Implementation of Multiple Multiplicative Inverses Modulo 2^w 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 multiple multiplicative inverses modulo 2^w using Intel Advanced Vector Extensions 512 (Intel AVX-512) instructions, where w is the bit width in the integer arithmetic operations. The proposed 52-bit implementation is based on a combination of Newton's method with quadratic convergence and a variant of Newton's method with cubic convergence. For multiple modular multiplicative inverses, the proposed 52-bit implementation using Intel AVX-512 instructions is up to approximately 9.95 and 1.92 times faster than a 64-bit implementation based on an inclusive scan using Intel 64 instructions and a 64-bit implementation based on Newton's method using Intel AVX-512 instructions, respectively, on an Intel Xeon Platinum 8468 processor.

Keywords: Modular multiplicative inverse · Newton's method · Intel AVX-512 instructions

1 Introduction

The modular multiplicative inverse is widely used in fields such as computational number theory and cryptography. The modular multiplicative inverse can be computed using the extended Euclidean algorithm, but Newton's method is more efficient when the modulus is a power of two [1–3,5,6,10,11]. An effective algorithm is presented for computing multiple modular multiplicative inverses for the same modulus [2].

We consider the computation of multiple multiplicative inverses modulo 2^w, where w is the bit width of the integer arithmetic operations. Such computations appear when computing Montgomery multiplications [12] for multiple moduli and exact divisions [9] for multiple divisors.

The multiple multiplicative inverses modulo 2^w are expected to be computed faster when vector instructions are used. Intel Advanced Vector Extensions 512 (Intel AVX-512) [7] is a 512-bit vector instruction set. In this paper, we propose an implementation of multiple multiplicative inverses modulo 2^w using Intel AVX-512 instructions.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025
O. Gervasi et al. (Eds.): ICCSA 2025, LNCS 15650, pp. 375–384, 2025.
https://doi.org/10.1007/978-3-031-96962-1_27

Algorithm 1. Newton's method for modular multiplicative inverse a^{-1} mod 2^{64} [11]

Input: a such that $0 < a < 2^{64}$, $2 \nmid a$
Output: $x = a^{-1} \bmod 2^{64}$
1: $x \leftarrow \{(3a) \oplus 2\} \bmod 2^{64}$
2: **for** i **from** 1 **to** 4 **do**
3: $\quad x \leftarrow x(2 - ax) \bmod 2^{64}$
4: **return** x.

```
uint64_t invmod64(uint64_t a)
/*  Compute x = a^-1 mod 2^64.
    Requires 0 < a < 2^64 and a mod 2 != 0. */
{
  uint64_t x;
  int i;

  x = (3 * a) ^ 2;
  for (i = 0; i < 4; i++)
    x *= 2 - a * x;

  return x;
}
```

Fig. 1. Modular multiplicative inverse of 64-bit integers.

The remainder of this paper is organized as follows. Section 2 describes the modular multiplicative inverse. Section 3 presents the implementation of multiple modular multiplicative inverses using Intel AVX-512 instructions. Section 4 presents the performance results. Finally, Sect. 5 presents the concluding remarks.

2 Modular Multiplicative Inverse

The modular multiplicative inverse of an integer a is an integer x such that $ax \equiv 1 \pmod{N}$. A necessary and sufficient condition for the existence of the inverse of a modulo N is that a and N are coprime (i.e., the greatest common divisor $\gcd(a, N)$ is 1).

Algorithm 1 shows Newton's method for modular multiplicative inverse a^{-1} mod 2^{64} [11]. Here, $(3a) \oplus 2$ is the correct multiplicative inverse modulo 2^5 (5 bits) [11], where $\oplus$ denotes the exclusive OR operation. Since Newton's method has quadratic convergence, four iterations are sufficient to obtain a^{-1} mod 2^{64}. 64-bit unsigned integer operations are performed with modulo 2^{64}, so there is no need to explicitly calculate the remainder divided by 2^{64}. Algorithm 1 requires nine multiplications, four subtractions, and one exclusive OR operation. Figure 1 shows the modular multiplicative inverse of 64-bit integers based on Algorithm 1.

