Fast multiplication of binary polynomials with the forthcoming vectorized VPCLMULQDQ instruction

Nir Drucker
University of Haifa, Israel,
and
Amazon Web Services Inc.[1]

Shay Gueron
University of Haifa, Israel,
and
Amazon Web Services Inc.[1]

Vlad Krasnov
CloudFlare, Inc.
San Francisco, USA

***Abstract*—Polynomial multiplication over binary fields $\mathbb{F}_{2^n}$ is a common primitive, used for example by current cryptosystems such as AES-GCM (with $n = 128$). It also turns out to be a primitive for other cryptosystems, that are being designed for the Post Quantum era, with values $n \gg 128$. Examples from the recent submissions to the NIST Post-Quantum Cryptography project, are BIKE, LEDAKem, and GeMSS, where the performance of the polynomial multiplications, is significant. Therefore, efficient polynomial multiplication over $\mathbb{F}_{2^n}$, with large n, is a significant emerging optimization target.**

Anticipating future applications, Intel has recently announced that its future architecture (codename "Ice Lake") will introduce a new vectorized way to use the current `VPCLMULQDQ` instruction. In this paper, we demonstrate how to use this instruction for accelerating polynomial multiplication. Our analysis shows a prediction for at least 2x speedup for multiplications with polynomials of degree 512 or more.

I. Introduction

Several modern encryption schemes use polynomial multiplication over $\mathbb{F}_{2^n}$ as one of their main primitives. One prominent example is AES-GCM whose (almost) XOR-universal hash function (GHASH) is an evaluation of polynomials with coefficients in $\mathbb{F}_{2^{128}}$. The nonce misuse resistant Authenticated Encryption AES-GCM-SIV is using a similar (almost) XOR-universal hash function (POLYVAL). These evaluations can be significantly sped up with dedicated instructions. Indeed, modern general-purpose processors are equipped with the "carry-less multiplication" instruction `PCLMULQDQ` [1], [2].

Multiplication of polynomials with higher degrees (≥ 128) is becoming a useful computational task for newly designed cryptosystems that are based on coding or multivariate polynomials. These are motivated by the NIST Post-Quantum Project [3] that targets the definition of the next generation of quantum-resistant public key and signature cryptosystems. We give three examples out of the recent 69 submissions to this project, that use polynomial multiplication in $\mathbb{F}_{2^n}$: BIKE [4] (MDPC codes) with $n = 32,749$, LEDAKem [5] (LDPC codes) with $n = 99,053$, and GeMSS [6] (multivariate polynomials) using $n = 354$. We note that the performance of the polynomial multiplication is significant in these algorithms, for example, the key generation and the encapsulation steps of BIKE are dominated by such multiplications.

[1]This work was done prior to joining Amazon.

Intel has recently announced [7] that its future architecture, codename "Ice Lake", will introduce a new instruction called `VPCLMULQDQ`. This instruction, together with the new vectorized AES instructions (`VAESENC` and `VAESDEC`), are useful for accelerating AES-GCM. However, we argue that even as a standalone instruction, `VPCLMULQDQ` is useful for some new emerging algorithms, which paper demonstrates how it can be used for accelerating "big" polynomial multiplications (with degree > 511).

The correctness of the algorithms (and the code) can be checked now, but actual performance measurements require a real CPU, which is currently unavailable. To address this difficulty, we use other techniques to estimate the future performance, such as counting instructions and running with some "stand in" replacements. The results of both techniques are similar, and this allows us to predict that polynomial multiplication is going to be at least 2x faster on these future CPUs, compared to the current software implementations on the current architectures.

The paper is organized as follows: Section II describes the new `VPCLMULQDQ` instruction. Section III presents some concepts that we use for polynomial multiplications. Section IV presents our new implementation. We show our experimental results in Section V, and conclude in Section VI.

