Faster multiplication over $F\_2[X]$ using AVX512 instruction set and VPCLMULQDQ instruction

Jean-Marc Robert, Pascal Véron

Noname manuscript No.
(will be inserted by the editor)

Faster Multiplication over $\mathbb{F}_2[X]$ using AVX512 instruction set and VPCLMULQDQ instruction

Jean-Marc Robert · Pascal Véron

This is a pre-print of an article published in "Journal of Cryptographic Engineering". The final authenticated version is available online at:
https://doi.org/10.1007/s13389-021-00278-3

the date of receipt and acceptance should be inserted later

Abstract Code based cryptography is one of the main proposition for the post-quantum cryptographic context, and several protocols of this kind have been submitted on the NIST platform. Among them, BIKE and HQC are part of the five alternate candidates selected in the third round of the NIST standardization process in the KEM category. These two schemes make use of multiplication of large polynomials over binary rings, and due to the polynomial size (from 10000 to 60000 bits), this operation is one of the costliest during key generation, encapsulation or decapsulation mechanisms. In BIKE-2, there is also a polynomial inversion which is time consuming and this problem has been addressed in [11]. In this work, we revisit the different existing constant-time algorithms for arbitrary polynomial multiplication. We explore the different Karatsuba and Toom-Cook constructions in order to determine the best combinations for each polynomial degree range, in the context of AVX2 and AVX512 instruction sets. This leads to different kernels and constructions in each case. In particular, in the context of AVX512, we use the VPCLMULQDQ instruction, which is a vectorized binary polynomial multiplication instruction. This instruction deals with up to four polynomial (of degree up to 63) multiplications, that is four operand pairs of 64-bit words with 128-bit word storing each results, the four results being stored in one single 512-bit word. This allows to divide by roughly 3 the retired instruction number of the operation in comparison with the AVX2 instruction set implementations, while the speedup is up to 39% in terms of processor clock cycles. These results are different than the ones estimated in [10]. To illustrate the benefit of the new VPCLMULQDQ instruction, we used the HQC code to evaluate our approaches. When implemented in the HQC protocol, for the security levels 128, 192 and 256, our approaches provide up to 12% speedup, for key pair generation.

Keywords

Finite field multiplication, Karatsuba, Toom-Cook, post-quantum cryptography, code based cryptography, AVX2, AVX512, VPCLMULQDQ.

1 Introduction

In 2017, the NIST launched a consultation dealing with the so-called "Post-Quantum Cryptography" (PQC) [24], leading to think that a practical quantum computer might appear in the next two or three decades. Among the candidates known to resist against quantum computers, several submissions on the NIST platform are code based protocols. The public key cryptosystem of Mc Eliece marked the beginning of code based cryptography [22]. The security of most of the code based protocols relies on a decision problem which can be stated without using the terminology of coding theory: the SD (Syndrome Decoding) problem.

Input	:	H a (k, n) matrix over $\mathbb{F}_2$, $s \in \mathbb{F}_2^k$ a column vector, p an integer.
Question	:	Is there a column vector $e \in \mathbb{F}_2^n$, with at most p non-zero coordinates, such that $He = s$?

J.-M. Robert, P. Véron
Institut de Mathématiques de Toulon
Université de Toulon, France
E-mail: jean-marc.robert@univ-tln.fr
E-mail: veron@univ-tln.fr

Although this problem is NP-complete [7], in practice, the efficiency of the probabilistic algorithms devoted to solve the SD problem [6] has as a consequence that code based cryptography usually suffers from huge keys. Numerous strategies have been deployed to obtain a compact representation of the key. Among them, the use of double circulant codes [13] leads to secure protocols with short keys.

Definition 1 An $n \times n$ matrix is called a circulant matrix if each row is obtained from the previous one by a cyclic shift over one position to the right.

$$A = \begin{pmatrix} a_0 & a_1 & \dots & a_{n-2} & a_{n-1} \\ a_{n-1} & a_0 & \dots & a_{n-3} & a_{n-2} \\ \vdots & \vdots & & \vdots & \vdots \\ a_1 & a_2 & \dots & a_{n-1} & a_0 \end{pmatrix}.$$

In the sequel, we use some coding theory terminology. The reader may refer to [21, chapter 1] for more information on coding theory.

Definition 2 Let $k, n \in \mathbb{N}$, an (n, k) linear code $\mathcal{C}$ over $\mathbb{F}_2$ is a k dimensional subspace of $\mathbb{F}_2^n$.

Definition 3 A parity check matrix of an (n, k) linear code $\mathcal{C}$ is an $(n-k) \times n$ matrix H over $\mathbb{F}_2$ such that $H\ {}^t c = 0$ iff $c \in \mathcal{C}$.

Definition 4 A $(2n, n)$ double circulant code $\mathcal{C}$ is a linear code such that :

$$(c_0, c_1, \dots, c_{n-2}, c_{n-1}, c_n, c_{n+1}, \dots, c_{2n-2}, c_{2n-1}) \in \mathcal{C}$$
$$\Downarrow$$
$$(c_{n-1}, c_0, \dots, c_{n-3}, c_{n-2}, c_{2n-1}, c_n, \dots, c_{2n-3}, c_{2n-2}) \in \mathcal{C}.$$

A parity check matrix H of the $(2n, n)$ double circulant code has the following form :

$$H = \left(A \;\middle|\; M \right),$$

where A and M are two $n \times n$ circulant matrices.

A parity check matrix of a $(2n, n)$ double circulant code can be stored, in a compact way, using only its first row. There is no general complexity result for the SD problem where H is the parity check matrix of a random double circulant code. However, in practice, up to a small factor, the best attacks against the SD problem in this case are the same as those for random binary codes. Indeed, according to [14], when n is prime and 2 is a primitive root of $\mathbb{Z}/n\mathbb{Z}$, almost all random double circulant codes lie on the Gilbert-Varshamov bound. As a result, the SD problem is considered hard by the cryptographic community for double circulant codes.

Let $y = (y_0, \dots, y_{2n-1}) \in \mathbb{F}_2^{2n}$ a column vector and let us define $y^{(1)}(X) = y_0 + y_1 X + \cdots + y_{n-1} X^{n-1}$ and $y^{(2)}(X) = y_n + y_{n+1} X + \cdots + y_{2n-1} X^{2n-1}$. Given that the algebra of $n \times n$ circulant matrices over $\mathbb{F}_2$ is isomorphic to the algebra of polynomials in the ring $\mathbb{F}_2[X]/(X^n - 1)$, through the mapping ψ such that $\psi(A) = a_0 + a_1 X + a_2 X^2 + \cdots + a_{n-1} X^{n-1}$, then the product Hy boils down to the computation of two polynomial multiplications, namely:

$$\psi(A) \times y^{(1)}(X) \pmod{X^n - 1}$$

and

$$\psi(M) \times y^{(2)}(X) \pmod{X^n - 1}.$$

BIKE [5] and HQC [3] make use of this isomorphism which maps a matrix-vector product into a polynomial multiplication in $\mathbb{F}_2[X]/(X^n - 1)$. Due to the polynomial size (from 10000 to 60000 bits) in both protocols, it turns out that this operation has an impact on key generation, key encapsulation and key decapsulation mechanisms.

For example, in the HQC submission, key generation requires one multiplication, encapsulation requires two multiplications and decapsulation requires three multiplications. These multiplications are computed over $\mathbb{F}_2[X]/(X^N - 1)$, with $17669 \leqslant N \leqslant 57637$.

Moreover, the multiplications are performed with one sparse operand, while the other one is dense. Sparse-dense multiplications are classically implemented using convolution approaches which is an adapted version of the schoolbook approach (for example, see Aranha *et al.* in [15]).

The report [4] mentions that side-channel resistance is a desirable security property for NIST PQC candidates. A minimum requirement for cryptographic primitives to ensure this property is to provide constant time implementations. In BIKE and HQC, some secret data are represented as sparse polynomials used as operand of a multiplication by an arbitrary polynomial, in the three steps of the protocol: key generation, encryption and decryption mechanisms. Thus, any multiplication algorithm taking advantage of the sparsity of the secret data may leak some information on it. An adversary able to exploit such source of leakage may recover information on secret data. That is why dense-dense approaches, which process the sparse operand as an arbitrary polynomial, are to be considered as mandatory.

Dense-dense multiplication over $\mathbb{F}_2[X]$ has been intensively studied in the past, for different applications:

- schoolbook approaches (with quadratic complexity);
- Karatsuba-Offmann [20] and Toom-Cook [8] subquadratic methods, with interpolation-evaluation algorithms;

– Schonhäge-Strassen [26] and Fürer [12] FFT based methods, and recent works (see Harvey *et al.* in [17, 16]) showing a quasi-linear complexity in $\mathcal{O}(n \log n)$ for integer multiplication.

One reference for the dense-dense operation is the general purpose NTL library (see [1]), which aims to provide the whole set of operations, and is based on the gf2x library [18] for the characteristic 2 operations. All the different approaches mentioned above are implemented. However, this library and the underlying gf2x have been designed for general purpose use and are optimized for generic operations with operand of any size.

In terms of operand size, the HQC [2] protocol deals with polynomials whose size is fixed as a protocol parameter. The range of sizes corresponds to the one for which the Karatsuba and Toom-Cook approaches are the best in the state-of-the-art. Indeed, in the gf2x library, over $\mathbb{F}_2[X]/(X^N - 1)$, with $17669 \leqslant N \leqslant 57637$, the computations make use of Karatsuba or Toom-Cook (split by 3 or 4) approaches, while the threshold for FFT approaches is above 240000 bits (*i.e.* the degree of the polynomials). These bounds are relevant with the recent results of Harvey *et al.* in [17,16].

In terms of software implementation on x86-64 platforms, until recently, the state-of-the-art was AVX2 instruction set implementations, especially using the PCLMULQDQ instruction (see the gf2x library). This instruction performs a binary polynomial (of degree at most 63) multiplication over $\mathbb{F}_2[X]$. It returns the result stored in a 128-bit word, either an xmm 128-bit register or a same size memory storage location. The AVX2 instruction deals with 256-bit registers or memory words and performs various vectorized operation on packed operands. In 256-bit words, one can store either 32 bytes, 16 16-bit words, 8 double words or 4 quadwords (whose size is 64 bits).

