An Implementation of Parallel Number-Theoretic Transform Using Intel AVX-512 Instructions

Daisuke Takahashi

Center for Computational Sciences, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577, Japan
daisuke@cs.tsukuba.ac.jp

Abstract. In this paper, we propose an implementation of the parallel number-theoretic transform (NTT) using Intel Advanced Vector Extensions 512 (AVX-512) instructions. The butterfly operation of the NTT can be performed using modular addition, subtraction, and multiplication. We show that a method known as the six-step fast Fourier transform algorithm can be applied to the NTT. We vectorized NTT kernels using the Intel AVX-512 instructions and parallelized the six-step NTT using OpenMP. We successfully achieved a performance of over 83 giga-operations per second on an Intel Xeon Platinum 8368 (2.4 GHz, 38 cores) for a 2^{20}-point NTT with a modulus of 51 bits.

Keywords: Number-theoretic transform · Modular multiplication · Intel AVX-512 instructions

1 Introduction

The fast Fourier transform (FFT) [5] is an algorithm that is widely used today in scientific and engineering computing. FFTs are often computed using complex or real numbers, but it is known that these transforms can also be computed in a ring and a finite field [14]. Such a transform is called the number-theoretic transform (NTT). The NTT is used for homomorphic encryption, polynomial multiplication, and multiple-precision multiplication.

Efficient arithmetic for NTTs has been proposed [7]. The number theory library (NTL) [15] is a C++ library for performing number-theoretic computations and implements NTT. Although the NTL is thread-safe, the parallel NTT is not supported. Spiral-generated modular FFTs have been proposed [11,12] and experiments were performed using 32-bit integers and 16-bit primes with Intel SSE4 instructions. An implementation of NTT using the Intel AVX-512IFMA (Integer Fused Multiply-Add) instructions has been proposed [2]. This implementation is available as the Intel Homomorphic Encryption (HE) Acceleration Library [3], an open-source C++ library that provides efficient implementations of integer arithmetic on finite fields. Intel HEXL targets the typical data size

$n = [2^{10}, 2^{17}]$ of NTTs used in homomorphic encryption [2] and is not parallelized by OpenMP.

In contrast, we consider accelerating NTT for larger data sizes by parallelization, targeting polynomial multiplication and multiple-precision multiplication. In this paper, we vectorize NTT kernels using the Intel AVX-512 instructions and parallelize NTT using OpenMP.

The remainder of this paper is organized as follows. Section 2 describes the number-theoretic transform (NTT). Section 3 presents the vectorization of the NTT kernels. Section 4 presents the proposed implementation of the parallel NTT. Section 5 presents the performance results. Finally, Sect. 6 presents concluding remarks.

2 Number-Theoretic Transform (NTT)

The discrete Fourier transform (DFT) is given by

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk}, \quad 0 \le k \le n-1, \tag{1}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$.

The DFT can be defined over rings and fields other than the complex field [14]. Equation (1) can be expressed in a field $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$, where p is a prime number:

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk} \bmod p, \quad 0 \le k \le n-1, \tag{2}$$

in which ω_n is the primitive n-th root of unity. For example, if $\omega_n = 157$ for $n = 16$ and $p = 1297$ $(= 81 \times 16 + 1)$, then $\omega_n^{16} \equiv 1 \bmod 1297$ and $\omega_n^{16/2} = \omega_n^8 \equiv 1296 \not\equiv 1 \bmod 1297$.

The n-point NTT in Eq. (2) is directly computed by $O(n^2)$ arithmetic operations, but by applying an algorithm similar to the FFT, the number of arithmetic operations can be reduced to $O(n \log n)$. Applications of the Stockham algorithm [4,16], known as an out-of-place FFT algorithm, to radix-2, 4, and 8 NTTs are shown in Algorithms 1, 2, and 3, respectively. The multiplication by $\omega_n^{n/4}$ in line 13 of Algorithm 2 can be performed by a trivial multiplication by $-i$ in the radix-4 FFT. As a result, the total number of real arithmetic operations for the n-point FFT is reduced from $5n \log_2 n$ in the radix-2 FFT to $4.25n \log_2 n$ in the radix-4 FFT. However, the number of arithmetic operations cannot be reduced by increasing the radix because there is no such trivial multiplication in the NTT. Intel HEXL includes radix-2 and 4 NTT implementations.

When computing the NTT, modular multiplication takes up most of the computation time. Modular multiplication includes modulo operations, which are slow due to the integer division process. However, Montgomery multiplication [13] and Shoup's modular multiplication [7] are known to avoid this problem.

