MS83 Next Generation FFT Algorithms in Theory and Practice: Parallel Implementations and Applications

- **Organizers:**
  - **Daisuke Takahashi**
    University of Tsukuba, Japan
  - **Franz Franchetti**
    Carnegie Mellon University, U.S.
  - **Samar A. Aseeri**
    King Abdullah University of Science & Technology (KAUST), Saudi Arabia

Aim of this minisymposium

- The fast Fourier Transform (FFT) is an algorithm used in a wide variety of applications, yet does not make optimal use of many current hardware platforms.
- Hardware utilization performance on its own does not however imply optimal problem solving.
- The purpose of this minisymposium is to enable exchange of information between people working on alternative FFT algorithms, to those working on FFT implementations, in particular for parallel hardware.
- http://www.fft.report

MS83

- **1:50-2:10 Parallel Implementation of FFT in a Finite Field**
  Daisuke Takahashi, University of Tsukuba, Japan
- **2:15-2:35 Updates on Sequential and Parallel FFTX**
  Franz Franchetti, Carnegie Mellon University, U.S.
- **2:40-3:00 A Comparison of Parallel Profiling Tools for Programs Utilizing the FFT**
  Samar A. Aseeri, King Abdullah University of Science & Technology (KAUST), Saudi Arabia; *Benson Muite,* University of Tartu, Estonia
- **3:05-3:25 Beyond 2D Parallelization of Multi-Dimensional FFTs**
  Doru Thom Popovici, Lawrence Berkeley National Laboratory, U.S.; *Martin D. Schatz,* University of Texas at Austin, U.S.; *Franz Franchetti* and *Tze Meng Low,* Carnegie Mellon University, U.S.

Parallel Implementation of FFT in a Finite Field

Daisuke Takahashi
Center for Computational Sciences
University of Tsukuba, Japan

Outline

- Background
- Objectives
- Number-theoretic Transform (NTT)
- Vectorization of NTT Kernels
- Parallel Implementation of NTT
- Performance Results
- Conclusion

Background (1/2)

- The fast Fourier transform (FFT) 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 they can also be computed in a ring and a finite field [Pollard 1971].
- Such a transform is called the number-theoretic transform (NTT).
- The NTT is used for homomorphic encryption and multiple-precision multiplication.

Background (2/2)

- 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 [Montgomery 1985] is known to avoid this.
- The butterfly operation of the NTT can be performed by using Montgomery multiplication.

Related Works

- NTL [Shoup et al.] is a C++ library for doing number theory, and it contains functions for NTTs.
    - Although the NTL is thread-safe, parallel NTT is not supported.
- SPIRAL-generated modular FFTs have been proposed [Meng et al. 2010].
    - Experiments were performed using 32-bit integers and 16-bit primes using Intel SSE4.1 instructions.
- An implementation of NTT using the Intel AVX-512IFMA (Integer Fused Multiply-Add) instructions has been proposed [Boemer et al. 2021].
    - This implementation is available as Intel HEXL in open source.

Objectives

- We vectorize NTT kernels using the Intel Advanced Vector Extensions 512 (AVX-512) instructions and parallelize NTT using OpenMP.

Number-theoretic Transform (NTT)

- The Discrete Fourier transform (DFT) can be defined over rings and fields other than the complex field [Pollard 1971].
- The definition of DFT can be expressed in a field $\mathbf{F}_p = \mathbf{Z}/p\mathbf{Z}$ where p is a prime number:

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

where ω_n is the primitive n-th root of unity.

