

Optimisation de codes correcteurs d'effacements par application de transformées polynomiales

Jonathan Detchart

18 février 2019



Plan de la présentation

1. Introduction et Contexte
2. Transformées polynomiales efficaces
3. Evaluation de performances
4. Conclusion

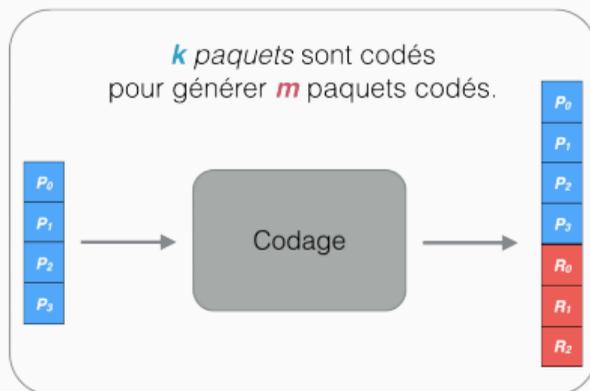
Introduction et Contexte

Que sont les *codes correcteurs d'effacements* et à quoi servent-ils ?

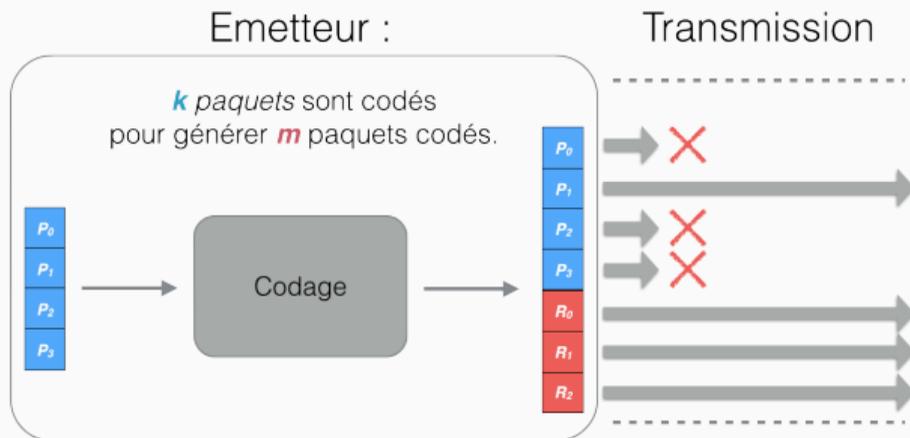
Les codes à effacement

Émetteur :

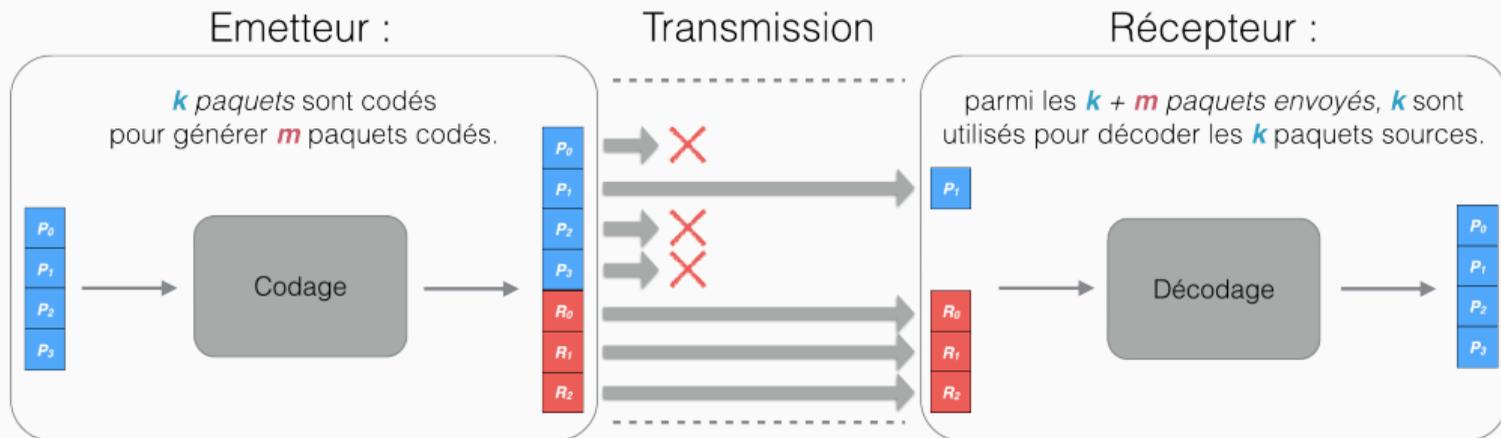
k paquets sont codés
pour générer m paquets codés.



