Acceleration of Classic McEliece Post-Quantum Cryptosystem with Cache Processing

Cyrius Nugier, Vincent Migliore

Abstract—The NIST Post-Quantum Cryptography standardization process is in its fourth round, with a first KEM standard based on LWE and three candidates based on ECCs. These primitives implementation are designed to be optimal on classical hardware architecture targets. However, emerging architectures with Processing In Memory, made to be multi-purpose contrary to cryptographic co-processors, have proven their efficiency in multiple use-cases and show better overall computational speed. In this paper, we show that the Classic McEliece performance can be improved on PIM architectures. Notably, the public key generation benefits of a 12.6x speed-up on architectures with bit-line operations. We also describe the open-source RISC-V simulator specifically developed for our experiments, including both in-cache and vectored operations. We discuss how these architecture changes may open the possibility of redesigning primitives or parameter sets for better efficiency.

Index Terms—Post Quantum Cryptography, RISC-V, Processing In Memory, NIST PQC Standardization

1 Introduction

One of the major challenges in the field of cryptography is the design of primitives not vulnerable to a quantum machine. Shor's and Grover's quantum algorithms are known to significantly reduce the security level of existing schemes, especially key exchange standards (RSA, DH and ECDH) since they rely on the hardness of factorisation and discrete logarithm problems. The identification of alternatives to the standard has recently gained momentum with the NIST standardization process to select the best quantum computer resistant primitives known as Post-Quantum Cryptography (PQC).

The first phase of the competition focused on the security assessment of the candidates. Next, the focus was on performance evaluation. Candidates are ranked on their performances on target platforms, specifically a general purpose processor and an ARM-type micro-controller for software evaluation, and an FPGA for hardware accelerators [1]. At this point, a first Key Encapsulation Mechanism (KEM) has been standardized (*CRYSTALS-KYBER*), and remaining ones are pushed into an additional round of competition [2].

This standardization effort will include additional algorithms in the standard for two reasons: first, quantum computing is an emerging research field, the proposed primitives might be broken in the upcoming years. Second, the proposed primitives have very different characteristics, and which is best depends on the target scenario.

The remaining candidates for KEM standardization are code-based (*Classic McEliece*, *BIKE*, *HQC*) and isogeny-based (*SIKE*). Out of these, Classic McEliece is considered one of the most reliable candidates. Introduced in 1978 as a competitor of RSA, its security has not been significantly reduced since then. Also, it has the shortest ciphertext size, which reduces bandwidth over time. However, it has an overly large public key and key generation time. BIKE and HQC are based on structured codes which improve performances, but the impact on security of the code structure is not well known.

Classic McEliece also displays a high level of arithmetic parallelism which would make it better on emerging parallel architectures. Among these, Processing In Memory (PIM) architectures redesign memory spaces such as caches or RAM to perform computations between operands of a much larger size than those of a processor's ALU, for a minimal additional die surface. An example is the technology called "bit-line computing". It allows to modify SRAMs by adding vertical sensors and horizontal activators. When two rows are activated, the bit-line sensors produce the results of a `nor` and a `and` operation and store them in a line which is destination-enabled. The robustness of this process is proven with high industry standards [3].

Experiments confirm the efficiency and robustness of this adaptation [4]. It enables computation in cache for only 8% of additional area. The cache size increases the level of parallelism beyond vector acceleration, with operands that can reach 8KB. The Compute Cache architecture [4] enables all basic binary operations, including copy and compare. This allows for smarter data management, lower cost copying, moving and searching through big amounts of data, and was more recently extended to machine learning applications [5], [6].

Cryptographic acceleration is often based on custom hardware co-processor development, however these are specialised and represent high costs for a single task. The architecture with general performance improvements which best fits cryptographic acceleration seems to be processing in cache: in many primitives, arithmetic and logic operators are fairly varied, which means that interactions with the processor are frequent. Additionally, caches are now fully integrated into modern processors.

Unfortunately the current standardization does not account for these emerging architectures, leading to a lack of oversight on the efficiency that post-quantum cryptographic algorithms might have once in use. Considering such archi-

• *CN, VM: LAAS-CNRS, INSA Toulouse, Université de Toulouse, France. E-mail: cnugier@laas.fr, vmiglior@laas.fr*

tectures in the discussion is of clear interest for a fair study of long-term performance.

In this paper, we propose the first acceleration of the Post-Quantum primitive Classic McEliece in a Processing In Memory architecture. For comparison purposes, we provide results with vector operations. Experiments were performed with a custom RISC-V simulator that we made available [7], which incorporates a vectorization modules, a cache with PIM capacities, and custom hardware performance counters to analyze behavior of the processor core and cache. To the best of our knowledge, this is the first simulator which can evaluate both vectorization and cache operations.

Our results show that compared to a classical implementation, the computation times for public key generation can be sped-up 12.6×, and the computation time for encryption by 4× with the standard algorithm and up to 63× if the public key is stored in transposed form.