Stockham Radix-2 NTT Algorithm

Algorithm 1 Stockham radix-2 NTT algorithm

Input: $n = 2^q$, $X_0(j) = x(j)$, $0 \le j \le n-1$, 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: $l \leftarrow n/2$
 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
```

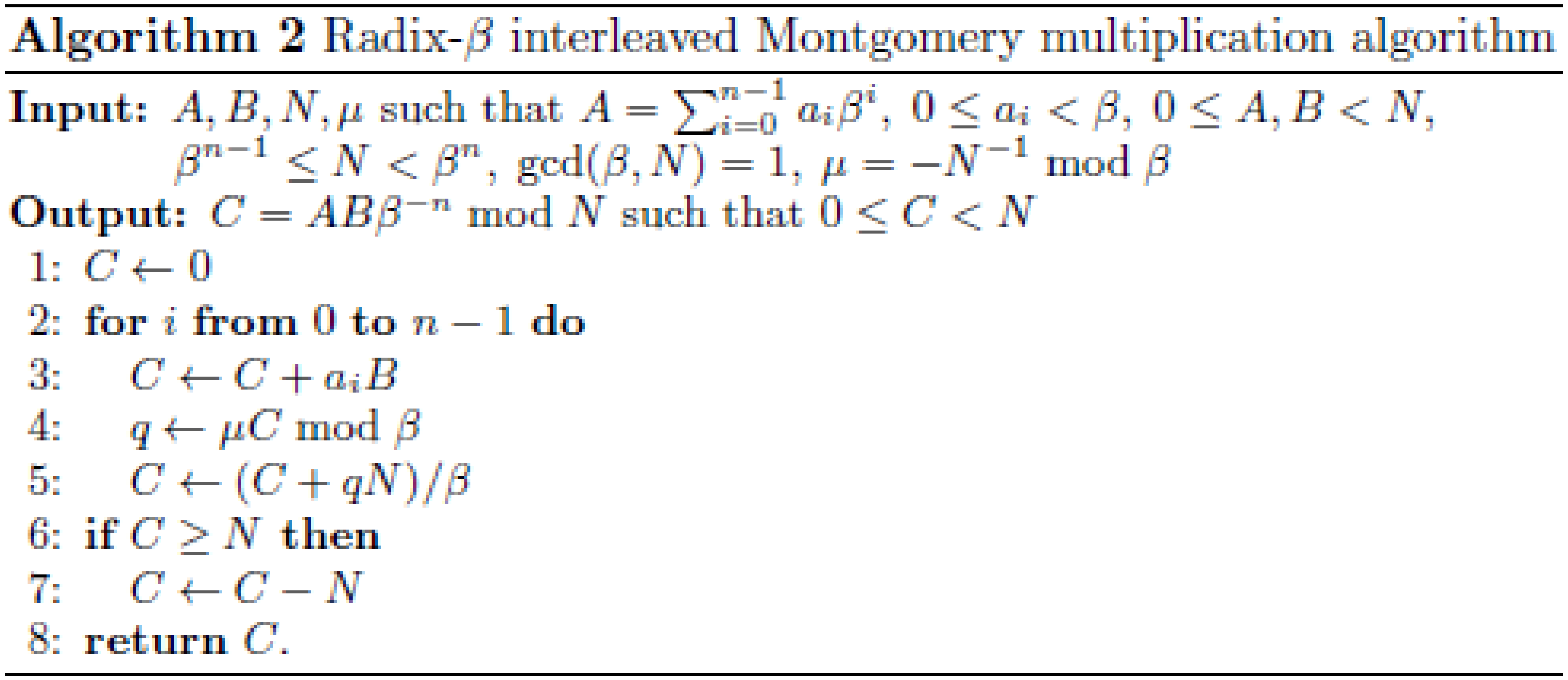
Vectorization of NTT Kernels (1/2)

- NTT kernels include modular addition, subtraction, and multiplication.
- The modular addition $c = (a + b) \bmod N$ for $a, b < N$ can be replaced by the addition $c = a + b$ 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.
- The Intel AVX-512F (Foundation) instruction set supports the **vpminuq** instruction for the 64-bit unsigned integer minimum operation.

Modular Additions and Subtractions of Packed 63-bit Integers Using Intel AVX-512 Intrinsics

```
__m512i _mm512_addmod_epu64(__m512i a, __m512i b, __m512i N)
/*  Compute (a[:] + b[:]) mod N[:]. */
{
  __m512i c;