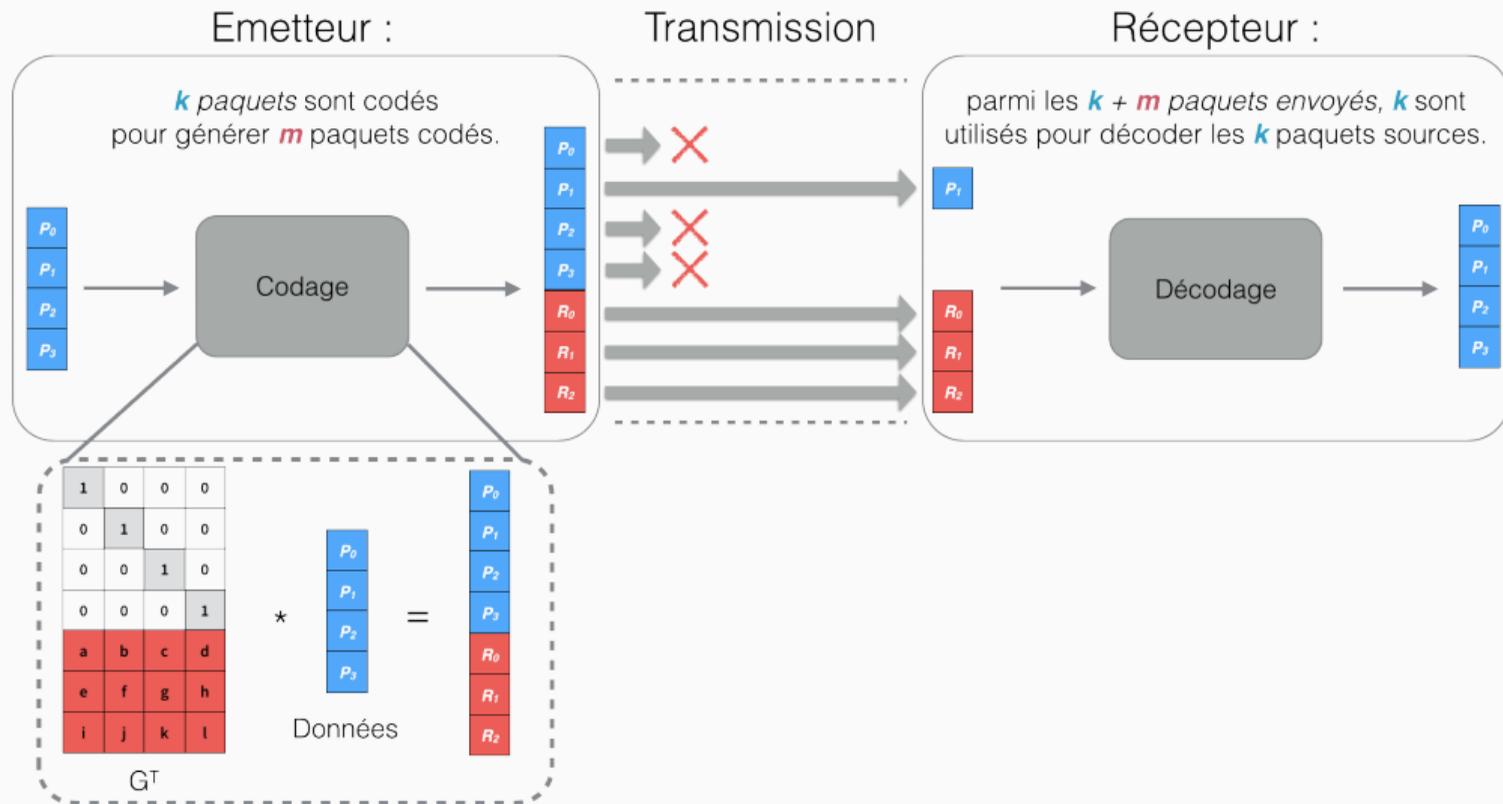
Les codes à effacement



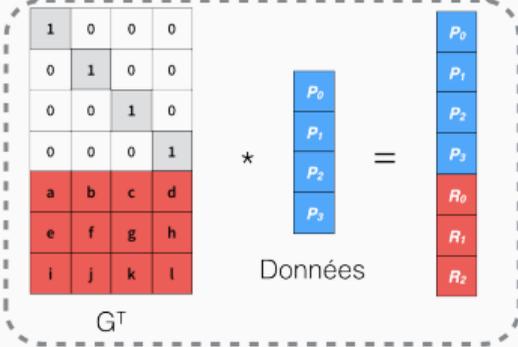
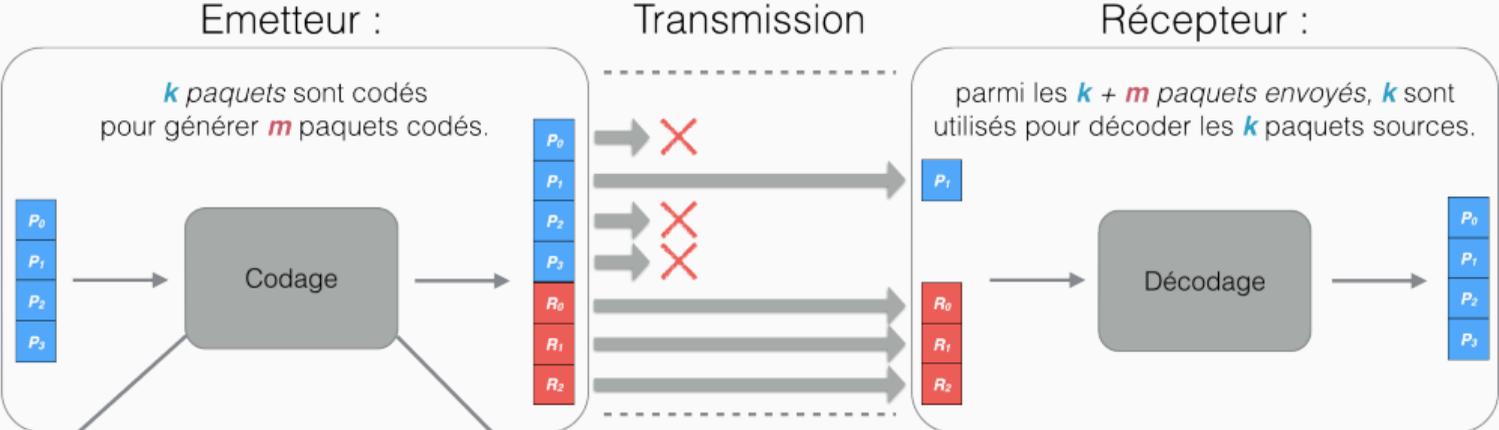
Les codes à effacement



Les codes à effacement



Les codes à effacement



$$R_0 = a * P_0 + b * P_1 + c * P_2 + d * P_3 \rightarrow \text{addmul}$$

$$R_1 = e * P_0 + f * P_1 + g * P_2 + h * P_3$$

$$R_2 = i * P_0 + j * P_1 + k * P_2 + l * P_3$$

* et + sont faites dans un **corps fini**

Décomposition de la multiplication

Sur \mathbb{F}_{2^w} , on décompose la multiplication en opérations dans \mathbb{F}_2 .

- Itoh et Tsujii, 1989 : *Structure of Parallel Multipliers for a Class of Fields GF(2^m)*
 - Implémentation hardware de la multiplication des corps finis
- Bloemer et al, 1995 : *An XOR-Based Erasure-Resilient Coding Scheme*
 - Implémentation software pour la transmission de données
- Blaum et Roth, 1999: *On lowest density MDS codes*
 - Propose une limite basse théorique en nombre de xor pour les codes MDS

Implémenter l'arithmétique des corps finis : méthode xor-based

Un élément du corps fini \mathbb{F}_{2^w} peut être représenté par une matrice binaire $w * w$:
 $a(x), c(x) \in \mathbb{F}_{2^4} = \mathbb{F}_2[X]/(x^4 + x + 1)$ avec $a(x) = 1 + x^2 + x^3$ et $c(x) = x + x^3$.

$$c(x) * a(x) = \begin{array}{|c|c|c|c|} \hline \text{white} & \text{red} & \text{white} & \text{red} \\ \hline \text{red} & \text{red} & \text{red} & \text{red} \\ \hline \text{white} & \text{red} & \text{red} & \text{red} \\ \hline \text{red} & \text{white} & \text{red} & \text{red} \\ \hline \end{array} * \begin{array}{|c|} \hline \text{red} \\ \hline \text{white} \\ \hline \text{red} \\ \hline \text{red} \\ \hline \end{array} = \begin{array}{|c|} \hline \text{red} \\ \hline \text{red} \\ \hline \text{white} \\ \hline \text{red} \\ \hline \end{array} = b(x)$$

- multiplication par $c(x)$: 12 *xors* dans $\mathbb{F}_2[X]/(x^4 + x + 1)$.
- Dans l'implémentation, il faut parcourir les w^2 bits un à un $\Rightarrow w^2$ conditions.