In 2018, Intel announced a new instruction set extension in the so-called Icelake processor generation, which extends the AVX512 instruction set already available on some XEON processors. In particular, this architecture introduces a vectorized VPCLMULQDQ instruction, which performs up to four polynomial PCLMULQDQ multiplications, the four 128-bit results being stored in one single 512-bit word. Following this announcement, Drucker *et al.* [10] proposed a software implementation (for polynomials up to degree $2^{16} - 1$) using this instruction set, but could only experiment simulations or adapted versions of their software implementations, since no platforms and no Icelake processors were available at this time. However, they claimed up to 50% lowering of retired instructions and predicted the same drop in terms of processor clock cycle number execution. Their implementations consist in three core flows that perform schoolbook multiplication:

1. a 4×4 quadwords (64 bits) multiplication, written in AVX, using xmm registers and the PCLMULQDQ instruction,
2. a 4×4 quadwords (64 bits) multiplication, written in AVX512, using ymm registers and the VPCLMULQDQ instruction,
3. a 8×8 quadwords (64 bits) multiplication, written in AVX512, using zmm registers and the VPCLMULQDQ instruction.

They also mention Karatsuba multiplications for operand sizes above 256 bits. They provide the source code only for the second approach (4×4 quadwords multiplication, using ymm registers).

Contributions

In this work, we explore the Karatsuba and Toom-Cook multiplication construction and we identify the best combinations to be used depending on the polynomial degrees. As an illustration, we applied these results on the hqc-128 and hqc-192 multiplications of the Optimized Implementation of the HQC release, 2020/10/01 version [2]. In this release, the multiplication implementation make use of a 2-recursive 3-split Karatsuba. We show that using a Tom-Cook-3 approach, this provides some speedup in comparison with the initial multiplication of this release. As a consequence, this has been integrated in the last official HQC release, 2021/06/06 version.

Then, in the context of AVX512 instruction set, now available since the Icelake microarchitecture processors, we propose new implementations designed for cryptographic use of polynomial multiplications over $\mathbb{F}_2[X]$. We show that the elementary multiplication construction has to be a schoolbook approach up to the 256 bit operand level, while in the state-of-the-art AVX2 context, a Karatsuba multiplication is required at the threshold of 128 bit operands.

We then implement tailor made vectorized subquadratic approaches (recursive Karatsuba and Toom-Cook-3) using the AVX512 instruction set and the vectorized VPCLMULQDQ instruction in order to improve the performances, in comparison with current state-of-the-art AVX2 implementations. We compare our implementations:

– with the gf2x library;
– with the multiplications provided or derived from the Optimized Implementation of HQC.

Drucker *et al.* in [10] estimated that, by using the new AVX512 instruction set and especially the new VPCLMULQDQ one, the retired instruction count might be

divided by two, and the clock cycle number might be lowered in the same proportion. At the time they submitted their paper, there were no actual processor available implementing the instruction set extension with `VPCLMULQDQ`. In this paper, we checked their claims. We show that while the instruction count reduction can be overtaken, in our tests, the clock cycle number is lowered by about 39% only.

This paper is organized as follows : in Section 2 we present the Karatsuba multiplication over $\mathbb{F}_2[X]$ and our `AVX512` software implementation, the timing results and comparison with the implementations of Drucker *et al.* [10,11], the `gf2x` library and state-of-the-art `AVX2` implementation; in Section 3 we present the Toom-Cook multiplication over $\mathbb{F}_2[X]$ and our `AVX512` software implementation, the timing results and comparison with the `gf2x` library and state-of-the-art `AVX2` implementation; in Section 4 we present the performances obtained with the HQC protocol:

- when using our `AVX2` multiplications in `hqc-128` and `hqc-192` in the HQC release (round 3), 2020/10/01 version;
- when implementing our `AVX512` multiplications in the last official HQC release, 2021/06/06 version.

Finally Section 5 provides some concluding remarks.

The source code of all of our implementations are available at `https://github.com/arithcrypto/`, in the `AVX512PolynomialMultiplication` repository.

2 Karatsuba multiplication: algorithms and implementations

The Karatsuba multiplication algorithm is the first subquadratic approach, which has been presented by Karatsuba and Offmann in [20]. This multiplication was first applied to large integers, but can be applied to polynomials. This classical approach has been extensively studied since then, and our work relies on all those previous works. Our main contribution here is the `AVX512` software implementation of this approach, in order to speedup the runtime execution of multiplications over $\mathbb{F}_2[X]$.

First, we review the subquadratic Karatsuba approaches for multiplication over $\mathbb{F}_2[X]$. We then present our implementations and the performance results.

2.1 Karatsuba algorithm

One wants to multiply two arbitrary polynomials of degree at most $N-1$, and the result is of degree at most $2 \cdot N - 2$. The Karatsuba complexity applied to polynomial multiplication over $\mathbb{F}_2[X]$ has been studied by Nègre and Robert in [23]. Let A and B be two binary polynomials of degree at most $N-1$. These polynomials are packed into an array of 64-bit words, whose size is $\lceil N/64 \rceil$. Let $t = 2^r$ with r the minimum value ensuring $t \geqslant \lceil N/64 \rceil$. Now, A and B are considered as polynomials of degree at most $64 \cdot t - 1$. We reproduce the Karatsuba algorithm in Algorithm 1 Appendix A. From [23], the complexity of the recursive Karatsuba multiplication is : $8t^{\log_2(3)} - 8t$ `XOR` between 64-bit words and $t^{\log_2(3)}$ native 64-bit multiplication. We assume that this native multiplication line 2 (denoted `Mult64`) is performed using a single processor instruction: this is the case of the Intel Cores i3, i5 and i7 and above.

There are variants of these approaches, splitting the operands in all number of parts, and using an elementary multiplication which can be all sort of Karatsuba multiplication for example. These variants have been extensively studied in Weimerskirch and Paar in [27]. Algorithms 2 and 3 present the 3-way and 5-way split Karatsuba (see Appendix A).

In Table 1, we remind the complexity of recursive Karatsuba multiplication (t is the size of the operands in 64-bit words).

Apart the Schoolbook, which presents the worst complexity in terms of elementary multiplications, one can verify that the Karatsuba approaches are ordered in growing complexity for equivalent sizes.

2.2 `AVX512` Implementation

We propose here a little survey of the possible approaches for `AVX512` implementations. Our goal is to review the state-of-the-art (to our knowledge) and possibly propose improvements. We started to evaluate the implementation approaches from [10], which were based on the schoolbook algorithm for the 256 bit and 512 bit operand sizes. Their main claim is a reduction of 50% of the instruction count for the kernels they presented in this paper, and while an actual processor were not available at the time of their work, they evaluated a similar improvement in terms of clock cycle number for the computation time. In a more recent work, Drucker *et al.* in [11] proposed a new approach based on the Karatsuba algorithm. In our evaluation, this last work outperforms the first of [10]. In this section, we briefly review the main feature of the `AVX512` instruction set, and then present the most interesting approaches for elementary multiplication for the 512 bit operand size, from [11] and our work.

mult.	#pclmul	#xor
Schoolbook	t^2	$4 \times t^2$
KaratRec	$t^{\log_2(3)}$	$8 \times t^{\log_2(3)}$
Karat3	$6 \times ((t/3)^{\log_2(3)})$	$48 \times ((t/3)^{\log_2(3)})$
Karat5	$15 \times ((t/5)^{\log_2(3)})$	$120 \times ((t/5)^{\log_2(3)})$

Table 1: Complexity of the Karatsuba's multiplication variants

2.2.1 AVX512 instruction set and special features

The AVX512 are 512 bit extensions to the SIMD (Single Instruction Multiple Data) 256 bit AVX (Advanced Vector Extension) instructions for x86 instruction set architecture. This was proposed by Intel since 2013 and consists of multiple extensions. In the AVX512 processors, in addition to the general purpose 64 bit registers, larger registers are also available in order to perform vectorized instructions. These registers are of type

- xmm : of size 128 bits, i.e. containing two quadwords, thus denoted $\{a_1, a_0\}$, a_0 and a_1 being the quadwords in register a;
- ymm : of size 256 bits, i.e. containing four quadwords, thus denoted $\{a_3, a_2, a_1, a_0\}$, a_0, a_1, a_2 and a_3 being the quadwords in register a;
- zmm : of size 512 bits, i.e. containing eight quadwords, thus denoted $\{a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0\}$, a_0, a_1, a_2 , a_3, a_4, a_5, a_6 and a_7 being the quadwords in register a;

The number of registers of each type is 32.

2.2.2 The elementary polynomial multiplication:

Let us first remind how the PCLMULQDQ instruction works.

The intrinsic available by including the immintrin.h file is :

```
__m128i _mm_clmulepi64_si128 (__m128i a,
          __m128i b, const int imm8)
```

The quadwords a_i and b_i represent binary polynomials of degree at most 63. The PCLMULQDQ instruction returns the results in an xmm register, that is $a_j \times b_i$, of degree at most 126. The selection of i and j, i.e. the corresponding quadword of the operand is made according to the value of imm8 :

1. imm8 $= 0x00 : i = 0, j = 0 \rightarrow$ PCLMULQDQ returns $a_0 \times b_0$;
2. imm8 $= 0x01 : i = 0, j = 1 \rightarrow$ PCLMULQDQ returns $a_1 \times b_0$;
3. imm8 $= 0x10 : i = 1, j = 0 \rightarrow$ PCLMULQDQ returns $a_0 \times b_1$;
4. imm8 $= 0x11 : i = 1, j = 1 \rightarrow$ PCLMULQDQ returns $a_1 \times b_1$;

The VPCLMULQDQ instruction now available on Icelake and above platforms has the following intrinsic:

```
__m512i _mm512_clmulepi64_epi128 (__m512i a,
                    __m512i b, const int Imm8)
```

This instruction computes in parallel 4 pclmul multiplications, i.e. carryless multiplications of binary polynomials of degree at most 63, stored in 4 quadwords in 512 bit registers, as seen above. Thus, four 128 bit results are stored in the zmm register as follows:

$$\{\underbrace{a_{6+j} \times b_{6+i}}_{128\ bits}, \underbrace{a_{4+j} \times b_{4+i}}_{128\ bits}, \underbrace{a_{2+j} \times b_{2+i}}_{128\ bits}, \underbrace{a_j \times b_i}_{128\ bits}\}$$

The selection of i and j is made as above according to the value of Imm8.

We now examine the implementation of four multiplications using this instruction set and these registers: the mul512 version, from Drucker *et al.* in [11], our new karat_1_512 using the mul128x4 procedure from [11] and our new karat_mult_1_512_SB using a schoolbook mul128x4 procedure, and the full schoolbook 512 bit implementation, as suggested in [10]. We provide the detailed source code of the karat_mult_1_512_SB, and corresponding explanations in Appendix C, while the source code of the other approaches are available in the github repository, as mentioned in the Introduction.

We chose not to present here the schoolbook approaches of [10], because in our tests, these versions are outperformed by the others. Likewise, we also implemented 256 bit kernels using schoolbook (SB256) and Karatsuba at the 256 bit level Karat256 with the AVX512 and VPCLMULQDQ instruction. These versions are also outperformed by the others, in particular by the 8x8 multiplication of [10].