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

  return c;
}

__m512i _mm512_submod_epu64(__m512i a, __m512i b, __m512i N)
/*  Compute (a[:] - b[:]) mod N[:]. */
{
  __m512i c;

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

  return c;
}
```

Radix-β interleaved Montgomery multiplication algorithm [Montgomery 1985, Bos et al. 2014]



Algorithm 2 Radix-β interleaved Montgomery multiplication algorithm

```
Input: $A, B, N, \mu$ such that $A = \sum_{i=0}^{n-1} a_i \beta^i$, $0 \le a_i < \beta$, $0 \le A, B < N$,
       $\beta^{n-1} \le N < \beta^n$, $\gcd(\beta, N) = 1$, $\mu = -N^{-1} \bmod \beta$
Output: $C = AB\beta^{-n} \bmod N$ such that $0 \le C < N$
1: $C \leftarrow 0$
2: for $i$ from $0$ to $n-1$ do
3:     $C \leftarrow C + a_i B$
4:     $q \leftarrow \mu C \bmod \beta$
5:     $C \leftarrow (C + qN)/\beta$
6: if $C \ge N$ then
7:     $C \leftarrow C - N$
8: return $C$.
```

Vectorization of NTT Kernels (2/2)

- The Intel 64 instruction set supports the **mulq** and **mulx** instructions, which perform 64-bit × 64-bit → 128-bit unsigned integer multiplication.
- In contrast, the Intel AVX-512F instruction set does not support 64-bit × 64-bit → 128-bit unsigned integer multiplication, but supports the **vpmuludq** instruction, which performs 32-bit × 32-bit → 64-bit unsigned integer multiplication.
- The radix-β interleaved Montgomery multiplication algorithm [Montgomery 1985, Bos et al. 2014] can be used to vectorize multiple Montgomery multiplications.
- In radix-2^{32} interleaved Montgomery multiplication, there are some overflows in 64-bit unsigned integer additions.
- A vectorized multiple Montgomery multiplications of 62-bit integers with $\beta = 2^{31}$ and $n = 2$ has been proposed to avoid these overflows [Takahashi 2020].

Montgomery Multiplication of Packed 62-bit Integers Using Intel AVX-512 Intrinsics

```
__m512i _mm512_mulmod_epu64(__m512i a, __m512i b, __m512i N, __m512i mu)
/*  Compute (a[:] * b[:] * 2^-62) mod N[:]. We need mu[:] = -N[:]^-1 mod 2^31. */
{
  __m512i a0, a1, b0, b1, c, N0, N1, q, t0, t1, t2, t3;

  a0 = _mm512_and_epi64(a, _mm512_set1_epi64(0x7FFFFFFF));
  a1 = _mm512_srli_epi64(a, 31);
  b0 = _mm512_and_epi64(b, _mm512_set1_epi64(0x7FFFFFFF));
  b1 = _mm512_srli_epi64(b, 31);
  N0 = _mm512_and_epi64(N, _mm512_set1_epi64(0x7FFFFFFF));
  N1 = _mm512_srli_epi64(N, 31);
  t0 = _mm512_mul_epu32(a0, b0);
  t1 = _mm512_mul_epu32(a0, b1);
  t2 = _mm512_mul_epu32(a1, b0);
  t3 = _mm512_mul_epu32(a1, b1);
  q = _mm512_and_epi64(_mm512_mul_epu32(t0, mu), _mm512_set1_epi64(0x7FFFFFFF));
  t0 = _mm512_add_epi64(_mm512_srli_epi64(_mm512_add_epi64(t0, _mm512_mul_epu32(q, N0)), 31),
                        _mm512_add_epi64(t1, _mm512_mul_epu32(q, N1)));
  t2 = _mm512_add_epi64(t2, _mm512_and_epi64(t0, _mm512_set1_epi64(0x7FFFFFFF)));
  t3 = _mm512_add_epi64(t3, _mm512_srli_epi64(t0, 31));
  q = _mm512_and_epi64(_mm512_mul_epu32(t2, mu), _mm512_set1_epi64(0x7FFFFFFF));
  t2 = _mm512_add_epi64(_mm512_srli_epi64(_mm512_add_epi64(t2, _mm512_mul_epu32(q, N0)), 31),
                        _mm512_add_epi64(t3, _mm512_mul_epu32(q, N1)));
  c = _mm512_min_epu64(t2, _mm512_sub_epi64(t2, N));

