IACR Communications in Cryptology
ISSN 3006-5496, Vol. 1, No. 3, 24 pages.
https://doi.org/10.62056/a3txl86bm
Check for updates

Truncated multiplication and batch software SIMD AVX512 implementation for faster Montgomery multiplications and modular exponentiation

Laurent-Stéphane Didier[1], Nadia El Mrabet[2], Léa Glandus[1] and Jean-Marc Robert[1]

[1] Toulon, Laboratoire IMath, Université de Toulon, France
[2] Saint-Etienne, Mines Saint-Etienne, CEA-LETI, Centre CMP, Department SAS, France

Abstract. This paper presents software implementations of batch computations, dealing with multi-precision integer operations. In this work, we use the Single Instruction Multiple Data (SIMD) `AVX512` instruction set of the `x86-64` processors, in particular the vectorized fused multiplier-adder `VPMADD52`. We focus on batch multiplications, squarings, modular multiplications, modular squarings and constant time modular exponentiations of 8 values using a word-slicing storage. We explore the use of Schoolbook and Karatsuba approaches with operands up to 4108 and 4154 bits respectively. We also introduce a truncated multiplication that speeds up the computation of the Montgomery modular reduction in the context of software implementation. Our Truncated Montgomery modular multiplication improvement offers speed gains of almost 20% over the conventional non-truncated versions. Compared to the state-of-the-art `GMP` and `OpenSSL` libraries, our speedup modular operations are more than 4 times faster. Compared to `OpenSSL BN_mod_exp_mont_consttimex2` using `AVX512` and `VPMADD52` in 256-bit registers, in fixed-window exponentiations of sizes 1024 and 2048, our 512-bit implementation provides speedups of respectively 1.75 and 1.38, while the 256-bit version speedups are 1.51 and 1.05 for 1024 and 2048-bit sizes (batch of 4 values in this case).

1 Introduction

The need of multi-precision computation in the context of cryptographic operations arose with the wide use of public key cryptography, linked to the RSA cryptosystem [RSA78], Digital Signature Algorithm (DSA) [Sch96], Elliptic Curve Cryptography [BSS99], or isogeny-based cryptography [BI21]. While these protocols are destined to give way to the future post-quantum ones in the long term, they are still widely used and the need of high throughput signature and/or verification remains vital. One direction to ensure fast signature or verification computations is to make use of the modern processor features. The `x86-64` processors include a 64-bit multiplier providing a 128-bit result. This instruction is the foundation of several multi-precision libraries like `GMP` (see [Ga]). Since the appearance of the SIMD (Single Instruction Multiple

This work has been partially funded by the AID (Agence pour l'Innovation de la Défense), project 2022151.

E-mail: laurent-stephane.didier@univ-tln.fr (Laurent-Stéphane Didier), nadia.el-mrabet@emse.fr (Nadia El Mrabet), lea.glandus@univ-tln.fr (Léa Glandus), jean-marc.robert@univ-tln.fr (Jean-Marc Robert)

This work is licensed under a "CC BY 4.0" license.

Received: 2024-07-02 Accepted: 2024-09-02

Data) instruction sets, especially the AVX and AVX2, allowing parallel computations, several works attempted to exploit the possibilities of these instructions. In 2012, Gueron and Krasnov in [GK12] proposed the so called RSA_Z implementation for OpenSSL. These software implementations take advantage of the 32-bit parallel multipliers, i.e. four 32-bit multiplications (providing four 64-bit results) can be performed simultaneously using the AVX2 256-bit registers. However, these improvements rely on the microarchitecture version used, since the complexity balance in terms of 32-bit multiplications leads to the same multiplication instruction numbers in comparison with conventional 64-bit sequential architecture. Gueron *et al.* in [GK16, DGK18] proposed other works using AVX512 to improve this balance. However, the carry management and word alignment of the additions often necessitates time consuming operations (shuffles or permutes, shifts, additions...) and the efficiency of the computation of a single multi-precision multiplication using SIMD instruction sets is not as efficient as expected. Furthermore, the carry management leads to the so called reduce-radix approach, that is using 28 or 27-bit operands in 32-bit multipliers, in order to keep spare bits to ensure the carry management, with a penalty in execution speed. Recently, the availability of the VPMADD52 instructions, computing up to 8 operations of the form $a + b \times c$ (AVX512) in 64-bit word-slicing with b and c of size 52 bits, brought some improvements. Nevertheless, the speedup remains small, see Gueron *et al.* in [GK16] (see also Bos *et al.* in [BMSZ14], Edamatsu and Takahashi in [DG19, ET20] and Takahashi in [Tak20]).

To counteract the carry management and word alignment penalty, another way is to design batch multipliers (and batch computations in general), using a word slicing representation of a corresponding batch of values. This technique is well known and used in the GPU context (see [BCC+09, Tre13, MC14, ELWW16, EZW18, ABS10, Bos12]).

In 2008, Grabher *et al.* [GGP08] present cryptographic pairing software implementations, some of them using a technique close to the word slicing approach with reduced radix (29 bit witdh) for inter-pairing parallel computations. Quite recently (2022), Buhrow *et al.* in [BGH22] improved the concept with 32-bit SIMD multipliers using the AVX512 instruction set (512-bit registers), applied to CRT-RSA decryptions. These implementations compute 8 modular exponentiations simultaneously, using the reduce-radix architecture, providing up to 1.9 speedup in comparison with the OpenSSL throughput, for 2048-bit CRT-RSA decryptions. With the AVX512 and VPMADD52 instructions, and in the context of post-quantum SIKE protocol, Cheng *et al.* in [CFG+21, CFGR22] proposed a batch implementation of the key exchange protocol. However, their implementations are specific to the SIKE parameters, in particular the Montgomery friendly primes for the modular operations. They make use of a slightly reduced radix approach (51-bit operands for the 52-bit multiplier of the VPMADD52). To the best of our knowledge, these multipliers are the fastest for the considered sizes (e.g. 503 bits), but with a relaxed carry management due to the specific modular reduction following each multiplication or squaring. Therefore, the implementations of Cheng *et al.* cannot be used or transposed in other contexts.

Recently, the OpenSSL library offers a new approach, which is intermediary between a parallelized computation of a single exponentiation and a batch version, implemented by Kyrillov and Matyukov (see[Pro]). This implementation computes two exponentiations simultaneously using the VPMADD52 instruction on 256-bit registers (when available on the platform). This avoids some mi-

croarchitecture issues of AVX512 while it is compliant to the Intel AVX10.1 new extension (see [Arc]). This function processes operands of 1024, 1536 and 2048 bits. To the best of our knowledge, these implementations are the state-of-the-art.

Another topic we have focused on in this paper is the concept of truncated operation applied to modular reduction. In his seminal paper, Barrett in [Bar86] mentioned the idea, but did not implement the concept further. Montgomery in [Mon85] did not pay attention to the idea either. Subsequent works on the Montgomery modular reduction or multiplication developed the concept of block or word approaches, like Koç in [KKAK96]. These are the CIOS (for Coarsely Integrated Operand Scanning) and variants, which aim to improve the performance by optimising the word addition numbers and the memory access patterns. However, these approaches are not suitable for truncated operations. Later on, Hars in [Har05] and [Har06] studied the concept of truncated multiplications and their application to various contexts, mainly Barrett and Montgomery multiplications. This work remained theoretical and focused on possible improvements in hardware implementations but no implementations have been developed. More recently, Ding *et al.* in [DL18] proposed a hardware implementation of a 256-bit ECC processor using a Barrett modular multiplication with a truncated multiplication. Later on in [DL20], the same authors proposed FPGA implementations of 256 and 512-bit Montgomery modular multipliers based on 3 and 4 way Karatsuba truncated multipliers. However, these approaches cannot be easily transferable to the context of software implementations. Furthermore, since they only present very few versions and sizes, their work is difficult to generalize.

In another recent work, Bos *et al.* in [BKP21] explore parallel implementations of Montgomery Multiplications in both cases: intra-parallelism of CIOS approach with a multi-thread implementation, and inter-multiplication parallelism, using a word-slicing representation. However, they do not provide implementation results.

