Algorithmes et Implémentations Optimisées de Calculs Cryptographiques sur les Courbes Elliptiques Binaires

Jean-Marc Robert

Algorithmes et Implémentations Optimisées de Calculs Cryptographiques sur les Courbes Elliptiques Binaires

Jean-Marc ROBERT

Team DALI/LIRMM, Université de Perpignan, France

Journées C2, Dinard, le 11 Octobre 2012

UPVD
Université de Perpignan Via Domitia

DALI
Digits Architectures et Logiciels Informatique

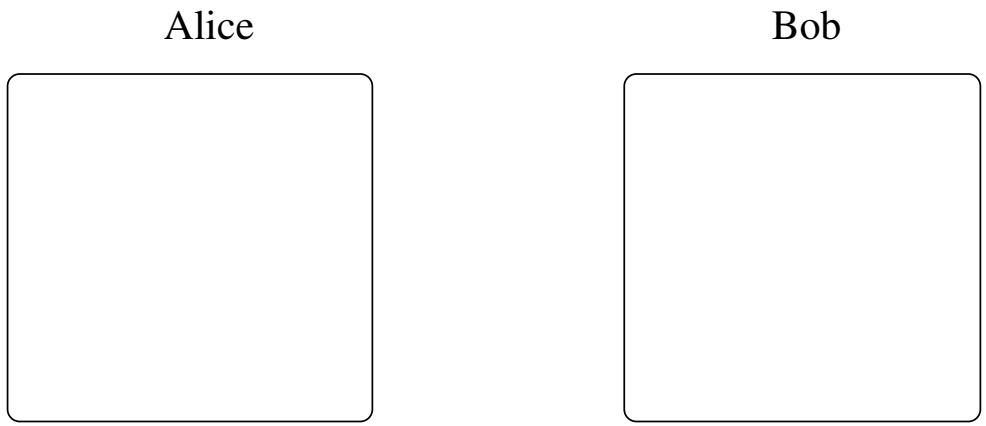
LIRMM

Table des matières

Algorithmes et Implémentations Optimisées

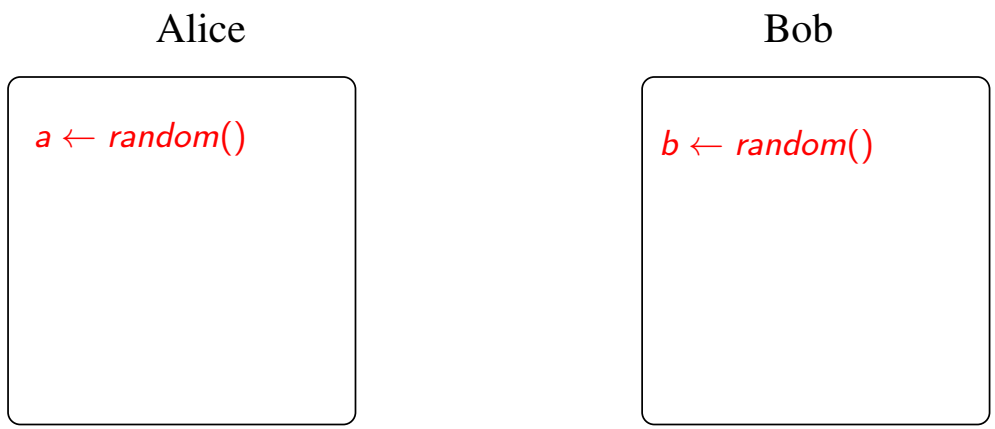
Échange de Clé de Diffie-Hellmann

Alice et Bob s'accordent sur un groupe $(G, +, \mathcal{O})$ et un point P, générateur du groupe.