Le nombre de xor n'est pas uniquement déterminé par le nombre de monomes :

$$b(x) = x^3 + 1 : \begin{array}{|c|c|c|c|} \hline \text{red} & \text{red} & \text{white} & \text{white} \\ \hline \text{white} & \text{white} & \text{red} & \text{white} \\ \hline \text{white} & \text{white} & \text{white} & \text{red} \\ \hline \text{red} & \text{white} & \text{white} & \text{white} \\ \hline \end{array}$$

Les méthodes utilisant les tables précalculées

- L. Rizzo, 1997 : Effective Erasure Codes For Reliable Computer Communication Protocols
 - Implémentation d'un code à effacement sur \mathbb{F}_{2^w}

Multiplication par accès mémoires vers des tables précalculées :

Sur \mathbb{F}_{2^8} :

```
1 void addmul(uint8_t* dst, uint8_t* src, uint8_t e, int sz) {
2     for (int i = 0; i < sz; i++) {
3         dst[i] ^= multable[e][src[i]];
4     }
5 }
```

Sur \mathbb{F}_{2^w} , la table précalculée pour la multiplication contiendra $2^w * 2^w$ valeurs.

Nouvelles instructions

- 1999 : Intel introduit **Streaming SIMD Extensions (SSE)**
- 2006 : Supplemental Streaming SIMD Extensions 3 \Rightarrow instruction *pshufb*

- H. Li et Q. Huang, 2008 : Parallelized network coding with SIMD instruction sets
 - Implémentation de la multiplication dans \mathbb{F}_{2^8} avec l'instruction SSE *pshufb*
- H. P. Anvin, 2009 : The mathematics of RAID-6
 - Implémentation de l'arithmétique des corps finis pour RAID-6 (\mathbb{F}_{2^8})
- J. S. Plank, K. M. Greenan et E. L. Miller, 2013 : Screaming fast Galois Field arithmetic using Intel SIMD instructions
 - Implémentation de la multiplication dans \mathbb{F}_{2^w} avec $w \in 4, 8, 16, 32$ (avec *pshufb*)

Méthode la plus efficace pour implémenter l'arithmétique des corps finis dans un code correcteur d'effacement

Les opérations dans un corps fini sont complexes. Nous effectuons les opérations dans un anneau.

- En utilisant des transformées rapides, on plonge les éléments d'un corps fini dans un anneau plus grand
- Les opérations de multiplication dans un anneau sont plus rapide à effectuer
- On applique une transformée inverse pour revenir dans un corps fini

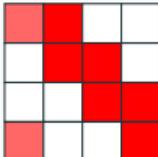
Opérations dans un anneau

Proposition :

Travailler dans un anneau polynomial $\mathbb{F}_2[x]/(x^n + 1)$

La multiplication est faite modulo $x^n + 1 \Rightarrow$ décalage cyclique

ex : $a(x) \in \mathbb{F}_2[x]/(x^4 + 1)$

$$a(x) = 1 + x^3 =$$


Le nombre de monome détermine le nombre de xor à effectuer !

Transformées polynomiales efficaces

*Comment transformer **efficacement** les éléments d'un corps fini en éléments d'un anneau et vice-versa ?*

Objectif : opérations de codage et décodage efficaces

$$\mathbb{F}_{2^w} : (\alpha_0, \dots, \alpha_{k-1}) \times \begin{pmatrix} \gamma_{0,0} & \cdots & \gamma_{0,n-1} \\ \vdots & \ddots & \vdots \\ \gamma_{k-1,0} & \cdots & \gamma_{k-1,n-1} \end{pmatrix}$$

$$R_{2,w+i} :$$

Objectif : opérations de codage et décodage efficaces

$$\mathbb{F}_{2^w} : \quad (\alpha_0, \dots, \alpha_{k-1}) \quad \times \quad \begin{pmatrix} \gamma_{0,0} & \cdots & \gamma_{0,n-1} \\ \vdots & \ddots & \vdots \\ \gamma_{k-1,0} & \cdots & \gamma_{k-1,n-1} \end{pmatrix}$$

↓

$$R_{2,w+i} :$$

Objectif : opérations de codage et décodage efficaces

$$\mathbb{F}_{2^w} : (\alpha_0, \dots, \alpha_{k-1}) \times \begin{pmatrix} \gamma_{0,0} & \dots & \gamma_{0,n-1} \\ \vdots & \ddots & \vdots \\ \gamma_{k-1,0} & \dots & \gamma_{k-1,n-1} \end{pmatrix}$$

↓

$$R_{2,w+i} : (a_0, \dots, a_{k-1})$$

Objectif : opérations de codage et décodage efficaces

$$\mathbb{F}_{2^w} : \quad (\alpha_0, \dots, \alpha_{k-1}) \quad \times \quad \begin{pmatrix} \gamma_{0,0} & \dots & \gamma_{0,n-1} \\ \vdots & \ddots & \vdots \\ \gamma_{k-1,0} & \dots & \gamma_{k-1,n-1} \end{pmatrix}$$

↓

↓

$$R_{2,w+i} : \quad (\mathbf{a}_0, \dots, \mathbf{a}_{k-1})$$

Objectif : opérations de codage et décodage efficaces

$$\begin{array}{ccc} \mathbb{F}_{2^w} : & (\alpha_0, \dots, \alpha_{k-1}) & \times \begin{pmatrix} \gamma_{0,0} & \dots & \gamma_{0,n-1} \\ \vdots & \ddots & \vdots \\ \gamma_{k-1,0} & \dots & \gamma_{k-1,n-1} \end{pmatrix} \\ & \Downarrow & \Downarrow \\ R_{2,w+i} : & (a_0, \dots, a_{k-1}) & \times \begin{pmatrix} g_{0,0} & \dots & g_{0,n-1} \\ \vdots & \ddots & \vdots \\ g_{k-1,0} & \dots & g_{k-1,n-1} \end{pmatrix} \end{array}$$

Objectif : opérations de codage et décodage efficaces

$$\begin{array}{ccc} \mathbb{F}_{2^w} : & (\alpha_0, \dots, \alpha_{k-1}) & \times \begin{pmatrix} \gamma_{0,0} & \cdots & \gamma_{0,n-1} \\ \vdots & \ddots & \vdots \\ \gamma_{k-1,0} & \cdots & \gamma_{k-1,n-1} \end{pmatrix} \\ & \Downarrow & \Downarrow \\ R_{2,w+i} : & (a_0, \dots, a_{k-1}) & \times \begin{pmatrix} g_{0,0} & \cdots & g_{0,n-1} \\ \vdots & \ddots & \vdots \\ g_{k-1,0} & \cdots & g_{k-1,n-1} \end{pmatrix} = (b_0, \dots, b_{n-1}) \end{array}$$

Objectif : opérations de codage et décodage efficaces

$$\begin{array}{ccc} \mathbb{F}_{2^w} : & (\alpha_0, \dots, \alpha_{k-1}) & \times \begin{pmatrix} \gamma_{0,0} & \dots & \gamma_{0,n-1} \\ \vdots & \ddots & \vdots \\ \gamma_{k-1,0} & \dots & \gamma_{k-1,n-1} \end{pmatrix} \\ & \Downarrow & \Downarrow \\ R_{2,w+i} : & (a_0, \dots, a_{k-1}) & \times \begin{pmatrix} g_{0,0} & \dots & g_{0,n-1} \\ \vdots & \ddots & \vdots \\ g_{k-1,0} & \dots & g_{k-1,n-1} \end{pmatrix} = (b_0, \dots, b_{n-1}) \end{array} \quad \Uparrow$$