Contributions. In this paper, we present software implementations of batch multi-precision multipliers, that is 8 simultaneous multiplications, using a word slicing representation in radix 2^{52} to take advantage of the AVX512 VPMADD52 instructions. We implement sizes up to 4108 bits, using the Schoolbook approach and up to 4154 bits using the Karatsuba approach. We use these multipliers in order to implement Montgomery modular multiplications, and propose, to the best of our knowledge, the first software implementation of Truncated Montgomery modular multiplication, which allows up to 20 % speedup over non-truncated versions and presents a more than 4 times speedup over the OpenSSL RSA_Z (not using the 256-bit VPMADD52) and GMP libraries. We make use of these modular multiplications in fixed-window modular exponentiations, for sizes 1024, 2048 and 4096 bits, with speedups up to nearly 4 (1024 bits) over the OpenSSL BN_mod_exp_mont_consttime and 1.75 over the OpenSSL BN_mod_exp_mont_consttimex2 which computes two exponentiations in parallel using the 256-bit VPMADD52. We also implemented a 256-bit Truncated Montgomery exponentiation of our batch AVX512 using VPMADD52 instructions one in order to compare with the corresponding OpenSSL BN_mod_exp_mont_consttimex2, with a maximum speedup of 1.51 and 1.05 for 1024 and 2048-bit operands respectively.

Organisation of the paper. This paper is organised as follows: Section 2 presents the implementation principles of the Schoolbook batch multiplications and squarings, Section 3 deals with the Karatsuba versions of the batch multiplications and squarings, Section 4 presents the Montgomery modular batch multiplications. In Section 5, we then present the Truncated Montgomery modular reduction and its batch implementation. This is followed by the presentations of the implementation performance in Section 6, including the fixed-window modular exponentiations. A conclusion ends the paper.

Notations.

- $>>$ represents a logic RIGHT SHIFT
- $\bigvee$ represents a logic OR
- & represents a logic AND

Multiprecision numbers will be represented by either bits denoted as a_{k_i}, or in 64 bit word arrays denoted as $A64_k[i]$, or in 52 bit word arrays denoted as $A_k[i]$.

2 Batch Schoolbook multiplications

The objective of our implementations of batch Schoolbook multiplications is to achieve eight simultaneous multiplications when using the `AVX512` instruction set, or a batch of four values when using the `AVX2` one and will further refer to their corresponding C variables types as `__m512i` and `__m256i` which are respectively 256 and 512 bit vectors, representing integers. In the sequel, we focus on the `__m512i` case since the conversion to the other case is straightforward. We use a word slicing approach of the operands as follows.
Let us have a batch of eight values $A_k, 0 \leq k < 8$ of t bits each. They can be sliced into $t_{64} = \lceil t/64 \rceil$ words of 64 bits each. To be able to use the `VPMADD52` instructions, we propose to split them into 52-bit words. This requires the use of $t_{52} = \lceil t/52 \rceil$ 512-bit lines to store the 8 A_k's.
More formally, we set:

$$A_k = \sum_{i=0}^{t} a_{ki} 2^i = \sum_{i=0}^{t_{64}} A64_k[i] 2^{64 \times i} = \sum_{i=0}^{t_{52}} A_k[i] 2^{52 \times i},$$

and store the eight values as shown in Table 1.

Table 1: Word Slicing storage of eight values in 52-bit words

	64 bits (52 bits)	64 bits (52 bits)	64 bits (52 bits)	64 bits (52 bits)	64 bits (52 bits)	64 bits (52 bits)	64 bits (52 bits)	64 bits (52 bits)
$A512_0$	$0 \mid A_0[0]$	$0 \mid A_1[0]$	$0 \mid A_2[0]$	$0 \mid A_3[0]$	$0 \mid A_4[0]$	$0 \mid A_5[0]$	$0 \mid A_6[0]$	$0 \mid A_7[0]$
$A512_1$	$0 \mid A_0[1]$	$0 \mid A_1[1]$	$0 \mid A_2[1]$	$0 \mid A_3[1]$	$0 \mid A_4[1]$	$0 \mid A_5[1]$	$0 \mid A_6[1]$	$0 \mid A_7[1]$
⋮	⋮							
⋮	t_{52} 512-bit lines							

We present the `VPMADD52` instructions, which are integer fused multiply–add instructions, and denote them using the C-intrinsics [Int]:

```
_mm512_madd52lo_epu64(a, b, c)
_mm512_madd52hi_epu64(a, b, c)
```

1. `__m512i _mm512_madd52lo_epu64(a, b, c)`
   for $i = 0 \ldots 7$ **do**
   // 52 least significant bits of $b \times c$
   $Dest_i \leftarrow a_i + [(b_i \times c_i) \bmod 2^{52}]$
   end for

2. `__m512i _mm512_madd52hi_epu64(a, b, c)`
   for $i = 0 \ldots 7$ **do**
   // 52 most significant bits of $b \times c$
   $Dest_i \leftarrow a_i + [(b_i \times c_i) >> 52]$
   end for

In the rest of the paper we rename these instructions to `madd52lo` and `madd52hi`.

2.1 Schoolbook multiplication

We target operands of RSA sizes, and more generally sizes which are multiples of 64. Because we also aim at the `VPMADD52` instructions, we convert radix 2^{64} representation into radix 2^{52} representation. This minimizes as much as possible the number of words required to represent the operands.
For instance, in [CFGR22], the authors use radix 2^{51} which is convenient for the SIKE 503 case they focused on. They use 10-word operands with Schoolbook or Karatsuba approaches. However, this requires 11 words in case of 512-bit operands whereas 10 words are enough in radix 2^{52}.
After testing several configurations, we choose the Algorithm 1 (`B_mul`) in order to limit the memory transfer of operands. Thus, we use the same 512-bit line of the first operand, parsing the second operand lines and storing the partial products in the corresponding line of the result. This configuration necessitates a carry management at the end which is performed using a binary mask *mask52*.

Algorithm 1 Batch Schoolbook multiplications, `B_mul`

Require: Two batches of 8 values A_k and B_k stored in 52-bit slices in t_{52} `__m512i` shares, $mask52$ is a 512-bit batch of eight 52-bit masks.
Ensure: A batch of 8 values $C_k = A_k \times B_k$ stored in 52-bit slices in $2 \times t_{52}$ `__m512i` shares.
1: **for** k from 0 to 7 **do in parallel**
2: $C_k[0] \leftarrow 0$
3: **for** i from 0 to $t_{52} - 1$ **do**
4: **for** j from 0 to $t_{52} - 1$ **do**
5: $C_k[i+j] \leftarrow$ `madd52lo`$(C_k[i+j],\ A_k[i],\ B_k[j])$
6: **end for**
7: **end for**
8: $carry_k \leftarrow 0$ ▷ carry management
9: **for** i from 2 to $2 \times t_{52} - 2$ **do**
10: $carry_k \leftarrow C_k[i-1] >> 52$
11: $C_k[i] \leftarrow C_k[i] + carry_k$
12: $C_k[i-1] \leftarrow C_k[i-1] \& mask52$
13: **end for**
14: $carry_k \leftarrow C_k[2 \times t_{52} - 2] >> 52$
15: $C_k[2 \times t_{52} - 1] \leftarrow C_k[2 \times t_{52} - 1] + carry_k$
16: **end for**
17: **return** C

2.2 Squaring

We applied a similar approach for the squaring operation (Alg. 2, `B_square`). In this algorithm, the number of `VPMADD52` operations is divided by nearly two compared to the multiplication implementation presented in Section 2.1.

Algorithm 2 Batch Schoolbook squaring `B_square`

Require: One batch of 8 values A_k stored in 52-bit slices in t_{52} `__m512i` shares, $mask52$ is a 512-bit batch of eight 52-bit masks.
Ensure: A batch of 8 values $C_k = A_k^2$ stored in 52-bit slices in $2 \times t_{52}$ `__m512i` shares.

1: $carry \leftarrow 0_{512bits}$
2: **for** k from 0 to 7 **do in parallel**
3:     **for** ℓ from 0 to $2 \times t_{52} - 1$ **do**
4:         $C_k[\ell] \leftarrow 0_{512}$
5:         **for** (i, j) such that $i + j = \ell$ and $j < i$ **do**
6:             $C_k[\ell] \leftarrow$ `madd52lo`$(C_k[\ell], A_k[i], A_k[j])$
7:         **end for**
8:         **for** $(i + j)$ such that $i + j = \ell - 1$ and $j < i$ **do**
9:             $C_k[\ell] \leftarrow$ `madd52hi`$(C_k[\ell], A_k[i], A_k[j])$
10:         **end for**
11:         $C_k[\ell] \leftarrow C_k[\ell] << 1$
12:         **if** $\ell \bmod 2 = 0$ **then**
13:             $i \leftarrow \ell/2$
14:             $C_k[\ell] \leftarrow$ `madd52lo`$(C_k[\ell], A_k[i], A_k[i])$
15:         **else**
16:             $i \leftarrow \lfloor \ell/2 \rfloor$
17:             $C_k[\ell] \leftarrow$ `madd52hi`$(C_k[\ell], A_k[i], A_k[i])$
18:         **end if**
19:         $C_k[\ell] \leftarrow C_k[\ell] + carry_k$
20:         $carry_k \leftarrow C_k[\ell] >> 52$
21:         $C_k[\ell] \leftarrow C_k[\ell] \& mask52$
22:     **end for**
23: **end for**
24: return C

2.3 Complexity comparison

As mentioned above, for t-bit operands split in 64-bit words, we store 8 operands in $t_{52} = \lceil t/52 \rceil$ 512-bit lines. The Schoolbook batch multiplication requires t_{52}^2 elementary multiplications, performed using both instructions (`madd52lo` and `madd52hi`), that is $2 \times t_{52}^2$ `VPMADD52` instructions. Algorithm 1 requires a few extra 512-bit additions in order to manage the carries.