II. The VPCLMULQDQ instruction

Algorithm 1 DST = `VPCLMULQDQ`(SRC1, SRC2, Imm8)

Inputs: SRC1, SRC2 (wide registers) Imm8 (8 bits)
Outputs: DST (a wide register)
procedure `VPCLMULQDQ`(SRC1, SRC2, Imm8)
  for i := 0 to $KL - 1$ **do**
    $j_1 = 2i + Imm8[0]$
    $j_2 = 2i + Imm8[4]$
    T1[63 : 0] = SRC1[$64(j_1 + 1) - 1 : 64j_1$]
    T2[63 : 0] = SRC2[$64(j_2 + 1) - 1 : 64j_2$]
    DST[$128(i + 1) - 1 : 128i$] = `PCLMULQDQ`(T1, T2)
  return DST

Alg. 1 [7] presents the extended instruction `VPCLMULQDQ`. It vectorizes polynomial (carry-less) multiplications, and is able to perform $KL = 1/2/4$ multiplications of two qwords (64 bits). The multiplicands are selected from two source operands, which are $128/256/512$-bit registers (named xmm, ymm, zmm, respectively), and the selection is determined by the value of the immediate byte. Note that the case $KL = 1$

							a_7	$\boldsymbol{a_6}$	a_5	$\boldsymbol{a_4}$	a_3	$\boldsymbol{a_2}$	a_1	$\boldsymbol{a_0}$
							b_7	$\boldsymbol{b_6}$	b_5	$\boldsymbol{b_4}$	b_3	$\boldsymbol{b_2}$	b_1	$\boldsymbol{b_0}$
							a_0b_7	$\boldsymbol{a_0b_6}$	a_0b_5	$\boldsymbol{a_0b_4}$	a_0b_3	$\boldsymbol{a_0b_2}$	a_0b_1	$\boldsymbol{a_0b_0}$
						$\boldsymbol{a_1b_7}$	a_1b_6	$\boldsymbol{a_1b_5}$	a_1b_4	$\boldsymbol{a_1b_3}$	a_1b_2	$\boldsymbol{a_1b_1}$	a_1b_0	
					a_2b_7	$\boldsymbol{a_2b_6}$	a_2b_5	$\boldsymbol{a_2b_4}$	a_2b_3	$\boldsymbol{a_2b_2}$	a_2b_1	$\boldsymbol{a_2b_0}$		
				$\boldsymbol{a_3b_7}$	a_3b_6	$\boldsymbol{a_3b_5}$	a_3b_4	$\boldsymbol{a_3b_3}$	a_3b_2	$\boldsymbol{a_3b_1}$	a_3b_0			
			a_4b_7	$\boldsymbol{a_4b_6}$	a_4b_5	$\boldsymbol{a_4b_4}$	a_4b_3	$\boldsymbol{a_4b_2}$	a_4b_1	$\boldsymbol{a_4b_0}$				
		$\boldsymbol{a_5b_7}$	a_5b_6	$\boldsymbol{a_5b_5}$	a_5b_4	$\boldsymbol{a_5b_3}$	a_5b_2	$\boldsymbol{a_5b_1}$	a_5b_0					
	a_6b_7	$\boldsymbol{a_6b_6}$	a_6b_5	$\boldsymbol{a_6b_4}$	a_6b_3	$\boldsymbol{a_6b_2}$	a_6b_1	$\boldsymbol{a_6b_0}$						
$\boldsymbol{a_7b_7}$	a_7b_6	$\boldsymbol{a_7b_5}$	a_7b_4	$\boldsymbol{a_7b_3}$	a_7b_2	$\boldsymbol{a_7b_1}$	a_7b_0							

Fig. 1. Schoolbook multiplication of 8×8 qwords (or 4×4 qwords in red dashed border). The size of each $a_i b_j$ is 128 bits.

using xmm registers, is exactly the `VPCLMULQDQ` instruction that is currently available in the modern CPU's.