Nevertheless, we give an overview on these multiplications and their performances Appendix B and Appendix F, and the source code is also provided in the github repository.

2.2.3 *mul512 version, from Drucker* et al. *in [11]*

In [11], Drucker *et al.* present a multiplication of 1024 bit operands. This multiplication is computed as follows:

- The `mul1024` is a Karatsuba wrapper which calls three `mul512` multiplications, along with a classical Karatsuba reconstruction. This implementation is similar to the `AVX2` equivalent implementations except the register size.
- The `mul512` is a Karatsuba multiplication which splits in four parts the 512 bit operands. They use a four 512 bit word table, storing the 5 elementary xored operands in addition to the operands themselves, for a total of 9 pairs of 128 bit operands. After this step of operand preparation, the `mul512` procedure calls three times a `mul128x4` function in order to compute the 9 elementary 128 bit operand multiplications.
- Finally, the `mul128x4` procedure performs four 128 bit operand multiplications in parallel, using the `VPCLMULQDQ` instruction. This instruction is called three times, corresponding to the Karatsuba construction using 128 bit operands, split in two 64 bit words.

One may notice that the `mul512` procedure invokes 9 times the `VPCLMULQDQ` instruction, that is 36 elementary 64 bit operand multiplications, while using only $3^3 = 27$ out of them due to the Karatsuba multiplication. We refer the reader to [11, Appendix B], for a complete and detailed explanation of the source code.

2.2.4 *Our new `karat_mult_1_512` using the `mul128x4` procedure, from [11]*

Starting from the previous implementation of [11], our goal is to check the difference between the Karatsuba and the schoolbook approach at the 256 bit operand level. For this sake, we modified the `mul512` procedure into classic Karatsuba construction, which split in two parts the 512 bit operands. Now, three elementary 256 bit schoolbook multiplications are computed. These multiplications invoke one single call to the `mul128x4` procedure from [11]. In our version, the code of the `mul128x4` procedure has been manually inlined.

The total number of `VPCLMULQDQ` instructions in this 512 bit multiplication is 9, the same as previously. However, this procedure now makes use of all 36 elementary 64 bit multiplication computed, while slightly simplifying the final reconstruction at the 256 bit level. This version is called `karat_mult_1_512`, and its source code can be found in the github repository of the paper.

2.2.5 *Our new `karat_mult_1_512_SB` using a schoolbook `mul128x4` procedure*

This configuration implements the schoolbook algorithm at the 128 bit and 256 bit multiplication levels. Our goal now is to check which algorithm between Karatsuba and schoolbook is the best at the 128 bit operand level. Indeed, the instruction count is lower, while making use of one more `VPCLMULQDQ` instruction. The latency and throughput of this instruction is higher than conventional instructions. However, the vectorized version changes this by performing simultaneously four elementary 64 bit operand multiplications.

This approach now uses $3 \times 4^2 = 48$ elementary 64 bit operand multiplications in total. This version is called `karat_mult_1_512_SB` and is presented in details in Appendix C page 17.

2.2.6 *Full schoolbook 512 bit multiplication `SB512`*

From the description of Drucker *et al.* in [10], we also wrote a full schoolbook approach at the 512 bit size level.

This version now uses $4 \times 4^2 = 64$ elementary 64 bit operand multiplications in total, and is called `SB512`.

2.2.7 *Instruction counts for the four 512 bit multiplication versions*

The instruction count for both configurations of `mul128x4` is shown Table 2.

Instruction counts for `mul512`, `SB512`, `karat_mult_1_512` and `karat_mult_1_512_SB` are shown Table 3. For the Drucker *et al.* [11] version, we count three times the `mul128x4` instruction count plus the `mul512` instructions.

These instruction counts give an overview of the potential differences between the schoolbook and Karatsuba approaches at different levels. The threshold for the use of Karatsuba algorithm is at the lowest level in `AVX2` implementation. However, the vectorized `VPCLMULQDQ` instruction performs 4 multiplications at a time, with similar latency and throughput. Thus, in the context of `AVX512`, the good level for this threshold is 256 bit multiplication.

As expected (see Table 3), the instruction count is the lowest for the `karat_mult_1_512_SB`, corresponding to the schoolbook approach until the 256 bit operand size. The `mul512` of [11] instruction count is the greatest among the three approaches, due to the more complex final reconstruction of each Karatsuba recursion step.

Instruction count of mul128x4 128 bit size operands		
Instructions	Drucker *et al.* [11]	schoolbook (this work)
_mm512_clmulepi64_epi128	3	4
XOR	4	1
_mm512_mask_xor_epi64	2	2
_mm512_permutex_epi64	2	-
_mm512_permutexvar_epi64	1	1
Total	**12**	**8**

Table 2: Instruction count for the mul128x4 bit multiplication versions

Instruction count 512 bit size operands				
Instructions	karat_mult_1_512_SB	karat_mult_1_512	SB512	mul512 [11]
_mm512_clmulepi64_epi128	12	9	16	9
XOR	11	20	5	22
_mm512_mask_xor_epi64	11	11	14	11
_mm512_permutex_epi64	-	-	-	6
_mm512_permutexvar_epi64	13	20	18	8
_mm512_permutex2var_epi64	3	3	5	5
_mm512_alignr_epi64	-	-	-	6
Total	**50**	**63**	**58**	**67**

Table 3: Instruction count for the 512 bit multiplication versions

2.2.8 Performances

We now check the performances of all the previous approaches. The test procedure is presented Appendix E page 19. Since the SB512 version is the slowest one, we chose to provide its performances Table 14 page 20, in Appendix F.

We present Table 4 the results for the first three approaches described above, and compare them to the classic AVX2 implementations from the gf2x library [18] and the one from the HQC submission [2]. The main conclusions are as follows:

- among the AVX512 implementations, the best version is the one based on the karat_mult_1_512_SB, i.e. a schoolbook algorithm applied to the 128 bit and 256 bit levels, and the Karatsuba approach at the highest level (512 bit operands);
- in comparison with the AVX2 implementation, this version achieves a retired instruction number divided by roughly three, and speedups (in terms of clock cycle number) are from 29.5% to up to 39.1% (KaratRec, size 131072 bit), due to the lower IPC of the AVX512 instruction set.

The gf2x library performances are slightly lower than the ones of the other implementations. The results shown Table 4 allows to evaluate the *Instructions per cycles* (IPC), which is the ratio between the retired instructions and the clock cycle number. The graph in Figure 1 shows that while the IPC of the gf2x library multiplications is about 4.5, the IPC of the AVX2 multiplications is nearly 3.0 and the IPC of our AVX512 multiplications (with karat_mult_1_512_SB) is about 1.5. In the gf2x case, the software implementation makes use of AVX2 instruction set. However, this all-purpose library includes some wrappers and tests, and especially offers the possibility to tune each operand size. This is costly, but is written with conventional instructions. This is the most likely explanation of the high IPC, while the clock cycle number is a little worse that the one of the AVX2 software implementation.

We compare also the IPC of our AVX512 multiplication (with karat_mult_1_512_SB) and the one using the mul512 of Drucker *et al.* from [11]. In this last case, the IPC is greater than the one obtained with our implementation. While the clock cycle numbers of our implementations are about 9% lower than the ones using the mul512, Table 4 shows that the instruction

KaratRec		AVX2		Our impl. of [11]	**This work** AVX512 new 512 bit op. mult.	
size		gf2x[18]	after [2]	mul512	karat_mult_1_512	karat_mult_1_512_SB
1024	# clock cycles	339	183	**137**	143	**129**
	# instructions	1224	612	**254**	228	**193**
2048	# clock cycles	998	610	**461**	447	**423**
	# instructions	3892	1867	**872**	693	**581**
4096	# clock cycles	2949	1929	**1425**	1383	**1287**
	# instructions	11079	5684	**2701**	2219	**1821**
8192	# clock cycles	8742	6038	**4424**	4236	**3977**
	# instructions	33182	17991	**8600**	7051	**6128**
16384	# clock cycles	26128	18327	**13314**	12893	**12078**
	# instructions	100163	54840	**26099**	21573	**18797**
32768	# clock cycles	78889	59613	**40582**	39038	**36811**
	# instructions	295755	166410	**79078**	65466	**57525**
65536	# clock cycles	226640	187305	**126386**	121348	**114620**
	# instructions	853977	503014	**238851**	198185	**174468**
131072	# clock cycles	667900	572984	**382669**	369495	**348982**
	# instructions	2516857	1515625	**719423**	597021	**527205**

Table 4: Performance comparison for Algorithm 1

numbers are 24% to 33% lower. This may be explained by the intensive use of the `_mm512_clmulepi64_epi128` we have in our case. Due to this instruction high latency and throughput, the lesser instruction count does not yield such a large decrease for the clock cycle numbers. These results may also vary versus the processor version, according to the corresponding `_mm512_clmulepi64_epi128` instruction latency and throughput.

The conclusion of this comparison is that the `AVX512` multiplication implementation presents an elementary 64 bit multiplication cost relatively low in comparison with the `AVX2` situation. In this case, the non vectorized multiplication cost leads to a Karatsuba application at the 128 bit level. Insofar as the vectorized instructions equivalently divides by four the latency and the throughput of the elementary multiplication at the 64 bit level, it becomes more interesting to apply the schoolbook approach at the 128 bit level and also at the 256 bit level. In our tests, we saw that at the 512 bit level, the Karatsuba approach becomes again the best one.

On the hardware point of view, our tests show another aspect of this `AVX512` implementation case, which differs from the initial estimations given in [10], concerning the potential speedups brought by the `AVX512` instruction set and `VPCLMULQDQ` instruction. Indeed, these results make clear that, in terms of processor hardware, at the microarchitectural level, the `AVX512` instruction set has not yet been implemented with the same integration level as the one of the other instruction sets (conventional and `AVX2`), at least on our platform. The Intel documentation does not provide a lot of details [19], however, one can assume that the hardware features are not homogeneous with the `AVX2` equivalents. This also means that future Intel processor generations might improve the IPC of the `AVX512` instruction set and potentially decrease the clock cycle number of our implementation. Indeed, between our `AVX2` and `AVX512` multiplication implementation, the retired instruction count is divided by nearly three. If the processor manufacturer improves the IPC of the `AVX512` and `AVX2` instruction sets, to get closer to the one of the conventional instruction set IPC, this means that one may observe speedups with our software implementations on future platforms.