The batch squaring requires only $t_{52}(t_{52} + 1)$ `VPMADD52` instructions and $t_{52} - 1$ left shifts (i.e. multiplications by 2) and additions also in order to handle the carries. These complexities are summarized in Table 2.

Table 2: Number of instructions of batch Schoolbook multiplications

	# `VPMADD52`	# shifts	# Additions	# maskings
Multiplication Alg. 1	$2 \times t_{52}^2$	$2 \times t_{52} - 2$	$2 \times t_{52} - 2$	$2 \times t_{52} - 2$
Squaring Alg. 2	$t_{52} \times (t_{52} + 1)$	$2 \times t_{52} - 2$	$2 \times t_{52} - 2$	$2 \times t_{52} - 2$

3 Batch Karatsuba multiplications

This batch construction is also adapted to Karatsuba multiplication. We remind here this construction. Let A and B be two t-bit operands. We assume that t is even. We first split the operands as follows:

$$A = a_\ell + 2^{t/2}a_h, B = b_\ell + 2^{t/2}b_h.$$

We then compute 3 elementary products (instead of four in the Schoolbook method) and we similarly split them:

$$\begin{cases} D_0 = D_{0\ell} + 2^{t/2}D_{0h} \leftarrow & a_\ell \times b_l, \\ D_1 = D_{1\ell} + 2^{t/2}D_{1h} \leftarrow & (a_\ell + a_h) \times (b_\ell + b_h), \\ D_2 = D_{2\ell} + 2^{t/2}D_{2h} \leftarrow & a_h \times b_h. \end{cases}$$

The low-level multiplications are performed with the Schoolbook method. The multiplication width is determined by $(a_\ell + a_h) \times (b_\ell + b_h)$. Finally, we obtain the result as follows:

$$\begin{aligned} A \times B = \ & D_{0\ell} \\ & +2^{t/2}(D_{0h} + D_{1\ell} - D_{0\ell} - D_{2\ell}) \\ & +2^{t}(D_{2\ell} + D_{1h} - D_{0h} - D_{2h}) \\ & +2^{3t/2}D_{2h}. \end{aligned}$$

The size of A and B and the size of the elementary multiplication are linked. For example, if the largest elementary product requires 520-bit operands, A and B have to be at most $2\times$ 519 bits long. This means that t =1038. The Table 3 sums up the sizes we consider in the rest of the paper.

Table 3: Operand and elementary multiplication size

Karatsuba mult. size t	518	1038	2078	4154*
Elementary product	260	520	1040	2078

* double Karatsuba

In this construction, we represent the operands in two radix $2^{t/2}$ shares. They are radix 2^{52} numbers in case of one Karatsuba stage. In the special case of 4154 bits which requires two Karatsuba stages, we have the following representation:

$$A = a_\ell + 2^{2077}a_h,$$

with $a_\ell = a_{\ell\ell} + 2^{1039}a_{\ell h}$ and $a_h = a_{h\ell} + 2^{1039}a_{hh}$. The shares are then represented with 52-bit words. All these representations are stored in memory in the batch way presented Section 2.

In terms of complexity, the number of Schoolbook elementary multiplications is 3 for 1038 and 2078 bits (respectively 520 and 1040-bit multiplications), and 9 Schoolbook elementary multiplications of size 1040 bits for the 4154-bit batch Karatsuba multiplications. We provide the instruction count in Table 4.

While the `VPMADD52` instruction count is smaller compared to the Schoolbook case, the other instructions are much more numerous. This explains why the Karatsuba approach is interesting only for the 4154-bit case for the squaring.

4 Batch Montgomery modular multiplications

Let us briefly remind of the Montgomery modular multiplication [Mon85]. In this operation, the product T of two t-bit operands is reduced modulo

Table 4: Number of instructions of batch Karatsuba multiplications

	# VPMADD52	# shifts	# Add./Sub.	# maskings
Multiplication with 3 elt. Alg. 1	$\frac{3}{2} \times t_{52}^2$	$8 \times t_{52} - 6$	$10 \times t_{52} - 7$	$8 \times t_{52} - 6$
Squaring with 3 elt. Alg. 2	$\frac{3}{4} t_{52} \times (t_{52} + 2)$	$10 \times t_{52} - 1$	$9 \times t_{52} + 3$	$7 \times t_{52} + 4$

a t-bit modulus N (Alg. 5). The returned result is $C \leftarrow T \times R^{-1} \bmod N$ where R is a power of 2 in binary implementations. In order to handle this multiplicative factor, the input operands are usually converted in the Montgomery representation. This consists of multiplying the initial operands by the square modulo R of N, using the same Montgomery modular reduction. By this, one gets:

$$C \leftarrow MontRed(T \times R^2, N) = \frac{A \times R^2}{R} \mod N$$

and

$$C = T \times R \bmod N.$$

This renders the representation stable in case of a sequence of multiple multiplications and squarings. A final Montgomery reduction is enough to convert the result from this Montgomery representation.

Batch Montgomery multiplication. With this context, it is possible to make a batch Montgomery reduction modulo 8 different moduli N_i using AVX512 instructions. We investigated several approaches.
The first is based on the Schoolbook multiplication and/or squaring (Alg. 1 and Alg. 2) followed by the batch Montgomery reduction (Alg. 3). In this case, we use a batch fused multiplier-adder (B_fma) operation at line 2 of Alg. 3. This operation has the same cost as a single multiplication because of the use of the VPMADD52 instructions. This makes free the addition required in the Montgomery reduction.
With the Karatsuba multiplication, it is not possible to use a batch Karatsuba B_fma because of the additions in the reconstruction phase. This explains why the speedups are slightly better for the small sizes in the Schoolbook case.
We also implemented the word-level Montgomery reduction, adapted from [KKAK96] also called CIOS and variants (BPS improvement in [BGH22]), only on the Montgomery multiplication. This approach is presented Algorithm 4.
In both Algorithms 3 and 4, the usual final subtraction is not required in our case, since the size of the multiplication is greater enough than the size of the modulus, see Gueron *et al.* in [GK12].

5 Truncated Batch Montgomery modular multiplications

The classic Montgomery reduction is reminded in Algorithm 5. This algorithm computes $T \times R^{-1} \bmod N$ where the modulus N is a t-bit integer (with $t \equiv 0 \bmod 64$) and the Montgomery constant $R = 2^{52 \times t_{52}}$.

Algorithm 3 Batch Montgomery reduction

Require: One batch of 8 values A stored in 52 word-slices in $2 \times t_{52}$ `__m512i` shares, $mask52$ is a 512-bit batch of eight 52-bit masks, the 8 moduli N, some precomputed values $N' = (-N)^{-1} \bmod R$ with $R = 2^{52 \times t_{52}}$.
Ensure: A batch of 8 values such that $C_k = A_k \times R^{-1} \bmod N_k$ stored in 52 word-slices in t_{52} `__m512i` shares.

1: $q \leftarrow$ `B_mul`$(A, N') \bmod R$
2: $T \leftarrow$ `B_fma`(q, N, A) ▷ $A + q \times N$
3: $C \leftarrow T/R$ ▷ returns the t higher words of T
4: return C

Algorithm 4 Batch CIOS inspired Montgomery multiplication

Require: Two batches of 8 values A_k and B_k stored in 52 word-slices in $2 \times t_{52}$ `__m512i` shares, $mask52$ is a 512-bit batch of eight 52-bit masks, the 8 moduli N_k, some precomputed values $N'_k = (-N_k)^{-1} \bmod R$ with $R = 2^{52 \times t_{52}}$.
Ensure: A batch of 8 values such that $C_k = A_k \times B_k \times R^{-1} \bmod N$ stored in 52 word-slices in t_{52} `__m512i` shares.

