

Fast Multiple Montgomery Multiplications Using Intel AVX-512IFMA 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 a fast implementation of multiple Montgomery multiplications using Intel AVX-512IFMA (Integer Fused Multiply-Add) instructions. The proposed implementation is based on a modified Montgomery multiplication. For Montgomery multiplication operands with 52 bits or fewer, the proposed implementation using Intel AVX-512IFMA instructions is up to approximately 12.22 and 4.30 times faster than the implementations using Intel 64 and Intel AVX-512F (Foundation) instructions on an Intel Core i3-8121U processor, respectively.

Keywords: Modular multiplication · Montgomery multiplication · Intel AVX-512IFMA instructions

1 Introduction

Modular multiplication, widely used in fields such as computer algebra and cryptography, includes modulo operations, which are slow because they involve an integer division process. Montgomery multiplication [1] avoids this drawback. Modern processors support single-instruction multiple-data (SIMD) vector instructions. The Intel Advanced Vector Extensions 512 (Intel AVX-512) [2] is a 512-bit vector instruction set. Montgomery multiplication algorithms that use vector instructions have been proposed [3,4], as has fast modular squaring with Intel AVX-512IFMA (Integer Fused Multiply-Add) [5]. Vector instructions have also been used to compute multiple Montgomery multiplications in parallel [4]. For multiple Montgomery multiplications, implementations based on Intel Streaming SIMD Extensions 2 (Intel SSE2) instructions [6] and the Cell processor [7] have been proposed.

Montgomery multiplication is usually performed on integers of several hundred bits or more; however, this paper considers the case of 52 bits or fewer. Multiple Montgomery multiplications with such numbers of bits are used in the modular fast Fourier transform (FFT) algorithm [8]. To the best of our knowledge, there are no existing implementations of multiple Montgomery multiplications using Intel AVX-512IFMA instructions. Here, we propose such an implementation and evaluate its performance.

© Springer Nature Switzerland AG 2020
O. Gervasi et al. (Eds.): ICCSA 2020, LNCS 12253, pp. 655–663, 2020.
https://doi.org/10.1007/978-3-030-58814-4_52

Algorithm 1. Montgomery multiplication algorithm [1]

Input: A, B, N such that $0 \leq A, B < N$, $\beta > N$, $\gcd(\beta, N) = 1$, $\mu = -N^{-1} \bmod \beta$
Output: $C = AB\beta^{-1} \bmod N$ such that $0 \leq C < N$
1: $C \leftarrow AB$
2: $q \leftarrow \mu C \bmod \beta$
3: $C \leftarrow (C + qN)/\beta$
4: **if** $C \geq N$ **then**
5: $\quad C \leftarrow C - N$
6: **return** C.

```
uint64_t mulmod63(uint64_t a, uint64_t b, uint64_t N, uint64_t mu)
/*  Compute mulmod63 = (a * b * 2^-63) mod N.
    We need mu = -N^-1 mod 2^63. */
{
  __uint128_t t;
  uint64_t c, q;

  t = (__uint128_t) a * b;
  q = ((uint64_t) t * mu) & 0x7FFFFFFFFFFFFFFF;
  c = (t + (__uint128_t) q * N) >> 63;
  if (c >= N)
    c -= N;

  return c;
}
```

Fig. 1. Montgomery multiplication of 63-bit integers

The remainder of this paper is organized as follows. Section 2 describes the vectorization of multiple Montgomery multiplications. Section 3 presents the proposed implementation of multiple Montgomery multiplications using Intel AVX-512IFMA instructions. Section 4 presents the performance results. Finally, Sect. 5 gives the concluding remarks.

2 Vectorization of Multiple Montgomery Multiplications

Montgomery multiplication [1] is shown in Algorithm 1. Figure 1 shows the Montgomery multiplication of 63-bit integers, which corresponds to $\beta = 2^{63}$ in Algorithm 1. This program uses the `__uint128_t` extension for 128-bit unsigned integer arithmetic supported by the GCC, Clang, and Intel C compilers. Because this program computes a single Montgomery multiplication, vectorization, which requires multiple Montgomery multiplications to be performed simultaneously, cannot be applied.