2.2.9 `Karat3` and `Karat5` implementations

These multiplications (Algorithms 2 and 3, Appendix A page 15) make use of the previous `KaratRec` multiplication (Algorithm 1) as elementary multiplications. The vectorized versions are again implemented using `AVX2` and `AVX512` instruction set. This leads to different sizes: while the `Karat3` multiplication has an operand size which is three times the one of the elemen-

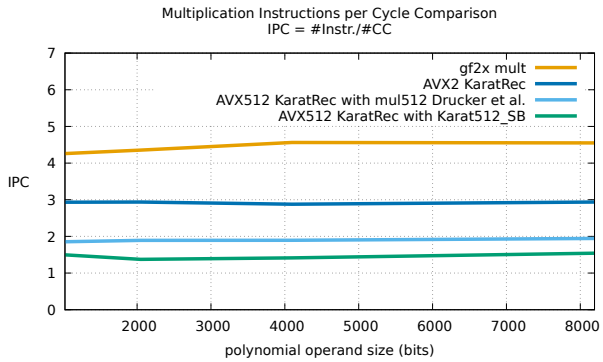

Fig. 1: Instructions per cycle comparison for Algorithm 1

	size	# clock cycles		
		gf2x	AVX2	AVX512
Karat3	1536	904	440	**302**
	3072	2486	1272	**871**
	6144	6104	3974	**2655**
	12288	18463	11998	**8118**
	24576	50311	37024	**24488**
	49152	150469	117707	**76147**
	98304	427281	357577	**231686**
Karat5	2560	1770	1135	**724**
	5120	4640	3378	**2251**
	10240	13814	10177	**6741**
	20480	40911	30545	**20486**
	40960	118334	97809	**63524**
	81920	323997	295147	**192378**

Table 5: Performance comparison for Algorithm 2 and 3

tary multiplication (for example 512,1024, 2048... bits), the Karat5 has an operand size which is five times the one of the elementary recursive Karatsuba multiplication. Thus, depending on the context, one may choose the most appropriate version according to the operand sizes. We present the performance results (# clock cycles) in Table 5.

The Karat3 or Karat5 multiplications can also use themselves as elementary multiplications. This leads to four extra combinations for multiplication. We present the performance results (# clock cycles) in Table 6. One may notice that the Karat3(Karat5), i.e. a Karat3 multiplication using a Karat5 elementary multiplication, deals with the same operand sizes that the Karat5(Karat3). However, we give the results for both versions, and observe that they present very close performances.

In all this experimentation, we only used, as the most elementary multiplication, the karat_mult_1_512_SB version, presented Section 2.2.5 page 6. The platform and experimentation process are the same as previously (see Appendix E page 19).

2.3 Conclusion

In this section, we presented our AVX512 implementation of recursive Karatsuba multiplication over $\mathbb{F}_2[X]$ for polynomial of degree at most 131071.

Our implementations show that the best AVX512 approach is a 512 bit kernel, using a schoolbook algorithm at the 128 and 256 bit level, and Karatsuba at the highest level, that is for the 512 bit size operands.

Used in AVX512 recursive Karatsuba multiplications of greater sizes, and in comparison with the AVX2 software implementation of [2], our implementations achieve a retired instruction number divided by roughly three, and speedups (in terms of clock cycle number) are from 29.5 % to up to 39.1 % (KaratRec, size 131072 bit), due to the lower IPC of the AVX512 instruction set.

The same achievements have been reached in all the Karatsuba variants (split in 3, 5 parts, and combinations).

3 Toom-Cook multiplication over $\mathbb{F}_2[X]$

In this section, we present the implementation issues of the Toom-Cook 3-5 approach applied to multiplication over $\mathbb{F}_2[X]$, especially with the AVX512 instruction set. We refer the reader to Appendix D page 18 for a general presentation of the Toom-Cook approach.

3.1 Toom-Cook multiplication complexity

There are several way to split the operands in the Toom-Cook approach:

- The Toom-Cook 3-5, which splits the operands in 3 parts, and involves 5 elementary multiplications;
- The Toom-Cook 4-7, which splits the operands in 4 parts, and involves 7 elementary multiplications;
- The Toom-Cook 5-9, which splits the operands in 5 parts, and involves 9 elementary multiplications;

We present Table 7 the general results versus the polynomial degrees, and the corresponding operand size in 64 bit words.

As $\log_2(3) > \log_3(5) > \log_4(7) > \log_5(9)$, this implies that the multiplication number is decreasing while

	size	# clock cycles		
		`gf2x`	`AVX2`	`AVX512`
Karat3(Karat3)	4608	4129	2707	**1920**
	9216	11261	8102	**5425**
	18432	34205	24498	**16340**
	36864	97586	76912	**51097**
	73728	270900	232362	**154065**
Karat5(Karat5)	12800	19495	17988	**11471**
	25600	56258	53132	**36417**
	51200	161740	159148	**107020**
	102400	438283	479256	**321361**

	size	# clock cycles		
		`gf2x`	`AVX2`	`AVX512`
Karat3(Karat5)	7680	8317	7075	**4496**
	15360	25110	20670	**13845**
	30720	73222	63386	**42454**
	61440	209872	192557	**127336**
	122880	618612	597357	**388414**
Karat5(Karat3)	7680	8242	7031	**4943**
	15360	26364	20586	**13775**
	30720	72897	63653	**42713**
	61440	216280	190317	**129419**
	122880	619802	580575	**387763**

Table 6: Performance comparison for recursive Algorithms 2 and 3

increasing the split. However, the hidden constant in the $\mathcal{O}$ notation is increasing, due to the more complex interpolation phase. This leads to a "gray zone" in which the different algorithms are very close to each other and this gray zone resides in our range for code based cryptographic protocols. In the `gf2x` implementation, a specific selection of the algorithm is made depending on each of the operand size (which can be different in this general purpose library).

3.2 Toom-Cook multiplication, implementation issues

Let us recall that in order to multiply two binary polynomials A and B of degree at most $N-1$, we consider them as polynomials of degree at most $64t-1$ where $t = 3n$ and n ensures $t \geqslant \lceil N/64 \rceil$. We now present how to choose n.

In the evaluation phase, the elementary products do not have the same operand size: $C(0), C(1)$ and $C(\infty)$ have operands of degree at most $64n-1$, while $C(x)$ and $C(x+1)$ have operands of degree at most $64n+2\cdot w-1$ (see Appendix D page 18). To take into account this characteristic, if the size of the elementary product is known, one has to set operands of size $n + 2w/64$ 64-bit words, padding with zeros in order to use the same elementary product. Now, we can specify the value of n mentioned Appendix D: n must be the minimum value ensuring $3n \geqslant \lceil N/64 \rceil$ such that $n + 2w/64$ is the size of the elementary multiplications computed during the evaluation phase. Thus, $n + 2w/64$ is either a power of 2 in case of a Karatsuba multiplication (as seen section 2.1), or a value which complies with another Toom-Cook multiplication.

In the interpolation phase, since the divisions by x and $(x+1)$ are exact, they can be implemented using the trick presented by Quercia and Zimmermann in [28] and [25]. One takes advantage of the size of the polynomial to replace these divisions by a one word right shift for the division by x, and by a special multiplication by $(x+1)^{-1} \bmod X^d$, $d >$ degree of $C(x)$ and $d \equiv 0 \bmod w$.

In the second case, to evaluate $(x+1)^{-1} \bmod X^d = (X^w+1)^{-1} \bmod X^d$, notice that:

$$(X^w+1)^{-1} \bmod X^n = \sum_{i=0}^{d/w-1} X^{w\cdot i}.$$

Example 1. *Here is a small toy example with polynomial over $\mathbb{F}_2[X]$: let us divide by $X+1$ polynomial whose degree is less than 8. One has $(X+1)^{-1} \bmod X^8 = X^7+X^6+X^5+X^4+X^3+X^2+X+1$. Now, $\forall P(X)$ such as $(X+1)|P(X)$ and degree(P)<8, $Q(X) = P(X)/(X+1)$ is computed as $P(X)\cdot(X+1)^{-1} \bmod X^8$. Since this division is exact, the result is the exact quotient.*
Let us set $P(X) = X^7+X^5+X^4+X$:

$$\begin{aligned} Q(X) &= (X^7+X^5+X^4+X)\cdot(X+1)^{-1} \bmod X^8 \\ &= (X^7+X^5+X^4+X) \\ &\quad \cdot(X^7+X^6+X^5+X^4+X^3+X^2+ \\ &\qquad X+1) \bmod X^8 \\ &= X^6+X^5+X^3+X^2+X. \end{aligned}$$

The vectorized implementation, while using 256-bit instructions, uses both values $w = 64$ or $w = 256$. We can even use $w = 512$ in the case of `AVX512` platforms. Notice that the division by $x+1$ is cheaper in case of $w = 256$ or 512, while n is slightly greater with $w = 64$.

To illustrate some Toom-Cook use cases, let us present some examples.

Example 2. *In order to use elementary Karatsuba multiplications, and with $w = 64$, let us consider n such that $n+2w/64 = 2^8$. Thus, one has $64\cdot n+2\cdot w = 8192$, and this elementary multiplication proceeds polynomials of degree at most 8191. We then have $n = 254$, $t = 3\cdot n = 762$, thus building a Toom-Cook multiplication which can multiply polynomials of degree at most $762\cdot 64-1 = 24191$.*

mult.	Complexity	size range (gf2x [18])	
		operand degrees	# 64 bit words
KaratRec	$\mathcal{O}(t^{\log_2(3)})$	$<$ 1343	$<$ 21
Toom-Cook 3-5	$\mathcal{O}(t^{\log_3(5)})$	1343 $<$ degrees $<$ 22143	21$<w<$346
Toom-Cook 4-7	$\mathcal{O}(t^{\log_4(7)})$	$>$ 22143	$>$ 346
Toom-Cook 5-9	$\mathcal{O}(t^{\log_5(9)})$	above	above

Table 7: Complexity of the Toom-Cook's multiplication variants

Example 3. *We now will use the previous multiplication in order to build a 1-recursive Toom-Cook multiplication, setting* $w = 256$*. One now chooses* n *such that* $64n + 2 \cdot 256 = 24192$*. This leads to* $n = 370$ *and* $t = 1110$*. This multiplication proceeds polynomials of degree at most* 71039*:*

- *the first split computes polynomials of degree at most* 23679*;*
- *we build the operands for the evaluation phase, whose size is up to* 24192*;*
- *we then call the Toom-Cook procedure of the previous example, which makes use of Karatsuba multiplications of size* 8192*.*

Example 4. *We now build a Toom-Cook multiplication, setting* $w = 512$ *using a* ***Karat5*** *elementary multiplication of size* 20480*. One now chooses* n *such that* $64n+2\cdot512 = 20480$*. This leads to* $n = 304$ *and* $t = 912$*. This multiplication proceeds polynomials of degree at most* 58368*. This multiplication fits with the hqc-256 protocol (see [2]), whose parameter* $N = 57669$*, and for an* ***AVX512*** *implementation.*

This can be adjusted for other sizes. To deal only with word shifts, this implies on the operand size that:

- elementary recursive Karatsuba multiplications are based on PCLMULQDQ or VPCLMULQDQ 64-bit elementary multiplications, whose size corresponds to the ones previously seen (KaratRec, Karat3, or Karat5...);
- the computation of $C(x)$ and $C(x + 1)$ needs construction of operands whose size has to be the elementary multiplication size;
- the split has to take into account the size of the elementary multiplication operand, by diminishing the size of the split by two words.