Algorithm 2. Multiple modular multiplicative inverses [2]

Input: $0 < a_1, \ldots, a_k < N$
Output: $x_1 = a_1^{-1} \bmod N, \ldots, x_k = a_k^{-1} \bmod N$
1: $b_1 \leftarrow a_1$
2: **for** i **from** 2 **to** k **do**
3: $\quad b_i \leftarrow b_{i-1} a_i \bmod N$
4: $q \leftarrow b_k^{-1} \bmod N$
5: **for** i **from** k **downto** 2 **do**
6: $\quad x_i \leftarrow q b_{i-1} \bmod N$
7: $\quad q \leftarrow q a_i \bmod N$
8: $x_1 \leftarrow q$.

Algorithm 2 shows an algorithm for computing multiple modular multiplicative inverses [2]. This algorithm requires only one modular multiplicative inverse (on line 4) and $3(k-1)$ modular multiplications. Therefore, it is faster than computing the modular multiplicative inverse k times when the modular multiplicative inverse requires more than three times the computational cost of the modular multiplication [2]. Lines 3 and 7 of Algorithm 2 contain an inclusive scan that can be vectorized and parallelized.

Figure 2 shows the vectorized multiple modular multiplicative inverses of 64-bit integers based on Algorithm 2. In this program, the first and second `for` loops include an inclusive scan that can be vectorized using the `#pragma omp scan inclusive` directive supported since OpenMP 5.0. The function `invmod64()` shown in Fig. 1 is called to compute a 64-bit modular multiplicative inverse. It is also possible to compute multiple modular multiplicative inverses of 64-bit integers by executing Algorithm 1 multiple times, rather than using Algorithm 2.

3 Implementation of Multiple Modular Multiplicative Inverses Using Intel AVX-512 Instructions

The Intel AVX-512 is a 512-bit vector instruction set with several extensions that can be implemented independently. Intel AVX-512 implementations require only the Intel AVX-512F (Foundation) core extension. The Intel AVX-512DQ (Doubleword and Quadword) instruction set [7] supports the `vpmullq` instruction, which calculates the lower half of the product of two 64-bit integers. The Intel AVX-512IFMA (Integer Fused Multiply-Add) instruction set [7] supports the `vpmadd52huq` and `vpmadd52luq` instructions, which multiply 52-bit unsigned integers, produce the high and low halves of the result, respectively, and add them to 64-bit accumulators. Intel provides intrinsics [8] to facilitate the use of Intel AVX-512 instructions.

The `vpmadd52luq` instruction requires fewer cycles than the `vpmullq` instruction [4]. In this case, since Newton's method has quadratic convergence, it is faster to compute the first three of the four iterations in Algorithm 1 using

```
void minvmod64(uint64_t *x, uint64_t *a, uint64_t *q, uint64_t *b)
/*  Compute x[:] = a[:]^-1 mod 2^64.
    Requires 0 < a[:] < 2^64 and a[:] mod 2 != 0. */
{
  uint64_t tmp;
  int i;

  b[0] = a[0];
  tmp = b[0];
#pragma omp simd reduction(inscan, *:tmp)
  for (i = 1; i < VLEN; i++) {
    tmp *= a[i];
#pragma omp scan inclusive(tmp)
    b[i] = tmp;
  }

  q[VLEN - 1] = invmod64(b[VLEN - 1]);
  tmp = q[VLEN - 1];
#pragma omp simd reduction(inscan, *:tmp)
  for (i = VLEN - 1; i >= 1; i--) {
    tmp *= a[i];
#pragma omp scan inclusive(tmp)
    q[i - 1] = tmp;
  }

#pragma omp simd
  for (i = VLEN - 1; i >= 1; i--)
    x[i] = q[i] * b[i - 1];
  x[0] = q[0];
}
```

Fig. 2. Vectorized multiple modular multiplicative inverses of 64-bit integers using inclusive scan.

52-bit unsigned integer operations and the last iteration using 64-bit unsigned integer operations. The Intel AVX-512IFMA instruction set does not support the fused multiply-subtract (FMS) instruction. In the calculation of $2 - ax$ in Algorithm 1, $-a$ can be calculated by subtracting a from 0 in advance and then using the fused multiply-add (FMA) instruction.

Figure 3 shows the vectorized multiple modular multiplicative inverses of 64-bit integers using Intel AVX-512 intrinsics. This program iterates Algorithm 1 `VLEN` (vector length) times on a vector `a[:]` of 64-bit integers.