III. Polynomial Multiplication

Obviously, polynomial multiplication can be executed by the standard schoolbook algorithms. However, for sufficiently high degrees, a Carry-less Karatsuba algorithm [1], [2], [8] is faster. For even higher degrees, other algorithms such as Toom-Cook [9], [10] may become useful. These algorithms (e.g., Karatsuba and Toom-Cook) work in a recursive way, where at each step multiplies smaller degree operands. After a sufficient number of iterations, the involved polynomial has a small enough degree, and then it pays to revert to the schoolbook multiplication, for the final step.

In [11], we showed that for polynomials of degree slightly smaller or equal to a power of two, the best performance is attained with a recursive Karatsuba algorithm, until the polynomials area the degree of 255 (4 qwords), and then revert to the schoolbook multiplication.

Here, we demonstrate how the new `VPCLMULQDQ` instruction can be used for two methods that multiply polynomials of degree 511 (8 qwords): a) A schoolbook multiplication of 8×8 qwords using zmm registers; b) A Karatsuba multiplication that invokes a 4×4 qwords schoolbook multiplication, as its "core", and uses ymm registers. We compare them to a reference implementation [11] that uses the existing version of `PCLMULQDQ`, for a Karatsuba multiplication that wraps a 4×4 qwords schoolbook flow.

IV. VPCLMULQDQ and schoolbook

We use the same algorithm for both the 8×8 qwords and 4×4 qwords multiplications. The only difference is in the sizes of the wide registers that are used (zmm and ymm, respectively). For brevity, we present only the 8×8 qwords version, but in each table, also mark the 4×4 qwords parts with red dashed lines. Fig. 2 shows a real (37-lines) code snippet of the 4×4 qwords multiplication, as an example.

Fig. 1 illustrates the basic schoolbook multiplication with polynomials of degree 511, namely a, b. They are represented by 8 qwords $a = a_7||a_6||, \dots, ||a_0$ and $b = b_7||b_6||, \dots, ||b_0$ (where $||$ denotes concatenation of qwords). Note that Fig. 2 (Steps 26-31 and 33-36) sums up the values in the even columns (bold) and the value in the odd columns separately (as recommended in [11]). Note also that the horizontal lines contain 16 qwords, and that this quantity fits perfectly in two zmm registers. This is why splitting each row into "even" and "odd" columns, where each part is 8 qwords long, seems natural. We use 16 zmm registers, namely $\texttt{Even}_0, \dots, \texttt{Even}_7$ and $\texttt{Odd}_0, \dots, \texttt{Odd}_7$, to accommodate these values. Table I presents the layout of the qwords in the registers.

TABLE I

A split presentation of the even and odd columns presented in Figure 1, the red dashed line marks the 4×4 qwords multiplication boundary

Register alias	Values (4 packets of 128-bits dashed for alignment)							
$\texttt{Even}_0$	-	-	-	-	a_0b_6	a_0b_4	a_0b_2	a_0b_0
$\texttt{Even}_1$	-	-	-	a_1b_7	a_1b_5	a_1b_3	a_1b_1	-
$\texttt{Even}_2$	-	-	-	a_2b_6	a_2b_4	a_2b_2	a_2b_0	-
$\texttt{Even}_3$	-	-	a_3b_7	a_3b_5	a_3b_3	a_3b_1	-	-
$\texttt{Even}_4$	-	-	a_4b_6	a_4b_4	a_4b_2	a_4b_0	-	-
$\texttt{Even}_5$	-	a_5b_7	a_5b_5	a_5b_3	a_5b_1	-	-	-
$\texttt{Even}_6$	-	a_6b_6	a_6b_4	a_6b_2	a_6b_0	-	-	-
$\texttt{Even}_7$	a_7b_7	a_7b_5	a_7b_3	a_7b_1	-	-	-	-
$\texttt{Odd}_0$	-	-	-	-	a_0b_7	a_0b_5	a_0b_3	a_0b_1
$\texttt{Odd}_1$	-	-	-	-	a_1b_6	a_1b_4	a_1b_2	a_1b_0
$\texttt{Odd}_2$	-	-	-	a_2b_7	a_2b_5	a_2b_3	a_2b_1	-
$\texttt{Odd}_3$	-	-	-	a_3b_6	a_3b_4	a_3b_2	a_3b_0	-
$\texttt{Odd}_4$	-	-	a_4b_7	a_4b_5	a_4b_3	a_4b_1	-	-
$\texttt{Odd}_5$	-	-	a_5b_6	a_5b_4	a_5b_2	a_5b_0	-	-
$\texttt{Odd}_6$	-	a_6b_7	a_6b_5	a_6b_3	a_6b_1	-	-	-
$\texttt{Odd}_7$	-	a_7b_6	a_7b_4	a_7b_2	a_7b_0	-	-	-