3.2.1 Toom3Mult implementations

Three Toom3Mult versions have been implemented (see Table 8):

- with KaratRec elementary multiplications whose size is among 512, 1024, 2048, 4096, 8192, 16384 and 32768 bits;
- with Karat3 elementary multiplications whose size is among 6144, 12288 and 32768 bits;
- with Karat5 elementary multiplications whose size is among 5120, 10240, 20480, 40960 bits.

The platform and test procedure are the ones described Appendix E page 19.

The speedup of our AVX512 implementation in comparison with the AVX2 one is as follows:

- Toom3 based on KaratRec (Toom3Mult(KaratRec)): the speedup starts from 33.5% for the 48768 bit operand size up to 36.7% for 97920 bit operand size;
- Toom3 based on Karat3 (Toom3Mult(Karat3)): the speedup starts from 30.4% for 36480 bit operand size up to 33.3% for the 73344 bit operand size;
- Toom3 based on Karat5 (Toom3Mult(Karat5)): the speedup starts from 32.4% for the 14976 bit operand size up to 33.9% for 122496 bit operand size.

3.3 Toom3 vs Karat3 comparison

The Toom-Cook multiplication presented above (Toom3) and the Karat3 multiplication (see Algorithm 2 page 15) look similar, both splitting the operands in three parts. However, the Toom3 needs 5 elementary multiplications while the Karat3 requires 6. In the gf2x library [18], the Toom-3 multiplication is used above the threshold of 21 64-bit words, i.e. polynomial of degree 1343.

In our AVX2 and AVX512 implementations, we compare the clock cycle numbers of multiplications using the same elementary Karatsuba multiplication. Consequently, the operand size in the Toom3 case is slightly lower. In Table 9, the clock cycle numbers of the Toom3 multiplication are lower for the considered sizes, and the speedup starts from 8% for the smallest (about 6000 bit operand size) up to 11.7% for the bigger sizes, the maximum potential speedup being theoretically 16.7%. Indeed, the costliest interpolation and reconstruction phase of the Toom3 approach lowers the speedup for the smallest sizes.

	size	# clock cycles		
		gf2x	AVX2	AVX512
Toom3Mult(KaratRec)	24192	47200	33501	**22238**
	48768	144960	104257	**69328**
	97920	427101	323719	**204596**
Toom3Mult(Karat3)	18048	34019	22032	**14872**
	36480	94378	67451	**46940**
	73344	272891	206601	**137791**
Toom3Mult(Karat5)	14976	25146	18570	**12664**
	30336	77548	57339	**38668**
	61056	211790	170316	**113783**
	122496	622830	513595	**339387**

Table 8: Performance comparison between AVX2 and AVX512 Toom3 multiplications, with various elementary Karatsuba multiplications

multiplication size			# clock cycles AVX2		AVX512	
Elt. Karat. size	Karat3	Toom3	Karat3	Toom3	Karat3	Toom3
2048	6144	5760	4150	**3987**	2655	-
3072	9216	8832	8405	**7726**	5425	-
4096	12288	11998	12508	**11480**	8118	-
6144	18432	18048	25264	**22032**	16340	**14872**
8192	24576	24192	37024	**33501**	24488	**22238**
12288	36864	36480	79727	**67451**	51097	**46940**
16384	49152	48768	117707	**104257**	76147	**69328**
24576	73728	73344	241548	**206601**	154065	**137791**
32768	98304	97920	357577	**323719**	231686	**204596**

Table 9: Performance comparison between AVX2 and AVX512 Toom3Mult and Karat3 multiplications

3.4 Conclusion

In this section, we presented our AVX512 implementation of Toom-Cook multiplication over $\mathbb{F}_2[X]$ for polynomial of degree at most 122496, based on various Karatsuba version elementary AVX512 multiplications (KaratRec, Karat3 and Karat5).

The speedup between AVX2 and our AVX512 implementation is again up to nearly 37% (97920 bit size, Toom3Mult(KaratRec)), as it has already been observed with our AVX512 Karatsuba multiplications. Thus, the same remark can be done about the potential in terms of future results.

4 HQC multiplications

In this section, we present the application of our AVX2 and AVX512 Toom-Cook multiplications in the context of the HQC protocol.

The sizes of the HQC multiplications are (see [2]):

- hqc-128 : PARAM_N = 17669;
- hqc-192 : PARAM_N = 35851;
- hqc-256 : PARAM_N = 57637.

In the NIST submission [2, updated submission package (round 3), 2020/10/01], the hqc-128 and the hqc-192 implementations make use of the Karat3 multiplication based on elementary Karat3 multiplications, while the hqc-256 is a Toom-Cook multiplication (3 part operand split) based on a Karat5 elementary multiplication. The reader may notice that the operand size is different in the hqc-256 case, in comparison with the Toom3Mult(Karat5) version presented in Table 8 page 12, whose operand size is 61056 bits, while the AVX2 operand size is 59904 bits, and the AVX512 one is 58368 bits, see Table 10. This is due to the word size considered in the Toom-Cook implementation: the first version of table 8 has a word size $w = 64$ bits, while the

AVX2 version has $w = 256$ and we chose $w = 512$ for the AVX512 version. The new sizes fit the hqc-256 PARAM_N = 57637, allowing a slightly better performance.

The platform and test procedure are the ones described Appendix E page 19. We kept the compiler flags used in the NIST submission [2, updated submission package (round 3), 2020/10/01]:
`-O3 -funroll-all-loops -flto -march=tigerlake` for AVX512 versions;
`-O3 -funroll-all-loops -flto -mavx -mavx2 -mbmi -mpclmul` for AVX2 versions.

In Table 10, we sum-up the performances of the respective multiplications, from Tables 8 page 12 and 9 page 12, along with the specific hqc-256 multiplications results, in the context of HQC protocol.

4.1 HQC implementation performances

This leads to the performances and also the speedups brought by our work in comparison with the NIST release [2, updated submission package (round 3), 2020/10/01], provided in Table 11.

As a consequence of this work, the AVX2 Toom3 multiplication based on Karat3 is now part of the last release of HQC (2021/06/06).

Table 12 provides the HQC performances when using our AVX512 multiplications in the 2021/06/06 release. One may notice that column one of this Table is different of the corresponding one of Table 11 because other parts of the HQC source code has been updated between both releases.

A maximum 11.8% speedup for the HQC implementation is reached with the hqc-256 Keygen.

5 Conclusion

In this paper, we considered the software AVX512 implementation of polynomial multiplication over $\mathbb{F}_2[X]$, using the vectorized 64-bit polynomial multiplication instruction VPCLMULQDQ. We studied the different combinations of schoolbook/Karatsuba constructions for the kernels up to 512 bit operands. We then implemented two different approaches: one based on the Karatsuba subquadratic approach and the other on the Toom-Cook approach. These implementations are competitive in comparison with state-of-the art general purpose library, HQC submissions, and other AVX512 software implementations of [10,11]. While the retired instruction count is divided by roughly three compared to the corresponding AVX2 implementations, we achieved a speedup up to nearly 40%, in terms of clock cycle numbers.

We implemented our approaches in the HQC protocol by patching the NIST submission released in october 2020, in order to experiment the potential benefits, and this leads to speedups up to 11.8% (hqc-256 Keygen.).

All the implementations of this work are available on github[1].

Funding This work has been partially funded by TPM Metropol (AAP2020-IPOCRAS project).

References

1. Ntl: a library for doing number theory. https://libntl.org, last accessed 21 Sep 2021.
2. Carlos Aguilar-Melchior, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Jean-Marc Robert, Pascal Véron, and Gilles Zémor. Hamming Quasi-Cyclic (HQC). In *NIST Post-Quantum Cryptography submissions, round 3*. NIST, october 2020. http://pqc-hqc.org/, last accessed 15 Sep 2021.
3. Carlos Aguilar-Melchior, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, and Gilles Zémor. Hamming Quasi-Cyclic (HQC). In *NIST Post-Quantum Cryptography submissions, round 2*. NIST, 2019. http://pqc-hqc.org/implementation.html, last accessed 15 Sep 2021.
4. Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status Report on the First Round of the NIST PQC Standardization Process, 2019. https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8240.pdf, last accessed 16 Sep 2021.
5. Nicolas Aragon, Slim Bettaieb, Paulo S.L.M. Barreto, Loïc Bidoux, Olivier Blazy, Jean-Christophe DeneuvillePhilippe Gaborit, Philippe Gaborit, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchior, Rafael Misocki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, and Gilles Zémor. Bit Flipping Key Encapsulation (BIKE). In *NIST Post-Quantum Cryptography submissions, round 2*. NIST, 2019. https://bikesuite.org/, last accessed 15 Sep 2021.
6. Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. A finite regime analysis of information set decoding algorithms. *Algorithms*, 12(10):209, 2019. doi:10.3390/a12100209.
7. Elwyn R. Berlekamp, Robert J. McEliece, and Henk C. A. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Trans. Inf. Theory*, 24(3):384–386, 1978. doi:10.1109/TIT.1978.1055873.
8. Marco Bodrato. Towards optimal Toom-Cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In Claude Carlet and Berk Sunar, editors, *WAIFI'07 proceedings*, volume 4547 of *LNCS*, pages 116–133. Springer, June 2007. doi:10.1007/978-3-540-73074-3_10.
9. Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster multiplication in gf(2)[x]. In *Algorithmic Number Theory, 8th International Symposium, ANTS-VIII, Banff, Canada, May 17-22, 2008, Proceedings*, pages 153–166, 2008. doi:10.1007/978-3-540-79456-1_10.