The Intel 64 instruction set supports `mulq` and `mulx` instructions, which perform 64-bit $\times$ 64-bit $\rightarrow$ 128-bit unsigned integer multiplication. In contrast, the Intel AVX-512F (Foundation) instruction set supports only the `vpmuludq`

Algorithm 2. Radix-β interleaved Montgomery multiplication algorithm [1, 4]

Input: A, B, N, μ 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: $\quad C \leftarrow C + a_i B$
4: $\quad q \leftarrow \mu C \bmod \beta$
5: $\quad C \leftarrow (C + qN)/\beta$
6: **if** $C \ge N$ **then**
7: $\quad C \leftarrow C - N$
8: **return** C.

instruction, which performs 32-bit × 32-bit → 64-bit unsigned integer multiplication. Furthermore, a statement that contains `__uint128_t` variables cannot be automatically vectorized by the Intel C compiler. The radix-β interleaved Montgomery multiplication algorithm [1,4], shown in Algorithm 2, can be used to vectorize multiple Montgomery multiplications. In radix-2^{32} interleaved Montgomery multiplication, there is some overflow in 64-bit unsigned integer addition. A vectorized multiple Montgomery squaring operation of 62-bit integers with $\beta = 2^{31}$ and $n = 2$ has been proposed to avoid this overflow [9].

We modified the vectorized multiple Montgomery squaring operation to achieve vectorized multiple Montgomery multiplications. For multiple Montgomery multiplications, performance is degraded by the conditional subtraction $C - N$ when $C \ge N$ on lines 6 and 7 of Algorithm 2. The conditional subtraction can be replaced by the minimum operation $\min(C, C - N)$ for unsigned integer values C and N with wrap-around two's complement arithmetic [9]. The Intel AVX-512F instruction set supports the `vpminuq` instruction for the 64-bit unsigned integer minimum operation.

Figure 2 shows the vectorized multiple Montgomery multiplications of 62-bit integers, which correspond to $\beta = 2^{31}$ and $n = 2$ in Algorithm 2. In Fig. 2, `#pragma ivdep` instructs the compiler to ignore assumed vector dependencies and `#pragma vector aligned` instructs the compiler to use aligned data movement instructions for all array references during vectorization. The vectorized multiple Montgomery multiplications can be further optimized using the Intel AVX-512 intrinsics [10], as shown in Fig. 3. In this program, the Intel AVX-512 intrinsics `_mm512_mul_epu32()` and `_mm512_min_epu64()` correspond to the `vpmuludq` and `vpminuq` instructions, respectively. This program assumes that the vector length `VLEN` is divisible by 8. If `VLEN` is not divisible by 8, a remainder loop needs to be executed.

```
void vmulmod62(uint64_t *c, uint64_t *a, uint64_t *b, uint64_t *N,
               uint32_t *mu)
/*  Compute c[:] = (a[:] * b[:] * 2^-62) mod N[:].
    We need mu[:] = -N[:]^-1 mod 2^31. */
{
  uint64_t t0, t1, t2, t3;
  uint32_t a0, a1, b0, b1, N0, N1, q;
  int i;

#pragma ivdep
#pragma vector aligned
  for (i = 0; i < VLEN; i++) {
    a0 = a[i] & 0x7FFFFFFF;
    a1 = a[i] >> 31;
    b0 = b[i] & 0x7FFFFFFF;
    b1 = b[i] >> 31;
    N0 = N[i] & 0x7FFFFFFF;
    N1 = N[i] >> 31;
    t0 = (uint64_t) a0 * b0;
    t1 = (uint64_t) a0 * b1;
    t2 = (uint64_t) a1 * b0;
    t3 = (uint64_t) a1 * b1;
    q = ((uint32_t) t0 * mu[i]) & 0x7FFFFFFF;
    t0 = ((t0 + (uint64_t) q * N0) >> 31) + (t1 + (uint64_t) q * N1);
    t2 += t0 & 0x7FFFFFFF;
    t3 += t0 >> 31;
    q = ((uint32_t) t2 * mu[i]) & 0x7FFFFFFF;
    t2 = ((t2 + (uint64_t) q * N0) >> 31) + (t3 + (uint64_t) q * N1);
    c[i] = min(t2, t2 - N[i]);
  }
}
```

Fig. 2. Vectorized multiple Montgomery multiplications of 62-bit integers

3 Implementation of Multiple Montgomery Multiplications Using Intel AVX-512IFMA Instructions

Intel AVX-512IFMA instructions [2] are supported by the Cannon Lake and Ice Lake microarchitectures, and will be supported by the Tiger Lake microarchitecture. 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.