The `__m512i` data type shown in Fig. 3 is supported by Intel AVX-512 intrinsics. The `__m512i` data type can hold 64 8-bit integer values, 32 16-bit integer values, 16 32-bit integer values, or 8 64-bit integer values. In this program, the intrinsics `_mm512_set1_epi64()`, `_mm512_load_epi64()`, `_mm512_sub_epi64()`, `_mm512_xor_epi64()`, and `_mm512_store_epi64()` correspond to the Intel AVX-512F `vpbroadcastq`, `vmovdqa64`, `vpsubq`, `vpxorq`, and `vmovdqa64` instr-

```
void vinvmod64(uint64_t *x, uint64_t *a)
/*  Compute x[:] = a[:]^-1 mod 2^64.
    Requires 0 < a[:] < 2^64 and a[:] mod 2 != 0. */
{
  __m512i a64, neg_a, three, two, x52, x64, zero;
  int i, j;

  zero = _mm512_set1_epi64(0);
  two = _mm512_set1_epi64(2);
  three = _mm512_set1_epi64(3);

  for (j = 0; j < VLEN; j += 8) {
    a64 = _mm512_load_epi64(&a[j]);
    neg_a = _mm512_sub_epi64(zero, a64);
    x52 = _mm512_xor_epi64(_mm512_madd52lo_epu64(zero, three, a64), two);
    for (i = 0; i < 3; i++)
      x52 = _mm512_madd52lo_epu64(zero, x52,
                                  _mm512_madd52lo_epu64(two, neg_a, x52));
    x64 = _mm512_mullo_epi64(x52, _mm512_sub_epi64(two,
                                    _mm512_mullo_epi64(a64, x52)));
    _mm512_store_epi64(&x[j], x64);
  }
}
```

Fig. 3. Vectorized multiple modular multiplicative inverses of 64-bit integers using Intel AVX-512 intrinsics.

uctions, respectively. In addition, the intrinsics `_mm512_madd52lo_epu64()` and `_mm512_mullo_epi64()` correspond to the `vpmadd52luq` and `vpmullq` instructions, respectively. If the vector length `VLEN` in the program shown in Fig. 3 is not divisible by 8, a remainder loop is required.

We propose a combination of Newton's method with quadratic convergence and a variant of Newton's method with cubic convergence for the modular multiplicative inverse $a^{-1} \bmod 2^{52}$, as shown in Algorithm 3. Here, a is the correct multiplicative inverse modulo 2^3 (3 bits) [6]. To obtain $a^{-1} \bmod 2^{52}$, it is sufficient to use a variant of Newton's method with cubic convergence twice and Newton's method with quadratic convergence once. Algorithm 3 requires eight multiplications, four additions, and three subtractions. The total number of operations in Algorithm 3 is greater than that in Algorithm 1. For a processor with FMA and FMS instructions, Algorithm 1 requires four FMS operations, five multiplications, and one exclusive OR operation. In contrast, Algorithm 3 requires four FMA operations, three FMS operations, and one multiplication.

Algorithm 3. Combination of Newton's method with quadratic convergence and variant of Newton's method with cubic convergence for modular multiplicative inverse $a^{-1} \bmod 2^{52}$

Input: a such that $0 < a < 2^{52}$, $2 \nmid a$
Output: $x = a^{-1} \bmod 2^{52}$
1: $x \leftarrow a$
2: **for** i **from** 1 **to** 2 **do**
3: $\quad t \leftarrow (1 - ax) \bmod 2^{52}$
4: $\quad x \leftarrow \left\{x + x(t + t^2)\right\} \bmod 2^{52}$
5: $x \leftarrow x(2 - ax) \bmod 2^{52}$
6: **return** x.

Figure 4 shows the vectorized multiple modular multiplicative inverses of 52-bit integers using Intel AVX-512 intrinsics. This program iterates Algorithm 3 `VLEN` (vector length) times on a vector `a[:]` of 52-bit integers.

The integer operations in Algorithm 3 are performed with modulo 2^{52}, but the intrinsic `_mm512_madd52lo_epu64()` returns the result of adding a 64-bit unsigned integer to the lower half of the product of two 52-bit unsigned integers (i.e., modulo 2^{64}). If the first argument of the intrinsic `_mm512_madd52lo_epu64()` is set to zero, the lower half of the product of two 52-bit unsigned integers (i.e., modulo 2^{52}) is returned. Therefore, it is sufficient to perform the multiplication of x and $2-ax$ modulo 2^{52} using Newton's method with quadratic convergence in line 5 of Algorithm 3.