1: **for** k from 0 to 7 **do in parallel**
2: $Y_k \leftarrow a_k[0] \cdot B_k$
3: $q_k \leftarrow |Y_k|_{2^{52}} \cdot N'_k \mod 2^{52}$
4: $Y_k \leftarrow (Y_k + q_k \cdot N_k)/2^{52}$
5: **for** $i = 1$ **to** $t_{52} - 1$ **do**
6: $Y_k \leftarrow Y_k + a_k[i] \cdot B_k$
7: $q_k \leftarrow |Y_k|_{2^{52}} \cdot N'_k \mod 2^{52}$
8: $Y_k \leftarrow (Y_k + q_k \cdot N_k)/2^{52}$
9: **end for**
10: **end for**
11: return $C \leftarrow Y$

At step 2 of Algorithm 5, the division by R is exact because $T + q \times N \equiv 0 \bmod R$. In other words, the $52 \times t_{52}$ least significant bits of $T + q \times N$ are all zeroes. Because we know the value of its least significant part, the computation of the $52 \times t_{52}$ least significant bits of $q \times N$ can be avoided. We only need to estimate the input carry in the sum of the $52 \times t_{52}$ most significant bits. These remarks lead to Algorithm 6.
In this algorithm, T is a $2t_{52}$-word integer and the modulus N is a t_{52}-word integer. We denote $T[0]$ the least significant 52-bit word of T. At step 2, we compute only the most significant bits of the multiplication $q \times N$, denoted $\widehat{qN} \leftarrow \lfloor \frac{q \times N}{R} \rfloor$. The carry c_{add} to be propagated from the lower part is computed at line 3. Finally, the result is computed at line 4, where $\lfloor T/R \rfloor$ is the t_{52} most significant words of T.
In the rest of this paper, we deal with batch SIMD software implementations, however, the idea may be applied to classical sequential implementations.
Theorem 1 provides the correctness of our Montgomery modular reduction with truncated multiplication. Theorem 2 describes how to efficiently compute the correct $\widehat{qN}$.

Theorem 1 (Correctness of the Truncated MontMul)**.** *With $T < 4N^2$, a t-bit modulus N, an integer $R = 2^{52 \times t_{52}}$, with $t_{52} \geq \lceil t/64 \rceil$, precomputed value $N' = (-N)^{-1} \bmod R$, Algorithm 6 correctly computes $C \equiv T \times R^{-1} \bmod N$ and $C < 2N$.*

Proof. The key point of Algorithm 6 is that we know that $T + q \times N \bmod R \equiv 0$. In other words, the $52 \times t_{52}$ least significant bits are zeroes. This makes easier

Algorithm 5 Montgomery modular reduction: *MontRed*

Require: $T < 4N^2$, N the t-bit modulus, $R = 2^{52\times t_{52}}$, with $t_{52} \geq t$, precomputed value $N' = (-N)^{-1} \bmod R$.
Ensure: $C \equiv T \times R^{-1} \bmod N$ and $C < 2N$
1: $q \leftarrow T \times N' \bmod R$ ▷ $q < R$
2: $C \leftarrow \frac{T+q\times N}{R}$ ▷ $C < 2N$
3: **return** C

Algorithm 6 Montgomery modular reduction with truncated multiplication

Require: $T < 4N^2$, N the t-bit modulus, $R = 2^{52\times t_{52}}$, with $t_{52} \geq t$, precomputed value $N' = (-N)^{-1} \bmod R$.
Ensure: $C \equiv T \times R^{-1} \bmod N$ and $C < 2N$
1: $q \leftarrow T \times N' \bmod R$ ▷ $q < R$
2: $\widehat{qN} \leftarrow \lfloor (q \times N)/R \rfloor$
3: $c_{add} = \bigvee_{i=0}^{t_{52}-1} T[i]$
4: $C \leftarrow \lfloor T/R \rfloor + \widehat{qN} + c_{add}$
5: **return** C

the computation of the carry c_{add}.
If the least significant bit of T is 0, then the least significant bit of $q \times N$ is also 0, and no carry is propagated to the next position. This property holds until the first bit of T equals 1.
If the ith bit of T is 1, then the ith bit of qN must also be 1 and a carry must be propagated to position $i+1$ in order to ensure that the ith bit of $T + q \times N$ is 0. Next, to ensure that the next bit of $T + q \times N$ is 0, only one of the $i + 1^{th}$ bits of T and qN must be 1. These conditions are summarized in the table below:

bits	i	$i+1$	
T	1	0	1
$q \times N$	1	1	0
generated carry c_{add}	1	1	

Thus, if a carry is generated at position i^{th}, then another carry is generated at position $i+1^{th}$. As a consequence, the carry generated at position $52 \times t_{52} - 1$ is 1 only if there is at least one of the $52 \times t_{52} - 1$ least significant bits that is 1. This carry is output from the $t_{52} - 1^{th}$ word.

$$c_{add} = c_{add\ 52\times(t_{52}-1)} = \bigvee_{i=0}^{t_{52}-1} T[i]. \tag{1}$$

Finally, in Algorithm 5, the result C is computed as follows:

$$C = \left(\sum_{i=0}^{2\times t_{52}-1} T[i]2^{i52} + \widehat{qN}2^{52\times t_{52}} + \underset{\smile}{qN} \right) >> (52 \times t_{52}),$$

where $\underset{\smile}{qN} = qN \bmod R$. As a consequence:

$$C = \sum_{i=t_{52}}^{2\times t_{52}-1} T[i]2^{52(i-t_{52})} + \widehat{qN} + \left(\sum_{i=0}^{t_{52}-1} T[i]2^{i52} + \underset{\smile}{qN} \right) >> (52 \times t_{52}).$$

The last term of this sum is c_{add} that we compute with equation (1) at line 3 in Algorithm 6. □

In Algorithm 6 the computation of $\widehat{qN}$ do not require to compute all the partial products of $q \times N$, as shown in Theorem 2.

Theorem 2 (Computation of $\widehat{qN}$)**.** *The correct computation of $\widehat{qN}$ requires only the partial products of $q \times N$ of weight at least $t_{52} - 1$.*

Proof. Since we know that $T + q \times N \bmod R \equiv 0$, the computation $\widehat{qN}$ can be simplified. Figure 1 illustrates the computation of $q \times N$. It is not necessary to compute all the least significant partial products of $q \times N$ and sum them in order to know which carries to propagate and to compute $\widehat{qN}$ correctly.

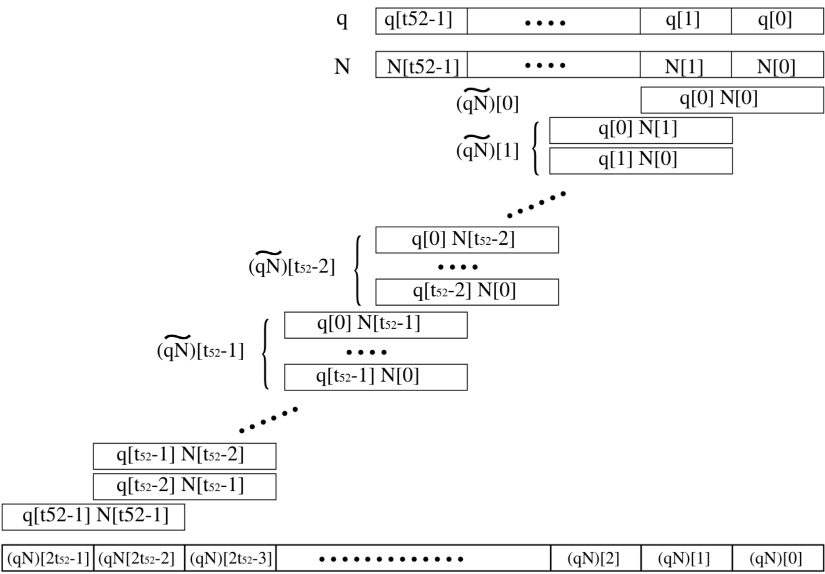

Figure 1: Detail of $q \times N$

Let us denote $\widetilde{(qN)}[i]$ the sum of the partial products of weight i. More formally, the partial product of weight $52t_{52} - 1$ is a $\lceil \log_2(t_{52} - 1) \rceil + 52$-bit word and is computed as follows:

$$
\begin{aligned}
\widetilde{(qN)}[t_{52} - 1] \leftarrow & \sum_{i=0, j=0, i+j=t_{52}-1}^{t_{52}-1} \texttt{mul52lo}(q[i], N[j]) \\
& + \sum_{i=0, j=0, i+j=t_{52}-2}^{t_{52}-2} \texttt{mul52hi}(q[i], N[j]) \\
& + \widetilde{(qN)}[t_{52} - 2] >> 52,
\end{aligned} \tag{2}
$$

where `mul52hi` and `mul52lo` compute respectively the 52 higher and lower bits of two 52-bit operands. Therefore, the $t_{52} - 1^{th}$ word of $T + qN$ (which is known to be 0) is:

$$
(T + qN)[t_{52} - 1] = (T[t_{52} - 1] + \widetilde{(qN)}[t_{52} - 1] + c_{add}) \bmod 2^{52}
$$

and then

$$
(T[t_{52} - 1] + \widetilde{(qN)}[t_{52} - 1] + c_{add}) \bmod 2^{52} = 0. \tag{3}
$$

If $\mathbf{c_{add}} = \mathbf{0}$: This means that all $T[i] = 0$ for $i \leq t_{52} - 1$ and so for $q[i]$, because $q = T \times N' \bmod R$ in Alg. 6. Therefore in equation (2), only the partial products of weight greater than or equal to $t_{52} - 1$ are needed.