TABLE II

Permutation of values from Table I for 8×8 multiplication

Register alias	Permutation mask (Original qword idx)	Output Values (4 packets of 128-bits)			
$\texttt{Even}_0$	-	$\boldsymbol{a_0b_6}$	$\boldsymbol{a_0b_4}$	$\boldsymbol{a_0b_2}$	$\boldsymbol{a_0b_0}$
$\texttt{Even}_1$	5 4 3 2 1 0 7 6	$\boldsymbol{a_1b_5}$	$\boldsymbol{a_1b_3}$	$\boldsymbol{a_1b_1}$	a_1b_7
$\texttt{Even}_2$	5 4 3 2 1 0 7 6	$\boldsymbol{a_2b_4}$	$\boldsymbol{a_2b_2}$	$\boldsymbol{a_2b_0}$	a_2b_6
$\texttt{Even}_3$	3 2 1 0 7 6 5 4	$\boldsymbol{a_3b_3}$	$\boldsymbol{a_3b_1}$	a_3b_7	a_3b_5
$\texttt{Even}_4$	3 2 1 0 7 6 5 4	$\boldsymbol{a_4b_2}$	$\boldsymbol{a_4b_0}$	a_4b_6	a_4b_4
$\texttt{Even}_5$	1 0 7 6 5 4 3 2	$\boldsymbol{a_5b_1}$	a_5b_7	a_5b_5	a_5b_3
$\texttt{Even}_6$	1 0 7 6 5 4 3 2	$\boldsymbol{a_6b_0}$	a_6b_6	a_6b_4	a_6b_2
$\texttt{Even}_7$	-	a_7b_7	a_7b_5	a_7b_3	a_7b_1
$\texttt{Odd}_0$	-	$\boldsymbol{a_0b_7}$	$\boldsymbol{a_0b_5}$	$\boldsymbol{a_0b_3}$	$\boldsymbol{a_0b_1}$
$\texttt{Odd}_1$	-	$\boldsymbol{a_1b_6}$	$\boldsymbol{a_1b_4}$	$\boldsymbol{a_1b_2}$	$\boldsymbol{a_1b_0}$
$\texttt{Odd}_2$	5 4 3 2 1 0 7 6	$\boldsymbol{a_2b_5}$	$\boldsymbol{a_2b_3}$	$\boldsymbol{a_2b_1}$	a_2b_7
$\texttt{Odd}_3$	5 4 3 2 1 0 7 6	$\boldsymbol{a_3b_4}$	$\boldsymbol{a_3b_2}$	$\boldsymbol{a_3b_0}$	a_3b_6
$\texttt{Odd}_4$	3 2 1 0 7 6 5 4	$\boldsymbol{a_4b_3}$	$\boldsymbol{a_4b_1}$	a_4b_7	a_4b_5
$\texttt{Odd}_5$	3 2 1 0 7 6 5 4	$\boldsymbol{a_5b_2}$	$\boldsymbol{a_5b_0}$	a_5b_6	a_5b_4
$\texttt{Odd}_6$	1 0 7 6 5 4 3 2	$\boldsymbol{a_6b_1}$	a_6b_7	a_6b_5	a_6b_3
$\texttt{Odd}_7$	1 0 7 6 5 4 3 2	$\boldsymbol{a_7b_0}$	a_7b_6	a_7b_4	a_7b_2