Objectif : opérations de codage et décodage efficaces

$$\begin{array}{ccc} \mathbb{F}_{2^w} : & (\alpha_0, \dots, \alpha_{k-1}) \times \begin{pmatrix} \gamma_{0,0} & \cdots & \gamma_{0,n-1} \\ \vdots & \ddots & \vdots \\ \gamma_{k-1,0} & \cdots & \gamma_{k-1,n-1} \end{pmatrix} & = (\beta_0, \dots, \beta_{n-1}) \\ & \Downarrow & \Downarrow \\ R_{2,w+i} : & (a_0, \dots, a_{k-1}) \times \begin{pmatrix} g_{0,0} & \cdots & g_{0,n-1} \\ \vdots & \ddots & \vdots \\ g_{k-1,0} & \cdots & g_{k-1,n-1} \end{pmatrix} & = (b_0, \dots, b_{n-1}) \\ & & \Uparrow \end{array}$$

$$\mathbb{F}_{2^w} : (\alpha_0, \dots, \alpha_{k-1}) \times \begin{pmatrix} \gamma_{0,0} & \dots & \gamma_{0,n-1} \\ \vdots & \ddots & \vdots \\ \gamma_{k-1,0} & \dots & \gamma_{k-1,n-1} \end{pmatrix} = (\beta_0, \dots, \beta_{n-1})$$

\Downarrow *Emb* ou *Par*

\Downarrow *Sparse*

\Uparrow *Emb*⁻¹ ou *Par*⁻¹

$$R_{2,w+1} : (a_0, \dots, a_{k-1}) \times \begin{pmatrix} g_{0,0} & \dots & g_{0,n-1} \\ \vdots & \ddots & \vdots \\ g_{k-1,0} & \dots & g_{k-1,n-1} \end{pmatrix} = (b_0, \dots, b_{n-1})$$

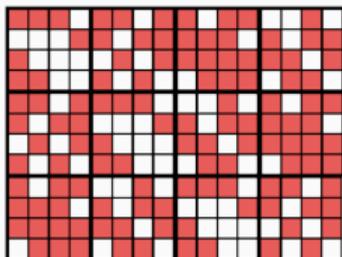
Quelles sont les opérations effectuées ?

matrice de Cauchy génératrice d'un code
systématique (7,4)

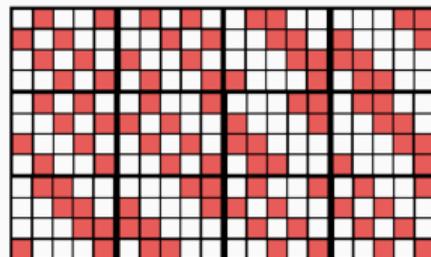
13	11	7	6
11	13	6	7
7	6	13	11

les éléments appartiennent au
corps \mathbb{F}_2^4 dans leur
représentation décimale :
13 représente $x^3 + x^2 + 1$

opérations xor-based dans le corps fini



opérations xor-based dans l'anneau



Evaluation de performances

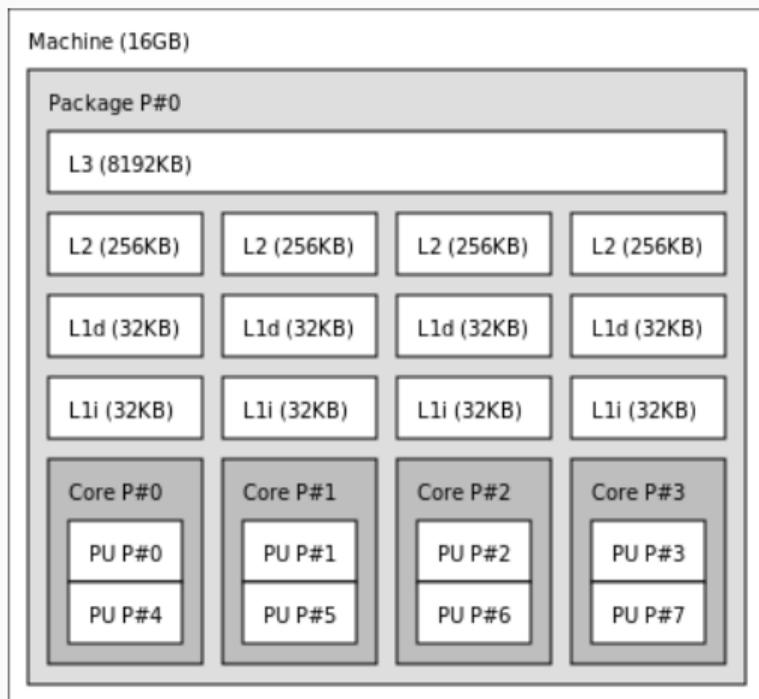
Implémentation d'un codec pour le code MDS sur \mathbb{F}_{2^4} et \mathbb{F}_{2^6} : Pyrit (PoLYnomial RiNg Transform)

- Codec utilisant des matrices de Cauchy (généralisées ou non)
- Calculs via les registres SIMD sur architecture Intel x86_64 (SSE, AVX) et ARM (Neon)
- Méthode Parity et Embedding selon les paramètres du code

Présentation des architectures testées

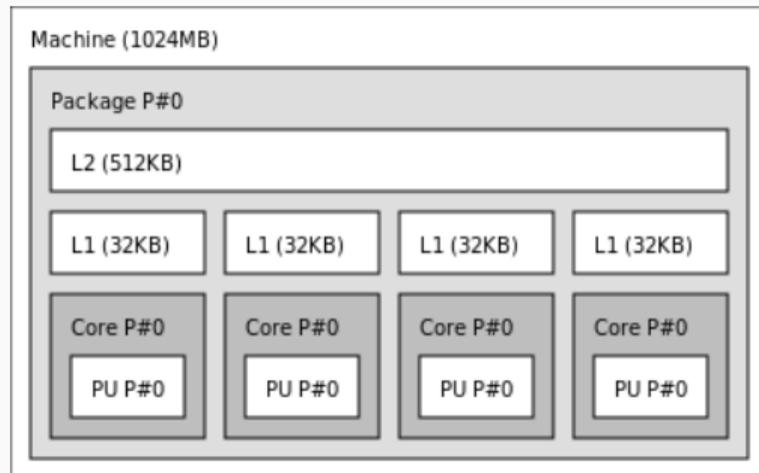
Machine x86 (Intel i7-6700) :

Comparaison de Pyrit avec ISA-L (Intel)