Algorithm 1. Stockham radix-2 NTT algorithm

```
Input: $n = 2^q$, $X_0(j) = x(j)$, $0 \le j \le n-1$, and $\omega_n$ is the primitive $n$-th root of unity.
Output: $y(k) = X_q(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk} \bmod p$, $0 \le k \le n-1$
 1: $l \leftarrow n/2$
 2: $m \leftarrow 1$
 3: for $t$ from 1 to $q$ do
 4:   for $j$ from 0 to $l-1$ do
 5:     for $k$ from 0 to $m-1$ do
 6:       $c_0 \leftarrow X_{t-1}(k+jm)$
 7:       $c_1 \leftarrow X_{t-1}(k+jm+lm)$
 8:       $X_t(k+2jm) \leftarrow (c_0+c_1) \bmod p$
 9:       $X_t(k+2jm+m) \leftarrow \omega_n^{jm}(c_0-c_1) \bmod p$
10:     end for
11:   end for
12:   $l \leftarrow l/2$
13:   $m \leftarrow 2m$
14: end for
```

Shoup's modular multiplication [7] is shown in Algorithm 4. In Algorithm 4, if β is a power-of-two integer, the truncated quotient of dividing AB' by β in line 1 can be calculated by right shifting, and the remainder of dividing $(AB - qN)$ by β in line 2 can be calculated by bit masking.

The decimation-in-frequency butterfly operation of the NTT is shown in the following expression:

$$\begin{cases} X = (x+y) \bmod p, \\ Y = \omega(x-y) \bmod p. \end{cases}$$

The value of Y in the butterfly operation can be calculated in Algorithm 4 using $A = x - y$, $B = \omega$, and $N = p$. Here, the value of $B' = \lfloor \omega\beta/p \rfloor$ can be calculated in advance.

When convolution is performed for polynomials using NTT, the modulus also needs to increase as the degree increases. If the modulus does not fit into the size of the machine word (e.g., 32 or 64 bits), it is known that the convolution can be performed by computing NTTs on multiple moduli and then reconstructing the moduli using the Chinese remainder theorem.

3 Vectorization of NTT Kernels

Intel Advanced Vector Extensions 512 (AVX-512) [8] is a 512-bit vector instruction set that consists of multiple extensions that can be implemented independently. All Intel AVX-512 implementations require only the core extension Intel AVX-512F (Foundation). The most direct way to use Intel AVX-512 instructions is to insert assembly-language instructions inline into the source code. However, this can be time-consuming and tedious. Intel thus provides API extension

Algorithm 2. Stockham radix-4 NTT algorithm

Input: $n = 4^q$, $X_0(j) = x(j)$, $0 \le j \le n-1$, and ω_n is the primitive n-th root of unity.

Output: $y(k) = X_q(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk} \bmod p$, $0 \le k \le n-1$

```
1:  $l \leftarrow n/4$
2:  $m \leftarrow 1$
3:  for $t$ from 1 to $q$ do
4:    for $j$ from 0 to $l-1$ do
5:      for $k$ from 0 to $m-1$ do
6:        $c_0 \leftarrow X_{t-1}(k+jm)$
7:        $c_1 \leftarrow X_{t-1}(k+jm+lm)$
8:        $c_2 \leftarrow X_{t-1}(k+jm+2lm)$
9:        $c_3 \leftarrow X_{t-1}(k+jm+3lm)$
10:       $d_0 \leftarrow (c_0+c_2) \bmod p$
11:       $d_1 \leftarrow (c_0-c_2) \bmod p$
12:       $d_2 \leftarrow (c_1+c_3) \bmod p$
13:       $d_3 \leftarrow \omega_n^{n/4}(c_1-c_3) \bmod p$
14:       $X_t(k+4jm) \leftarrow (d_0+d_2) \bmod p$
15:       $X_t(k+4jm+m) \leftarrow \omega_n^{jm}(d_1+d_3) \bmod p$
16:       $X_t(k+4jm+2m) \leftarrow \omega_n^{2jm}(d_0-d_2) \bmod p$
17:       $X_t(k+4jm+3m) \leftarrow \omega_n^{3jm}(d_1-d_3) \bmod p$
18:     end for
19:   end for
20:   $l \leftarrow l/4$
21:   $m \leftarrow 4m$
22: end for
```

sets, referred to as intrinsics [9], to facilitate implementation. The GCC [6], Clang [19], and Intel C compilers [9] support automatic vectorization using Intel AVX-512 instructions.

The NTT kernels include modular addition, subtraction, and multiplication. The modular addition $c = (a+b) \bmod N$ for $0 \le a, b < N$ can be replaced by the addition $c = a + b$ and the conditional subtraction $c - N$ when $c \ge N$. Such conditional subtraction involves a branch. However, the branch can be avoided by replacing it with the minimum operation $\min(c, c-N)$ for unsigned integer values c and N with wrap-around two's complement arithmetic [18]. The Intel AVX-512F instruction set supports the `vpminuq` instruction for the 64-bit unsigned integer minimum operation. Here, it is sufficient that each of a and b be less than 2^{63} in order for the calculation of $c = a + b$ not to overflow with the 64-bit unsigned integer addition. Similarly, the modular subtraction $c = (a-b) \bmod N$ for $0 \le a, b < N$ can be replaced by the subtraction $c = a - b$ and the minimum operation $c = \min(c, c+N)$ for unsigned integer values a, b, c, and N with wrap-around two's complement arithmetic.