Algorithm 3 shows a modified Montgomery multiplication algorithm. $C+qN$ on line 3 of Algorithm 1 is divisible by β. That is, the lower-half bits of $C+qN$ are 0, and thus do not need to be computed. In this case, if q on line 2 of Algorithm 1 is 0, the carry added to the sum of the upper-half bits of C and the upper-half

```
void vmulmod62(uint64_t *c, uint64_t *a, uint64_t *b, uint64_t *N,
               uint64_t *mu)
/*  Compute c[:] = (a[:] * b[:] * 2^-62) mod N[:].
    We need mu[:] = -N[:]^-1 mod 2^31. */
{
  __m512i a0, a1, b0, b1, N0, N1, q, t0, t1, t2, t3;
  int i;

  for (i = 0; i < VLEN; i += 8) {
    a0 = _mm512_and_epi64(_mm512_load_epi64(&a[i]),
                          _mm512_set1_epi64(0x7FFFFFFF));
    a1 = _mm512_srli_epi64(_mm512_load_epi64(&a[i]), 31);
    b0 = _mm512_and_epi64(_mm512_load_epi64(&b[i]),
                          _mm512_set1_epi64(0x7FFFFFFF));
    b1 = _mm512_srli_epi64(_mm512_load_epi64(&b[i]), 31);
    N0 = _mm512_and_epi64(_mm512_load_epi64(&N[i]),
                          _mm512_set1_epi64(0x7FFFFFFF));
    N1 = _mm512_srli_epi64(_mm512_load_epi64(&N[i]), 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, _mm512_load_epi64(&mu[i])),
                         _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, _mm512_load_epi64(&mu[i])),
                         _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)));
    _mm512_store_epi64(&c[i], _mm512_min_epu64(t2, _mm512_sub_epi64(t2,
                              _mm512_load_epi64(&N[i]))));
  }
}
```

Fig. 3. Vectorized multiple Montgomery multiplications of 62-bit integers using Intel AVX-512 intrinsics

bits of qN is 0; otherwise, it is 1. We note that the lower-half bits of qN need not be calculated in Algorithm 3.

If the Montgomery multiplication operand is 52 bits or fewer, $\beta = 2^{52}$ can be set in Algorithm 3 by using the Intel AVX-512IFMA instructions. Line 1 of Algorithm 3 calculates the remainder of dividing AB by β, and thus only the low 52 bits of AB need to be calculated. We can use the `vpmadd52luq` instruction for this calculation. Line 2 of Algorithm 3 calculates the remainder of dividing

Algorithm 3. Modified Montgomery multiplication algorithm

Input: A, B, N such that $0 \leq A, B < N$, $\beta > N$, $\gcd(\beta, N) = 1$, $\mu = -N^{-1} \bmod \beta$
Output: $C = AB\beta^{-1} \bmod N$ such that $0 \leq C < N$
1: $C \leftarrow AB \bmod \beta$
2: $q \leftarrow \mu C \bmod \beta$
3: $C \leftarrow \lfloor AB/\beta \rfloor + \lfloor qN/\beta \rfloor$
4: **if** $q \neq 0$ **then**
5: $\quad C \leftarrow C + 1$
6: **if** $C \geq N$ **then**
7: $\quad C \leftarrow C - N$
8: **return** C.

```
void vmulmod52(uint64_t *c, uint64_t *a, uint64_t *b, uint64_t *N,
               uint64_t *mu)
/*  c[:] = (a[:] * b[:] * 2^-52) mod N[:]. */
    We need mu[:] = -N[:]^-1 mod 2^52. */
{
  __m512i q, t;
  int i;

  for (i = 0; i < VLEN; i += 8) {
    t = _mm512_madd52lo_epu64(_mm512_set1_epi64(0),
        _mm512_load_epi64(&a[i]), _mm512_load_epi64(&b[i]));
    q = _mm512_madd52lo_epu64(_mm512_set1_epi64(0), t,
                              _mm512_load_epi64(&mu[i]));
    t = _mm512_madd52hi_epu64(_mm512_min_epu64(q, _mm512_set1_epi64(1)),
                              q, _mm512_load_epi64(&N[i]));
    t = _mm512_madd52hi_epu64(t, _mm512_load_epi64(&a[i]),
                              _mm512_load_epi64(&b[i]));
    _mm512_store_epi64(&c[i], _mm512_min_epu64(t, _mm512_sub_epi64(t,
                       _mm512_load_epi64(&N[i]))));
  }
}
```