[1] https://github.com/arithcrypto/AVX512PolynomialMultiplication

		This work	
	NIST release [2]	improved `AVX2`	`AVX512`
`hqc-128`	Karat9 ($N = 18432$)	Toom3Karat3 ($N = 18048, w = 64$)	
`PARAM_N` $= 17669$	24498	22032	**14872**
`hqc-192`	Karat9 ($N = 36864$)	Toom3Karat3 ($N = 36480, w = 64$)	
`PARAM_N` $= 35851$	76912	67451	**46940**
`hqc-256`	Toom3Karat5 ($N = 59904, w = 256$)		Toom3Karat5 ($N = 58368, w = 512$)
`PARAM_N` $= 57637$	168975		**110568**

Table 10: `AVX2` and `AVX512` multiplication version performances, clock cycles numbers, in the context of the HQC protocol

# clock cycles		NIST release 2020/10/01	**This work** improved `AVX2`
`hqc-128`	Keygen	111073	110009 (-1.0 %)
	Encaps	185741	181154 (-2.4 %)
	Decaps	344154	337594 (-2.0 %)
`hqc-192`	Keygen	250184	239908 (-5.3 %)
	Encaps	430689	410090 (-4.8 %)
	Decaps	722899	696779 (-3.6 %)

Table 11: HQC performances, `AVX2` clock cycles numbers

# clock cycles		NIST release 2021/06/06	**This work** `AVX512`
`hqc-128`	Keygen	70171	64825 (-7.6 %)
	Encaps	1723219	158377 (-8.1 %)
	Decaps	311434	300661 (-3.5 %)
`hqc-192`	Keygen	168397	154486 (-8.3 %)
	Encaps	395367	361838 (-8.5 %)
	Decaps	646313	617853 (-4.4 %)
`hqc-256`	Keygen	338137	298331 (-11.8 %)
	Encaps	768537	680602 (-11.4 %)
	Decaps	1290132	1194526 (-7.4 %)

Table 12: HQC performances, `AVX512` clock cycles numbers

10. 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.
11. Nir Drucker, Shay Gueron, and Dusan Kostic. Fast polynomial inversion for post quantum QC-MDPC cryptography. In Shlomi Dolev, Vladimir Kolesnikov, Sachin Lodha, and Gera Weiss, editors, *Cyber Security Cryptography and Machine Learning - Fourth International Symposium, CSCML 2020, Be'er Sheva, Israel, July 2-3, 2020, Proceedings*, volume 12161 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2020. doi:10.1007/978-3-030-49785-9_8.
12. Martin Fürer. Faster integer multiplication. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 57–66, 2007. doi:10.1145/1250790.1250800.
13. Philippe Gaborit. Shorter keys for code-based cryptography. In *Proceedings of Workshop on Codes and Cryptography*, pages 81–90, France, 2005. WCC 2005.
14. Philippe Gaborit and Marc Girault. Lightweight code-based identification and signature. In *IEEE International Symposium on Information Theory, ISIT 2007, Nice, France, June 24-29, 2007*, pages 191–195. IEEE, 2007. doi:10.1109/ISIT.2007.4557225.

15. Antonio Guimarães, Diego Aranha, and Edson Borin. Secure and efficient software implementation of qc-mdpc code-based cryptography. In *XX Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 116–117, 11 2019. doi:10.5753/wscad_estendido.2019.8710.
16. David Harvey and Joris van der Hoeven. Faster polynomial multiplication over finite fields using cyclotomic coefficient rings. *J. Complexity*, 54, 2019. doi:10.1016/j.jco.2019.03.004.
17. David Harvey, Joris van der Hoeven, and Grégoire Lecerf. Faster polynomial multiplication over finite fields. *J. ACM*, 63(6):52:1–52:23, 2017. doi:10.1145/3005344.
18. Inria. gf2x library. In *gf2x Library*, 2019. https://www.gforge.inria.fr/frs/?group_id=1874, last accessed 15 Sep 2021.
19. Intel. Intel® 64 and ia-32 architectures software developer manuals. Intel website, 2021. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html, last accessed 15 Sep 2021.
20. A. Karatsuba and Yu Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akad. Nauk SSSR*, volume 145, pages 293–294, 1962.
21. F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error Correcting Codes*. Number ptie. 2 in Mathematical Library. North-Holland Publishing Company, 1977.
22. R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, January 1978.
23. Christophe Nègre and Jean-Marc Robert. Impact of Optimized Field Operations AB, AC and $AB + CD$ in Scalar Multiplication over Binary Elliptic Curve. In *Progress in Cryptology - AFRICACRYPT, 6th International Conference on Cryptology in Africa, June 22-24.*, LNCS, pages 279–296, 2013. doi:10.1007/978-3-642-38553-7_16.
24. NIST. Post-Quantum Cryptography, 2019. https://csrc.nist.gov/projects/post-quantum-cryptography, last accessed 15 Sep 2021.
25. Michel Quercia and Paul Zimmermann. Irred-ntl patch. In *Irred-ntl source code*, 2003. https://members.loria.fr/PZimmermann/irred/.
26. Arnold Schönhage and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971. doi:10.1007/BF02242355.
27. André Weimerskirch and Christof Paar. Generalizations of the karatsuba algorithm for efficient implementations. Cryptology ePrint Archive, Report 2006/224, 2006. https://eprint.iacr.org/2006/224, last accessed 15 Sep 2021.
28. Paul Zimmermann. Irred-ntl patch. In *ntl Library*, 2008. https://members.loria.fr/PZimmermann/irred/.

Appendix A Karatsuba algorithms

We reproduce here the Karatsuba algorithms:

- Algorithm 1 reproduces the recursive multiplication with a two halves split, from [23];
- Algorithm 2 shows the three parts split corresponding approach;
- Algorithm 3 shows the five parts split corresponding approach;

Algorithm 1 KaratRec(A,B,t), from [23]

Require: A and B on $t = 2^r$ computer words.
Ensure: $R = A \times B$
if $t = 1$ **then**
  return ($Mult64(A, B)$)
else
  // Split in two halves of word size $t/2$.
  $A = A_0 + x^{64t/2} A_1$
  $B = B_0 + x^{64t/2} B_1$
  // Recursive multiplications
  $R_0 \leftarrow \texttt{KaratRec}(A_0, B_0, t/2)$
  $R_1 \leftarrow \texttt{KaratRec}(A_1, B_1, t/2)$
  $R_2 \leftarrow \texttt{KaratRec}(A_0 + A_1, B_0 + B_1, t/2)$
  // Reconstruction
  $R \leftarrow R_0 + (R_0 + R_1 + R_2)X^{64t/2} + R_1 X^{64t}$
  return (R)

Algorithm 2 Karat3(A,B,t), from [27]

Require: A and B on $t = 3 \times 2^r$ computer words.
Ensure: $R = A \times B$
// Split in three thirds of word size $t/3$.
$A = A_0 + x^{64t/3} A_1 + x^{2\times 64t/3} A_2$
$B = B_0 + x^{64t/3} B_1 + x^{2\times 64t/3} B_2$
// Recursive multiplications
$R_0 \leftarrow \texttt{KaratRec}(A_0, B_0, t/3)$
$R_1 \leftarrow \texttt{KaratRec}(A_1, B_1, t/3)$
$R_2 \leftarrow \texttt{KaratRec}(A_2, B_2, t/3)$
$R_3 \leftarrow \texttt{KaratRec}(A_0 + A_1, B_0 + B_1, t/3)$
$R_4 \leftarrow \texttt{KaratRec}(A_0 + A_2, B_0 + B_2, t/3)$
$R_5 \leftarrow \texttt{KaratRec}(A_1 + A_2, B_1 + B_2, t/3)$
// Reconstruction
$R \leftarrow R_0 + (R_0 + R_1 + R_3)X^{64t/3} + (R_0 + R_1 + R_2 + R_4)X^{2\times 64t/3} + (R_1 + R_2 + R_5)X^{64t} + R_2 X^{4\times 64t/3}$
return (R)

Appendix B Source code for 256-bit operand size multiplication

B.1 Source code for the 4×4 256 bit multiplication of this work based on the schoolbook approach

We present here our schoolbook AVX512 implementation of the 256 bit operand size multiplication, with comments and explanations.

```
__m512i mask_middle= (__m512i){0x0UL,
            0xffffffffffffffffUL ,
            0xffffffffffffffffUL ,0x0UL,0x0UL,
            0xffffffffffffffffUL ,
            0xffffffffffffffffUL ,0x0UL};

__m512i idx_b=(__m512i){0x0UL,0x1UL,0x2UL,0x3UL
                ,0x2UL,0x3UL,0x0UL,0x1UL};
```

Algorithm 3 `Karat5`(A,B,t), from [27]

Require: A and B on $t = 5 \times 2^r$ computer words.
Ensure: $R = A \times B$

1: // Split in five parts of word size $t/5$.
2: $A = A_0 + x^{64t/5}A_1 + x^{2\times 64t/5}A_2 + x^{3\times 64t/5}A_3 + x^{4\times 64t/5}A_4$
3: $B = B_0 + x^{64t/5}B_1 + x^{2\times 64t/5}B_2 + x^{3\times 64t/5}B_3 + x^{4\times 64t/5}B_4$
4: // Recursive multiplications
5: $R_0 \leftarrow \texttt{KaratRec}(A_0, B_0, t/5)$
6: $R_1 \leftarrow \texttt{KaratRec}(A_1, B_1, t/5)$
7: $R_2 \leftarrow \texttt{KaratRec}(A_2, B_2, t/5)$
8: $R_3 \leftarrow \texttt{KaratRec}(A_3, B_3, t/5)$
9: $R_4 \leftarrow \texttt{KaratRec}(A_4, B_4, t/5)$
10: $R_{01} \leftarrow \texttt{KaratRec}(A_0 + A_1, B_0 + B_1, t/5)$
11: $R_{02} \leftarrow \texttt{KaratRec}(A_0 + A_2, B_0 + B_2, t/5)$
12: $R_{03} \leftarrow \texttt{KaratRec}(A_0 + A_3, B_0 + B_3, t/5)$
13: $R_{04} \leftarrow \texttt{KaratRec}(A_0 + A_4, B_0 + B_4, t/5)$
14: $R_{12} \leftarrow \texttt{KaratRec}(A_1 + A_2, B_1 + B_2, t/5)$
15: $R_{13} \leftarrow \texttt{KaratRec}(A_1 + A_3, B_1 + B_3, t/5)$
16: $R_{14} \leftarrow \texttt{KaratRec}(A_1 + A_4, B_1 + B_4, t/5)$
17: $R_{23} \leftarrow \texttt{KaratRec}(A_2 + A_3, B_2 + B_3, t/5)$
18: $R_{24} \leftarrow \texttt{KaratRec}(A_2 + A_4, B_2 + B_4, t/5)$
19: $R_{34} \leftarrow \texttt{KaratRec}(A_3 + A_4, B_3 + B_4, t/5)$
20: // Reconstruction
21: $R \leftarrow R_0 + (R_0 + R_1 + R_{01})X^{64t/5} + (R_0 + R_1 + R_2 + R_{02})X^{2\times 64t/5} + (R_0 + R_1 + R_2 + R_3 + R_{03} + R_{12})X^{3\times 64t/5} + (R_0 + R_1 + R_2 + R_3 + R_4 + R_{04} + R_{13})X^{4\times 64t/5} + (R_1 + R_2 + R_3 + R_4 + R_{14} + R_{23})X^{64t} + (R_3 + R_2 + R_4 + R_{24})X^{6\times 64t/5} + (R_3 + R_4 + R_{34})X^{7\times 64t/5} + R_4X^{8\times 64t/5}$
22: **return** (R)

```
__m512i idx_1=(__m512i){0x0UL,0x1UL,0x8UL,0x9UL,
                        0x2UL,0x3UL,0xaUL,0xbUL};
__m512i idx_2=(__m512i){0x0UL,0x1UL,0x6UL,0x7UL,
                        0x2UL,0x3UL,0x4UL,0x5UL};
__m512i idx_3=(__m512i){0x0UL,0x1UL,0x4UL,0x5UL,
                        0x2UL,0x3UL,0x6UL,0x7UL};
__m512i idx_4=(__m512i){0x8UL,0x0UL,0x1UL,0x2UL,
                        0x3UL,0x4UL,0x5UL,0x8UL};
__m512i idx_5=(__m512i){0x8UL,0x8UL,0x8UL,0x6UL,
                        0x7UL,0x8UL,0x8UL,0x8UL};
__m512i idx_6=(__m512i){0x0UL,0x0UL,0x4UL,0x5UL,
                        0xcUL,0xdUL,0x0UL,0x0UL};
__m512i idx_7=(__m512i){0x0UL,0x0UL,0x6UL,0x7UL,
                        0xeUL,0xfUL,0x0UL,0x0UL};
```