If $\mathbf{c_{add}} = 1$: The only way the equation (3) can be verified is if the binary vector $T[t_{52}-1] + \widetilde{(qN)}[t_{52}-1]$ has all its 52 least significant bits set to 1. Using Eq. (2), equation (3) can be written as follows:

$$(UP + \widetilde{(qN)}[t_{52}-2] >> 52 + c_{add}) \bmod 2^{52} = 0, \tag{4}$$

where

$$\begin{aligned} UP \leftarrow T[t_{52}-1] + & \sum_{i=0,j=0,i+j=t_{52}-1}^{t_{52}-1} \texttt{mul52lo}(q[i], N[j]) \\ + & \sum_{i=0,j=0,i+j=t_{52}-2}^{t_{52}-2} \texttt{mul52hi}(q[i], N[j]). \end{aligned}$$

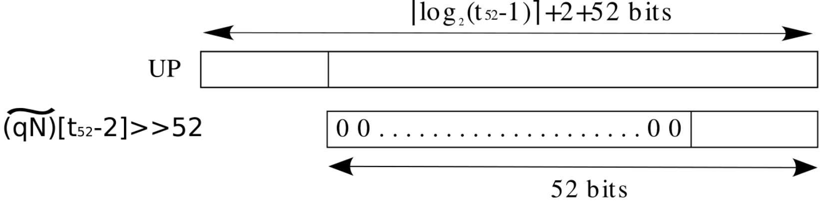

Figure 2: Detail of $UP + (\widetilde{(qN)}[t_{52}-2] >> 52)$

Therefore, $\widetilde{(qN)}[t_{52}-2] >> 52$ can be computed with the least significant bits of UP and its sum with UP does not generate any carry out of the $52th$ bit. Thus $\widetilde{(qN)}[t_{52}-1] >> 52$ is computed only with the most significant bits of UP and the carry c_{add} which is propagated to the 53rd bit of UP.

As a conclusion $\widehat{(qN)}[0]$ is computed only with the partial products of weight at least equal $t_{52}-1$:

$$\begin{aligned} \widehat{(qN)}[0] \leftarrow & \sum_{i=0,j=0,i+j=t_{52}}^{t_{52}} \texttt{mul52lo}(q[i], N[j]) \\ & + \sum_{i=0,j=0,i+j=t_{52}-1}^{t_{52}-1} \texttt{mul52hi}(q[i], N[j]) \\ & + \widetilde{(qN)}[t_{52}-1] >> 52 + c_{add}. \end{aligned} \tag{5}$$

□

5.1 Complexity of the Truncated Montgomery multiplication

The truncated multiplication in Algorithm 6 at line 2 can be done in several ways.

Schoolbook multiplication. Here, we compute only the $t_{52}+1$ most significant words of the $q \times N$ product. The instruction count in Table 5 shows that the instruction count is divided by nearly two for the truncated `B_fma`, which replaces the operation line 2 Algorithme 3. Thus, for the whole Montgomery multiplication, this leads to a global complexity of slightly more than 2 multiplications, instead of 2.5 in the classical approaches, including the CIOS.

Table 5: Batch Schoolbook truncated B_fma used in Alg. 3, instruction number comparison

	# VPMADD52	# shifts	# Additions	# maskings	# OR
B_fma	$2t_{52}^2$	$2t_{52}-2$	$2t_{52}-2$	$2t_{52}-2$	-
trunc. B_fma	$t_{52}^2+3/2t_{52}-1$	t_{52}	$t_{52}+2$	$t_{52}+2$	$t_{52}-1$

Karatsuba multiplication. In this approach of the truncated multiplication, the operands have to be split with the following construction:

$$trunc(A \times B) = \quad 2^t(D_{2\ell} + D_{1h} - D_{0h} - D_{2h}) + 2^{3t/2}D_{2h}.$$

We therefore compute only $D_{2\ell}, D_{2h}, D_{1h}$ and D_{0h}, that is one and two halves of elementary multiplications instead of three. The cost of this truncated multiplication is roughly two-thirds of a complete multiplication. And we need also half of a whole addition to achieve the last step of the Montgomery reduction.
The halves of elementary multiplications are computed using the same approach than the one for the Schoolbook. The correctness is ensured by the same kind of carry evaluation and propagation.
Thus, in this case, for the whole Montgomery multiplication, this leads to a global complexity of a little more than 2.17 multiplications with Karatsuba, instead of 2.0 for the truncated approaches with Schoolbook, or 2.5 for the conventional approaches, either classical or CIOS (word level Montgomery).

6 Performances of the implementations

Our batch multiplications, batch Montgomery multiplications and the corresponding exponentiations have been implemented in C. All the source codes are available at https://github.com/lea-gl/TruncatedBatchSIMDAVX512MontgomeryMultiplicationsModularExponentiation.git.
In this section, after the presentation of the performance measurement procedure, we provide the results of the batch AVX512 multiplications, Batch Montgomery multiplications and squarings, and corresponding exponentiations. This section concludes with the performances of the 256-bit batch exponentiation counterparts. All these experiments are compared to state-of-the-art GMP and OpenSSL performances, compiled and run on the same platform and using the same measurement procedure.

6.1 Performances measurement procedure

It is known that the intensive use of vectorized computation with AVX2 and AVX512 extensions can lead to penalties due to the resulting high power consumption [Kra17]. Intel processors have a limited power budget and may reduce their frequency when executing complex AVX2 and AVX512 instructions. This might also affect the execution of adjacent non-vectorized code. However, the impact of this issue depends on the power budget allowed by the processor and how efficiently it is cooled. Our goal here is to provide a fair evaluation of the sequential and our parallel algorithms. In order to perform this comparison as fairly as possible, we focused on measuring clock cycles. Measurements were performed on a Dell Inspiron laptop with a tiger lake processor.

```
vendor_id : GenuineIntel
```

```
cpu family : 6
model  : 140
model name : 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
```

The compiler is `gcc` version 9.4.0, the compiler options are as follows:
`-O3 -funroll-all-loops -g -march=native -lgmp -lcrypto`.
We kept the `-funroll-all-loops` option though it does not provide significant improvements. We follow the same kind of test procedure as described in [DGK18] or in [RV22]:

- the *Turbo-Boost®* is deactivated during the tests;
- 1000 runs are executed in order to "warm-up" the cache memory;
- 50 random data sets are generated, and for each data set the minimum of the execution clock cycle numbers over a batch of 1000 runs is recorded;
- the performance is the average of all these minimums;

The clock cycles have been counted with `rdtsc/rdtscp` instructions.

6.2 Performances of the batch multipliers

We have implemented both batch squaring and multiplication for several sizes. The timings are shown in Table 6 where the best results are in bold. We target operands having between 260 and 4154 bits. Due to the Karatsuba splitting, the size of Karatsuba implementations are slightly different. Karatsuba multiplication is inefficient for small operands so we have not implemented it for 260-bit operands. The 4154-bit operands are large enough to permit a double Karatsuba splitting.
We have compared our implementations with 8 successive `GMP` low level multiplications `mpn_mul_n()`. It can be observed that the batch approach takes advantage of vector instructions for both squaring and multiplication. The batch approach is 5 to 9 times faster than `GMP`. For large enough operands, the Karatsuba approach is the fastest.