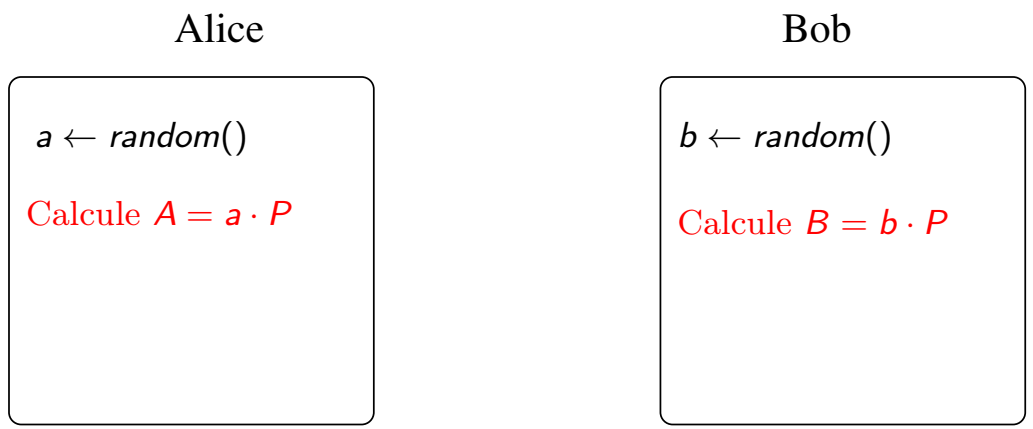
Échange de Clé de Diffie-Hellmann

Alice et Bob s'accordent sur un groupe $(G, +, \mathcal{O})$ et un point P, générateur du groupe.



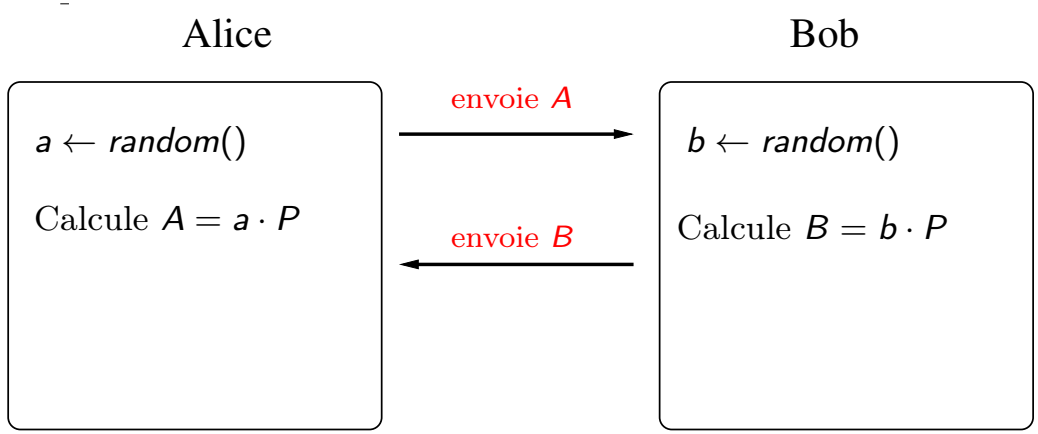
Échange de Clé de Diffie-Hellmann

Alice et Bob s'accordent sur un groupe $(G, +, \mathcal{O})$ et un point P, générateur du groupe.



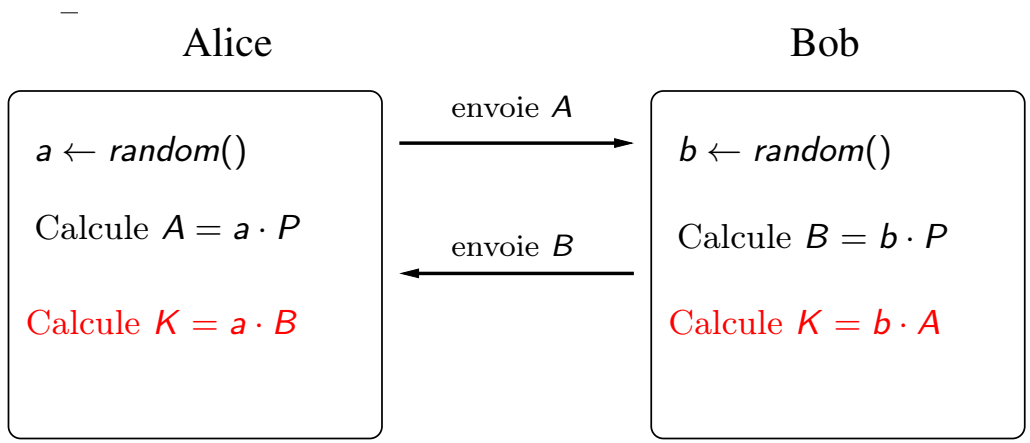
Échange de Clé de Diffie-Hellmann

Alice et Bob s'accordent sur un groupe $(G, +, \mathcal{O})$ et un point P, générateur du groupe.