Figures 1 and 2 show the modular additions and subtractions of packed 63-bit integers using Intel AVX-512 intrinsics. The intrinsics support the `__m512i` data type in Figs. 1 and 2. The `__m512i` data type can hold 64 8-bit integer

Algorithm 3. Stockham radix-8 NTT algorithm

```
Input: $n = 8^q$, $X_0(j) = x(j)$, $0 \le j \le n-1$, and $\omega_n$ is the primitive $n$-th root of unity.
Output: $y(k) = X_q(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk} \bmod p$, $0 \le k \le n-1$
 1: $l \leftarrow n/8$
 2: $m \leftarrow 1$
 3: for $t$ from $1$ to $q$ do
 4:   for $j$ from $0$ to $l-1$ do
 5:     for $k$ from $0$ to $m-1$ do
 6:       $c_0 \leftarrow X_{t-1}(k+jm)$
 7:       $c_1 \leftarrow X_{t-1}(k+jm+lm)$
 8:       $c_2 \leftarrow X_{t-1}(k+jm+2lm)$
 9:       $c_3 \leftarrow X_{t-1}(k+jm+3lm)$
10:       $c_4 \leftarrow X_{t-1}(k+jm+4lm)$
11:       $c_5 \leftarrow X_{t-1}(k+jm+5lm)$
12:       $c_6 \leftarrow X_{t-1}(k+jm+6lm)$
13:       $c_7 \leftarrow X_{t-1}(k+jm+7lm)$
14:       $d_0 \leftarrow (c_0+c_4) \bmod p$
15:       $d_1 \leftarrow (c_0-c_4) \bmod p$
16:       $d_2 \leftarrow (c_2+c_6) \bmod p$
17:       $d_3 \leftarrow \omega_n^{n/4}(c_2-c_6) \bmod p$
18:       $d_4 \leftarrow (c_1+c_5) \bmod p$
19:       $d_5 \leftarrow (c_1-c_5) \bmod p$
20:       $d_6 \leftarrow (c_3+c_7) \bmod p$
21:       $d_7 \leftarrow \omega_n^{n/4}(c_3-c_7) \bmod p$
22:       $e_0 \leftarrow (d_0+d_2) \bmod p$
23:       $e_1 \leftarrow (d_0-d_2) \bmod p$
24:       $e_2 \leftarrow (d_4+d_6) \bmod p$
25:       $e_3 \leftarrow \omega_n^{n/4}(d_4-d_6) \bmod p$
26:       $e_4 \leftarrow (d_1+d_3) \bmod p$
27:       $e_5 \leftarrow (d_1-d_3) \bmod p$
28:       $e_6 \leftarrow \omega_n^{n/8}(d_5+d_7) \bmod p$
29:       $e_7 \leftarrow \omega_n^{3n/8}(d_5-d_7) \bmod p$
30:       $X_t(k+8jm) \leftarrow (e_0+e_2) \bmod p$
31:       $X_t(k+8jm+m) \leftarrow \omega_n^{jm}(e_4+e_6) \bmod p$
32:       $X_t(k+8jm+2m) \leftarrow \omega_n^{2jm}(e_1+e_3) \bmod p$
33:       $X_t(k+8jm+3m) \leftarrow \omega_n^{3jm}(e_5+e_7) \bmod p$
34:       $X_t(k+8jm+4m) \leftarrow \omega_n^{4jm}(e_0-e_2) \bmod p$
35:       $X_t(k+8jm+5m) \leftarrow \omega_n^{5jm}(e_4-e_6) \bmod p$
36:       $X_t(k+8jm+6m) \leftarrow \omega_n^{6jm}(e_1-e_3) \bmod p$
37:       $X_t(k+8jm+7m) \leftarrow \omega_n^{7jm}(e_5-e_7) \bmod p$
38:     end for
39:   end for
40:   $l \leftarrow l/8$
41:   $m \leftarrow 8m$
42: end for
```

Algorithm 4. Shoup's modular multiplication algorithm [7]

```
Input: $A$, $B$, $N$ such that $0 \leq A, B < N$, $N < \beta/2$
       precomputed $B' = \lfloor B\beta/N \rfloor$
Output: $C = AB \bmod N$
1: $q \leftarrow \lfloor AB'/\beta \rfloor$
2: $C \leftarrow (AB - qN) \bmod \beta$
3: if $C \geq N$ then
4:     $C \leftarrow C - N$
5: return $C$.
```

```
__m512i _mm512_addmod_epu64(__m512i a, __m512i b, __m512i N)
/*  Compute (a + b) mod N. Requires 0 <= a, b < N < 2^63. */
{
  __m512i c;

  c = _mm512_add_epi64(a, b);
  c = _mm512_min_epu64(c, _mm512_sub_epi64(c, N));

  return c;
}
```

Fig. 1. Modular additions of packed 63-bit integers using Intel AVX-512 intrinsics

values, 32 16-bit integer values, 16 32-bit integer values, or 8 64-bit integer values. In addition, the intrinsics `_mm512_add_epi64()` and `_mm512_sub_epi64()` correspond to the `vpaddq` and `vpsubq` instructions, respectively.

We consider performing modular multiplication $c = ab \bmod N$ using Shoup's modular multiplication. If we set $\beta = 2^{64}$ in Algorithm 4, then the upper 64-bit half of the 64-bit $\times$ 64-bit $\rightarrow$ 128-bit unsigned integer multiplication is required. The Intel AVX-512DQ (Doubleword and Quadword) instruction set [8] supports the `vpmullq` instruction for the lower 64-bit half of the 64-bit $\times$ 64-bit $\rightarrow$ 128-bit integer multiplication, but does not support the upper 64-bit half of the 64-bit $\times$ 64-bit $\rightarrow$ 128-bit unsigned integer multiplication.

