Code based Cryptography
Vectorized implementation of HQC round 2 NIST PQC submission

Oleg BESSONOV, Jean-Marc ROBERT and Pascal VÉRON

1 PATCH, version 2, 19/07/2019

In this section, we present the modification of the optimized HQC source code (HQC NIST round 2 submission, 10/04/2019, see Aguilar-Melchior *et al.* [1]) in order to improve the performances. This modification deals with the vectorization of two functions: the sparse-dense polynomial multiplication (vectorized fastConvolution presented above) and the syndrome generation (`syndrome_gen`). We chose these functions after a profiling analysis which gave the three main functions whose load on the computation time is the heaviest :

- `sparse_dense_mul`
- `syndrome_gen`
- `repetition_code_decode`

The vectorization makes use of the `avx` and `avx2` instruction set, which is based on the utilization of the `ymm` 256 bit registers. In the sequel, the platform is a dell 9020 whose CPU is as follows:

```
model name : Intel(R) Core(TM) i5-4690S CPU @ 3.20GHz
cache size : 6144 KB
```

The *Turbo-Boost*®is deactivated during the tests.
The test procedure is as follows :

- 1000 runs in order to "heat" the cache memory
- one generates 50 random data sets
  - for each data set, one takes the minimum of the execution clock cycle numbers over a batch of 1000 runs
- the performance is the average of all these minimums

This procedure is a standard and a recommended one in the architecture community.
In comparison with the previous version of the patch, some small optimisations has been written in the Fast Convolution multiplication (improved carry management, loop optimisations...). These allows to take a better advantage of the vectorization.

The speed-up of our patched HQC implementation is about 34 to 46 % in comparison with the current optimized version mentioned above, see table 4.

1.1 Fast convolution

1.1.1 Algorithm and implentation

This section remains unchanged in comparison with the previous version of the 18/07/2019, except the performance results, which shows about 40% improvement on the multiplication in comparison with the one of the previous patch (18/07/2019). This operation is a sparse-dense polynomial multiplication over the anticyclic ring $\mathcal{R} = \mathbb{F}_2[X]/(X^N - 1)$ (N is denoted `PARAM_N` in the implementation). The multiplication is implemented according to algorithm 1.

Algorithm 1 *Fast Convolution*

Require: A dense polynomial $A[X] = \sum_0^{N-1} a_i X^i \in \mathcal{R}$ and its N bit representation A, and a sparse polynomial $B[X] = \sum_{\forall b_i \neq 0} b_i X^i \in \mathcal{R}$ of Hamming weight ω, represented as a ω position vector vB.
Ensure: $C[X] = A[X] \cdot B[X] \bmod (X^N - 1)$

1: $C \leftarrow 0$
2: **for** $i = 0$ to $\omega - 1$ **do**
3: $\quad T \leftarrow A << vB_i$
4: $\quad C \leftarrow C \wedge T$
5: **end for**
6: $P[X] \leftarrow C[X] \bmod (X^N - 1)$
7: **return** $P[X]$

In table 1, we present the performances (including a vectorized reduction modulo $(X^N - 1)$), compared to the multiplication implementation used in the optimized `HQC` source code (HQC, [1]), which is inspired by the vectorized one of the `ntl` library(see Quercia and Zimmermann in [3] and [2]).

Tab. 1: Performance comparison between multiplications

algorithm/ op. size (bits)	Optimized HQC `sparse_dense_mult`	`fastConvolution WithRed`	Optimized HQC `sparse_dense_mult`	`fastConvolution WithRed`
BASIC				
	w = 67		w = 77	
24677	100360	24784	110952	26468
ADVANCED				
	w = 101		w = 117	
43669	236792	62216	266008	71796
46747	253016	65664	284300	75820
PARANOIAC				
	w = 133		w = 153	
63587	426472	118012	479228	136080
126120 67699	453776	126120	509728	144496
70853	474040	132480	532516	152468

The speed-up of our version 2 vectorized fast convolution multiplication implementation is now about 70-75 %.

1.1.2 Patch over the `HQC` source code

We implemented a patch using this multiplication. The performances are presented table 2.

Tab. 2: `HQC` Performance comparison without and with vectorized fast convolution multiplication patch, # clock cycles

	keygen		encryption		decryption	
algorithm/ `PARAM_N` (bits)	`HQC` `origin`	`HQC` `with patch`	`HQC` `origin`	`HQC` `with patch`	`HQC` `origin`	`HQC` `with patch`
BASIC						
24677 speed-up	238628	**155351** 35.1%	505886	**334838** 33.7%	1085102	**840277** 22.6%
ADVANCED						
43669 speed-up	478133	**290251** 39.3%	984790	**574678** 41.4%	1783812	**1215340** 31.9%
46747 speed-up	503797	**306005** 39.3%	1037276	**609119** 41.2%	1837189	**1214742** 33.8%
PARANOIAC						
63587 speed-up	783288	**450261** 42.5%	1584447	**883910** 44.2%	2658767	**1681677** 28.4%
67699 speed-up	816450	**474132** 34.9%	1678787	**923592** 37.1%	2811135	**1756628** 37.5%
70853 speed-up	869592	**497743** 42.8%	1749131	**957072** 45.2%	2916507	**1818599** 37.6%

1.2 The `syndrome_gen` patch

The `syndrome_gen` is used in the decryption algorithm, and, as its name says, computes the set of $2 \times \delta$ syndroms of the received BCH codeword.