Échange de Clé de Diffie-Hellmann

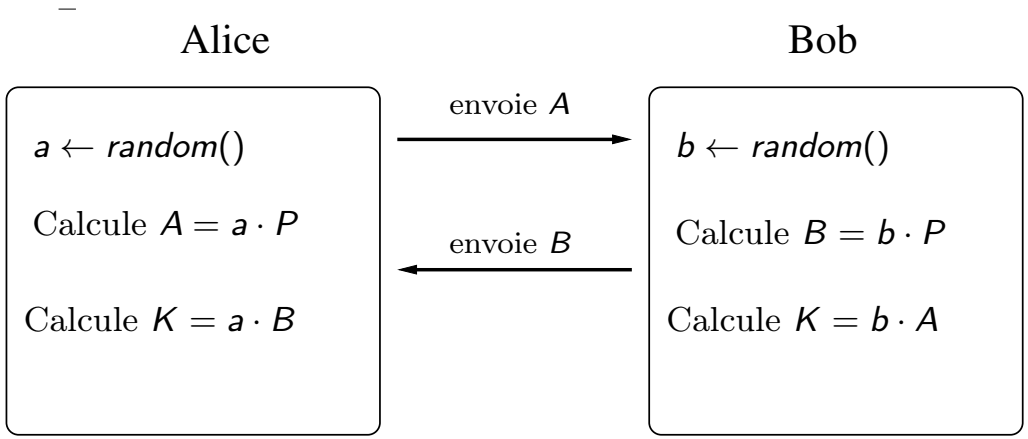
Alice et Bob s'accordent sur un groupe $(G, +, \mathcal{O})$ et un point P, générateur du groupe.



Clé secrète Partagée $K = a \cdot b \cdot P$

Échange de Clé de Diffie-Hellmann

Alice et Bob s'accordent sur un groupe $(G, +, \mathcal{O})$ et un point P, générateur du groupe.