Fig. 4. Vectorized multiple Montgomery multiplications of 52-bit integers using Intel AVX-512 intrinsics

μC by β, and thus only the low 52 bits of μC need to be calculated. We can use the `vpmadd52luq` instruction for this calculation. On line 3 of Algorithm 3, the high 52 bits of AB and qN can be calculated using the `vpmadd52huq` instruction. The conditional increment $C + 1$ when $q \neq 0$ on lines 4 and 5 of Algorithm 3 can be replaced by the minimum operation and the addition $C + \min(q, 1)$ for unsigned integer values q and C. The minimum operation $\min(q, 1)$ is performed using the `vpminuq` instruction. The conditional subtraction on lines 6 and 7 of Algorithm 3 is also performed using the `vpminuq` instruction, as described in Sect. 2.

Table 1. Specifications of the test platform

Platform	Intel NUC 8 Home, a Mini PC
Number of cores	2
Number of threads	4
CPU Type	Intel Core i3-8121U Cannon Lake-U 2.2 GHz
L1 Cache (per core)	I-Cache: 32 KB D-Cache: 32 KB
L2 Cache (per core)	256 KB
L3 Cache	4 MB
Main Memory	LPDDR4-2666 4 GB
OS	Linux 4.15.0-74-generic

Figure 4 shows the vectorized multiple Montgomery multiplications of 52-bit integers using Intel AVX-512 intrinsics. In this program, the Intel AVX-512 intrinsics `_mm512_madd52lo_epu64()`, `_mm512_madd52hi_epu64()`, and `_mm512_min_epu64()` correspond to the `vpmadd52luq`, `vpmadd52huq`, and `vpminuq` instructions, respectively. This program assumes that the vector length `VLEN` is divisible by 8. If `VLEN` is not divisible by 8, a remainder loop needs to be executed.

4 Performance Results

For performance evaluation, a comparison among multiple 52-bit Montgomery multiplications using Intel AVX-512IFMA instructions, multiple 62-bit Montgomery multiplications using Intel AVX-512F instructions, and multiple 63-bit Montgomery multiplications using Intel 64 instructions was performed.

Since a processor based on the Ice Lake microarchitecture was not available at the time of writing this paper, the performance was measured on an Intel Core i3-8121U processor, which is the only processor based on the Cannon Lake microarchitecture.

In the computation of the Montgomery multiplication $C = AB\beta^{-1} \bmod N$, N is an odd random number in the range of 1–$2^{52}-1$, and A and B are random numbers in the range of $0 \leq A, B < N$. The modular multiplicative inverses μ in Algorithms 1, 2, and 3 were prepared in advance.

The batch size of Montgomery multiplications was varied from 64 to 1024. Each batch of Montgomery multiplications was executed one million times. The number of Montgomery multiplications per second (Mulmod$\times 10^9$/s) was calculated based on the average elapsed time.

The specifications of the test platform are shown in Table 1. The Intel Core i3-8121U processor has two cores. However, we evaluated the performance on a single core and a single thread to focus on vectorization. The Intel C compiler (version 19.0.5.281) was used. The compiler options were `icc -O3 -xCANNONLAKE`. The compiler option `-O3` enables the most aggressive optimizations for maximum speed. The compiler option `-xCANNONLAKE` generates instructions for the Cannon Lake microarchitecture.