Table 6: Batch Schoolbook (SB) and Karatsuba multiplications, clock cycles.

op. size (bits) SB op. size (bits) Karatsuba	260	520 518	1040 1038	2080 2078	4108 4154
# clock cycles					
`GMP mpn_mul_n()` (×8)	574	1526	5213	16361	50099
`GMP mpn_sqr()` (×8)	376	967	2994	9871	31322
Mul. Schoolbook	**61**	**223**	890	4076	16326
Mul. Karatsuba		258	**849**	**3054**	**10367**
Squaring Schoolbook	**49**	**149**	**501**	**1898**	8923
Squaring Karatsuba		220	622	1964	**6882**

A few comments on the results:

- Concerning the multiplication, the Schoolbook Algorithm 1 is better than its Karatsuba counterpart for the 260 and 520-bit sizes. This is consistent with the complexities
- Concerning the squaring, one can see that the Karatsuba approach is better only in the 4154-bit case. This is explained by the fact that the Schoolbook squaring (Algorithm 2) makes intensive use of the `madd52lo`

and `madd52hi` instructions with no use of additions except for the carry management, whereas in the Karatsuba case, except for the elementary squarings, the final reconstruction requires a lot of additions which can not be saved in the same way as in Schoolbook case. Thus, the threshold for better efficiency in the Karatsuba case is much higher in comparison with the multiplication case.

6.3 Performances of the batch Montgomery multiplication and squaring

The performances of the batch Montgomery multiplications and squarings are given in Table 7, for the classical and truncated versions. We evaluated our implementations for 1024, 2048 and 4096-bit operands.

Because of the interleaved word-size multiplications, Algorithm 4 has only been implemented with the Schoolbook multiplication. The 1024-bit Karatsuba multiplication is slower than the Schoolbook multiplication. As a consequence, we have not implemented Truncated Karatsuba versions for this size.

For comparison sake, we implemented Montgomery modular multiplication and squaring using `GMP mpn` Montgomery operations, and `OpenSSL BN_mod_mul_montgomery` functions. However, we have not found a specific `OpenSSL` counterpart of the Montgomery modular squaring. The best timings are in bold in Table 7. The best implementations are more than 4 times faster than `OpenSSL` in all cases. Furthermore, the truncated approach is about 21 % faster than the conventional Schoolbook Montgomery multiplication, and up to nearly 28 % faster than the 4108-bit squaring.

For the Karatsuba approaches, the truncated version is almost 21 % faster than the classical multiplication for 2048-bit operands and up. In the case of the squaring the improvement is about 15 %. Even for the largest size (4154 bits), the Truncated Karatsuba modular squaring remains slightly slower than its Schoolbook counterpart, while the Karatsuba modular multiplication is nearly 10 % faster than the Schoolbook version.

Table 7: Batch Montgomery multiplications and squarings (normal and truncated) # clock cycles.

		Montgomery modular multiplications			Montgomery modular squarings		
Modulus Size		1024	2048	4096	1024	2048	4096
`GMP` (×8)		9862	35497	120342	8358	30465	101373
`OpenSSL` (×8)		7949	29928	116898	-	-	-
Batch	**Multiplication**	**This work**					
Algorithm 3	Schoolbook	2276	8696	38609	1885	7154	32107
Algorithm 4	Schoolbook	2162	8936	42730	-	-	-
Algorithm 3	Truncated Schoolbook	**1847**	7306	30492	**1439**	**5548**	**23413**
speedup vs `GMP`		5.34	4.86	3.95	5.81	5.49	4.33
speedup vs `OpenSSL`		4.31	4.10	3.69	-	-	-
Algorithm 3	Karatsuba	2423	8400	34628	2191	7330	28841
Algorithm 3	Truncated Karatsuba	-	**7286**	**27594**	-	6202	23736
speedup vs `GMP`		4.07	4.87	4.36	3.81	4.91	4.27
speedup vs `OpenSSL`		3.28	4.12	4.24	-	-	-

6.4 Window exponentiation with Truncated Montgomery Modular Mutiplication

We have implemented Left-to-Right fixed-window exponentiation in a constant time fashion for 1024, 2048 and 4096-bit operands, i.e. the modulus size. These implementations make use of the batch Montgomery squarings, `B_fma`s and multiplications mentioned above.

Algorithm 7 Constant-time Batch Fixed-Window Left-to-Right Exponentiation

---

Require: Eight values a_k, the corresponding eight s-bit exponents and moduli e_k and m_k, all stored in 64-bit word arrays, the window width w.
Ensure: The eight modular exponentiations $y_k = a_k^{e_k} \bmod m_k$

    //batch of 8 values A_k stored in 52-bit slices in t_{52} `__m512i` shares
1: $A_k \leftarrow$`expand`$(a_0, \ldots, a_7)$
    //batch of 8 moduli M_k stored in 52-bit slices in t_{52} `__m512i` shares
2: $M_k \leftarrow$`expand`$(m_0, \ldots, m_7)$
    //batch of 8 exponents E_k stored in 64-bit slices in t_{64} `__m512i` shares
3: $E_k \leftarrow$`expand`$_{64}(e_0, \ldots, e_7)$
4: **for** k from 0 to 7 **do in parallel**
5:     $Y_k \leftarrow \mathbf{1}$ // batch of ones
6:     **for** i from 0 to 2^w **do** //precomputation
7:         $G_k[i] \leftarrow A_k^i \bmod M_k$
8:     **end for**
9:     **for** i from $s - 2w$ to 0 by w **do**// main loop
10:         $b_k \leftarrow E_k[i, i + w - 1]$ // w bits of E_k
11:         $tmp_k \leftarrow G_k(b_k)$ // constant-time batch selection
12:         $Y_k \leftarrow Y_k^{2^w} \bmod M_k$ // w-batch Montgomery squarings
13:         $Y_k \leftarrow Y_k \times tmp_k \bmod M_k$
14:     **end for**
15:     loop epilog if necessary
16: **end for**
    //backward conversion of the 8 results in 64-bit word arrays
17: $y_k \leftarrow$ `contract`(Y_k)
18: return the eight results $y_k = a_k^{e_k} \bmod m_k$

They aim to compare our approach for the modular operations with state-of-the-art modular exponentiations. The exponentiation functions take as arguments big integers represented by 64-bit word arrays, identical to those used in the low-level functions of the `GMP` library. The result is stored in the same fashion.
Thus, the exponentiation is processed as follows:

- conversion of a batch of operands stored in 64-bit word arrays in word-slicing representation.
- computation of the batch exponentiation
- backward conversion to a batch of results stored in 64-bit word arrays.

This conversion is implemented in conventional C using maskings and shiftings. Since the complexity is linear in the operand size, the cost remains negligible. Nevertheless, we provide in Table 8 the timings of our implementations in clock cycles. The forward conversion from the conventional representation to the batch 52-bit representation is called `expand` and the backward conversion is called `contract`.
We tested window sizes from 1 (L-R Square-and-Multiply-always) to 5. The best window size is 4 for 1024-bit moduli and 5 in the other cases. We used

Table 8: Batch conversions, $\#10^3$ clock cycles.

	size	1040	2080	4108
# clock cycles	expand	1165	2411	4790
	contract	1726	3274	6505

the fastest modular squaring and multiplication. For 1024-bit moduli, we used Algorithm 4. In the other cases, we used Algorithm 3. This approach is shown Algorithm 7.

6.4.1 Experimentation of AVX512 versions

The timings are summarized in the last three columns of Table 9, for the implemented modulus sizes (1024, 2048 and 4096 bits).
Whatever the multiplication used (Schoolbook or Karatsuba), the truncated version is always faster than the classical version. The improvement ranges from 13% faster with the 4096-bit Truncated Karatsuba, to 20 % faster with the 1024-bit Truncated Schoolbook version. As expected, the Karatsuba approach is only better for the largest size of 4096 bits.
We compare these results with GMP and OpenSSL libraries, providing the timings for eight modular exponentiation computations.

- The GMP version is the 6.2.0 and we used the function mpn_sec_powm, which is specifically designed for cryptographic use.
- The OpenSSL version is the 3.2.1. This version is compiled on our platform and provides the RSA_Z operations, which make use of the AVX2 instruction set. We measured the performance of two functions:
  - BN_mod_exp_mont_consttime, which is the RSA_Z implementation (AVX2 version, see [GK12]).
  - BN_mod_exp_mont_consttimex2, which computes 2 exponentiations simultaneously. For 1024 and 2048-bit operands, this function implements the VPMADD52 instructions, but in 256-bit registers. For the 4096-bit operands, this function falls back on two calls of BN_mod_exp_mont_consttime. This explains why we do not present 4096-bit performance results for this function Table 9.

  These functions are constant-time fixed-window exponentiations, and implement a CIOS-like Montgomery modular multiplication and squaring.