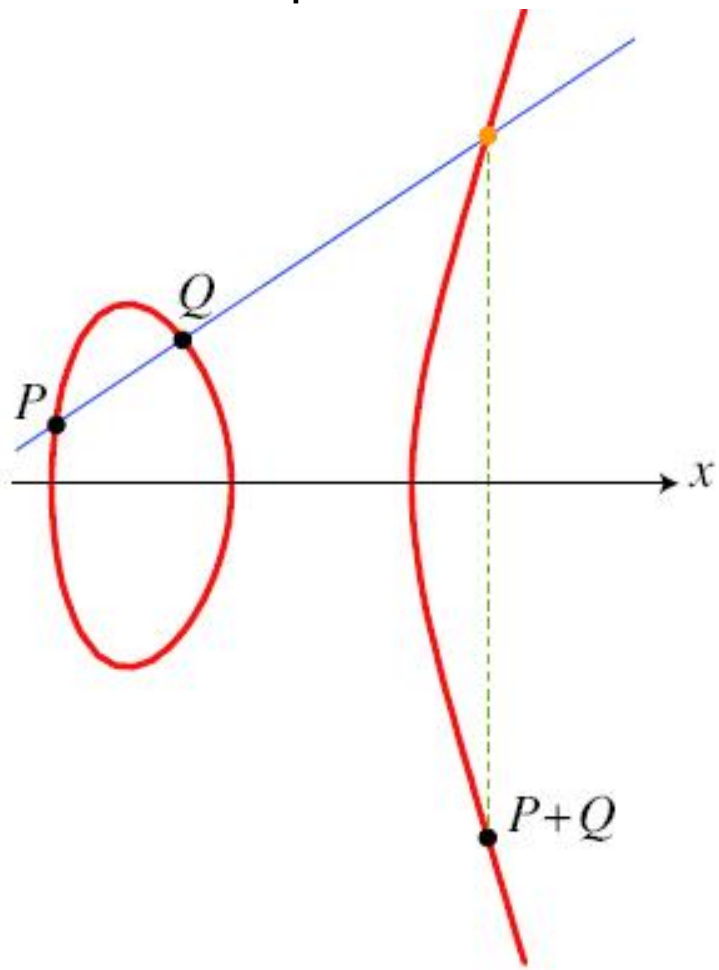


Clé secrète partagée $K = a \cdot b \cdot P$

$\rightarrow$ Le produit scalaire $a \cdot P$ est la principale opération.

Courbe Elliptique sur $\mathbb{F}_{2^m}$

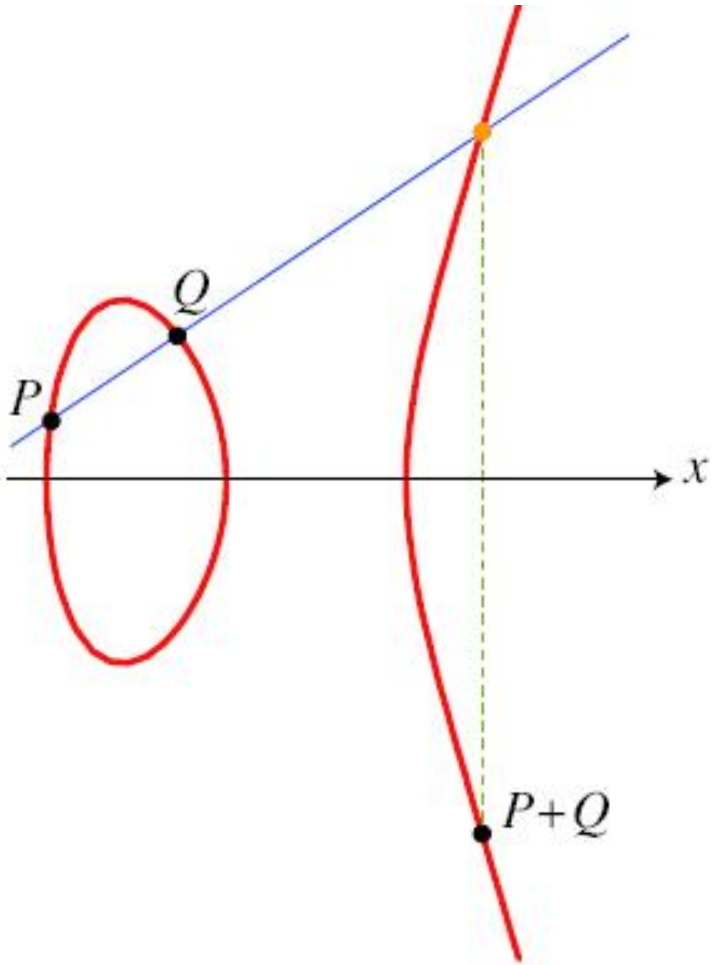
Exemples sur $\mathbb{R}$:



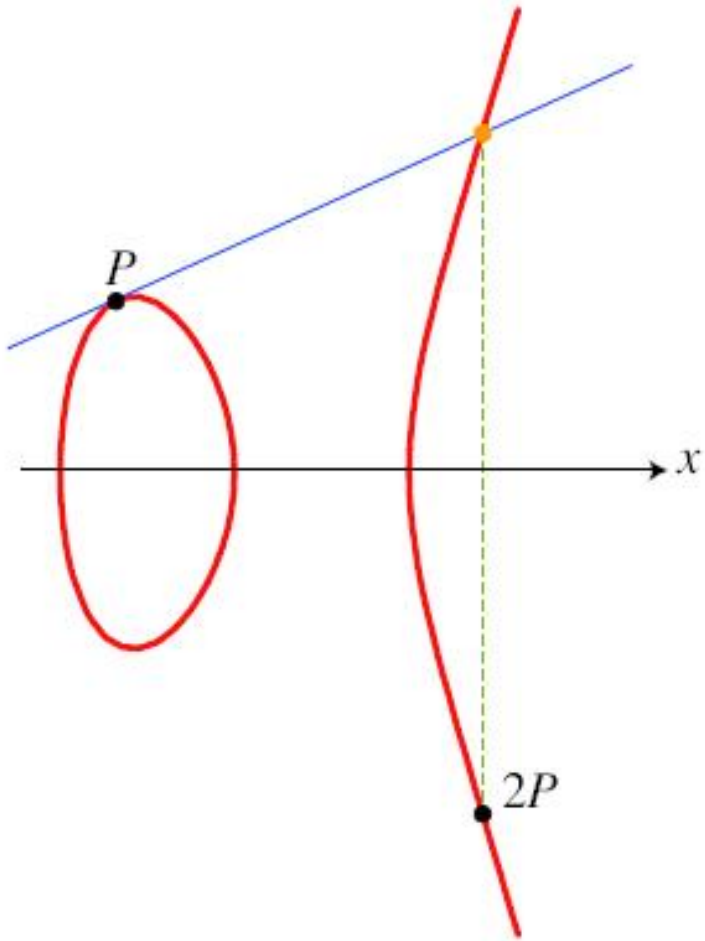
Addition de points

Courbe Elliptique sur $\mathbb{F}_{2^m}$

Exemples sur $\mathbb{R}$:



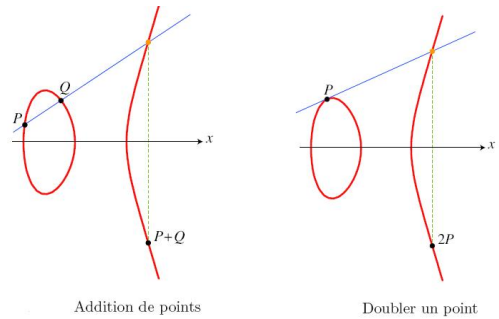
Addition de points



Doubler un point

Courbe Elliptique sur $\mathbb{F}_{2^m}$

Notre courbe est sur $\mathbb{F}_{2^m}$ (et non $\mathbb{R}$) :



Addition de points — Doubler un point

$$E : Y^2 + XY = X^3 + aX^2 + b, \quad a, b \in \mathbb{F}_{2^m}.$$

- Les coordonnées des points appartiennent à $\mathbb{F}_{2^m} = \mathbb{F}_2[x]/(f(x) \cdot \mathbb{F}_2[x])$
- Soit $A = \sum_{i=0}^{m-1} a_i \cdot x^i$ et $B = \sum_{i=0}^{m-1} b_i \cdot x^i, \quad a_i, b_i \in \{0, 1\}$

$$\text{alors : } A + B = \sum_{i=0}^{m-1} (a_i + b_i) \cdot x^i,$$

$$\text{et : } A \times B = A \cdot B \mod f.$$

Algorithmes et Implémentations Optimisées

Opérations sur Courbes elliptiques

Les formules pour le doublement et l'addition sont :

$$\begin{cases} x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 = (x_1 + x_3)\lambda + x_3 + y_1 \end{cases} \quad \text{avec} \quad \begin{cases} \lambda = \frac{y_1+y_2}{x_1+x_2} \text{ if } P_1 \neq P_2 \\ \\ \lambda = \frac{y_1}{x_1} + x_1 \text{ if } P_1 = P_2 \end{cases}$$

Opérations sur Courbes elliptiques

Les formules pour le doublement et l'addition sont :

$$\begin{cases} x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 = (x_1 + x_3)\lambda + x_3 + y_1 \end{cases} \quad \text{avec} \quad \begin{cases} \lambda = \frac{y_1+y_2}{x_1+x_2} \text{ if } P_1 \neq P_2 \\ \\ \lambda = \frac{y_1}{x_1} + x_1 \text{ if } P_1 = P_2 \end{cases}$$