The paper is structured as follows: section 2 presents how classical architectures can be adapted for cache computation. Section 3 presents the Classic McEliece algorithms and how to accelerate them with cache computation. Section 4 presents our open-source simulator experimental results. Section 5 draws conclusions.

2 Architecture adaptation for Processing in Cache

The memory cell adaptation that allows for computation in SRAM elements is introduced in [3]. The main design change is the introduction of vertical sensors over the cells. Combined with horizontal activators, they allow to read the results of a `and` and a `not` and write the results on enabled cells. This approach was validated for reliability on on multiple 28nm CMOS test chips.

By using this technology on a cache and combining the gates into column peripherals at the extremities of the sensors, [4] implemented a variety of operations. Vocabulary used in this paper is similar to [4], and illustrated in figure 1. Experimentation in this paper is based on simulations of this architecture, which is described in the rest of this section.

One big challenge is data mapping in cache which is addressed by modifying the address structure. Usually, address structure is as follows: `| TAG | SET | OFFSET |`. For cache operations, field `SET` is subdivided to regroup cache lines sharing their bit-lines (sensors). Therefore, address structure for cache computation is as follows: `| TAG | SET IN BP | BANK | BP | OFFSET |`. This way, two bits are cache-aligned if their respective bytes addresses share the same `BANK`, `BP`, and `OFFSET` fields, and they share the same position in their byte. The length of the fields depends on the geometry of the cache.

In [4], cache computation operations are bit-wise binary operations over cache-aligned bits. Called bit-line operations, they are applied to full cache lines in the same block partitions (sharing vertical sensors). The result is stored in a destination-enabled cache line of the same block partition, or in a register. The same bit-line operation can be undertaken in all BPs at the same time. We consider the following operations:

```
CCPY(addr_dst, addr_a, n_lines)
CCMP(ret, addr_a, addr_b, nb_lines)
```

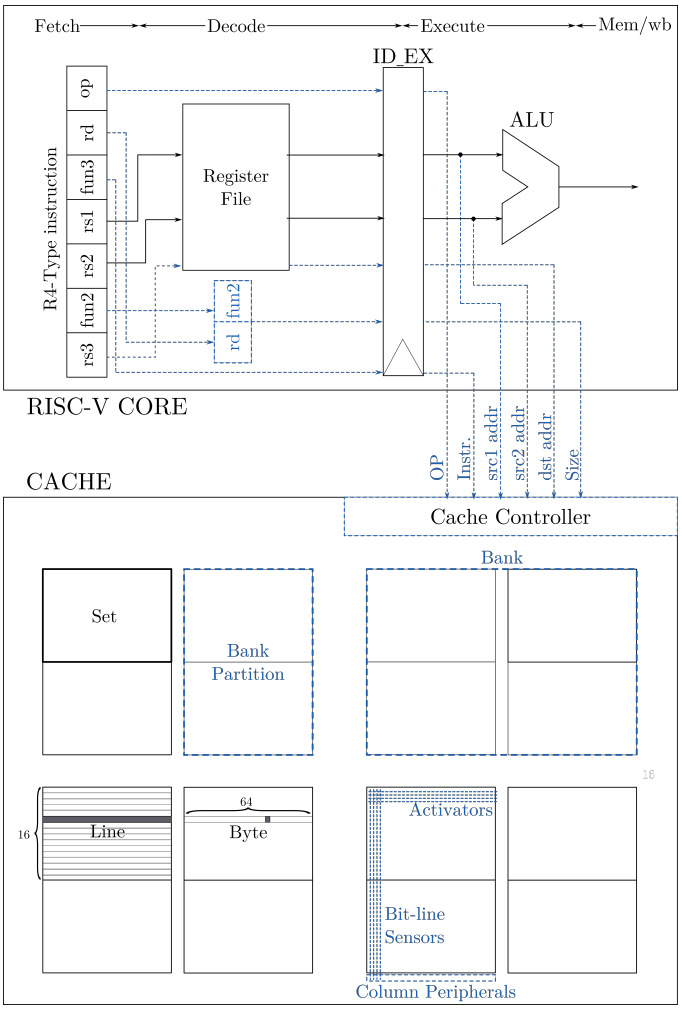


Fig. 1: Architecture of the proposed solution. Elements and notations in black are from classical RISC-V and cache architectures. Elements in blue dotted lines are modifications made to allow bit-line computing. Cache geometry notations in blue follow [4].

```
CAND(addr_dst, addr_a, addr_b, n_lines)
COR(addr_dst, addr_a, addr_b, n_lines)
CXOR(addr_dst, addr_a, addr_b, n_lines)
CNOT(addr_dst, addr_a, n_lines)
CSET(addr_dst, val, n_lines)
```