4 Performance Results

To evaluate performance, we compared the performance of the following implementations of multiple modular multiplicative inverses:

- 64-bit implementations of Algorithms 1 and 2 using Intel 64 instructions
- 64-bit implementations of Algorithms 1 and 2 using Intel AVX-512 instructions
- 52-bit implementation of Algorithm 3 using Intel AVX-512 instructions

In the 64-bit implementation of Algorithm 2 using Intel 64 instructions, vectorization was suppressed by commenting out the directives beginning with `#pragma omp` in the program shown in Fig. 2.

The batch size of the modular multiplicative inverses was varied from 512 to 4096. The input integer a was set to be a random odd number in the range $[1, 2^{64}-1]$ for the 64-bit implementations of Algorithms 1 and 2, and in the range $[1, 2^{52}-1]$ for the 52-bit implementation of Algorithm 3. Each batch of modular multiplicative inverses was executed one million times. The number of modular multiplicative inverses per second (Invmod$\times 10^9$/s) was calculated based on the average elapsed time.

The specifications of the platform used in the evaluations are listed in Table 1. The Intel Xeon Platinum 8468 processor has 48 cores. However, to focus on

```
void vinvmod52(uint64_t *x, uint64_t *a)
/*  Compute x[:] = a[:]^-1 mod 2^52.
    Requires 0 < a[:] < 2^52 and a[:] mod 2 != 0. */
{
  __m512i a52, neg_a, one, t, two, x52, zero;
  int i, j;

  zero = _mm512_set1_epi64(0);
  one = _mm512_set1_epi64(1);
  two = _mm512_set1_epi64(2);

  for (j = 0; j < VLEN; j += 8) {
    a52 = _mm512_load_epi64(&a[j]);
    neg_a = _mm512_sub_epi64(zero, a52);
    x52 = a52;
    for (i = 0; i < 2; i++) {
      t = _mm512_madd52lo_epu64(one, neg_a, x52);
      x52 = _mm512_madd52lo_epu64(x52, x52,
                                  _mm512_madd52lo_epu64(t, t, t));
    }
    x52 = _mm512_madd52lo_epu64(zero, x52,
                                _mm512_madd52lo_epu64(two, neg_a, x52));
    _mm512_store_epi64(&x[j], x52);
  }
}
```

Fig. 4. Vectorized multiple modular multiplicative inverses of 52-bit integers using Intel AVX-512 intrinsics.

vectorization, performance was evaluated on a single core and a single thread. The program shown in Fig. 2 uses four arrays of 64-bit unsigned integers, so the dataset fits into the 48 KB L1 data cache for batch sizes of up to 1024. The programs shown in Figs. 3 and 4 use two arrays of 64-bit unsigned integers, so the dataset fits into the 48 KB L1 data cache for batch sizes of up to 2048.

The Intel oneAPI DPC++/C++ Compiler (version 2024.0.2) was used. The compiler options were `icx -O3 -xSAPPHIRERAPIDS -qopenmp-simd -qopt-zmm-usage=high`. The compiler option `-O3` enables optimizations for speed and more aggressive loop transformations. The compiler option `-xSAPPHIRERAPIDS` generates instructions for the Sapphire Rapids microarchitecture. The compiler option `-qopenmp-simd` enables OpenMP SIMD compilation. The compiler option `-qopt-zmm-usage=high` sets the level of zmm register usage to high.

Table 2 shows the performance of the various implementations of multiple modular multiplicative inverses. For a batch size of 2048, the 52-bit implementation of Algorithm 3 using Intel AVX-512 instructions was approximately 9.95 and 1.92 times faster than the 64-bit implementation of Algorithm 2 using Intel 64

Table 1. Specifications of platform used in evaluations

Platform	Intel Xeon Platinum processor
Number of cores	48
Number of threads	96
CPU type	Intel Xeon Platinum 8468 Sapphire Rapids 2.1 GHz
L1 cache (per core)	I-cache: 32 KB D-cache: 48 KB
L2 cache (per core)	2 MB
L3 cache	105 MB
Main memory	DDR5-4400 128 GB
OS	Ubuntu 22.04 (kernel 5.15.0-100-generic)

Table 2. Performance of various implementations of multiple modular multiplicative inverses (Invmod$\times 10^9$/s)

Batch size	Intel 64		Intel AVX-512		
	64-bit		64-bit		52-bit
	Algorithm 1	Algorithm 2	Algorithm 1	Algorithm 2	Algorithm 3
512	0.38605	0.49243	2.40270	0.37828	4.60871
1024	0.38681	0.48999	2.40460	0.34734	4.60600
2048	0.38712	0.46430	2.40560	0.38465	4.62205
4096	0.38731	0.45022	2.40201	0.33912	3.95695

instructions and the 64-bit implementation of Algorithm 1 using Intel AVX-512 instructions, respectively, on the Intel Xeon Platinum 8468 processor.