The OpenSSL functions offers better results than GMP ones.
Our implementation uses a constant time fixed-window approach similar as the one of the OpenSSL functions. Compared to OpenSSL BN_mod_exp_mont_consttime, the best speedup of our implementations is achieved for the 1024-bit operands, with almost 4 times fewer cycles per exponentiation (our Truncated Schoolbook). The best other cases provide speedups of 3.71 and 3.37, respectively, for the 2048-bit Schoolbook and the 4096-bit Karatsuba.
Compared to OpenSSL BN_mod_exp_mont_consttimex2, the best speedup of our implementations is achieved for the 1024-bit operands, with 1.75 times fewer cycles per exponentiation (our Truncated Schoolbook). The best other cases provide speedups of 1.38 and 1.27, respectively, for the 2048-bit Schoolbook and Karatsuba.

One may notice that the non-truncated versions remain better than their RSA_Z BN_mod_exp_mont_consttime or BN_mod_exp_mont_consttimex2 counterparts.

Table 9: Batch of 8 modular fixed-window exponentiations, $\#10^3$ clock cycles.

$\times 8$ Modular Fixed-Window exponentiation ($\times 10^3$#cc)				
Modulus Size		1024	2048	4096
GMP ($\times 8$)		11333	81288	614637
OpenSSL BN_mod_exp_mont_consttime ($\times 8$)		8313	58445	434359
OpenSSL BN_mod_exp_mont_consttimex2 ($\times 4$)		3648	21674	442533
Batch	**multiplication**	**This work**		
Algorithm 3 & 4	Schoolbook	2589	19678	177095
Algorithm 3	Truncated Schoolbook	**2090**	**15736**	131753
speedup vs GMP		5.42	5.17	4.66
speedup vs OpenSSL BN_mod_exp_mont_consttime		3.98	3.71	3.30
speedup vs OpenSSL BN_mod_exp_mont_consttimex2		1.75	1.38	-
Algorithm 3	Karatsuba	-	19717	144567
Algorithm 3	Truncated Karatsuba	-	17124	**129015**
speedup vs GMP		-	4.75	4.76
speedup vs OpenSSL BN_mod_exp_mont_consttime		-	3.41	3.37
speedup vs texttttOpenSSL BN_mod_exp_mont_consttimex2		-	1.27	-

6.4.2 Experimentation of 256-bit versions

In order to provide a fair comparison with the BN_mod_exp_mont_consttimex2 OpenSSL implementation using the 256-bit vectorized fused multiplier-adder (_mm256_madd52*_epu64(a, b, c)), we derived 256-bit versions from the previous implementations, computing a batch of four values instead of eight for 512-bit versions.

Comparison between 256-bit and 512-bit implementations. We discuss here the register size impact. In our implementations, the difference between both 512-bit and 256-bit versions is in the registers used: respectively zmm and ymm. Thus, one might expect the retired instruction number to be the same between the two versions. However, this is not the case for the clock cycle numbers, since the instruction throughput in the 256-bit case is much lower than that of the 512-bit instructions. In order to check the performance level of both versions, we provide the clock cycle numbers per exponentiation of both versions. In other words, this is the batch delay divided by the number of operations in the batch, 8 and 4 respectively (Table 10).
In any case, the clock cycle number per exponentiation is better in the AVX512 configurations. The advantage ranges from 14.4% (Truncated - 1024 bits) to 31.5 % (Truncated - 2048 bits) while it is between 20 and 30 % in the other configurations.

Comparison with OpenSSL BN_mod_exp_mont_consttimex2 function.
Since the BN_mod_exp_mont_consttimex2 function provides vectorized computations for sizes from 1024 to 2048 bits, we provide the comparison between our work and the OpenSSL function for these sizes (Table 11). We compare our

Table 10: Batch Modular fixed-window exponentiations, $\#10^3$ clock cycles per exponentiation.

Modular Fixed-Window exponentiation			
Register Type		512-bit $\times 10^3\#cc/8$	256-bit $\times 10^3\#cc/4$
Mod.size	**Multiplication**	**This work**	
1024 bits	Schoolbook	325	419
	Truncated Schoolbook	263	301
2048 bits	Schoolbook	2401	2918
	Truncated Schoolbook	1965	2583

batch implementation, which computes 4 exponentiations simultaneously, with two successive runs of the `OpenSSL` `BN_mod_exp_mont_consttimex2`.

Table 11: Batch of 4 Modular fixed-window exponentiations, $\#10^3$ clock cycles.

$\times 4$ **Modular Fixed-Window exponentiation ($\times 10^3\#cc$)**				
Modulus Size bits		1024	2048	ratio 2048/1024
`OpenSSL` `BN_mod_exp_mont_consttimex2` ($\times 2$)		1824	10837	5.94
Batch	**Multiplication**	**This work**		
Algorithm 3 & 4	Schoolbook	1678	11632	6.96
Algorithm 3	Truncated Schoolbook	**1203**	**10290**	8.59
speedup truncated vs `BN_mod_exp_mont_consttimex2`		**1.52**	**1.05**	

Some comment on this table about the 256-bit versions of our modular exponentiation:

- The best speedup is achieved for the 1024-bit modulus size. Our implementation using the Truncated Montgomery reduction provides a speedup of 1.51 (window width = 5).
- Our 1024-bit modulus implementation using the conventional Montgomery reduction is also slightly faster than the `BN_mod_exp_mont_consttimex2` function, with a 1.09 speedup (window width = 4).
- For the 2048-bit modulus, our implementation using the conventional Montgomery reduction is slightly slower than the `BN_mod_exp_mont_consttimex2` function. We remind here that for a square-and-multiply variant algorithm based implementation, the theoretical ratio should be 8 between the 2048-bit and the 1024-bit versions. These ratios are indicated in the right column of Table 11. One can see that this ratio is better in the `OpenSSL` case, especially compared to our versions using the Truncated Montgomery reduction. As a conclusion, compared to the `BN_mod_exp_mont_consttimex2` function, while our implementation using the conventional Montgomery reduction is slightly slower by around 7 % (window width = 5), the one using our truncated approach for the modular reduction gives a 1.05 speedup (window width = 4).

As a conclusion, this shows the potential of the Truncated Montgomery reduction applied to various implementation situations. In every case explored

here, our proposed approach is, to the best of our knowledge, faster than the state-of-the-art implementations.

7 Conclusion

In this paper, we present new software implementations using `AVX512` instruction set and taking advantage of the `VPMADD52` instructions, which compute a vectorized fused multiplication-addition. We implemented multi-precision multiplications and squarings, for sizes from 260 to 4154 bits. We used these implementations in Montgomery modular multiplications and squaring along with CIOS Montgomery multiplications. We also present a new approach of Truncated Montgomery multiplication computing the most significant higher half part of one of the multiplications involved in the Montgomery modular reduction in order to speedup the computation. Our implementations of this approach are more than 4 times faster than the `OpenSSL` ones. Moreover, used in fixed-window exponentiations of sizes 1024, 2048 and 4096 bits, compared to `BN_mod_exp_mont_consttime`, the best speedups are respectively 3.98, 3.71, 3.37 for our implementations using our new Truncated Schoolbook or Karatsuba Montgomery modular multiplications. Compared to `BN_mod_exp_mont_consttimex2` using `madd52*` in 256-bit registers, in fixed-window exponentiations of sizes 1024 and 2048, our `AVX512` implementations provide speedups of 1.75 and 1.38 respectively, while their 256-bit counterparts give speedups of 1.51 and 1.05 for 1024 and 2048-bit sizes (batch of 4 values in this case).
The speedups are good because the batch operations are highly parallel and can be easily vectorized. Similar results could be obtained on other processors whose architecture has SIMD instruction sets. The NEON instruction set on ARM processors [ARM] offers such possibilities. However, these instructions are not exactly identical to `AVX512` extensions. For example, `VPMADD52` instructions have only a 32-bit equivalent on this architecture. This could lead to a different word slicing of the operands. Very significant gains over sequential implementations can be reach, but may not be the same to those with `AVX512` instructions.

Perspectives. The improvements presented in this work could be adapted to other contexts as well. Batch computations have demonstrated their value and could be utilized, for instance, in post-quantum schemes. While our study of Truncated Montgomery multiplication was focused on RSA, there are other potential applications worth studying. Indeed, schemes relying on supersingular elliptic curves like pairing-based cryptography (see [MJ17] and [DL06]) or isogeny based post-quantum protocols (see [FKL+20]), also require large integer modular multiplications.
In addition, homomorphic encryption protocols based on large integers may take advantage of our approach, among them, Coron *et al.* [CMNT11] and Dyer *et al.* [DDX19].

References

[ABS10] S. Antão, J.-C. Bajard, and L. Sousa. Elliptic Curve Point Multiplication on GPUs. In *ASAP 2010-21st IEEE International Confer-*

ence on Application-specific Systems, Architectures and Processors, pages 192–199. IEEE, 2010. doi:10.1109/ASAP.2010.5541000.