This function has been rewrited in order to take advantage of vectorization technics, and has been tested. The performance results are presented below :

```
PARAM_N1 : 766, VEC_N1_SIZE_BYTES : 96
500 tests computed with syndrome_gen and syndrome_gen2, all equal !
Chronométrage !
---------------
meanTimer syndrome_gen : 400209 et 0.117709ms
meanTimer syndrome_gen2 : 18101 et 0.005324ms

PARAM_N1 : 796, VEC_N1_SIZE_BYTES : 100
500 tests computed with syndrome_gen and syndrome_gen2, all equal !
Chronométrage !
---------------
meanTimer syndrome_gen : 533545 et 0.156925ms
meanTimer syndrome_gen2 : 21805 et 0.006413ms
```

We implemented a patch using our vectorized `syndrome_gen` function. The performances are presented table 3

1.3 Patch cumulation

We tested the cumulation of both patches. Although this cumulation only improves the decryption function, we present all the results including key generation and encryption. The results are presented table 4.

Tab. 3: Performance comparison with vectorized `syndrome_gen` function patch, # clock cycles

	decryption	
algorithm/ PARAM_N (bits)	HQC origin	HQC with patch
BASIC		
24677 speed-up	1085102	**832015** 23.3%
ADVANCED		
43669 speed-up	1783812	**1555275** 12.8%
46747 speed-up	1837189	**1600054** 9.3%
PARANOIAC		
63587 speed-up	2658767	**2442694** 8.1%
67699 speed-up	2811135	**2562646** 8.8%
70853 speed-up	2916507	**2682426** 8.1%

Tab. 4: HQC Performance comparison without and with both multiplication and syndrome computation patches, # clock cycles

	keygen		encryption		decryption	
algorithm/ PARAM_N (bits)	HQC origin	HQC with patch	HQC origin	HQC with patch	HQC origin	HQC with patch
BASIC						
24677 speed-up	238628	**155351** 35.1%	505886	**334838** 33.7%	1085102	**593841** 45.2%
ADVANCED						
43669 speed-up	478133	**290251** 39.3%	984790	**574678** 41.4%	1783812	**980663** 45.0%
46747 speed-up	503797	**306005** 39.3%	1037276	**609119** 41.2%	1837189	**986286** 46.3%
PARANOIAC						
63587 speed-up	783288	**450261** 42.5%	1584447	**883910** 44.2%	2658767	**1454480** 45.3%
67699 speed-up	816450	**474132** 34.9%	1678787	**923592** 37.1%	2811135	**1527811** 45.6%
70853 speed-up	869592	**497743** 42.8%	1749131	**957072** 45.2%	2916507	**1574772** 46.0%

2 PATCH version 3

The main modifications deal with the reduction $(X^N - 1)$ (modification on the final masking) and the tests are performed on a new platform which is now :

- the platform is now a Dell Precision 3530;
- Intel 8$^{\text{th}}$ generation processor :
  - `model name : Intel(R) Core(TM) i5-8400H CPU @ 2.50GHz`
  - `cache size : 8192 KB`
- the source code is compiled with `gcc (Ubuntu 7.4.0-1ubuntu1 18.04.1) 7.4.0`
- he *Turbo-Boost*®is deactivated during the tests.

In addition, this version of the patch has been tested using the KAT tests provided in the HQC submission (see [1]), in order to check the correctness of the calculation. The `PQCkemKATxxxx` has been pairwise compared, and the results are equals for all datasets.

This section mainly presents the update of the performance experimentation results: subsection deals with the fast convolution multiplication, subsection 2.2 presents the tests for the multiplication patch, subsection 2.3 the tests for the syndrom patch, and subsection 2.4 the patch cumulation.

2.1 FastConvolution version 3

Table 5 are the test results on the new platform.

Tab. 5: Performance comparison between multiplications, Intel 8$^{\text{th}}$ generation processor

algorithm/ op. size (bits)	Optimized HQC `sparse_dense_mult`	`fastConvolution WithRed`	Optimized HQC `sparse_dense_mult`	`fastConvolution WithRed`
BASIC				
	w = 67		w = 77	
24677	81494	22344	87698	25678
ADVANCED				
	w = 101		w = 117	
43669	177586	56480	194688	65402
46747	189562	59990	207888	69270
PARANOIAC				
	w = 133		w = 153	
63587	305118	105620	336138	121348
67699	324002	111956	357126	129282
70853	338760	117728	372918	135026

The performance is now about 64 % (PARANOIAC-III, $\omega = 153$) to nearly 73 % (BASIC, $\omega = 67$).