Table 2. Performance of multiple Montgomery multiplications using various instruction sets (Mulmod $\times 10^9$/s)

Batch size	Intel 64	AVX-512F	AVX-512IFMA
	63-bit	62-bit	52-bit
64	0.45860	1.32821	5.57394
128	0.46901	1.33310	5.72937
256	0.47624	1.33138	4.90638
512	0.47970	1.34336	5.28162
1024	0.47235	1.32351	4.14454

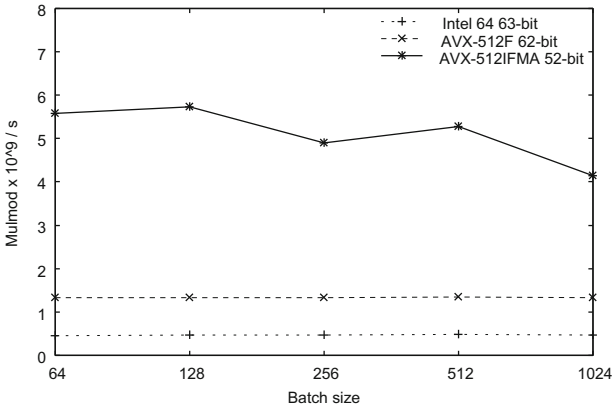

Fig. 5. Performance of multiple Montgomery multiplications using various instruction sets

Table 2 and Fig. 5 show the performance of multiple Montgomery multiplications using Intel 64, Intel AVX-512F, and Intel AVX-512IFMA instructions. When the batch size was 128, the proposed implementation using Intel AVX-512IFMA instructions was approximately 12.22 and 4.30 times faster than the implementations using Intel 64 and Intel AVX-512F instructions on the Intel Core i3-8121U processor, respectively.

The reason for this is that while the implementation using Intel 64 instructions calculates one integer at a time, the implementation using Intel AVX-512IFMA instructions calculates eight integers simultaneously. Furthermore, the number of instructions is smaller in the implementation using Intel AVX-512IFMA instructions than in the implementation using Intel AVX-512F instructions.

5 Conclusion

In this paper, we proposed a fast implementation of multiple Montgomery multiplications using Intel AVX-512IFMA instructions. The proposed implementation is based on a modified Montgomery multiplication algorithm. For Montgomery multiplication operands with 52 bits or fewer, the proposed implementation using Intel AVX-512IFMA instructions outperforms implementations using Intel 64 and Intel AVX-512F instructions on the Intel Core i3-8121U processor.

Our future work is to demonstrate the effectiveness of the proposed implementation on other processors that support the Intel AVX-512IFMA instructions.

Acknowledgments. This research was partially supported by JSPS KAKENHI Grant Number JP19K11989.

References

1. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**, 519–521 (1985)
2. Intel Corporation: Intel 64 and IA-32 architectures software developer's manual, volume 1: Basic architecture (2019). https://software.intel.com/sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf
3. Gueron, S., Krasnov, V.: Software implementation of modular exponentiation, using advanced vector instructions architectures. In: Özbudak, F., Rodríguez-Henríquez, F. (eds.) WAIFI 2012. LNCS, vol. 7369, pp. 119–135. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31662-3_9
4. Bos, J.W., Montgomery, P.L., Shumow, D., Zaverucha, G.M.: Montgomery multiplication using vector instructions. In: Lange, T., Lauter, K., Lisoněk, P. (eds.) SAC 2013. LNCS, vol. 8282, pp. 471–489. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43414-7_24
5. Drucker, N., Gueron, S.: Fast modular squaring with AVX512IFMA. In: Latifi, S. (ed.) 16th International Conference on Information Technology-New Generations (ITNG 2019). AISC, vol. 800, pp. 3–8. Springer, Cham (2019)
6. Page, D., Smart, N.P.: Parallel cryptographic arithmetic using a redundant Montgomery representation. IEEE Trans. Comput. **53**, 1474–1482 (2004)
7. Bos, J.W.: High-performance modular multiplication on the Cell processor. In: Hasan, M.A., Helleseth, T. (eds.) WAIFI 2010. LNCS, vol. 6087, pp. 7–24. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13797-6_2
8. Meng, L., Johnson, J.R., Franchetti, F., et al.: Spiral-generated modular FFT algorithms. In: Proceedings of 4th International Workshop on Parallel and Symbolic Computation (PASCO 2010), pp. 169–170 (2010)
9. 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)
10. Intel Corporation: Intel C++ compiler 19.0 developer guide and reference (2019). https://software.intel.com/sites/default/files/cpp_dev_guide_190_u5_1.pdf