The Intel AVX-512F instruction set supports the `vpmuludq` instruction, which performs 32-bit $\times$ 32-bit $\rightarrow$ 64-bit unsigned integer multiplication. The upper 64-bit half of the 64-bit $\times$ 64-bit $\rightarrow$ 128-bit unsigned integer multiplication can be implemented by dividing the multiplicand and multiplier of a 64-bit unsigned integer into the upper and lower 32-bit unsigned integers, respectively, and using the `vpmuludq` instruction for 32-bit $\times$ 32-bit $\rightarrow$ 64-

```
__m512i _mm512_submod_epu64(__m512i a, __m512i b, __m512i N)
/*  Compute (a - b) mod N. Requires 0 <= a, b < N < 2^63. */
{
  __m512i c;

  c = _mm512_sub_epi64(a, b);
  c = _mm512_min_epu64(c, _mm512_add_epi64(c, N));

  return c;
}
```

Fig. 2. Modular subtractions of packed 63-bit integers using Intel AVX-512 intrinsics

```
__m512i _mm512_mulhi_epu64(__m512i a, __m512i b)
/*  Compute floor((a * b) / 2^64). Requires 0 <= a, b < 2^64. */
{
  __m512i a0, a1, b0, b1, c, t0, t1, t2, t3;

  a0 = _mm512_and_epi64(a, _mm512_set1_epi64(0xFFFFFFFF));
  a1 = _mm512_srli_epi64(a, 32);
  b0 = _mm512_and_epi64(b, _mm512_set1_epi64(0xFFFFFFFF));
  b1 = _mm512_srli_epi64(b, 32);
  t0 = _mm512_mul_epu32(a0, b0);
  t1 = _mm512_mul_epu32(a0, b1);
  t2 = _mm512_mul_epu32(a1, b0);
  t3 = _mm512_mul_epu32(a1, b1);
  t1 = _mm512_add_epi64(t1, _mm512_srli_epi64(t0, 32));
  t2 = _mm512_add_epi64(t2, _mm512_and_epi64(t1,
                            _mm512_set1_epi64(0xFFFFFFFF)));
  c = _mm512_add_epi64(_mm512_srli_epi64(t1, 32),
                       _mm512_add_epi64(_mm512_srli_epi64(t2, 32), t3));

  return c;
}
```

Fig. 3. The upper 64-bit half of the 64-bit $\times$ 64-bit $\rightarrow$ 128-bit unsigned integer multiplications of packed 64-bit integers using Intel AVX-512 intrinsics

bit unsigned integer multiplication. Figure 3 shows the upper 64-bit half of the 64-bit $\times$ 64-bit $\rightarrow$ 128-bit unsigned integer multiplications of packed 64-bit integers using Intel AVX-512 intrinsics. The intrinsics `_mm512_and_epi64()`, `_mm512_set1_epi64()`, `_mm512_srli_epi64()`, and `_mm512_mul_epu32()` correspond to the `vpandq`, `vpbroadcastq`, `vpsrlq`, and `vpmuludq` instructions, respectively.

Figure 4 shows Shoup's modular multiplications of packed 63-bit integers using Intel AVX-512 intrinsics, which correspond to $\beta = 2^{64}$ in Algorithm 4. In this program, the function `_mm512_mulhi_epu64()` shown in Fig. 3 is used. The intrinsic `_mm512_mullo_epi64()` corresponds to the `vpmullq` instruction. The conditional subtraction on lines 3 and 4 of Algorithm 4 is also performed

```
__m512i _mm512_mulmod_epu64(__m512i a, __m512i b, __m512i bb, __m512i N)
/*  Compute (a * b) mod N. Precomputed bb = floor((b * 2^64) / N).
    Requires 0 <= a, b < N < 2^63. */
{
  __m512i c, q;

  q = _mm512_mulhi_epu64(a, bb);
  c = _mm512_sub_epi64(_mm512_mullo_epi64(a, b),
                       _mm512_mullo_epi64(q, N));
  c = _mm512_min_epu64(c, _mm512_sub_epi64(c, N));

  return c;
}
```

Fig. 4. Shoup's modular multiplications of packed 63-bit integers using Intel AVX-512 intrinsics

```
__m512i _mm512_mulmod_epu64(__m512i a, __m512i b, __m512i bb, __m512i N)
/*  Compute (a * b) mod N. Precomputed bb = floor((b * 2^52) / N).
    Requires 0 <= a, b < N < 2^51. */
{
  __m512i c, q;

  q = _mm512_madd52hi_epu64(_mm512_set1_epi64(0), a, bb);
  c = _mm512_sub_epi64(
      _mm512_madd52lo_epu64(_mm512_set1_epi64(0), a, b),
      _mm512_madd52lo_epu64(_mm512_set1_epi64(0), q, N));
  c = _mm512_and_epi64(c, _mm512_set1_epi64(0x000FFFFFFFFFFFFF));
  c = _mm512_min_epu64(c, _mm512_sub_epi64(c, N));

  return c;
}
```

Fig. 5. Shoup's modular multiplications of packed 51-bit integers using Intel AVX-512 intrinsics

Table 1. Inner-loop operations for radix-2, 4, and 8 NTT kernels

	Radix-2	Radix-4	Radix-8
Loads	2	4	8
Stores	2	4	8
Modular multiplications	1	4	12
Modular additions/subtractions	2	8	24
Total arithmetic operations	3	12	36
Byte/Operation ratio	10.667	5.333	3.556

using the `vpsubq` and `vpminuq` instructions in the same way as in the modular addition.