  return c;
}
```

In-Cache NTT Algorithm

- We use Stockham radix-2, 4, and 8 NTT algorithms for in-cache NTTs.
- It is known that the radix-4 or 8 FFT reduces the number of arithmetic operations compared to the radix-2 FFT.
- On the other hand, 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.

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	16
Modular additions/subtractions	2	8	32
Total arithmetic operations	3	12	48
Byte/Operation ratio	10.667	5.333	2.667

Six-Step NTT Algorithm

- If n has factors n_1 and n_2 $(n = n_1 \times n_2)$, in the same way as the six-step FFT algorithm [Bailey90], the following six-step NTT algorithm is derived:
- Step 1: Transposition
- Step 2: n_1 individual n_2-point multicolumn NTTs
- Step 3: Twiddle factor ($\omega_n^{j_1 k_2}$) multiplication
- Step 4: Transposition
- Step 5: n_2 individual n_1-point multicolumn NTTs
- Step 6: Transposition

Parallelization of Six-Step NTT

```
  uint64_t a[n1 * n2], b[n1 * n2], i, ii, j, jj, omega, p;
#pragma omp parallel {
#pragma omp 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];
#pragma omp for
  for (j = 0; j < n1; j++)
    ntt2(&b[j * n2], n2, omega, p);
  …
  }
```