2.2 Patch with FastConvolution version 3

Table 6 presents the test results on the new platform.

Tab. 6: HQC Performance comparison without and with vectorized fast convolution multiplication patch on the new platform with Intel 8^{th}, # clock cycles

	keygen		encryption		decryption	
algorithm/ PARAM_N (bits)	HQC origin	HQC with patch	HQC origin	HQC with patch	HQC origin	HQC with patch
BASIC						
24677 speed-up	203391	**137002** 32.5%	432944	**299190** 30.9%	926563	**732291** 21.0%
ADVANCED						
43669 speed-up	392523	**257535** 34.3%	804541	**521544** 35.1%	1475946	**1064522** 27.8%
46747 speed-up	417655	**270789** 35.1%	850973	**549459** 35.3%	1498599	**1062802** 29.1%
PARANOIAC						
63587 speed-up	614957	**395800** 35.6%	1251156	**784782** 37.2%	2119301	**1454984** 31.3%
67699 speed-up	648822	**412670** 36.3%	1322885	**820829** 37.9%	2233020	**1508916** 32.4%
70853 speed-up	670848	**432448** 35.6%	1364479	**854091** 37.3%	2287196	**1563462** 31.7%

2.3 Patch with `syndrome_gen2`

The `syndrome_gen2` function is unchanged from patch version 2.

```
PARAM_N1 : 766, VEC_N1_SIZE_BYTES : 96
500 tests computed with syndrome_gen and syndrome_gen2, all equals !
Chronométrage !
-----------------------------------------------------------------------------------------------------
meanTimer syndrome_gen : 406919 et 0.119682ms
meanTimer syndrome_gen2 : 21856 et 0.006428ms

PARAM_N1 : 796, VEC_N1_SIZE_BYTES : 100
500 tests computed with syndrome_gen and syndrome_gen2, all equals !
Chronométrage !
-----------------------------------------------------------------------------------------------------
meanTimer syndrome_gen : 447492 et 0.131615ms
meanTimer syndrome_gen2 : 21117 et 0.006211ms
```

We tested this patch on the new platform. The performances are presented table 7.

2.4 Patch cumulation version 3

Table 8 presents the test results on the new platform. As for the former version of the patches, we recapitulate all the results.

3 Conclusion and future work

The tests on both platforms shows a similar improvement in terms of performances, in comparison with the source code published in [1] for both patches cumulation:

- On the Intel 4^{th} generation, the improvement is about 45-46 %
- On the Intel 8^{th} generation, the improvement is about 40-43 %

The `repetition_code_decode` function is the next to be vectorized.

Tab. 7: Performance comparison with vectorized `syndrome_gen` function patch on the new platform with Intel 8^{th}, # clock cycles

	decryption	
algorithm/ `PARAM_N` (bits)	`HQC origin`	`HQC with patch`
BASIC		
24677 speed-up	926563	**722236** 22.1%
ADVANCED		
43669 speed-up	1475946	**1259253** 14.7%
46747 speed-up	1498599	**1279999** 14.6%
PARANOIAC		
63587 speed-up	2119301	**1917560** 9.5%
67699 speed-up	2233020	**1994663** 10.6%
70853 speed-up	2287196	**2075867** 9.2%

Tab. 8: `HQC` Performance comparison without and with vectorized fast convolution multiplication patch on the new platform with Intel 8^{th}, # clock cycles

	keygen		encryption		decryption	
algorithm/ `PARAM_N` (bits)	`HQC origin`	`HQC with patch`	`HQC origin`	`HQC with patch`	`HQC origin`	`HQC with patch`
BASIC						
24677 speed-up	203391	**137002** 32.5%	432944	**299190** 30.9%	926563	**526298** 43.2%
ADVANCED						
43669 speed-up	392523	**257535** 34.3%	804541	**521544** 35.1%	1475946	**869507** 41.1%
46747 speed-up	417655	**270789** 35.1%	850973	**549459** 35.3%	1498599	**864054** 42.3%
PARANOIAC						
63587 speed-up	614957	**395800** 35.6%	1251156	**784782** 37.2%	2119301	**1270741** 40.0%
67699 speed-up	648822	**412670** 36.3%	1322885	**820829** 37.9%	2233020	**1305217** 41.6%
70853 speed-up	670848	**432448** 35.6%	1364479	**854091** 37.3%	2287196	**1344243** 41.2%

References

[1] 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.

[2] Michel Quercia and Paul Zimmermann. Irred-ntl patch. In *Irred-ntl source code*, 2003.

[3] Paul Zimmermann. Irred-ntl patch. In *ntl Library*, 2008.