Intel AVX-512IFMA instructions [8] are supported by the Cannon Lake, Ice Lake, and Tiger Lake microarchitectures. The Intel AVX-512IFMA instruction set supports the `vpmadd52luq` and `vpmadd52huq` instructions, which multiply 52-bit unsigned integers and produce the low and high halves, respectively, of a 104-bit intermediate result. These halves are added to 64-bit accumulators. Since such operations are not supported in the C language, it is necessary to use Intel AVX-512 intrinsics or insert assembly-language instructions inline into the source code in order to use the Intel AVX-512IFMA instructions.

Figure 5 shows Shoup's modular multiplications of packed 51-bit integers using Intel AVX-512 intrinsics, which correspond to $\beta = 2^{52}$ in Algorithm 4. The intrinsics `_mm512_madd52lo_epu64()` and `_mm512_madd52hi_epu64()` correspond to the `vpmadd52luq` and `vpmadd52huq` instructions, respectively.

The Stockham radix-2, 4, and 8 NTTs are vectorized using the functions in Figs. 1, 2, 3, 4, and 5. Table 1 shows the inner-loop operations for radix-2, 4, and 8 NTT kernels. As mentioned in Sect. 2, the radix-4 or 8 NTT does not reduce the number of arithmetic operations compared to the radix-2 NTT. However, in view of the Byte/Operation ratio, the radix-8 NTT is preferable to the radix-2 and 4 NTTs. Although higher radix NTTs require more registers to hold intermediate results, processors that support the Intel AVX-512 instructions have 32 ZMM 512-bit registers.

A power-of-two point NTT (except for the 2-point NTT) can be performed by a combination of radix-8 and radix-4 steps containing at most two radix-4 steps. In other words, the power-of-two NTTs can be performed as a length $n = 2^p = 4^q 8^r$ $(p \geq 2,\ 0 \leq q \leq 2,\ r \geq 0)$.

4 Parallel Implementation of Number-Theoretic Transform

In Eq. (2), if n has factors n_1 and n_2 $(n = n_1 \times n_2)$, then the indices j and k can be expressed as:

$$j = j_1 + j_2 n_1, \quad k = k_2 + k_1 n_2. \tag{3}$$

We can define x and y in Eq. (2) as two-dimensional arrays (in column-major order):

$$x(j) = x(j_1, j_2), \qquad 0 \leq j_1 \leq n_1 - 1, \qquad 0 \leq j_2 \leq n_2 - 1, \tag{4}$$

$$y(k) = y(k_2, k_1), \qquad 0 \leq k_1 \leq n_1 - 1, \qquad 0 \leq k_2 \leq n_2 - 1. \tag{5}$$

Substituting the indices j and k in Eq. (2) with the indices in Eq. (3), and using the relation of $n = n_1 \times n_2$, we can derive the following equation:

$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_n^{j_2 k_2 n_1} \omega_n^{j_1 k_2} \omega_n^{j_1 k_1 n_2} \bmod p. \tag{6}$$

In the same way as the six-step FFT algorithm [1,20], the following six-step NTT algorithm is derived from Eq. (6):

Step 1: Transposition

$$x_1(j_2, j_1) = x(j_1, j_2).$$

Step 2: n_1 individual n_2-point multicolumn NTTs

$$x_2(k_2, j_1) = \sum_{j_2=0}^{n_2-1} x_1(j_2, j_1) \omega_n^{j_2 k_2 n_1} \bmod p.$$

Step 3: Twiddle factor multiplication

$$x_3(k_2, j_1) = x_2(k_2, j_1) \omega_n^{j_1 k_2} \bmod p.$$

Step 4: Transposition

$$x_4(j_1, k_2) = x_3(k_2, j_1).$$

Step 5: n_2 individual n_1-point multicolumn NTTs

$$x_5(k_1, k_2) = \sum_{j_1=0}^{n_1-1} x_4(j_1, k_2) \omega_n^{j_1 k_1 n_2} \bmod p.$$

Step 6: Transposition

$$y(k_2, k_1) = x_5(k_1, k_2).$$

In the six-step NTT algorithm, two multicolumn NTTs are performed in steps 2 and 5. The locality of the memory reference in the multicolumn NTT is high. On the other hand, the three transpose steps (steps 1, 4, and 6) are typically the chief bottlenecks in cache-based processors. We can use cache blocking to reduce the number of cache misses in matrix transposition. An example of matrix transposition with cache blocking is shown in Fig. 6. Parameter `NBLK` is the blocking parameter. In Fig. 6, the outermost loop length may not have sufficient parallelism for manycore processors. A loop collapsing makes the length of a loop long by collapsing nested loops into a single-nested loop. By using the OpenMP collapse clause, the parallelism of the outermost loop can be expanded [17].