The `CSET` operation fills the lines with zeros if `val` $= 0$, with ones. The number of lines indicates the number of matching bits in the addresses of the operands. It only differs from [4]'s `cc_buz` by a conditional `not` gate

Bit-Array operations are compositions of bit-line operations that allows a function to be performed in parallel on elements of arbitrary size. This means that all bits of each element are cache-aligned (elements are stored "vertically"). The number of bit-line operations is the number of binary gates for the function computation for one element. For example:

```
BA_ADD(addr_dst,addr_a,addr_b,size,n_lines)
BA_MUL(addr_dst,addr_a,addr_b,size,n_lines)
```

The achievable levels of parallelism yields the best throughput in bit-array operations with small binary com-

plexities and a large number of operands multiple of the cache line size. On the contrary, float operations binary representations grow exponentially with their precision, so they are badly adapted for high-precision computation.

To benefit more from this computation method, SRAMs with two-dimensional data access called Transpose Gateway Units (TGU), enable the use of both bit-parallel operations (on horizontal data) and bit-serial operations (vertical data) at runtime with minimal efficiency loss. Chips integrating these technologies and bit-serial operators can be found in [8].

The core also requires changes to support cache instructions. Notably, for a RISC-V core, the modification of the register file to support reading three registers in parallel, i.e. two for the source addresses and one for the destination address. Note that no modifications are needed for cores supporting floating point arithmetic since they already implement that feature. To ease the implementation, we reused the R4-type instructions (i.e. floating point instructions) since the three register fields are already present, and adapted the other fields to our needs: `funct3` encodes the instruction type, and `rd` plus `funct2` encodes the number of cache lines processed. These changes are illustrated in figure 1.

3 Design considerations for Classic McEliece acceleration

3.1 Classic McEliece Cryptosystem

Classic McEliece [9] is based on Error Correcting Codes. Their original purpose is to encode data and transmit it on a noisy channel, allowing the receiver to remove the errors to get the correct message. If the decoder is kept secret and cannot be deduced from the encoder, it makes encoding with errors a one-way trapdoor function: the sender encodes with the public encoder and adds as many errors as the decoder can remove. The receiver with the decoder is then the only one who can remove the errors and read the message. Classic McEliece was designed based on Goppa codes for their fast decoding algorithm. Until now, this cryptosystem remains unbroken.

KeyGen:

1) `sk_gen()`: The secret key is generated by picking a polynomial $g \in \mathbb{F}_q[x]$ where $q = 2^m$ with $m = 12$ or 13 according to the parameter set, n elements $\alpha_1 \cdot \alpha_n$ of $\mathbb{F}_q$ and a string $s \in \mathbb{F}_2^n$
2) `pk_gen()`: The public key is generated as follows:
   a) **Initialization**: Sorts the elements of the secret key to get the support L.
   b) **Roots**: Applies the polynomial g to each element of L.
   c) **Inversion**: Finds the inverse of $g(L)$ and stores it in inv.
   d) **Zeroing**: Zeroes the matrix mat.
   e) **Filling**: Does t times the following:
      - Transpose every successive 8 elements of inv (m bits) onto m bytes in mat so the $\mathbb{F}_q$ elements are vertically stored.
      - Set inv to be the coefficient-wise multiplication of inv and L
   f) **Gaussian**: mat is reduced to its systematic form (the left square is the identity matrix) via Gaussian elimination. If this is not possible, restart KeyGen.
   g) **Copy**: copy the non-identity part of mat to bitpack it and output it as the public key.
3) `controlbits()`: This changes the representation of the α list in the secret key to improve decryption times. It uses a Beneš network to find pairwise swaps of $\mathbb{F}_q$ elements that put the α list as the first elements.

Encrypt:

1) `gen_e()`: Generates a error vector filled with zeros except in a precise number of random places.
2) `syndrome()`: Multiplies the public key matrix by the error vector error and returns the result (the syndrome) as the ciphertext.

Decrypt:

1) `support_gen()`: uses the control bits of the secret key to return the support L
2) `synd()`: From the Goppa polynomial g of the secret key, the support L and the ciphertext, outputs the syndrome s
3) `bm()`: the Berlekamp-Massey algorithm, outputs the minimal polynomial for s
4) `root()`: returns the roots of the polynomial over the support L. The errors vector has ones on the positions of these roots. If it has the correct weight it is the plaintext.

The bottleneck of KeyGen and `pk_gen()` is the Gaussian step. Comparatively, the generation of the secret key is negligible. In our experiments, the Gaussian elimination represented 94.4% of the public key generation (see Section 4). Also, matrix systematization may fail. To detect problems early, the diagonalization is first done on the left part of the matrix to ensure that there are no zeroes in the diagonal. Then, the right part is computed in case of success. Gaussian elimination is the best known algorithm, so optimisation comes from arithmetic operations. However, these operations are fundamentally parallelizable: elimination requires to compute identical operations on each line, which makes it excellent with tools such as the vectorization modules of processors (Intel AVX, ARM NEON, ...).