- Un doublement nécessite 1 inversion, 2 multiplications, 1 élévation au carré , et 8 additions de polynômes ;
- Une addition nécessite 1 inversion, 2 multiplications, 1 élévation au carré , et 9 additions de polynômes ;

Points de Courbes Elliptiques : Coordonnées Projectives

- Lopez-Dahab ont montré l'intérêt des coordonnées "*projectives*" . Un point en coordonnées affines est transformé comme suit :

$$P = (x, y) \text{ devient } (X : Y : Z) \text{ avec } \begin{cases} x = \frac{X}{Z} \\ y = \frac{Y}{Z^2} \end{cases}$$

Points de Courbes Elliptiques : Coordonnées Projectives

- Lopez-Dahab ont montré l'intérêt des coordonnées "*projectives*" . Un point en coordonnées affines est transformé comme suit :

$$P = (x, y) \text{ devient } (X : Y : Z) \text{ avec } \begin{cases} x = \frac{X}{Z} \\ y = \frac{Y}{Z^2} \end{cases}$$

- Maintenant, on effectue le doublement de la façon suivante :

$$2.(X : Y : Z) = (X_1 : Y_1 : Z_1) \text{ avec } \begin{cases} X_1 = X^4 + b \cdot Z^4 \\ Y_1 = bZ^4 \cdot Z_1 + X_1 \cdot (aZ_1 + Y^2 + bZ^4) \\ Z_1 = X^2 \cdot Z^2 \end{cases}$$

Points de Courbes Elliptiques : Coordonnées Projectives

- Lopez-Dahab ont montré l'intérêt des coordonnées "*projectives*" . Un point en coordonnées affines est transformé comme suit :

$$P = (x, y) \text{ devient } (X : Y : Z) \text{ avec } \begin{cases} x = \frac{X}{Z} \\ y = \frac{Y}{Z^2} \end{cases}$$

- Maintenant, on effectue le doublement de la façon suivante :

$$2.(X : Y : Z) = (X_1 : Y_1 : Z_1) \text{ avec } \begin{cases} X_1 = X^4 + b \cdot Z^4 \\ Y_1 = bZ^4 \cdot Z_1 + X_1 \cdot (aZ_1 + Y^2 + bZ^4) \\ Z_1 = X^2 \cdot Z^2 \end{cases}$$

- Les coordonnées *PL* (Kim-Kim) diminue encore la complexité.

Operation	Affine	Lopez-Dahab	Kim-Kim
Doublement	2M + 1S + 3R + 1I	4M + 4S + 8R	4M + 5S + 7R
Addition	2M + 1S + 3R + 1I	9M + 5S + 14R	8M + 5S + 9R

L'Algorithme de Produit Scalaire : *Double-and-add*

- Soit $k = (k_{n-1}, ..., k_1, k_0)_2 \in \mathbb{N}, P \in E(\mathbb{F}_{2^m})$

$$\begin{aligned} k \cdot P &= (\sum_{i=0}^{n-1} 2^i k_i) \cdot P \\ &= (k_0 + 2(k_1 + 2(k_2 + 2(\ldots + 2k_{n-1})\ldots)))) \cdot P \\ &= (k_0 \cdot P + 2(k_1 \cdot P + 2(k_2 \cdot P + 2(\ldots + 2(k_{n-2} \cdot P + 2k_{n-1} \cdot P)\ldots)))) \end{aligned}$$

L'Algorithme de Produit Scalaire : *Double-and-add*

- Soit $k = (k_{n-1}, ..., k_1, k_0)_2 \in \mathbb{N}, P \in E(\mathbb{F}_{2^m})$

$$\begin{aligned} k \cdot P &= (\sum_{i=0}^{n-1} 2^i k_i) \cdot P \\ &= (k_0 + 2(k_1 + 2(k_2 + 2(\ldots + 2k_{n-1})\ldots)))) \cdot P \\ &= (k_0 \cdot P + 2(k_1 \cdot P + 2(k_2 \cdot P + 2(\ldots + 2(k_{n-2} \cdot P + 2k_{n-1} \cdot P)\ldots)))) \end{aligned}$$

- On traduit cela dans l'algorithme suivant :

1: $Q \leftarrow \mathcal{O}$.
2: **for** i de $n-1$ à 0 **do**
3: $\quad Q \leftarrow 2 \cdot Q$.
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$.
6: $\quad$ **end if**
7: **end for**
8: **return** (Q).