A loop collapsing makes the length of a loop long by collapsing nested loops into a single-nested loop.

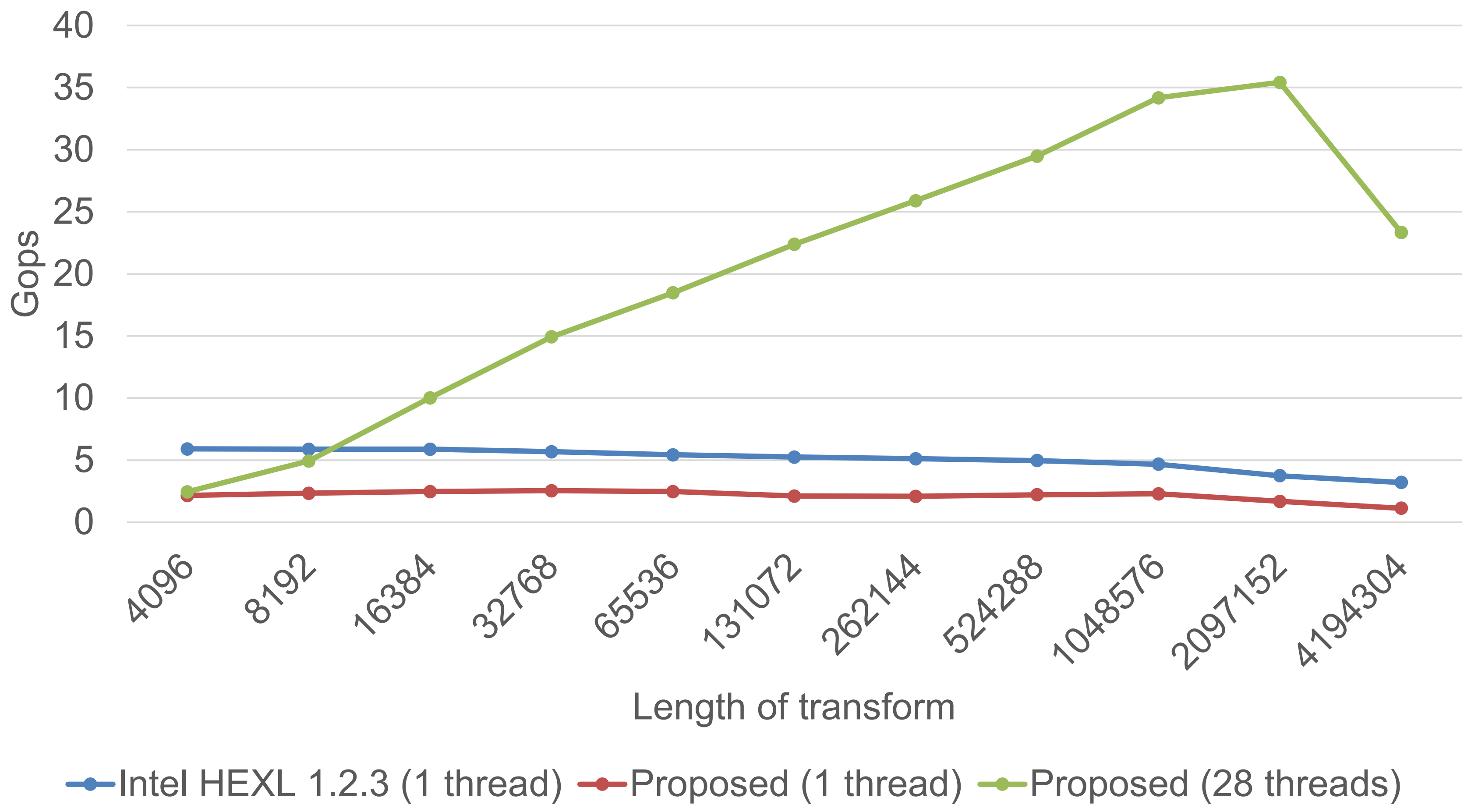
Performance Results

- For performance evaluation, a comparison between the implemented parallel NTT and the Intel HEXL (version 1.2.3) [Boemer et al. 2021] was performed.
- In the proposed implementation, NTT is performed with a modulus of 62 bits, while in Intel HEXL, NTT is performed with a modulus of 55 bits.
- The proposed implementation was run with 1 to 28 threads and the elapsed time was measured.
- Since Intel HEXL does not support parallel execution, it was executed in a single thread.
- The Giga Operations Per Second (Gops) values are each based on $(3/2)n \log_2 n$ for a transform of size $n = 2^m$.

Evaluation Environment

- HPE Superdome Flex
  - CPU: Intel Xeon Platinum 8280M (28 cores, Cascade Lake 2.7 GHz, DDR4 2933 MHz 24 TB)
  - Compiler: Intel C compiler 19.1.3.304 (for proposed)
    GNU C/C++ C compiler 8.4.0 (for Intel HEXL)
  - Compiler option: “icc -O3 -xCASCADELAKE -fno-alias -qopenmp -qopt-zmm-usage=high” (for proposed)
    “gcc -O3” (for Intel HEXL)

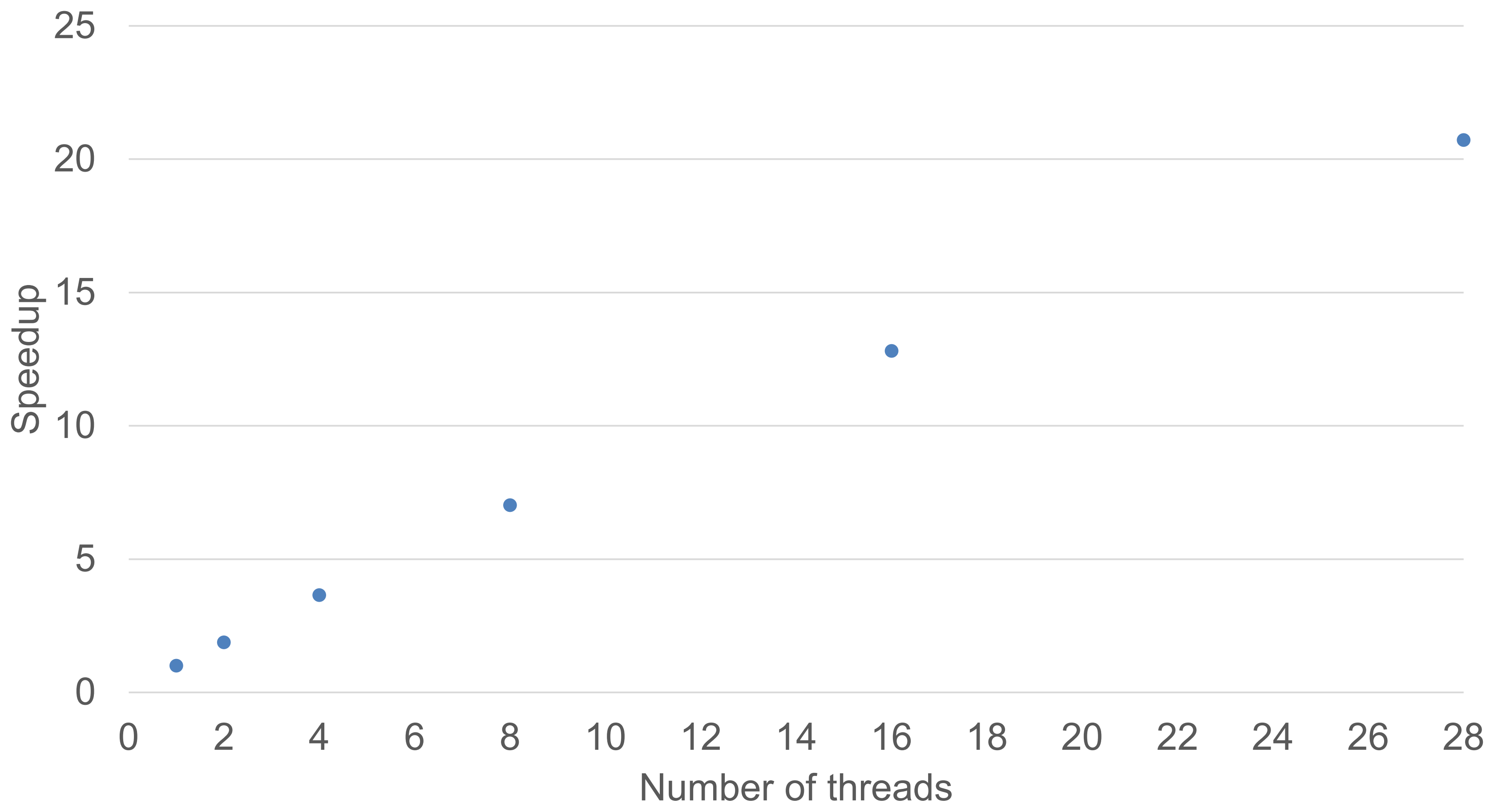
Performance of NTTs (Intel Xeon Platinum 8280M, 28 cores)



Discussion

- The proposed implementation is slower than Intel HEXL in a single-thread execution.
- The reason for this is that the modulus size is reduced to 55 bits in Intel HEXL which the proposed implementation has a modulus size of 62 bits.
- While the six-step NTT is suitable for parallelization, it requires three matrix transpositions, and the overhead of these matrix transpositions may be the reason why it is slower than Intel HEXL.
- Intel HEXL is highly optimized using Intel AVX-512DQ (Doubleword and Quadword) intrinsic.
- The proposed implementation is faster than Intel HEXL for $n \geq 2^{14}$ on 28 threads.

Speedup for 2^{22} -point NTTs (Intel Xeon Platinum 8280M, 28 cores)



Conclusion

- We proposed the implementation of the parallel number-theoretic transform (NTT).
- The butterfly operation of the NTT can be performed by using Montgomery multiplication.
- We vectorized NTT kernels using the Intel AVX-512 instructions and parallelized the six-step NTT by using OpenMP.
- Performance results demonstrate that the implemented parallel NTT utilizes cache memory effectively and exploits the Intel AVX-512 instructions.