3.2 Accelerating Classic McEliece in cache

3.2.1 Security parameters and operands size

Classic McEliece primitive provides 5 parameter sets to fit various security levels, summarized in Table 1. Since matrix size varies considerably from one set to another, a discussion on how efficiently store and process it is essential. Normally, the matrix is bit-packed, i.e. written contiguously in memory. However this representation limits bit-line operations.

The full public key matrix might be too large to fit into a L1 cache. However, computations are performed row by row, which requires a fraction of a typical cache. For

instance, for a 16 Kb LD1 cache (256 lines of 64 bytes), the largest matrix uses 6.25 % of the memory for a full row (16 cache lines).

To compute a full row in one operation (for maximum parallelism), the number of lines computed in parallel with PIM varies from 7 to 16 depending on the parameter set. This is typically the range of parallelism achieved with existing implementations [4] (one line per BP in parallel).

Therefore, some configurations underuse hardware resources because parameters sets were not designed to optimize cache operations. However, a good balance between performances and resource utilisation is achieved with an 8-line parallelism. This disadvantages parameter set 460896 which requires 9 lines per row, but optimizes parameter set 8192128 and is nearly optimal for all others, with some row padding. Speed-up provided by PIM encourages revising parameter sets to let row sizes fit the parallelism provided by cache computations.

Parameter Set	Matrix dimension		Cache lines used		PK size (bits)	Secu. Cat.
	Rows	Cols	Row	Matrix		
348864	768	3488	7	5 232	2 088 960	1
460896	1248	4608	9	11 232	4 193 280	3
6688128	1664	6688	14	21 736	8 359 936	5
6960119	1508	6960	14	20 500	8 378 552	5
8192128	1664	8192	16	26 624	10 862 736	5

TABLE 1: Parameter sets for Classic McEliece and consequences on the number of cache lines used (64-byte lines).

Note: High-end multi-core processors are often equipped with a large L3 cache (10MB and above) that can store the entire matrix. This should improve computation time by exploiting the L3 cache prefetching to reduce the RAM access overhead. In this paper, we target single core processors with L1 caches, but it would be interesting to extend this work to more complex architectures.

3.2.2 KeyGen

First, `sk_gen` cannot be particularly sped-up with PIM since it is mainly composed of a seed expansion and operations between $\mathbb{F}_q$ elements which do not require larger operands. Fortunately, `sk_gen` represents negligible computation times in KeyGen.

In `pk_gen`, we change the representation of mat to enable computation in cache: the matrix is not bit-packed but padded with zeroes to keep rows aligned in memory.

Initialization, Roots, and **Inversion:**

These steps create the first row of the matrix of $\mathbb{F}_q$ elements: inv. They consist in a lot of $\mathbb{F}_q$ operations which are best performed horizontally. They computationally represent a negligible part of `pk_gen` (confirmed in our experiments, figure 2c). We do not focus on optimizing them.

Zeroing:

Zeroing is straightforward with computations in cache, reducing the number of operations by a factor of $r/(l_s \times n)$, where l_s is a cache line size, n the number of lines computed in parallel, and r the register size.

```
for (i = 0; i < PK_NROWS; i++)
for (j = 0; j < SYS_N/8; j++)
  mat[i][j] = 0;
/* Can be optimised as */
for (i = 0; i < PK_NROWS; i++)
    cc_set(mat[i],0,8);
```

To do so, the matrix address has to be such that a row has the same `TAG` and `SET IN BP` fields. In C, `__attribute__((aligned (CACHE_SIZE)))` is used so start the row with the first set.

Filling:

As with the three first steps, we did not optimize the computation of all $\alpha_j^{(i-1)}/g(\alpha_j)$, which requires iterative multiplications in $\mathbb{F}_q$ and cannot straightforwardly benefit from larger operands. This is not an issue since it represents only a small fraction of the computation times.

Gaussian:

The slowest step is the computation of the systematic form. Gaussian elimination consists in creating a mask and then xoring each pair of rows in the matrix. The mask is filled with ones or zeroes to compute the Gaussian elimination in constant time. Elimination is first done on the lower triangle to check if the matrix admits a systematic form, and once this is checked, elimination proceeds on the rest of the the matrix. It is a bottleneck because the complexity of the algorithm and its implementation differ of a factor n/r, where r the memory word size. Thus, creating operands large enough to contain a whole row removes this factor.

```
for (k = row + 1; k < PK_NROWS; k++){
  mask = mat[row][i] ^ mat[k][i];
  mask >>= j;
  mask &= 1;
  mask = -mask;

  for (c = 0; c < SYS_N/8; c++)
    mat[row][c] ^= mat[k][c] & mask;
}
/* Can be optimised as */
for (k = row + 1; k < PK_NROWS; k++){
  bit = mat[row][i] ^ mat[k][i];
  bit >>= j;
  bit &= 1;
  CSET(mask, bit, 8); //The mask is stored in 8
    cache lines instead of a byte
  CAND(aux, mask, mat[k], 8); //The auxiliary value
    is also stored in 8 cache lines
  CXOR(mat[row], mat[row], aux, 8);
}
```

The xoring of two rows is done in one instruction. The mask is no longer defined as an integer, but as a full row of 4096 ones or zeroes. This allows the code to remain constant-branch and constant-index. Additionally, diagonalizing only the left part of the matrix to check if the matrix is systematic is no longer useful, since operations on the whole rows are computed in the same time than the main square.