Values as formatted as in Table I, are loaded to wide registers as follows: load b to the wide register $\texttt{B}$; split the second polynomial a across four wide registers $\texttt{A}_0, \dots, \texttt{A}_3$, where

TABLE III
PERMUTATION OF VALUES FROM TABLE I FOR 4×4 MULTIPLICATION

Register alias	Permutation mask	Output values	
$\mathtt{Even}_0$	-	$\boldsymbol{a_0b_2}$	$\boldsymbol{a_0b_0}$
$\mathtt{Even}_1$	1 0 3 2	$\boldsymbol{a_1b_1}$	a_1b_3
$\mathtt{Even}_2$	1 0 3 2	$\boldsymbol{a_2b_0}$	a_2b_2
$\mathtt{Even}_3$	-	a_3b_3	a_3b_1
$\mathtt{Odd}_0$	-	$\boldsymbol{a_0b_3}$	$\boldsymbol{a_0b_1}$
$\mathtt{Odd}_1$	-	$\boldsymbol{a_1b_2}$	$\boldsymbol{a_1b_0}$
$\mathtt{Odd}_2$	1 0 3 2	$\boldsymbol{a_2b_1}$	a_2b_3
$\mathtt{Odd}_3$	1 0 3 2	$\boldsymbol{a_3b_1}$	a_3b_3

$\mathtt{A_i} = \mathtt{a_{2i+1}}||\mathtt{a_{2i}}||\mathtt{a_{2i+1}}||\mathtt{a_{2i}}||\mathtt{a_{2i+1}}||\mathtt{a_{2i}}||\mathtt{a_{2i+1}}||\mathtt{a_{2i}}$. We carry out the multiplication by using `VPCLMULQDQ` instructions as described in Fig. 2, Steps 12-19.

For efficient (vectorized) schoolbook accumulation, we use the `VPERMPD` instruction that can shuffle the qwords in each register. This instruction receives a wide register $R[511:0]$ and the mask $M[511:0]$, and outputs $O[511:0]$, where $O[64(i+1):64i] = R[64(t+1):64t]$, $t = M[64i+2:64i]$, $i = 0, \ldots, 7$. The masks that define the permutations are given in Table II (or in Table III for the 4×4 multiplication).

For example, in Fig. 2 Steps 21-24, we use the wide register `Perm` which holds the mask "1 0 3 2". Subsequently, we accumulate the "high part" (the boldface values in Tables II and III), and the "low part" of the even columns in `EvenSumHigh` and `EvenSumLow`, respectively. We do the same for the odd columns, accumulating them in `OddSumHigh` and `OddSumLow`, respectively (Fig. 2, Steps 26-31 and 33-36). Finally, we shift the value `OddSumHigh`||`OddSumLow` by one qword to the left (Fig. 2, Steps 38-39), and add the 16-qword result to `EvenSumHigh`||`EvenSumLow`, to obtain the final product.