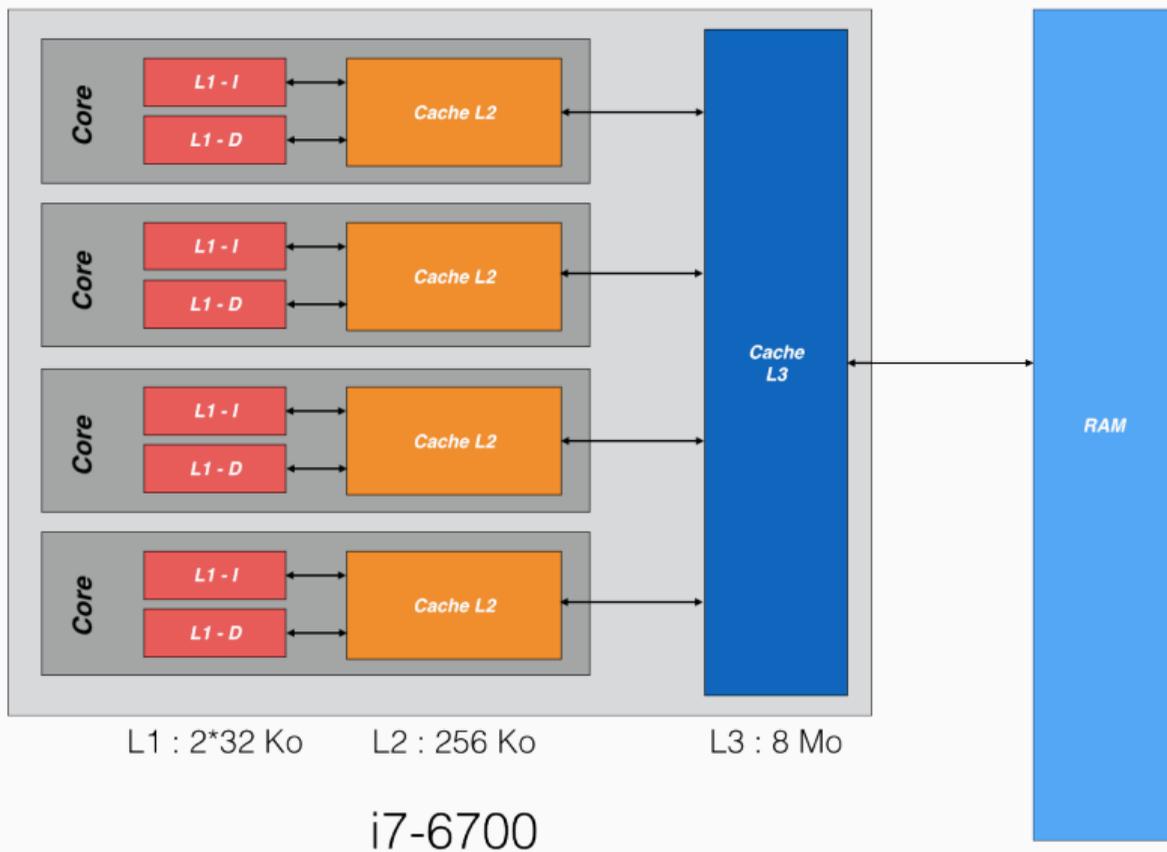


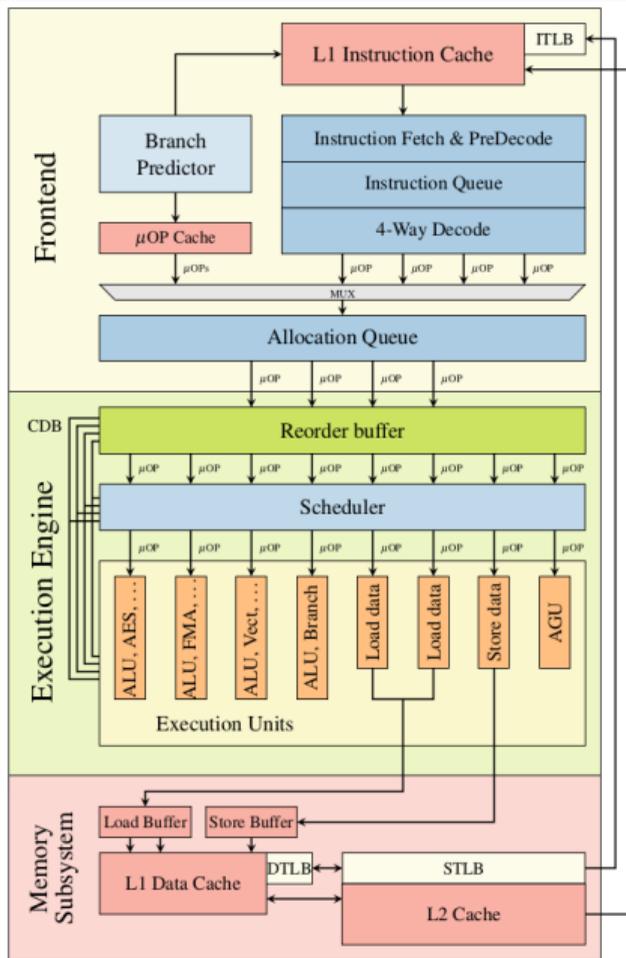
Raspberry Pi 3 (ARMv7) :

Comparaison de Pyrit avec Jerasure



Implémentation efficace





Implémentation efficace

```
1: function ADDMUL(dst,src,e,sz) 19:          r1 ← r1 xor r5          37:          r4 ← r4 xor r5
2:   for i = 0 to sz step 64 do 20:          r2 ← r2 xor r6          38:          r0 ← r0 xor r6
3:     r0 ← dst[i]                21:          r3 ← r3 xor r7          39:          r1 ← r1 xor r7
4:     r1 ← dst[i + 1 * 16]        22:          r4 ← r4 xor r8          40:          r2 ← r2 xor r8
5:     r2 ← dst[i + 2 * 16]        23:   end if                          41:   end if
6:     r3 ← dst[i + 3 * 16]        24:   if e & 4 then                42:
7:     r4 ← 0                        25:     r2 ← r2 xor r5            43:     r0 ← r0 xor r4
8:     r5 ← source[i]              26:     r3 ← r3 xor r6            44:     r1 ← r1 xor r4
9:     r6 ← source[i + 1 * 16]      27:     r4 ← r4 xor r7            45:     r2 ← r2 xor r4
10:    r7 ← source[i + 2 * 16]      28:     r0 ← r0 xor r8            46:     r3 ← r3 xor r4
11:    r8 ← source[i + 3 * 16]      29:   end if                          47:
12:    if e & 1 then                30:   if e & 8 then                48:     dst[i] ← r0
13:      r0 ← r0 xor r5              31:     r3 ← r3 xor r5            49:     dst[i + 1 * 16] ← r1
14:      r1 ← r1 xor r6              32:     r4 ← r4 xor r6            50:     dst[i + 2 * 16] ← r2
15:      r2 ← r2 xor r7              33:     r0 ← r0 xor r7            51:     dst[i + 3 * 16] ← r3
16:      r3 ← r3 xor r8              34:     r1 ← r1 xor r8            52:   end for
17:    end if                          35:   end if                          53: end function
18:    if e & 2 then                36:   if e & 16 then
```