Problems arise in direct-mapping caches, in which two or more operands of a cache operation may share the same location in the cache. To avoid collisions, we reserve a fixed space in the cache for the row-sized variables (aux and $mask$), and change the addresses of the rows of the matrix so that they never map to these reserved spaces. This improves performances since in case of collision operations would be done classically, but also increases memory requirements by $2/$`cache_ways`.

Copying:

Increasing matrix size by padding does not slow down the computation. The final output is a byte string. To reduce its size, the normal algorithm ignores the identity part of the matrix, so bytes of the matrix are individually copied into

the output. When the matrix is padded with zeroes, these operation remains the same.

Overall, cache computation should give a net reduction in the number of arithmetic operations, and additional gains in data movement.

3.2.3 Encrypt

The error vector creation does not benefit from larger operands. It consists in randomness manipulation to fill a vector with zeroes, except in a specific number of places.

In the reference code, the public key is expanded to add the identity matrix to its left. Then, the error vector is multiplied the matrix to obtain the syndrome, which is the ciphertext. The multiplication is carried the classically, row by row, with byte by byte multiplication, accumulation, and parity giving the corresponding bit of the syndrome. We propose variations to improve performances:

NoId: The matrix does not need to be extended in systematic form. Instead, we copy the first bytes of the error vector in the result and accumulate from there. Time should be gained because of the shorter rows. We applied this modification to all the following variations.

With Cache Operations: Multiplications are performed on the entire row simultaneously, leaving accumulation and parity to be done bytewise. For this, the error vector and the rows have to be cache-aligned. This means the matrix rows have to be cache-aligned, which requires more space. To ensure that the full row can have simultaneous computation, we map the start of the row to the start of the first set (a row takes 2 to 4 sets according to the security parameters).

```
for (j = 0; j < SYS_N/8; j++)
  b ^= row[j] & e[j];
/* is replaced by */
CAND(row, row, e, PK_NROWS/LINE_SIZE +1);
for (j = 0; j < SYS_N/8; j++)
  b ^= row[j];
```

Transposed: Computing multiplications column by column instead of row by row makes the accumulation and parity operation combine. For this, the public key has to be in transposed form. For the i^{th} row of the transposed matrix, we set a byte according to the i^{th} bit of the error vector (255 if the bit is 1 otherwise 0), multiply and accumulate directly on the result.

Transposed with cache operations: To increase parallelism, we multiply more rows of the transposed matrix at once: in the smallest parameter set, rows are 768 bits and fit in 2 BPs. In a 8-BP cache, we can align four rows to the start of each BP pair, set the four multiplying elements separately for each, and then multiply and accumulate them all at once. In the end, the accumulators of each BP pair still need to be aggregated into one result.

Transposition during Encrypt may outweigh the time saved on matrix-vector multiplication. Therefore, if the efficiency gain is good enough, we could either "prepare" the public key in KeyGen the same way `controlbits()` prepares the secret key, or, since it only has to be done once, when the public key is received, before encryption. A Neural Cache-like[5] Architecture adaptation includes TGUs, making data transposition in cache faster and removing the involvement of the core. In these architectures public key transposition would be more desirable.

3.2.4 Decrypt

The decryption process retrieves the error code from the ciphertext. It finds the closest code-word to the ciphertext in the Goppa code defined by the secret key, calculates the difference with the ciphertext, and checks if the weight matches the expected one. TGUs would be very useful during decryption, since in the subroutine `support_gen`, uses iteratively transpositions to extract the support from the control bits. Hardware accelerations might also have visible results here, but bit-line computing would not be of much use except for memory management (zeroing of big variables).

4 Experimentation

4.1 RISC-V workbench

For the evaluation of the proposed optimizations, a suitable workbench implementing both the RISC-V core, the vector extension, a cache hierarchy and operations in cache is necessary. Finding a suitable simulator or hardware implementation is possible, even for the vector extension (now officially supported by the RISC-V Spike simulator for instance). However, for cache operations, to the best of our knowledge, no implementation exists at the moment.

Integrating cache operations into an existing tool would be difficult since cache operations must be integrated at the micro-architecture level to handle the specific management of cache geometry (cache sets, bank partitions), the management of bit-line operation conflicts, and the accuracy of data transfers time between the RAM and the cache.