Our implementation uses the mask registers $\%\mathrm{k1}, \ldots, \%\mathrm{k6}$ (such registers are part of the AVX512 architecture), in order to perform operations on parts of a wide register. For example, consider Steps 27-36 in Fig. 2: the halves (i. e., upper/lower qwords) of the ymm registers are xored separately by using the mask registers $\mathrm{k1} = 0\mathrm{xc}$ (upper) and $\mathrm{k2} = 0\mathrm{x}03$ (lower). Note that the code snippets in Fig. 2 assume that the mask and the permutation registers are already initialized to the relevant value (e. g., before the recursive Karatsuba flow).

```
1  vmovdqu64 (a), A0
2  vmovdqu64 (b), B
3  vpermpd A0, PermA1, A1
4  vpermpd A0, PermA0, A0
5
6  vxorpd OddSumLow, OddSumLow, OddSumLow
7  vxorpd OddSumHigh, OddSumHigh, OddSumHigh
8  vxorpd EvenSumLow, EvenSumLow, EvenSumLow
9  vxorpd EvenSumHigh, EvenSumHigh, EvenSumHigh
10 vxorpd Zero, Zero, Zero
11
12 vpclmulqdq A0, B, Even0, 0x00
13 vpclmulqdq A0, B, Even1, 0x11
14 vpclmulqdq A1, B, Even2, 0x00
15 vpclmulqdq A1, B, Even3, 0x11
16 vpclmulqdq A0, B, Odd0, 0x10
17 vpclmulqdq A0, B, Odd1, 0x01
18 vpclmulqdq A1, B, Odd2, 0x10
19 vpclmulqdq A1, B, Odd3, 0x01
20
21 vpermpd Even1, Perm, Even1
22 vpermpd Even2, Perm, Even2
23 vpermpd Odd2, Perm, Odd2
24 vpermpd Odd3, Perm, Odd3
25
26 vmovdqa64 Even0, EvenSumLow
27 vxorpd EvenSumLow, Even1, EvenSumLow{%k1}
28 vxorpd EvenSumLow, Even2, EvenSumLow{%k1}
29 vmovdqa64 Even3, EvenSumHigh
30 vxorpd EvenSumHigh, Even1, EvenSumHigh{%k2}
31 vxorpd EvenSumHigh, Even2, EvenSumHigh{%k2}
32
33 vxorpd Odd1, Odd0, OddSumLow
34 vxorpd Odd2, OddSumLow, OddSumLow{%k1}
35 vxorpd Odd3, OddSumLow, OddSumLow{%k1}
36 vxorpd Odd3, Odd2, OddSumHigh{%k2}
37
38 valignq 0x3, OddSumLow, OddSumHigh, OddSumHigh
39 valignq 0x3, Zero, OddSumLow, OddSumLow
40
41 vxorpd OddSumLow, EvenSumLow, EvenSumLow
42 vxorpd OddSumHigh, EvenSumHigh, EvenSumHigh
43
44 vmovdqu64 EvenSumLow, (res)
45 vmovdqu64 EvenSumHigh, 0x20(res)
```

Fig. 2. Schoolbook multiplication of 4×4 qwords: $res[1023:0] = a[511:0] \cdot b[511:0]$.

V. RESULTS

This section provides the performance results of our study. To this end, we wrote three core flows that performs schoolbook multiplication: a) A reference code that performs 4×4 qwords multiplication, written in AVX (xmm registers) and uses `PCLMULQDQ`instructions; b) A 4×4 qwords multiplication written with AVX512 and the new `VPCLMULQDQ` instruction; c) A 8×8 qwords multiplication written with AVX512 and the new `VPCLMULQDQ` instruction. We also wrote a recursive Karatsuba wrapper, optimized with AVX instructions that run on modern CPUs. Subsequently, we replaced the underlying core flow and collected the results. The core functionality was written in x86 assembly, and wrapped by assisting C code compiled with gcc (version 5.4.0) in 64-bit mode, using the "O3" Optimization level.

Currently no real processor with `VPCLMULQDQ` instruction exists. Therefore, to predict the potential improvement on future Intel architectures we used the Intel Software Developer Emulator (SDE) [12]. This tool allows us to count the number of instructions executed during each of the tested functions. We marked the start/end boundaries of each function with "SSC marks" 1 and 2, respectively. This is done by executing "`movl ssc_mark, %ebx; .byte 0x64, 0x67, 0x90`" and invoking the SDE with the flags "-start_ssc_mark 1 -stop_ssc_mark 2 -mix -icl". The rationale is that a reduced number of instructions typically indicates improved performance that will be observed on a real processor (although the exact relation between the instructions count and the eventual cycles count is not known in advanced).

Figure 3 compares the instructions count for performing polynomial multiplication, using different software "core flows" for polynomials of degree up to $2^{16}-1$. Figure 4 zooms in to the same comparison for polynomials of smaller degrees. We see that the two 4×4 implementations use twice as many instructions as our zmm 8×8 implementation. For the lower

TABLE IV
INSTRUCTIONS COUNT FOR THE "CORE FLOWS". THE xmm AND ymm IMPLEMENTATIONS OPERATE ON 4×4 QWORDS THEREFORE THEIR COUNT IS MULTIPLIED BY 3 (FOR THE KARATSUBA ALGORITHM).