When using Intel AVX-512 instructions, the 52-bit implementation of Algorithm 3 shows a speedup that exceeds the bit width ratio (64 bits / 52 bits $\approx$ 1.23 times) over the 64-bit implementation of Algorithm 1. This is because the 52-bit implementation of Algorithm 3 uses fewer instructions than the 64-bit implementation of Algorithm 1. In addition, the 64-bit implementation of Algorithm 1 uses the `vpmullq` instruction in the last iteration of Newton's method, which is slower than the `vpmadd52luq` instruction.

The 64-bit implementation of Algorithm 1 using Intel AVX-512 instructions is faster than the 64-bit implementation of Algorithm 1 using Intel 64 instructions. This is because the Intel AVX-512 instructions calculate eight 64-bit integers simultaneously, whereas the Intel 64 instructions calculate one 64-bit integer at a time.

When using Intel 64 instructions, the 64-bit implementation of Algorithm 2 is faster than the 64-bit implementation of Algorithm 1. The computational cost of the modular multiplicative inverse on line 4 of Algorithm 2 is negligible if k is sufficiently large, and lines 3, 6, and 7 of Algorithm 2 require only three

multiplications in total per iteration. In contrast, Algorithm 1 requires nine multiplications, four subtractions, and one exclusive OR operation.

On the other hand, when using Intel AVX-512 instructions, the 64-bit implementation of Algorithm 2 is slower than the 64-bit implementation of Algorithm 1. This is because the `vpmullq` instruction is slow, and vectorizing the inclusive scan increases the number of operations.

5 Conclusion

In this paper, we proposed an implementation of multiple multiplicative inverses modulo 2^w using Intel AVX-512 instructions. The proposed 52-bit implementation is based on a combination of Newton's method with quadratic convergence and a variant of Newton's method with cubic convergence. For multiple modular multiplicative inverses, the proposed 52-bit implementation using Intel AVX-512 instructions is up to approximately 9.95 and 1.92 times faster than the 64-bit implementation based on an inclusive scan using Intel 64 instructions and the 64-bit implementation based on Newton's method using Intel AVX-512 instructions, respectively, on an Intel Xeon Platinum 8468 processor.

Acknowledgments. This work was supported by JSPS KAKENHI Grant Number JP22K12045.

References

1. Arazi, O., Qi, H.: On calculating multiplicative inverses modulo 2^m. IEEE Trans. Comput. **57**, 1435–1438 (2008)
2. Brent, R.P., Zimmermann, P.: Modern Computer Arithmetic. Cambridge University Press, Cambridge (2010)
3. Dumas, J.G.: On Newton-Raphson iteration for multiplicative inverses modulo prime powers. IEEE Trans. Comput. **63**, 2106–2109 (2014)
4. Fog, A.: Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs (2022). https://www.agner.org/optimize/instruction_tables.pdf
5. Granlund, T., Montgomery, P.L.: Division by invariant integers using multiplication. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 61–72 (1994)
6. Hurchalla, J.: An improved integer modular multiplicative inverse (modulo 2^w). Comput. Res. Repos. (2022). https://arxiv.org/abs/2204.04342
7. Intel Corporation: Intel 64 and IA-32 architectures software developer's manual, volume 1: Basic architecture (2024). https://cdrdv2-public.intel.com/671436/253665-sdm-vol-1.pdf
8. Intel Corporation: Intel intrinsics guide (2024). https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
9. Jebelean, T.: An algorithm for exact division. J. Symb. Comput. **15**, 169–180 (1993)

10. Koç, Ç.K.: A new algorithm for inversion modp^k. IACR Cryptol. ePrint Arch. (2017). https://eprint.iacr.org/2017/411
11. Mayer, E.W.: Efficient long division via Montgomery multiply. Comput. Res. Repos. (2016). https://arxiv.org/abs/1303.0328
12. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**, 519–521 (1985)