Therefore, we designed our own simulator in C that implements a mono-core RISC-V with full RV32I and RV32M support, part of the vector extension (which covers the needs for Classic McEliece), a cache hierarchy (direct-mapped, fully configurable geometry) and the standard instruction set for cache operations (listed in [4]). For a better estimate of the number of cycles, we integrated the handling of some micro-architectural events (RTL), such as the management of control flow hazard, the memory controller between cache and RAM, and the internal micro-operations executed during cache instructions. The simulator is available on Gitlab [7]. We benchmarked it against a cycle-accurate well-known simulator Marss [10] on a AES workload. On average, an AES block required 4791 cycles for our simulator, and 4897 for Marss (2%). A difference was expected because Marss is a full-system simulator, which makes it run slowly in real time. Anyway, the results are similar enough to be confident on the results brought by our simulator.

To better understand the interaction between the RISC-V, the cache, and the RAM, we integrated relevant Hardware performance counters (HPCs) : Instruction based HPCs (number and nature of instructions executed), cycle based HPCs (global cycles, RISC-V cycles, cache cycles and control flow hazard cycles) and memory based HPCs (number of transfers between cache and ram, heap and stack usage and allocation). The complete list of features is available on the dedicated repository.

We followed a flexible approach for the hardware components interconnection to allow quick architecture configuration. For example, if the cache is not needed for

experimentation, the RISC-V can be directly connected to the RAM with a few lines of code. This allows to evaluate the performance of a hardware accelerator in a complete architecture. In fact, the cache computation controller is seen as an hardware accelerator in the code.

4.2 Implementation Results

We implemented three versions: *reference* (with no accelerators), *vector* (with vector operations), and *cache* (with cache computation acceleration). We adapted the AVX2 implementation proposed to the NIST, which is an Intel vector extension, to the RISC-V vector extension. Performances for Classic McEliece on various architectures can be found on [11], however to the best of our knowledge, none of them includes computing in cache.

Our results are presented for parameter set 348864 (see Table 1). We configured the simulator with 8 BPs to match the number of BPs in [4]. To study the impact of the cache size on computation times and memory, we varied the number of ways from 16 to 1024. Notice that for 1024 ways, the public key matrix fits entirely into the cache, which is similar to RAM computation.

4.2.1 KeyGen

Figure 2c provides an overview of the computation times and memory requirements across the subroutines of KeyGen (1024 ways cache, -O3 GCC optimisation flag) while Figure 2a shows the number of cycles of the Gaussian step for several number of cache ways. We plotted these as a function of the number of ways, taking 16 ways as the reference, for the three architectures, in Figure 2b.

The shape of the curves are very similar. In each configuration, the number of cycles saved by using vectorization is about 450 million, and another 300 million cycles are saved with cache computing. Therefore, these numbers seem to represent almost exclusively the change in arithmetical operations. Therefore, the speedup observed when increasing the ways count is mostly based on reduction of data transfer between cache and memory.

In a 64-way cache, 20% of computation times is due to data transfer, 38% for vector computing and 93% for in-memory computing. This explains why with 1024 ways the Gaussian step with cache computations has a $19\times$ speedup compared to vectorization and $46\times$ compared to reference. Therefore, data transfer has a similar impact across operation types, but becomes the limiting factor as arithmetical operations are faster.

Therefore, for small caches, the bottleneck is the time needed to move data to the cache. Also, while computing in small caches gives noticeable time saves, PIM in larger memories such as the RAM have a superior effect.

Figure 2c provides implementation results for arithmetical operations speed evaluation (i.e. 1024 ways, -O3 GCC optimisation flag). These parameters allows the whole public key matrix to fit in cache.

The Zeroing operation was sped-up $20\times$, and the Gaussian systemization by $46.4\times$. The overall `pk_gen` computation was $12.6\times$ faster. In `pk_gen` proportion of computation times, the Gaussian step drops from 94.4% to 25.7%. In counterpart the Filling step increases from 3% to 41.8%. Therefore, contrary to what previous knowledge suggested, it is no longer a negligible part of public key generation.

Its optimization would require redesigning the step to increase parallelizable operations. The operations undertaken are $\mathbb{F}_q$ multiplications as horizontal 12 or 13-bits elements, and transposing of the results into a vertical form. One could first transpose `inv` and the roots α_j (which is faster than transposing the whole matrix), and then do all multiplications vertically with bit-array operations. However, experiments are needed to confirm that the slowdown from the extra number of operations in bit-array multiplication is offset by the time saved by a $n = 2^q$ parallelism. An architecture like [5] allows these operations, and include TGUs, which would make the transpositions faster.

Optimisations to Zeroing had it stay negligible, 0.04% to 0.02% instead of 0.5%. This confirms the results of [4] showing that Processing In Memory and in cache gives overall multipurpose improvements. We observe low RAM usage in the Zeroing and Gaussian steps as all computation is performed in cache.