Method	VPCLMULQDQ	VPXORPD	VMOVDQU	VPERMPD	Other	total
xmm	48	60	42	0	0	150
ymm	24	45	12	18	21	120
zmm	16	31	4	16	11	78

degree polynomials this becomes even more noticable (up to a factor of 3.43). This can also be observed in Table IV that summarizes the instructions count for a single call to the 8×8 qwords multiplication (with zmm). For a fair comparison, the values reported for the 4×4 qwords multiplications (with ymm or xmm) were multiplied by 3 in order to account for their use in the Karatsuba algorithm. This count overlooks the additional instructions that the Karatsuba wrapper requires.

The exact relation between the instructions count and the eventual cycles count is only an indicator that could be confirmed when a real processor is available. To further substantiate our prediction we complement our report with a second method that we call "stand in". We replace the VPCLMULQDQ instructions with the VPMULDQ instruction that has a latency of 5 cycles and throughput of 1 cycle [13]. This is an similar enough to the non-vectorized PCLMULQDQ instruction that have the same throughput and latency between 4 to 7 cycles depends on the architecture. This approach allows us to run experiments on a real processor (although the results are not functionally correct we are only interested in approximating the anticipated performance).

These experiments were carried out on a platform equipped with the latest 7^{th} Generation Intel® CoreY^{TM} processor ("Kaby Lake") - Intel® Xeon® Platinum 8124M CPU at 3.00 GHz Core® $i5-750$. The platform has 70 GB RAM, 32K L1d and L1i cache, 1,024K L2 cache, and 25,344K L3 cache. It was configured to disable the Intel® Turbo Boost Technology, and the Enhanced Intel Speedstep® Technology. The runs were carried out on a Linux (Ubuntu 16.04.3 LTS) OS.

The performance is reported in processor cycles, where lower is better, reflecting the performance per a single core. We use the follow measurement methodology: Each measured function was isolated, run 25 times (warm-up), followed by 100 iterations that were clocked (using the $RDTSC$ instruction) and averaged. To minimize the effect of background tasks running on the system, each such experiment was repeated 10 times, and the minimum result was recorded.

Figure 5 shows a speedup of 2.2x and 1.25x between the reference implementation and the zmm 8×8 and ymm 4×4 implementations, respectively.

VI. CONCLUSION

This paper showed an algorithm that can leverage Intel's new instruction VPCLMULQDQ for fast polynomial multiplication. We used two different prediction indicators and their results match. Based on these findings, we predict that future processors equipped with VPCLMULQDQ would be able to multiply binary polynomials at least 2 times faster than they do today.

Finally, to show the potential impact we note that polynomial multiplications (with high degrees) is a noticeable part of the workload in some of the new post-quantum proposals [3] such as BIKE, LEDAKem, and GeMSS.

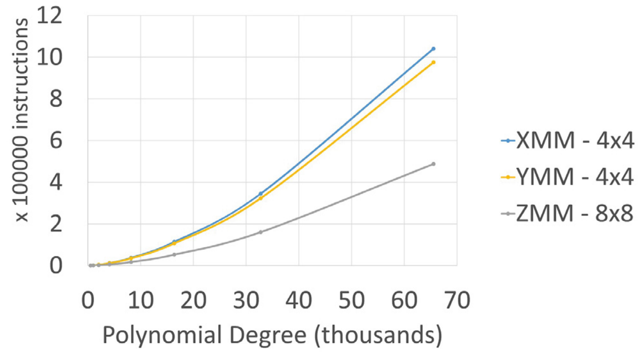

Fig. 3. Comparison of instructions count (lower is better).