[Arc] Architecture specification. Intel Advanced Vector extensions 10. https://www.intel.com/content/www/us/en/content-details/784267/intel-advanced-vector-extensions-10-intel-avx10-architecture-specification.html.

[ARM] ARM. Neon intrinsics reference. https://developer.arm.com/architectures/instruction-sets/intrinsics/.

[Bar86] P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. *Advances in Cryptology — CRYPTO' 86*, 263:311–323, 1986. doi:10.1007/3-540-47721-7_24.

[BCC+09] D. J. Bernstein, H.-C. Chen, M.-S. Chen, C.-M. Cheng, C.-H. Hsiao, T. Lange, Z.-C. Lin, and B.-Y. Yang. The Billion-Mulmod-Per-Second PC. In *Workshop record of SHARCS*, volume 9, pages 131–144, 2009.

[BGH22] B. Buhrow, B. Gilbert, and C. Haider. Parallel Modular Multiplication using 512-bit Advanced Vector Instructions: RSA Fault-Injection Countermeasure via Interleaved Parallel Multiplication. *Journal of Cryptographic Engineering*, 12(1):95–105, 2022. doi:10.1007/s13389-021-00256-9.

[BI21] C. Bouvier and L. Imbert. An Alternative Approach for SIDH Arithmetic. In Juan A. Garay, editor, *Public-Key Cryptography – PKC 2021*, pages 27–44, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-75245-3_2.

[BKP21] J. W. Bos, T. Kleinjung, and D. Page. *Efficient Modular Multiplication*, chapter 8. Lecture note series. Cambridge University Press, 2021. URL: www.cambridge.org/9781108795937.

[BMSZ14] J. W. Bos, P. L. Montgomery, D. Shumow, and G. M. Zaverucha. Montgomery Multiplication Using Vector Instructions. In *Selected Areas in Cryptography – SAC 2013*, pages 471–489, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. doi:10.1007/978-3-662-43414-7_24.

[Bos12] J. W. Bos. Low-latency Elliptic Curve Scalar Multiplication. *International Journal of Parallel Programming*, 40:532–550, 2012. doi:10.1007/s10766-012-0198-5.

[BSS99] I. F. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999. doi:10.1017/CBO9781107360211.

[CFG+21] H. Cheng, G. Fotiadis, J. Groszschädl, P. Y.A. Ryan, and P. Roenne. Batching CSIDH Group Actions using AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(4):618–649, 2021. doi:10.46586/tches.v2021.i4.618-649.

[CFGR22] H. Cheng, G. Fotiadis, J. Groszschädl, and P. Y.A. Ryan. Highly Vectorized SIKE for AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022(2):41–68, 2022. doi:10.46586/tches.v2022.i2.41-68.

[CMNT11] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 487–504, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-22792-9_28.

[DDX19] J. Dyer, M. Dyer, and J. Xu. Practical homomorphic encryption over the integers for secure computation in the cloud. *International Journal of Information Security*, 18(5):549–579, Oct 2019. doi:10.1007/s10207-019-00427-0.

[DG19] N. Drucker and S. Gueron. Fast Modular Squaring with AVX512IFMA. In *16th International Conference on Information Technology-New Generations (ITNG 2019)*, pages 3–8, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-14070-0_1.

[DGK18] N. Drucker, S. Gueron, and V. Krasnov. Fast Multiplication of Binary Polynomials with the Forthcoming Vectorized VPCLMULQDQ Instruction. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 115–119, 2018. doi:10.1109/ARITH.2018.8464777.

[DL06] S. Duquesne and T. Lange. *Pairing-based cryptography*, chapter 24, pages 573–590. Discrete Mathematics and Its Applications. Chapman and Hall/CRC Press, 2006. doi:10.1201/9781420034981.ch24.

[DL18] J. Ding and S. Li. A Modular Multiplier Implemented With Truncated Multiplication. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(11):1713–1717, 2018. doi:10.1109/TCSII.2017.2771239.

[DL20] J. Ding and S. Li. A Low-Latency and Low-Cost Montgomery Modular Multiplier Based on NLP multiplication. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(7):1319–1323, 2020. doi:10.1109/TCSII.2019.2932328.

[ELWW16] N. Emmart, J. Luitjens, C. Weems, and C. Woolley. Optimizing Modular Multiplication for Nvidia's Maxwell GPUs. In *2016 IEEE 23nd symposium on computer arithmetic (ARITH)*, pages 47–54. IEEE, 2016. doi:10.1109/ARITH.2016.21.

[ET20] T. Edamatsu and D. Takahashi. Accelerating Large Integer Multiplication Using Intel AVX-512IFMA. In *Algorithms and Architectures for Parallel Processing*, pages 60–74, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-38991-8_5.

[EZW18] N. Emmart, F. Zheng, and C. Weems. Faster Modular Exponentiation using Double Precision Floating Point Arithmetic on the GPU. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 130–137. IEEE, 2018. doi:10.1109/ARITH.2018.8464792.

[FKL+20] L. De Feo, D. Kohel, A. Leroux, C. Petit, and B. Wesolowski. SQISign: compact post-quantum signatures from quaternions and isogenies. Cryptology ePrint Archive, Paper 2020/1240, 2020. URL: https://eprint.iacr.org/2020/1240, doi:10.1007/978-3-030-64837-4_3.

[Ga] T. Granlund and al. GNU multiple Precision Arithmetic Library 6.1.2. https://gmplib.org/.

[GGP08] P. Grabher, J. Groszschaedl, and D. Page. On Software Parallel Implementation of Cryptographic Pairings. Cryptology ePrint Archive, Paper 2008/205, 2008. URL: https://eprint.iacr.org/2008/205, doi:doi.org/10.1007/978-3-642-04159-4_3.

[GK12] S. Gueron and V. Krasnov. Software Implementation of Modular Exponentiation, using Advanced Vector Instructions Architectures. In *Arithmetic of Finite Fields: 4th International Workshop, WAIFI 2012, Bochum, Germany, July 16-19, 2012. Proceedings 4*, pages 119–135. Springer, 2012. doi:10.1007/978-3-642-31662-3_9.

[GK16] S. Gueron and V. Krasnov. Accelerating Big Integer Arithmetic using Intel IFMA Extensions. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 32–38. IEEE, 2016. doi:10.1109/ARITH.2016.22.

[Har05] L. Hars. Fast Truncated Multiplication for Cryptographic Applications. In *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 211–225. Springer Berlin Heidelberg, 2005. doi:10.1007/11545262_16.

[Har06] L. Hars. Applications of Fast Truncated Multiplication in Cryptography. *EURASIP Journal on Embedded Systems*, 2007(1):061721, Dec 2006. doi:10.1155/2007/61721.

[Int] Intel. Intel intrinsics guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[KKAK96] C. Kaya Koc, T. Acar, and B.S. Kaliski. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16(3):26–33, 1996. doi:10.1109/40.502403.

[Kra17] V. Krasnov. On the dangers of intel's frequency scaling, 2017. URL: https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/.

[MC14] E. M Mahé and J.-M. Chauvet. Fast GPGPU-Based Elliptic Curve Scalar Multiplication. *Cryptology ePrint Archive*, 2014. URL: https://eprint.iacr.org/2014/198.

[MJ17] N.E. Mrabet and M. Joye. *Guide to Pairing-Based Cryptography*. Chapman and Hall/CRC Cryptography and Network Security Series. CRC Press, 2017. URL: https://books.google.fr/books?id=jmwNDgAAQBAJ.

[Mon85] P. L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, 1985. doi:10.2307/2007970.

[Pro] The OpenSSL Project. Openssl. https://www.openssl.org/.

[RSA78] R. L. Rivest, A. Shamir, and L. M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. doi:10.1145/359340.359342.

[RV22] J. M. Robert and P. Véron. Faster Multiplication over $\mathbb{F}_2[X]$ using AVX512 Instruction Set and VPCLMULQDQ Instruction. *Journal of Cryptographic Engineering*, January 2022. URL: https://cnrs.hal.science/hal-03520854, doi:10.1007/s13389-021-00278-3.

[Sch96] B. Schneier. *Applied Cryptography - Protocols, Algorithms, and Source Code in C, 2nd Edition*. Wiley, 1996. URL: https://www.worldcat.org/oclc/32311687.

[Tak20] D. Takahashi. Fast multiple Montgomery Multiplications Using Intel AVX-512IFMA Instructions. In *Computational Science and Its Applications – ICCSA 2020*, pages 655–663, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-58814-4_52.

[Tre13] Wilke Trei. Efficient modular arithmetic for SIMD devices. Cryptology ePrint Archive, Report 2013/652, 2013. URL: https://eprint.iacr.org/2013/652.