We parallelized the six-step NTT using OpenMP. Figure 7 shows a parallel implementation of the six-step NTT. In this program, `transpose()` is a function to transpose a matrix, `mulmod()` is a function to perform a modular multiplication, `ntt2()` is the Stockham NTT, and a variable `omega` is the primitive n-th root of unity.

5 Performance Results

For performance evaluation, we compared the performance of the following six implementations:

- Proposed implementation of the Stockham NTT (AVX-512DQ) with a modulus of 63 bits
- Proposed implementation of the six-step NTT (AVX-512DQ) with a modulus of 63 bits
- Proposed implementation of the Stockham NTT (AVX-512IFMA) with a modulus of 51 bits
- Proposed implementation of the six-step NTT (AVX-512IFMA) with a modulus of 51 bits

Table 2. Specifications of the platform

Platform	Intel Xeon Platinum processor
Number of cores	38
Number of threads	76
CPU type	Intel Xeon Platinum 8368 Ice Lake 2.4 GHz
L1 cache (per core)	I-cache: 32 KB D-cache: 48 KB
L2 cache (per core)	1.25 MB
L3 cache	57 MB
Main memory	DDR4-3200 256 GB
Theoretical peak performance	2.918 TFlops
OS	Linux 4.18.0-305.25.1.el8_4.x86_64

```
void transpose(uint64_t *a, uint64_t *b, uint64_t n1, uint64_t n2)
{
  uint64_t i, ii, j, jj;

#pragma omp parallel for collapse(2) private(i, j, jj)
  for (ii = 0; ii < n1; ii += NBLK)
    for (jj = 0; jj < n2; jj += NBLK)
      for (i = ii; i < min(ii + NBLK, n1); i++)
        for (j = jj; j < min(jj + NBLK, n2); j++)
          b[j + i * n2] = a[i + j * n1];
}
```

Fig. 6. Example of matrix transposition with cache blocking

```
  uint64_t a[n1 * n2], b[n1 * n2], w[n1 * n2], ww[n1 * n2];
  uint64_t i, j, omega, p;

  ...
/* Step 1: transpose n1*n2 to n2*n1 */
  transpose(a, b, n1, n2);

/* Step 2: n1 individual n2-point multicolumn NTTs */
#pragma omp parallel for
  for (j = 0; j < n1; j++)
    ntt2(&b[j * n2], n2, omega, p);

/* Step 3: twiddle factor multiplication modulo p */
#pragma omp parallel for
  for (i = 0; i < n1 * n2; i++)
    a[i] = mulmod(b[i], w[i], ww[i], p);

/* Step 4: transpose n2*n1 to n1*n2 */
  transpose(a, b, n2, n1);

/* Step 5: n2 individual n1-point multicolumn NTTs */
#pragma omp parallel for
  for (j = 0; j < n2; j++)
    ntt2(&b[j * n1], n1, omega, p);

/* Step 6: transpose n1*n2 to n2*n1 */
  transpose(b, a, n1, n2);
```

Fig. 7. Parallel implementation of the six-step NTT

- Intel HEXL 1.2.4 (AVX-512DQ) with a modulus of 62 bits
- Intel HEXL 1.2.4 (AVX-512IFMA) with a modulus of 50 bits

Intel HEXL uses a modified Shoup butterfly [7] that requires $p < \beta/4$ to reduce the number of conditional subtractions [2]. Therefore, the modulus sizes of Intel HEXL (AVX-512DQ) with $\beta = 2^{64}$ and Intel HEXL (AVX-512IFMA) with $\beta = 2^{52}$ are 62 bits and 50 bits, respectively.

The specifications of the platform are shown in Table 2. Note that Hyper-Threading [10] was enabled on the platform. The Intel C compiler (version 19.1.3.304) was used for the proposed implementations. The compiler options were `icc -O3 -xICELAKE-SERVER -fno-alias -qopenmp`. The compiler option `-O3` enables optimizations for speed and more aggressive loop transformations. The compiler option `-xICELAKE-SERVER` specifies the generation of instructions for the Ice Lake microarchitecture. The compiler option `-fno-alias` specifies that aliasing is not assumed in a program. The compiler option `-qopenmp` specifies the enabling of the compiler to generate multi-threaded code based on the OpenMP directives. Intel HEXL could not be built successfully with the Intel C/C++ compiler, so the GNU C/C++ compiler (version 8.3.1) was used for Intel HEXL. The compiler option was `gcc -O3`.

Since the proposed implementation of the Stockham NTT is not parallelized, it was executed in a single thread. The proposed implementation of the six-step NTT was run with 1 to 76 threads. In the proposed implementations of the Stockham NTT and six-step NTT, the number of repetitions was doubled until the elapsed time of the forward NTT was greater than 1 second, and the average elapsed time was measured. The table for twiddle factors was prepared in advance. Since Intel HEXL does not support parallel execution, it was executed in a single thread. The performance of Intel HEXL was measured using the benchmark program included in the Intel HEXL source code.