These lines define the constant indexes for the `_mm512_permutexvar_epi64` and `_mm512_permutex2var_epi64` instructions. These instructions are explained on the fly.

```
void mult_256_256_512(__m512i * Out,
            const __m256i * A256,
            onst __m256i * B256)
{

__m512i A512, B512 ;
__m512i R0_512,R1_512,R2_512, R3_512,
        middle, tmp;

A512 =_mm512_broadcast_i64x4(*A256);
tmp  =_mm512_broadcast_i64x4(*B256);
B512 =_mm512_permutexvar_epi64 (idx_b, tmp);
```

The `_mm512_broadcast_i64x4(*A256)` instruction duplicates the 256 bits of `*A256` in the `A512` register, the same for the `*B256`.

The `_mm512_permutexvar_epi64 (idx_b, tmp)` spreads the 64 bit words following the index `idx_b`. This allows to shuffle the 64 bit words of `*B256` in the `B512` register, in order to prepare the elementary multiplications.

We thus have:

`A512` $\leftarrow \{a_3, a_2, a_1, a_0, a_3, a_2, a_1, a_0\}$
`tmp` $\leftarrow \{b_3, b_2, b_1, b_0, b_3, b_2, b_1, b_0\}$
`B512` $\leftarrow \{b_1, b_0, b_3, b_2, b_3, b_2, b_1, b_0\}$

```
R0_512=_mm512_clmulepi64_epi128(A512,B512,0x00);
R1_512=_mm512_clmulepi64_epi128(A512,B512,0x10);
R2_512=_mm512_clmulepi64_epi128(A512,B512,0x01);
R3_512=_mm512_clmulepi64_epi128(A512,B512,0x11);
```

We now compute all the elementary 64 bit multiplications, providing all the 128 bit results as follows:

`R0` $\leftarrow \{a_2 \times b_0, a_0 \times b_2, a_2 \times b_2, a_0 \times b_0\}$
`R1` $\leftarrow \{a_2 \times b_1, a_0 \times b_3, a_2 \times b_3, a_0 \times b_1\}$
`R2` $\leftarrow \{a_3 \times b_0, a_1 \times b_2, a_3 \times b_2, a_1 \times b_0\}$
`R3` $\leftarrow \{\underbrace{a_3 \times b_1}_{128bits}, \underbrace{a_1 \times b_3}_{128bits}, \underbrace{a_3 \times b_3}_{128bits}, \underbrace{a_1 \times b_1}_{128bits}\}$

```
tmp = _mm512_permutex2var_epi64
      (R0_512,idx_1,R3_512);
```

The `tmp` register now contains all the $a_i \times b_i$ elementary products coming form the `R0_512` and `R3_512` registers:

`tmp` $\leftarrow \{a_3 \times b_3, a_2 \times b_2, a_1 \times b_1, a_0 \times b_0\}$

It remains now to compute the middle part of the result to be added to `tmp`, in order to get the final result.

```
middle = _mm512_permutexvar_epi64(idx_2, R1_512);
middle ^=_mm512_permutexvar_epi64(idx_3, R2_512);
```

The `middle` register now contains the addition (XOR) between `R1_512` and `R2_512`, reordered with the `_mm512_permutexvar_epi64`:

`middle` $\leftarrow \{a_0b_3 \oplus a_3b_0, a_2b_3 \oplus a_3b_2,$
$a_1b_2 \oplus a_2b_1, a_1b_0 \oplus a_0b_1\}$

```
tmp ^= _mm512_permutex2var_epi64
       (middle,idx_4,idx_b);
tmp ^= _mm512_permutex2var_epi64
       (middle,idx_5,idx_b);
```

The tmp register is added (XOR) with the elementary products of the middle register, and nearly contains the result, except some of the products of the middle part:

$$\texttt{tmp} \leftarrow \begin{array}{c|c|c|c} a_3\times b_3 & a_2\times b_2 & a_1\times b_1 & a_0\times b_0 \\ \oplus & a_2b_3\oplus a_3b_2 & a_1b_2\oplus a_2b_1 & a_1b_0\oplus a_0b_1 \\ \oplus & & a_0b_3\oplus a_3b_0 & \end{array}$$

```
middle = _mm512_permutex2var_epi64(R0_512,
                                   idx_6,R3_512);
middle ^= _mm512_permutex2var_epi64(R0_512,
                                    idx_7,R3_512);
```

The remaining products of the middle part to be added with tmp are put in place in the middle register:

$$\texttt{middle} \leftarrow \{\underbrace{0x0UL}_{128bits}, a_1b_3\oplus a_3b_1, a_0b_2\oplus a_2b_0, \underbrace{0x0UL}_{128bits}\}$$

```
        *Out = tmp^middle;
}
```

Out gets the final reconstruction :

$$\texttt{Out} \leftarrow \begin{array}{c|c|c|c} a_3\times b_3 & a_2\times b_2 & a_1\times b_1 & a_0\times b_0 \\ \oplus & a_2b_3\oplus a_3b_2 & a_1b_2\oplus a_2b_1 & a_1b_0\oplus a_0b_1 \\ \oplus & & a_1b_3\oplus a_3b_1 & a_0b_2\oplus a_2b_0 \\ \oplus & & a_0b_3\oplus a_3b_0 & \end{array}$$

Appendix C Source code for 512-bit operand size multiplications

We detail now the source code of the karat_mult_1_512_SB procedure.

First, the preamble declares the constant indexes for the _mm512_permutex2var_epi64 and _mm512_permutexvar_epi64 instructions.

```
inline static void karat_mult_1_512(__m512i * C,
        const __m512i * A, const __m512i * B)
{
 const __m512i perm_al = (__m512i){0x0UL,0x1UL,
                                   0x0UL,0x1UL,
                                   0x2UL,0x3UL,
                                   0x2UL,0x3UL};
 const __m512i perm_ah = (__m512i){0x4UL,0x5UL,
                                   0x4UL,0x5UL,
                                   0x6UL,0x7UL,
                                   0x6UL,0x7UL};
 const __m512i perm_bl = (__m512i){0x0UL,0x1UL,
                                   0x2UL,0x3UL,
                                   0x0UL,0x1UL,
                                   0x2UL,0x3UL};
 const __m512i perm_bh = (__m512i){0x4UL,0x5UL,
                                   0x6UL,0x7UL,
                                   0x4UL,0x5UL,
                                   0x6UL,0x7UL};
 const __m512i mask_R1 = _mm512_set_epi64
                          (6,7,4,5,2,3,0,1);
 const __m512i perm_h = (__m512i){0x4UL,0x5UL,
                                   0x0UL,0x1UL,
                                   0x2UL,0x3UL,
                                   0x6UL,0x7UL};
 const __m512i perm_l = (__m512i){0x0UL,0x1UL,
                                   0x4UL,0x5UL,
                                   0x6UL,0x7UL,
                                   0x2UL,0x3UL};
 const __m512i mask = _mm512_set_epi64
                       (15,14,13,12,3,2,1,0);
```

Next, we compute the registers al, ah, bl, bh, sa and sb so that they contain the split parts for the 256 bit operands, and the corresponding sums for the Karatsuba 256 bit middle multiplication.

```
__m512i al = _mm512_permutexvar_epi64
                          (perm_al, *A );
__m512i ah = _mm512_permutexvar_epi64
                          (perm_ah, *A );
__m512i bl = _mm512_permutexvar_epi64
                          (perm_bl, *B );
__m512i bh = _mm512_permutexvar_epi64
                          (perm_bh, *B );

__m512i sa = al^ah;
__m512i sb = bl^bh;
```

We compute now the three 256 bit multiplications in order to prepare the 512 bit registers cl, ch and cm containing their results.

```
// first multiplication 256 : AlBl

__m512i R0_512=_mm512_clmulepi64_epi128
                          (al,bl,0x00);
__m512i R1_512=_mm512_clmulepi64_epi128
                          (al,bl,0x01);
__m512i R2_512=_mm512_clmulepi64_epi128
                          (al,bl,0x10);
__m512i R3_512=_mm512_clmulepi64_epi128
                          (al,bl,0x11);

R1_512 = _mm512_permutexvar_epi64
              ( mask_R1 , R1_512^R2_512 ) ;

__m512i l =  _mm512_mask_xor_epi64( R0_512 ,
                          0xaa , R0_512 , R1_512 ) ;
__m512i h =  _mm512_mask_xor_epi64( R3_512 ,
                          0x55 , R3_512 , R1_512 ) ;
```

These lines computes four 128 bit operand size multiplications in parallel, using a schoolbook approach. This procedure acts like the mul128x4 procedure of Drucker *et al.* [11], which is based on the Karatsuba algorithm[2].

The 512 bit registers l and h now contains the four elementary 256 bit results.

```
__m512i cl = _mm512_permutex2var_epi64(l,mask,h);
l = _mm512_permutexvar_epi64(perm_l, l );
h = _mm512_permutexvar_epi64(perm_h, h );

__m512i middle = _mm512_maskz_xor_epi64(0x3c,h,l);

cl ^= middle;
```

This is the schoolbook reconstruction for the first 256 bit multiplication. The register cl now contains the 512 bit result of al×bl.