On the downside, `controlbits()` now represents 82.6% of KeyGen, it is a new bottleneck, the target of further optimizations. The `controlbits()` algorithm structure alternates between two types of operations. Ones could be parallelized if the operands were to be written vertically such as:

```
for (x = 0;x < n;++x) B[x]=(A[x]<<16)|(B[x]&0xffff);
```

Such operations would be very fast with cache computation. However, other operations require swapping multiple elements (such as sorting), which cannot be done efficiently with bit-line constraints.

Solutions are found in the literature: co-processors show good performance for permutation operations [12], or for more generic approaches, multi-dimensional access memory, such as TGUs. However the size of the data to transpose may be too large for these to provide good performances. Additionally, architectures with large multi-dimensional access do not seem realistic in the near future.

There is an impact on memory allocation requirements: the reference bit-packed matrix took 768×3488 bits, but our cache-aligned matrix takes $768 \times 8 \times 512$ bits. The theoretical increase for row length is 17.4%. However, constraints on the addresses of cache operands makes the compiler separate them from other variables and the measured increase is 79.7%. A more precise control over non-cache variables could reduce this drawback.

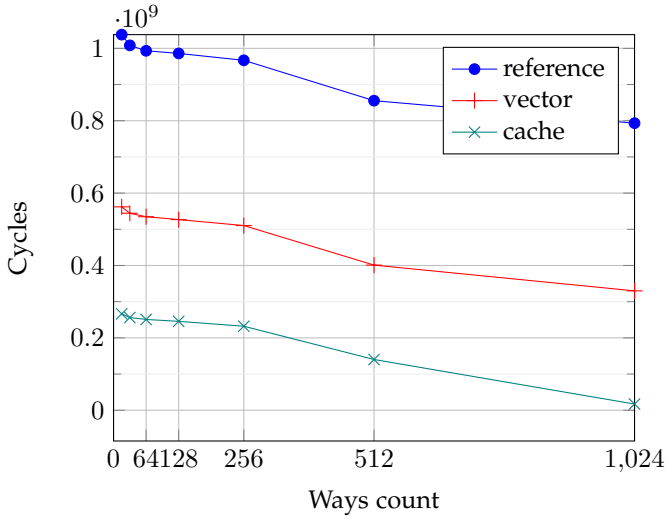
4.2.2 Encrypt

Results are presented in figure 2d. Removing the multiplication of the identity matrix removes 22% of computation times, matching the 22% reduction in row length. Computing in cache removes an additional 67%, and the product of the transposed matrix removes 64%. Computation in transposed form with cache computation is $47.6\times$ faster than NoId, $64\times$ faster than reference.

As KeyGen, the number of cycles saved by cache computation is stable at around 2.25 million cycles, and 700 000 additional cycles saved with transposition. The bottleneck remains cache fetching time, and efficiency increases with memory size.

Ways	Reference	Vector	Cache
16	1 037 890 479	562 045 422	266 384 330
32	1 008 175 343	544 386 638	256 151 754
64	993 256 239	534 758 990	250 866 890
128	985 981 839	526 796 846	245 638 090
256	966 934 895	510 378 350	232 160 458
512	855 361 103	401 017 326	140 247 338
1024	793 189 167	329 896 718	17 098 761

(a) Gaussian systematization performances for various cache sizes (in cycles). Optimisation flag -O3.



(b) Impact of changing the arithmetical operations technology and changing the number of lines in a set of cache.

Measure	Cycles			RAM Used (Bytes)			RAM Allocated (Bytes)		
Architecture	Reference	Vector	Cache	Reference	Vector	Cache	Reference	Vector	Cache
SKgen	26 683 052	26 683 052	26 683 052	42 549	42 549	42 549	51 042	51 042	51 042
Initialization	5 053 223	5 167 998	5 053 223	64 582	64 582	64 586	424 106	424 090	1 007 782
Roots	14 089 820	14 089 820	14 089 820	14 114	14 114	14 114	14 146	14 146	14 146
Inversion	2 005 620	2 005 620	2 005 620	7 100	7 100	7 100	14 238	14 238	14 238
Zeroing	335 772	338 076	16 614	334 848	334 852	0	334 850	390 922	0
Filling	25 421 134	25 448 910	27 876 551	348 804	348 804	348 800	381 726	390 926	440 068
Gaussian	793 189 167	329 896 718	17 098 761	334 860	344 069	73 728	381 726	390 934	392 802
Copy	456 603	458 908	508 421	522 244	522 248	552 244	691 554	700 746	1 142 442
PKgen	840 551 389	377 406 084	66 463 458	667 722	676 931	667 718	691 678	700 894	1 242 574
Controlbits	449 397 559	449 397 559	449 424 663	56 948	56 948	56 948	90 854	90 854	90 854
Keygen	1 316 632 015	853 486 710	542 756 873	691 375	700 584	733 323	691 662	700 584	1 242 558

(c) Key generation performances for three architectures and instruction sets: standard RISC-V, vector operations, and bit-line cache operations. Optimisation flag -O3.

