	CKKS Multiply Relinearize Rescale	3.24x
PALISADE	CKKS Rotate 1	2.68x

6.2 HE Library Integration

We benchmark the performance of two HE libraries that have adopted Intel HEXL, Microsoft SEAL [21] (version 3.6.5) and PALISADE [20] (version 1.11.13). For each library, we compile with and without Intel HEXL support and compare the benchmark throughput using $N = 8192$ and 3 50-bit moduli. We compile PALISADE using *WITH_OPENMP=OFF* and *WITH_NATIVEOPT=ON* for best single-threaded performance.

Table 2 shows the runtime speedup in Microsoft SEAL [21] and PALISADE [20] 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 and the efficiency of the HE library's native implementation. Intel HEXL has a flexible enough API to support different HE libraries, and integration at the polynomial level improves overall performance of the HE library.

7 FUTURE WORK

Future work improving Intel HEXL includes exploring additional NTT implementations, such as higher-radix implementations. We also plan to integrate Intel HEXL with additional open-source HE libraries, and expand the API to encompass more applications.

ACKNOWLEDGEMENTS

We would like to thank Ilya Albrekht for guidance on the AVX512 implementation, Kim Laine and Wei Dai for support integrating Intel HEXL to Microsoft SEAL [21], and Kurt Rohloff and Yuriy Polyakov for support integrating Intel HEXL to PALISADE [20].

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, Tzipora 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, 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.
[6] 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.
[7] Intel Corporation. 2021. Intel Intrinsics Guide. (2021). Retrieved 2021-06-11 from https://software.intel.com/sites/landingpage/IntrinsicsGuide/#avx512techs=AVX512IFMA52
[8] 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.
[9] 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.
[10] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. (2012). https://eprint.iacr.org/2012/144.
[11] Kjell Magne Fauske. 2006. Texample.net. (2006). Retrieved 2021-06-11 from https://texample.net/tikz/examples/radix2fft/
[12] Pierre Fortin, Ambroise Fleury, François Lemaire, and Michael Monagan. 2020. High performance SIMD modular arithmetic for polynomial evaluation. *arXiv preprint arXiv:2004.11571* (2020).
[13] Rémi Géraud, Diana Maimuț, and David Naccache. 2016. Double-speed barrett moduli. In *The New Codebreakers*. Springer, 148–158.
[14] Shay Gueron and Vlad Krasnov. 2016. Accelerating big integer arithmetic using intel IFMA extensions. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. IEEE, 32–38.
[15] David Harvey. 2014. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation* 60 (2014), 113–119.
[16] Joris Van Der Hoeven, Grégoire Lecerf, and Guillaume Quintin. 2016. Modular SIMD arithmetic in Mathemagix. *ACM Transactions on Mathematical Software (TOMS)* 43, 1 (2016), 1–37.
[17] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Jongmin Kim, Namhoon Kim, Keewoo Lee, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. 2021. Accelerating Fully Homomorphic Encryption Through Architecture-Centric Analysis and Optimization. *IEEE Access* 9 (2021), 98772–98789.
[18] 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.
[19] 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.
[20] Kurt Rohloff. 2018. The PALISADE Lattice Cryptography Library. (2018). Retrieved 2021-06-11 from https://palisade-crypto.org/software-library/
[21] SEAL 2020. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL. (Nov. 2020). Microsoft Research, Redmond, WA.
[22] Victor Shoup et al. 2001. NTL: A library for doing number theory. (2001).