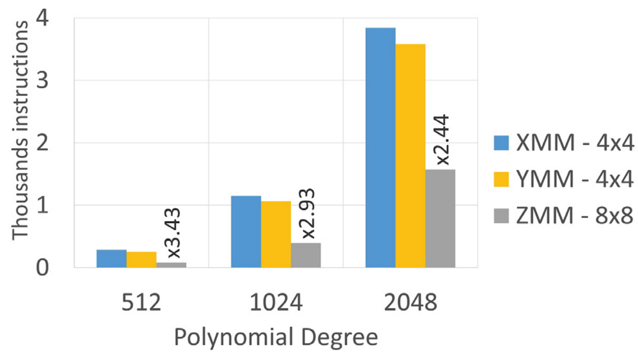

Fig. 4. Comparison of instructions count (lower is better).

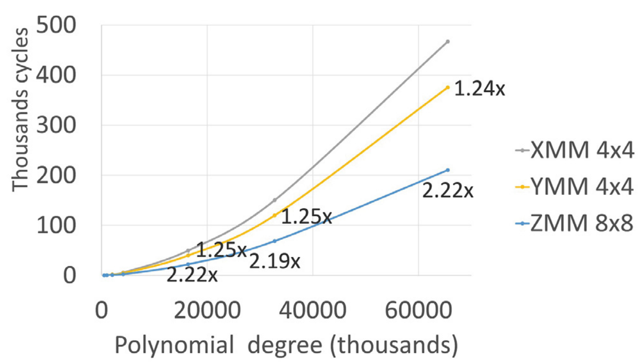

Fig. 5. Comparison of cycles count (lower is better). The VPCLMULQDQ instructions were replaced with VPMULDQ instructions

Acknowledgements

This research was supported by: The PQCRYPTO project, which was partially funded by the European Commission

Horizon 2020 research Programme, grant #645622; The Israel Science Foundation (grant No. 1018/16); The Ministry of Science and Technology, Israel, and the Department of Science and Technology, Government of India; The BIU Center for Research in Applied Cryptography and Cyber Security, in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office; The Center for Cyber Law and Policy at the University of Haifa.

REFERENCES

[1] S. Gueron and M. Kounavis, "Efficient implementation of the galois counter mode using a carry-less multiplier and a fast reduction algorithm," *Information Processing Letters*, vol. 110, no. 14, pp. 549 – 553, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002001901000092X

[2] S. Gueron and M. E. Kounavis, "Intel® carry-less multiplication instruction and its usage for computing the gcm mode," *White Paper*, 2010.

[3] —, "Nist:post-quantum cryptography - call for proposals," September 2017, https://csrc.nist.gov/Projects/Post-Quantum-Cryptography.

[4] N. Aragon, P. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Guneysu, C. A. Melchor *et al.*, "Bike: Bit flipping key encapsulation," 2017.

[5] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "Ledakem: a post-quantum key encapsulation mechanism based on QC-LDPC, codes," *CoRR*, vol. abs/1801.08867, 2018. [Online]. Available: http://arxiv.org/abs/1801.08867

[6] A. Casanova, J.-C. Faug'ere, G. Macario-Rat, J. Patarin, L. Perret, and J. Ryckeghem, "GeMSS: A Great Multivariate Short Signature," https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/GeMSS.zip, November 2017.

[7] —, "Intel architecture instruction set extensions programming reference," https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, October 2017.

[8] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Soviet Physics Doklady*, vol. 7, p. 595, Jan 1963.

[9] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," in *Soviet Mathematics Doklady*, vol. 3, no. 4, 1963, pp. 714–716.

[10] S. A. Cook and S. O. Aanderaa, "On the minimum computation time of functions," *Transactions of the American Mathematical Society*, vol. 142, pp. 291–314, 1969. [Online]. Available: http://www.jstor.org/stable/1995359

[11] N. Drucker and S. Gueron, "A toolbox for software optimization of QC-MDPC code-based cryptosystems," Cryptology ePrint Archive, Report 2017/1251, 2017, https://eprint.iacr.org/2017/1251.

[12] _, "Intel®software development emulator," https://software.intel.com/en-us/articles/intel-software-development-emulator.

[13] —, "Intel ®64 and IA-32 architectures optimization reference manual," June 2016.