Measure	Cycles	RAM Used (Bytes)	RAM Allocated (Bytes)
Reference	3 012 219	262 088	269 286
No Id	2 239 978	261 992	269 190
Cache	734 918	261 748	270 818
Transposed	796 970	349 208	350 426
Transposed + Cache	47 311	916	256 102

(d) Encryption performances for five syndrome algorithms. Optimisation flag -O3.

Ways	Reference	Cache	T. + Cache
16	3 195 547	872 646	212 008
32	3 169 211	845 350	211 048
64	3 156 475	832 230	210 856
128	3 150 395	825 990	210 664
256	3 144 251	819 686	210 568
512	3 120 571	818 854	182 600
1024	3 012 219	734 918	47 311

(e) Encryption performances various cache sizes (in cycles). Optimisation flag -O3.

This indicates that if no major reduction in `controlbits()` time is found by changing the architecture, the public key should be transposed and padded during KeyGen. Our simulation showed that even a very naive software transposition costs only 29 542 317 cycles. Even if the gain is only 687 607 cycles in the encryption, it is a 15.5× encrypt speedup for a 5.4% slowdown of the KeyGen. This modification should also be applied for classical architectures, in which bit-packing is possible.

5 Conclusion

The main takeaway of this paper is that efficiency of post-quantum cryptography will be influenced a positive side effect of generic-purpose hardware accelerators integration such as PIM.

This is the case for Classic McEliece, an encryption scheme known to have small ciphertexts but limited performances. Our results show that both KeyGen and Encryption can be improved with PIM.

For KeyGen, cache computations allowed an overall speedup of 12.6×. We learned that steps of the public key generation such as the matrix filling, might have to be reworked to accommodate these new architectures. For Encrypt, we shown that when the public key is given in transposed form, this type of architecture provides up to 28× speedup.

Additionally, it would be interesting to repeat the experiments with a k-associative cache instead of direct mapping, as well as with various replacement policies. Experiments should be extended with an accurate Neural Cache simulator to study the computational speedup of bit-array and TGUs. Also, FPGA implementation would increase the confidence in our results.

References

[1] "Round 2 of the nist pqc "competition" - what was nist thinking?" https://csrc.nist.gov/CSRC/media/Presentations/Round-2-of-the-NIST-PQC-Competition-What-was-NIST/images-media/pqcrypto-may2019-moody.pdf, accessed: 2021-08-01.

[2] NIST, "Round 3 candidates," https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions, 2021, accessed: 2021-04-28.

[3] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, 2016.

[4] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 481–492.

[5] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 383–396.

[6] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 397–410.

[7] V. Migliore and C. Nugier, "Risc-v simulator with pim operations," 2023. [Online]. Available: https://gitlab.laas.fr/vmiglior/risc-v-simulator

[8] J. Wang, X. Wang, C. Eckert, A. Subramaniyan, R. Das, D. Blaauw, and D. Sylvester, "A 28-nm compute sram with bit-serial logic/arithmetic operations for programmable in-memory vector computing," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 1, pp. 76–86, 2019.

[9] R. J. McEliece, "A public-key cryptosystem based on algebraic," *Coding Thv*, vol. 4244, pp. 114–116, 1978.

[10] G. Kothariand, "Marss-riscv: Micro-architectural system simulator for risc-v," 2020. [Online]. Available: https://github.com/bucaps/marss-riscv

[11] D. J. Bernstein and T. L. (editors), "ebacs: Ecrypt benchmarking of cryptographic systems," https://bench.cr.yp.to, Nov 2021.

[12] L. Kohútka and V. Stopjaková, "A new efficient sorting architecture for real-time systems," in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, Jun. 2017, pp. 1–4.

Cyrius Nugier is a third year PhD student, supervised by Vincent Migliore. His doctoral work explores the consequences of the transition from classical cryptography to post-quantum cryptography, at a theoretical and practical level. He holds an engineering degree in IT from ENSEEIHT, and a Master's degree in Algebra and Cryptography from University of Versailles. His research is supported by LAAS-CNRS where he is part of the Safety and Security team.

Vincent Migliore (PhD 2016) is an associate professor at INSA-Toulouse and member of the Safety and Security team. His work covers the acceleration of emerging post-quantum cryptographic algorithms (including homomorphic encryption), the design of secure implementations regarding side-channel attacks and the integration of low level security mechanisms. Dr. Migliore co-advised 3 PhD students and 2 Post-Doc. He is involved in research projects and partnerships with industrialists in the field of hardware and software security. For instance, he is currently the coordinator of the IDROMEL French project that addresses side-channel attacks at the micro-architecture level.