On the Intel Xeon Platinum 8368, the environment variable `KMP_AFFINITY=granularity=fine,compact` was specified. The giga-operations per second (Gops) values are each based on $(3/2)n \log_2 n$ for a transform of size $n = 2^m$. This Gops value is calculated with modular addition, subtraction, and multiplication as one operation each, but several instructions are required to actually perform modular addition, subtraction, and multiplication.

Figure 8 shows the performance of NTTs using Intel AVX-512DQ instruction. As shown in Fig. 8, the proposed implementations of the Stockham NTT and six-step NTT (AVX-512DQ) are slower than Intel HEXL (AVX-512DQ) in a single-thread execution. One possible reason for this is that the modulus size of the proposed implementations of the Stockham NTT and six-step NTT (AVX-512DQ) is 63 bits, while the modulus size of Intel HEXL (AVX-512DQ) is 62 bits, reducing the number of instructions. While the six-step NTT is suitable for parallelization, it requires three matrix transpositions, and the overhead of these matrix transpositions is the reason why the proposed implementation of the six-step NTT is slower than the proposed implementation of the Stockham NTT in a single-thread execution. The Intel Xeon Platinum 8368 processor used in this performance evaluation has 57 MB of L3 cache, so up to 2^{20}-point NTT fits into the L3 cache. Although the six-step NTT and matrix transposition with cache blocking are effective when the data do not fit into the cache, Intel HEXL was only able to execute up to 2^{22}-point NTT, which may not have demonstrated the superiority of the proposed implementation of the six-step NTT. The proposed

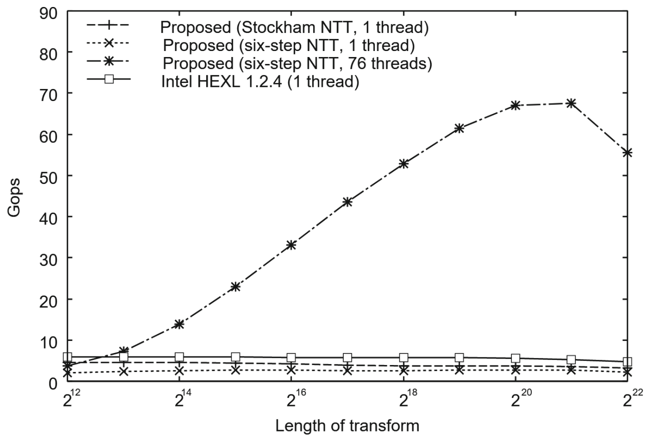

Fig. 8. Performance of NTTs using Intel AVX-512DQ instruction (Intel Xeon Platinum 8368, 38 cores)

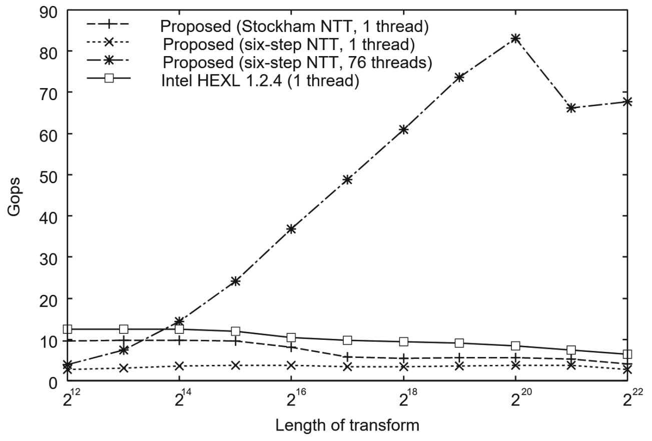

Fig. 9. Performance of NTTs using Intel AVX-512IFMA instruction (Intel Xeon Platinum 8368, 38 cores)

implementation of the six-step NTT (AVX-512DQ) is faster than Intel HEXL (AVX-512DQ) for $n \geq 2^{13}$ on 76 threads.

Figure 9 shows the performance of NTTs using Intel AVX-512IFMA instructions. The proposed implementations of the Stockham NTT and six-step NTT (AVX-512IFMA) are slower than Intel HEXL (AVX-512IFMA) in a single-thread execution, as shown in Fig. 9. One possible reason for this is that the modulus size of the proposed implementations of the Stockham NTT and six-step NTT (AVX-512IFMA) is 51 bits, while the modulus size of Intel HEXL (AVX-512IFMA) is 50 bits, reducing the number of instructions. The proposed implementation of the six-step NTT (AVX-512IFMA) is faster than Intel HEXL (AVX-512IFMA) for $n \geq 2^{14}$ on 76 threads.