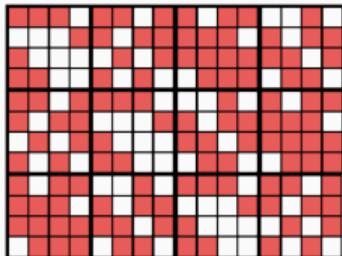
Opérations effectuées

matrice de Cauchy génératrice d'un code
systématique (7,4)

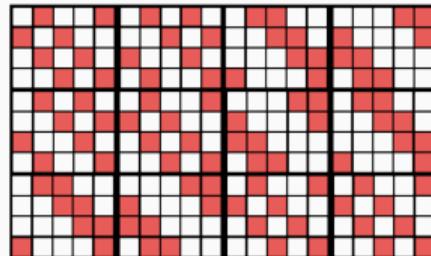
13	11	7	6
11	13	6	7
7	6	13	11

les éléments appartiennent au
corps \mathbb{F}_2^4 dans leur
représentation décimale :
13 représente $x^3 + x^2 + 1$

opérations xor-based dans le corps fini



opérations xor-based dans l'anneau



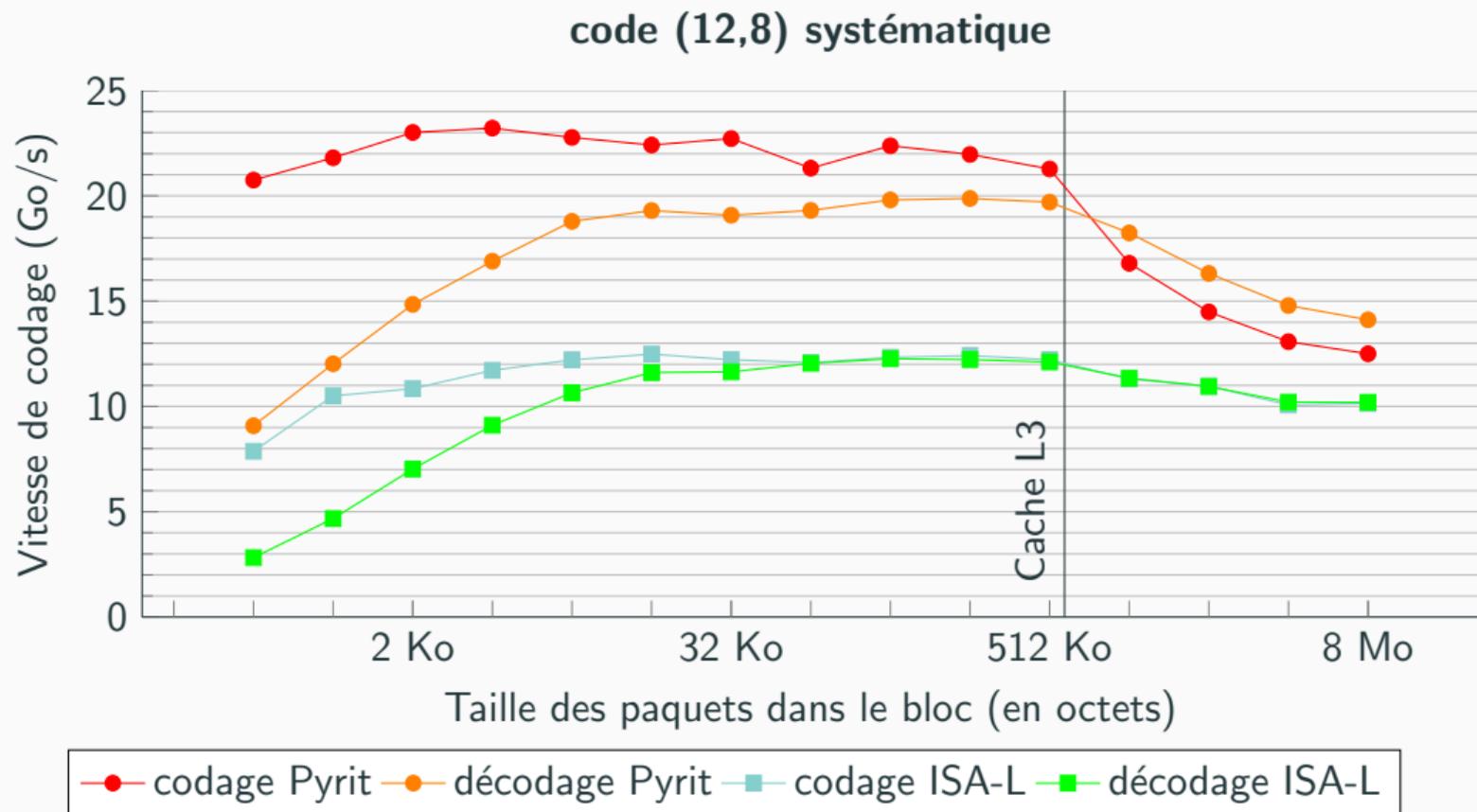
Présentation des codecs testés³

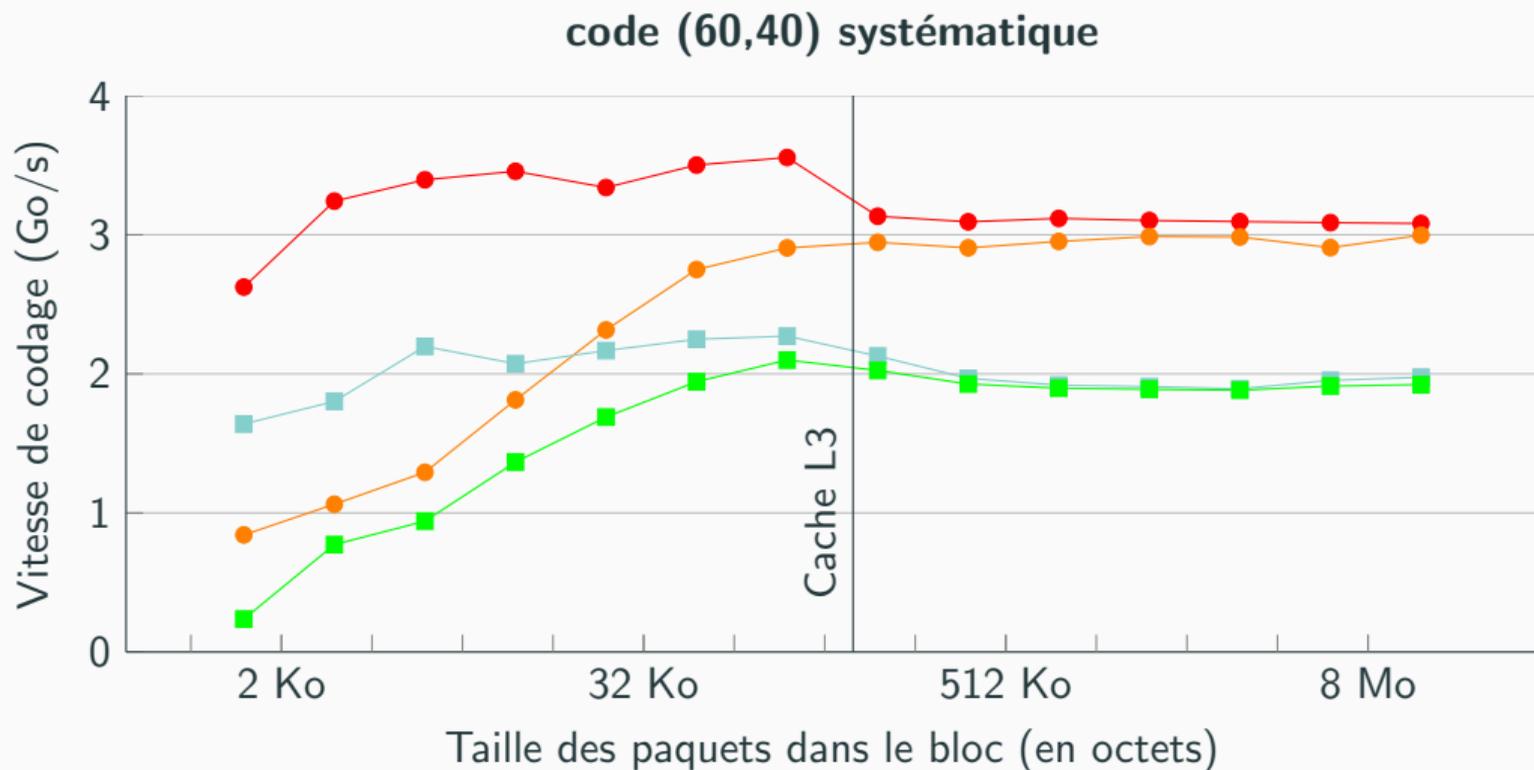