We now compute the same the two remaining 256 bit operand size multiplications:

```
// second multiplication 256 : AhBh
...
ch ^= middle;
```

[2] We do not present in detail our variant karat_mult_1_512 based on the mul128x4, however, we refer the reader to their paper [11] for its presentation.

```
// third multiplication 256 : SASB
...
cm ^= middle^cl^ch;
```

The register `ch` now contains the 512 bit result of `ah`×`bh`.

The result `cm` is directly added (XOR) to the other results `cl` and `ch` in order to prepare the final Karatsuba reconstruction, and `cm` now contains the 512 bit result of `(sa`×`sa)`⊕`cl`⊕`ch`:

```
// final reconstruction (Karatsuba)

const __m512i perm_cm = (__m512i){0x4UL,0x5UL,
          0x6UL,0x7UL,0x0UL,0x1UL,0x2UL,0x3UL};

cm = _mm512_permutexvar_epi64(perm_cm, cm );

C[0]= _mm512_mask_xor_epi64(cl,0xf0,cl,cm);
C[1]= _mm512_mask_xor_epi64(ch,0x0f,ch,cm);
}
```

This ends the computation, the final lines stores the result: the 512 least significant bits in the memory place `C[0]`, and the most significant bits in `C[1]`.

Appendix D Toom-Cook multiplication general algorithm

Several approaches to multiply two arbitrary polynomials over $\mathbb{F}_2[X]$ of degree at most $N-1$, using the Toom-Cook algorithm, have been presented by Bodrato in [8], Brent *et al.* in [9], and software implementations have been provided by Quercia and Zimmermann, in the context of the `ntl` and the `gf2x` library, see [28] and [25]. Let A and B be two binary polynomials of degree at most $N-1$. These polynomials are packed into an array of 64-bit words, whose size is $\lceil N/64 \rceil$. Let $t = 3n$ with n a value ensuring $t \geqslant \lceil N/64 \rceil$. Now, A and B are considered as polynomials of degree at most $64 \cdot t - 1$. We discuss the value of n in section 3.2.

A and B are split in three parts. One wants now to evaluate the result $C = A \cdot B$ with

$$A = a_0 + a_1 \cdot X^{64n} + a_2 \cdot X^{2 \cdot 64n} \in \mathbb{F}_2[X],$$

$$B = b_0 + b_1 \cdot X^{64n} + b_2 \cdot X^{2 \cdot 64n} \in \mathbb{F}_2[X],$$

(of maximum degree $64t-1$, and a_i, b_i of maximum degree $64n-1$) and,

$$C = c_0 + c_1 \cdot X^{64n} + c_2 \cdot X^{2 \cdot 64n} + c_3 \cdot X^{3 \cdot 64n} + c_4 \cdot X^{4 \cdot 64n}$$

of maximum degree $6 \cdot 64n - 2$.

The "word-aligned" version evaluates the polynomial for the values 0, 1, $x = X^w$, $x+1 = X^w + 1$, ∞, w being the word size, typically 64 in modern processors. Furthermore, on Intel processors, one can set $w = 256$ to take advantage of the vectorized instruction set `AVX-AVX2`, and even $w = 512$ (`AVX512` extension), at the cost of a slight operand size reduction.
For the evaluation phase, one has:

$$\begin{aligned}
C(0) &= a_0 \cdot b_0 \\
C(1) &= (a_0 + a_1 + a_2) \cdot (b_0 + b_1 + b_2) \\
C(x) &= (a_0 + a_1 \cdot x + a_2 \cdot x^2) \cdot (b_0 + b_1 \cdot x + b_2 \cdot x^2) \\
C(x+1) &= (a_0 + a_1 \cdot (x+1) + a_2 \cdot (x^2+1)) \cdot \\
&\quad (b_0 + b_1 \cdot (x+1) + b_2 \cdot (x^2+1)) \\
C(\infty) &= a_2 \cdot b_2
\end{aligned}$$

The implementation of this phase is straightforward, providing that the multiplication $a_i \cdot b_i$ is either another Toom-Cook or Karatsuba multiplication. Notice that the multiplications by x or x^2 are virtually free word shifts.
For the interpolation phase, one has the following equations:

$$\begin{aligned}
C(0) &= c_0 \\
C(1) &= c_0 + c_1 + c_2 + c_3 + c_4 \\
C(x) &= c_0 + c_1 \cdot x + c_2 \cdot x^2 + c_3 \cdot x^3 + c_4 \cdot x^4 \\
C(x+1) &= c_0 + c_1 \cdot (x+1) + c_2 \cdot (x^2+1) \\
&\quad + c_3 \cdot (x^3 + x^2 + x + 1) + c_4 \cdot (x^4 + 1) \\
C(\infty) &= c_4
\end{aligned}$$

The matrix associated to this system of equations is given by:

$$M = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
1 & x & x^2 & x^3 & x^4 \\
1 & x+1 & x^2+1 & x^3+x^2+x+1 & x^4+1 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

and one has :

$$M^{-1} = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
\frac{(x^2+x+1)}{(x^2+x)} & 1 & 1/x & \frac{1}{x+1} & x^2+x \\
0 & \frac{1}{x^2+x} & \frac{1}{x+1} & 1/x & x^2+x+1 \\
\frac{1}{x^2+x} & \frac{1}{x^2+x} & \frac{1}{x^2+x} & \frac{1}{x^2+x} & 0 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

Finally, the interpolation phase gives :

$$\begin{aligned}
c_0 &= C(0) \\
c_1 &= (x^2+x+1)/(x^2+x) \cdot C(0) + C(1) + C(x)/x \\
&\quad + C(x+1)/(x+1) + (x^2+x) \cdot C(\infty) \\
c_2 &= C(1)/(x^2+x) + C(x)/(x+1) + C(x+1)/x \\
&\quad + (x^2+x+1) \cdot C(\infty) \\
c_3 &= C(0)/(x^2+x) + C(1)/(x^2+x) + C(x)/(x^2+x) \\
&\quad + C(x+1)/(x^2+x) \\
c_4 &= C(\infty)
\end{aligned}$$

Appendix E Experimentation procedure

Measurements were performed on a Dell Inspiron laptop with an Intel 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 10.2.0, the compiler options are as follows:
`-O3 -g -march=tigerlake -funroll-all-loops -lm -lgf2x`.
We kept the `-funroll-all-loops` option though it does not provide significant improvements. We follow the same kind of test procedure that the one described in [10] :

- the *Turbo-Boost*® is deactivated during the tests;
- 1000 runs are executed in order to "heat" the cache memory;
- one generates 50 random data sets, 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;
- this procedure is run on console mode, to avoid system perturbations, and obtain the most accurate cycle counts.

The clock cycle counter is `rdtsc` and the instruction counter is `rdpmc` with the corresponding selection. The results for the smallest sizes (i.e. 256 bit and 512 bit operand sizes) are not very reliable since `rdtsc` and `rdpmc` are not serializing instructions (see [19]). For such sort of small functions, we wanted to avoid the insertion of a costly serializing instruction as `cpuid`, while the instruction count and the clock cycle number may be less than 20. We chose not to present them. The first size considered is 1024 bits, i.e. binary polynomial of degree at most 1023 operands.

Appendix F Instruction count and performances

F.1 Instruction count comparison

In Table 13, we provide the comparison between the instruction count of our schoolbook and Karatsuba versions. Moreover, we compare this two approaches with the current state-of-the-art AVX2 reference. Such an AVX2 implementation can be found in the source code of the optimized version of HQC [2]. It uses the AVX2 instruction set and the non vectorized PCLMULQDQ instruction. Finally, we also put in Table 13 the instruction number of the assembly source code for the same multiplication presented by Drucker *et al.* in [10]. Here are some comments on these results:

- The best version is our implementation of the schoolbook approach, dividing by more than 2 the instruction number in comparison with the state-of-the-art AVX2 implementation.
- Our Karatsuba approach presents more instructions but only 3 VPCLMULQDQ instead of 4 for the schoolbook version. Thus, the performance comparison may vary according to the latency and throughput of the instructions.
- Drucker *et al.*'s version has 8 VPCLMULQDQ instructions and a larger instruction number (31, instead of 19 for our implementation of the schoolbook approach). This is due to the fact that they only use 2 elementary 64 bit multiplications per VPCLMULQDQ instruction (ymmm version of the instruction), while we use 4. This also implies more XOR's in their case.

F.2 Performances for the 256 bit level kernels

We present Table 14 the performances of the AVX512 Karatsuba multiplications using the 256 bit kernels presented above. We also include the results of the multiplications using our 8x8 SB-512 kernel.

Instruction count 256 bit size operands	VPCLMULQDQ			AVX2
Instructions	SB version	Karat. version	Drucker *et al.* [10]	Karat. Rec.
_mm512_clmulepi64_epi128	4	3	8	
_mm_clmulepi64_si128				9
XOR	5	7	15	25
AND	0	1		
_mm512_broadcast_i64x4	2	2		
_mm512_permutexvar_epi64	3	6	6	
_mm512_permutex2var_epi64	5	5		
_mm512_alignr_epi64			2	
_mm_loadu_si128				4
_mm_shuffle_epi32				6
_mm_setzero_si128				6
_mm_unpacklo_epi64				3
_mm_unpackhi_epi64				3
Total	**19**	**24**	**31**	**56**
additional vmovdqa64's	8	9	6	8

Table 13: Instruction count for the 256 bit multiplication versions

KaratRec			Drucker *et al.* [10]		AVX2 [2]	**This work** vpclmulqdq-512	
size		gf2x	4x4 - 256	8x8 - 512		SB-256	Karat.-256
1024	# clock cycles	339	239	169	183	167	186
	# instructions	1224	326	221	612	276	339
2048	# clock cycles	998	744	532	610	541	627
	# instructions	3892	1137	736	1867	908	1102
4096	# clock cycles	2949	2262	1621	1929	1685	1937
	# instructions	11079	3632	2358	5684	2769	3346
8192	# clock cycles	8742	6960	4926	6038	5204	6205
	# instructions	33182	11535	7547	17991	8718	10552
16384	# clock cycles	26128	21154	14940	18327	15675	18043
	# instructions	100163	35483	23313	54840	26482	32068
32768	# clock cycles	78889	65758	45244	59613	47592	54881
	# instructions	295755	108329	70941	166410	80736	97278
65536	# clock cycles	226640	203855	140347	187305	147572	169844
	# instructions	853977	328772	214955	503014	244222	293873
131072	# clock cycles	667900	621942	425430	572984	446811	511939
	# instructions	2516857	992919	648533	1515625	736345	885164

Table 14: Performance comparison for Algorithm 1