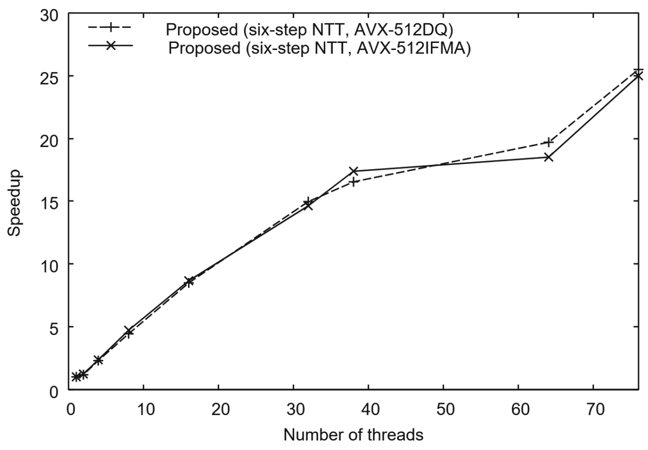

Fig. 10. Speedup for 2^{22}-point NTTs (Intel Xeon Platinum 8368, 38 cores)

Comparing Figs. 8 and 9, the proposed implementations of the Stockham NTT and six-step NTT (AVX-512IFMA) are faster than the proposed implementations of the Stockham NTT and six-step NTT (AVX-512DQ). The reason for this is that the proposed implementations of the Stockham NTT and six-step NTT (AVX-512IFMA) require fewer instructions to perform Shoup's modular multiplication using the Intel AVX-512DQ instruction. However, the modulus size is 63 bits for the proposed implementations of the Stockham NTT and six-step NTT (AVX-512DQ), while the modulus size is reduced to 51 bits for the proposed implementations of the Stockham NTT and six-step NTT (AVX-512IFMA).

Figure 10 shows the speedup for 2^{22}-point NTTs on the Intel Xeon Platinum 8368 when 1 to 76 threads are used. The results indicate that Hyper-Threading is effective for the proposed implementations of the six-step NTT (AVX-512DQ and AVX-512IFMA).

6 Conclusion

In this paper, we proposed the implementation of the parallel NTT using Intel AVX-512 instructions. The butterfly operation of the NTT can be performed using modular addition, subtraction, and multiplication. We showed that a method known as the six-step FFT algorithm could be applied to the NTT. We vectorized NTT kernels using the Intel AVX-512 instructions and parallelized the six-step NTT using OpenMP. We succeeded in obtaining a performance of over 83 Gops on an Intel Xeon Platinum 8368 (2.4 GHz, 38 cores) for a 2^{20}-point NTT with a modulus of 51 bits. These performance results demonstrate that the implemented parallel NTT uses cache memory effectively and exploits the Intel AVX-512 instructions.

Acknowledgments. This work was supported by JSPS KAKENHI Grant Number JP19K11989.

References

1. Bailey, D.H.: FFTs in external or hierarchical memory. J. Supercomput. **4**, 23–35 (1990)
2. Boemer, F., Kim, S., Seifu, G., de Souza, F.D.M., Gopal, V.: Intel HEXL: accelerating homomorphic encryption with Intel AVX512-IFMA52. In: Proceedings of 9th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC 2021), pp. 57–62 (2021)
3. Boemer, F., et al.: Intel HEXL. https://github.com/intel/hexl
4. Cochran, W.T., et al.: What is the fast Fourier transform? IEEE Trans. Audio Electroacoust. **15**, 45–55 (1967)
5. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**, 297–301 (1965)
6. Free Software Foundation Inc: GCC, the GNU Compiler Collection. https://gcc.gnu.org/
7. Harvey, D.: Faster arithmetic for number-theoretic transforms. J. Symb. Comput. **60**, 113–119 (2014)
8. Intel Corporation: Intel 64 and IA-32 architectures software developer's manual, volume 1: Basic architecture. https://software.intel.com/content/dam/develop/public/us/en/documents/253665-sdm-vol-1.pdf (2020)
9. Intel Corporation: Intel C++ compiler 19.1 developer guide and reference (2020). https://software.intel.com/content/dam/develop/external/us/en/documents/19-1-cpp-compiler-devguide.pdf
10. Marr, D.T., et al.: Hyper-threading technology architecture and microarchitecture. Intel. Technol. J. **6**, 1–11 (2002)
11. Meng, L., Johnson, J.: Automatic parallel library generation for general-size modular FFT algorithms. In: Gerdt, V.P., Koepf, W., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2013. LNCS, vol. 8136, pp. 243–256. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02297-0_21
12. Meng, L., Johnson, J.R., Franchetti, F., Voronenko, Y., Maza, M.M., Xie, Y.: Spiral-generated modular FFT algorithms. In: Proceedings of 4th International Workshop on Parallel and Symbolic Computation (PASCO 2010), pp. 169–170 (2010)
13. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**, 519–521 (1985)
14. Pollard, J.M.: The fast Fourier transform in a finite field. Math. Comput. **25**, 365–374 (1971)
15. Shoup, V.: NTL: a library for doing number theory. https://libntl.org
16. Swarztrauber, P.N.: FFT algorithms for vector computers. Parallel Comput. **1**, 45–63 (1984)
17. Takahashi, D.: An implementation of parallel 1-D real FFT on Intel Xeon phi processors. In: Gervasi, O., et al. (eds.) ICCSA 2017. LNCS, vol. 10404, pp. 401–410. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62392-4_29
18. Takahashi, D.: Computation of the 100 quadrillionth hexadecimal digit of π on a cluster of Intel Xeon phi processors. Parallel Comput. **75**, 1–10 (2018)
19. The Clang Team: clang: a C language family frontend for LLVM. https://clang.llvm.org/
20. Van Loan, C.: Computational Frameworks for the Fast Fourier Transform. SIAM Press, Philadelphia, PA (1992)