		<i>Pyrit</i>	<i>ISA-L</i>	<i>Jerasure</i>
Arch.	x86 (SSE, AVX)	✓	✓	✓
	ARM (Neon)	✓	-	✓
Lib.	Langage	C,asm	asm	C
	Lignes de code	18935	26884	18556
Corps Fini	\mathbb{F}_{2^4}	✓	-	✓
	\mathbb{F}_{2^6}	✓	-	-
	\mathbb{F}_{2^8}	✓	✓	✓

⇒ méthode *Split table* utilisée pour tous les cas testés

[3] ISA-L: <https://github.com/01org/isa-l>, Jerasure: <http://jerasure.org/>

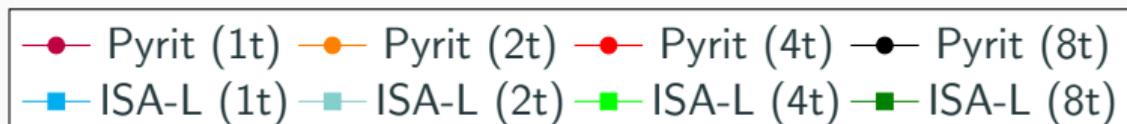
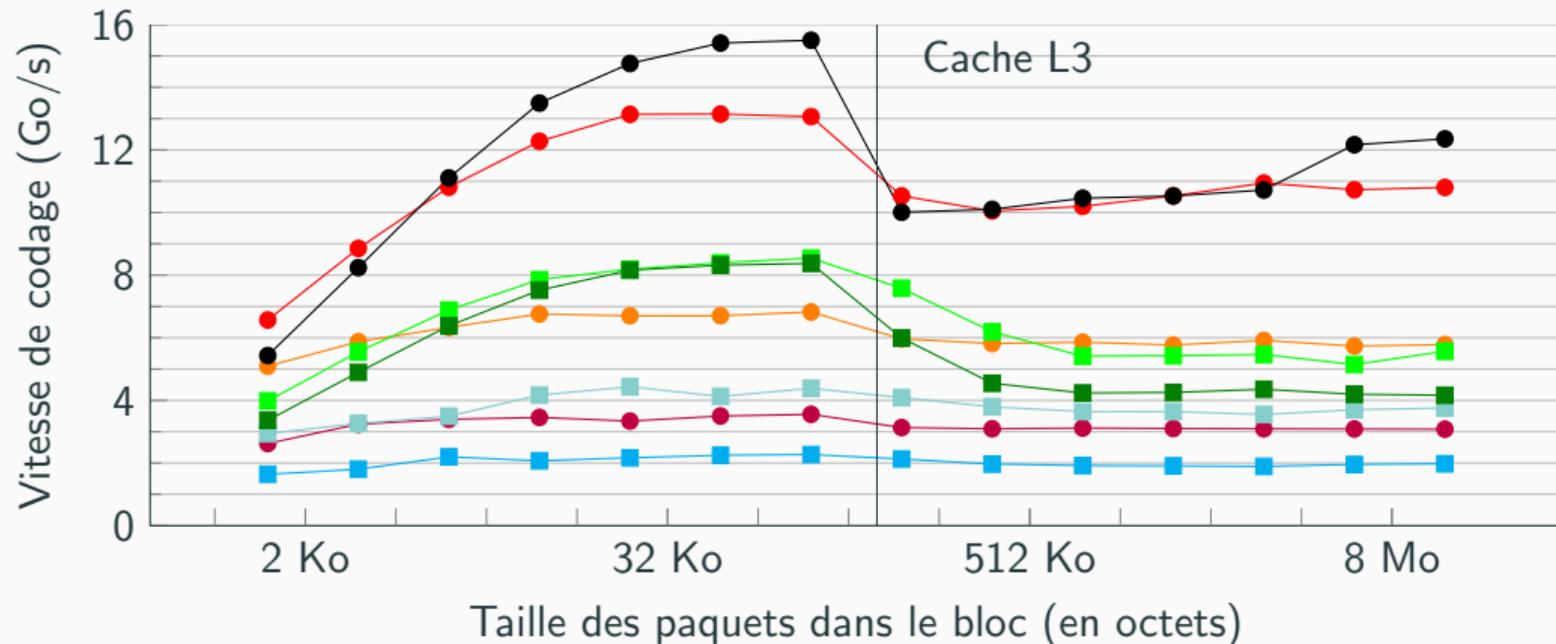
Code $(n, k) = (12, 8)$ sur \mathbb{F}_{2^4} (i7-6700)

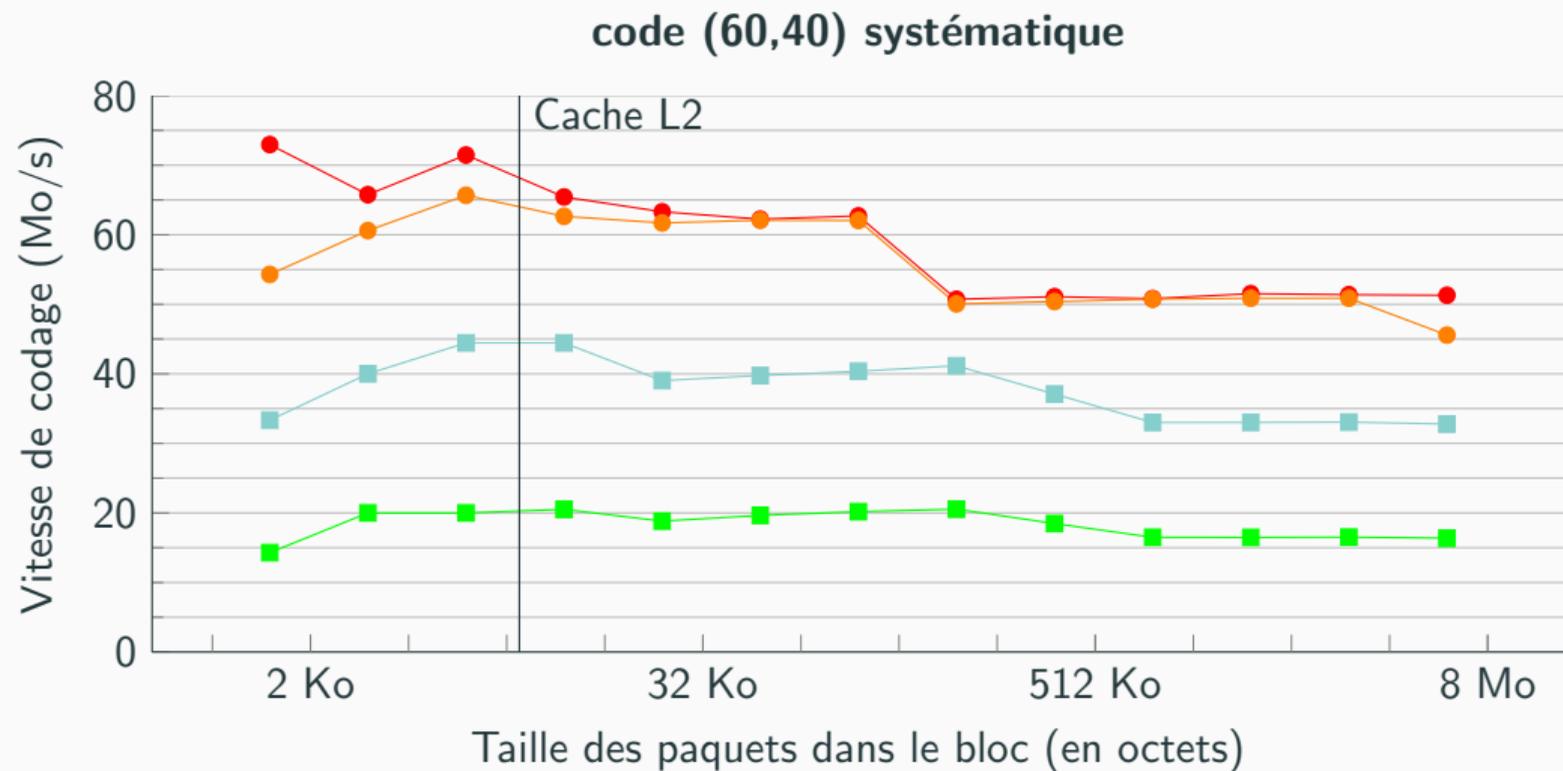




● Pyrit (codage)
 ● Pyrit (décodage)
 ■ ISA-L (codage)
 ■ ISA-L (décodage)

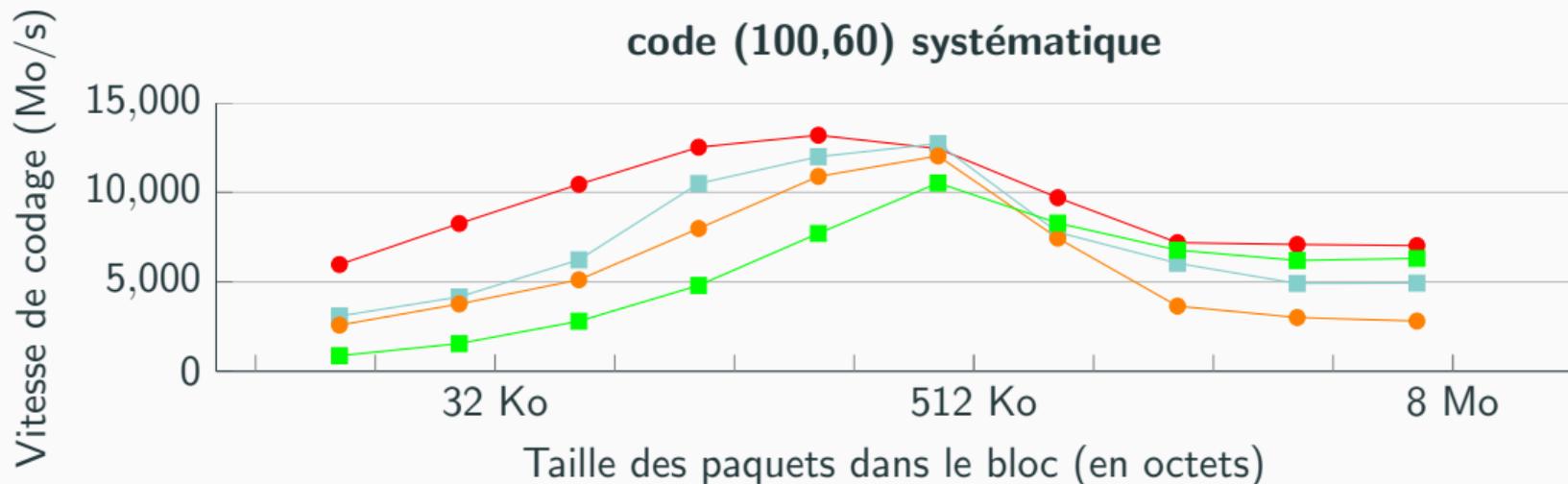
Code (60,40) systématique : codage sur plusieurs cœurs





Machine x86 (Xeon 8164 Scalable Processor)

- 26 cœurs, 32 registres SIMD, support de AVX512
- Construction du corps fini \mathbb{F}_{2^8} par composition sur \mathbb{F}_{2^4}



—●— Pyrit (codage) —■— ISA-L (codage) —●— Pyrit (decodage) —■— ISA-L (decodage)

Conclusion

Par l'utilisation de transformées rapides :

- le nombre d'opérations au codage et au décodage est réduit
- grâce à la structure cyclique des anneaux polynomiaux utilisés, la matrice génératrice est parcourue plus efficacement
- l'implémentation ne nécessite pas d'instructions particulières

Perspectives : quelles améliorations sont possibles ?

Poursuite des travaux sur \mathbb{F}_{2^8} :

- Implémentation sur ARMv8 (32 registres SIMD)
- Améliorations sur x86

Travaux sur les code autres que les codes MDS :

- LDPC non binaires, Locally Repairable Codes, codes à fenêtres glissantes
- Codes correcteurs d'erreurs

Analyse de performance sur du massivement parallèle (GPGPU)

Merci de votre attention.

Questions ?