Amélioration de *Double-and-add* : *NAF* et *W-NAF*.

- *NAF* remplace les séquences de 1 consecutifs : soit $k \in \mathbb{N}$ tel que $k = 2^i - 1$, alors

$$(k)_2 = \underbrace{111...1}_{i \text{ fois}} \text{ et on peut écrire : } (k)_{NAF} = \underbrace{100...00 - 1}_{i+1 \text{ chiffres}}.$$

- Cette représentation diminue le nombre moyen de chiffres non nuls de $n/2$ à $n/3$.

Amélioration de *Double-and-add* : *NAF* et *W-NAF*.

- *NAF* remplace les séquences de 1 consecutifs : soit $k \in \mathbb{N}$ tel que $k = 2^i - 1$, alors

$$(k)_2 = \underbrace{111...1}_{i \text{ fois}} \text{ et on peut écrire : } (k)_{NAF} = \underbrace{100...00 - 1}_{i+1 \text{ chiffres}}.$$

- Cette représentation diminue le nombre moyen de chiffres non nuls de $n/2$ à $n/3$.

- *W-NAF* réduit davantage encore le nombre moyen de chiffres non nuls à $n/(w+1)$ en utilisant des chiffres plus grand :

$$\{-2^{w-1} + 1, ..., -5, -3, -1, 0, 1, 3, 5, ..., 2^{w-1} - 1\}.$$

- Bilan sur la complexité du produit scalaire sur $E(\mathbb{F}_{2^m})$:

	nb. de doublements	nb. d'additions
Double-and-add	n	$n/2$
NAF Double-and-add	n	$n/3$
W-NAF Double-and-add	n	$n/(w+1) + 2^{w-2}$

Algorithmes et Implémentations Optimisées

Opération sur le corps : Multiplication de polynômes

Algorithme de Karatsuba :

Avec cet algorithme, pour $A(x)$ et $B(x)$ de degré $m-1$, on calcule $C(x) = A(x) \cdot B(x)$ de la manière suivante :

$$\begin{cases} A(x) = A_h(x)x^{m/2} + A_l(x) \\ \\ B(x) = B_h(x)x^{m/2} + B_l(x) \end{cases} \quad \text{On calcule} \quad \begin{cases} D0 = A_l.B_l \\ D1 = (A_l + A_h).(B_l + B_h) \\ D2 = A_h.B_h \end{cases}$$

$$\text{Et enfin : } C(x) = x^m D_2(x) + x^{m/2}(D_0 + D_1 + D_2) + D_0$$

Cet algorithme peut être implémenté avec l'algorithme L-R Comb W pour les produits élémentaires.

Multiplication de polynômes Left-to-Right Comb W, 64 bits

$$\text{Soit } a(x) = \sum_{i=0}^{m-1} a_i \cdot x^i \text{ et } b(x) = \sum_{i=0}^{m-1} b_i \cdot x^i, \quad a_i, b_i \in \{0, 1\}$$

Multiplication de polynômes Left-to-Right Comb W, 64 bits

$$\text{Soit } a(x) = \sum_{i=0}^{63} \sum_{j=0}^{t-1} x^{64j} \cdot a_{i+64j} \cdot x^i, \text{ avec } t = \frac{\lceil m-1 \rceil}{64}$$

Multiplication de polynômes Left-to-Right Comb W, 64 bits

$$\text{Soit } a(x) = \sum_{i=0}^{63} \sum_{j=0}^{t-1} x^{64j} \cdot a_{i+64j} \cdot x^i, \text{ avec } t = \frac{\lceil m-1 \rceil}{64}$$

$$= \sum_{k=0}^{15} \sum_{j=0}^{t-1} x^{4k+64j} \cdot u_{kj}(x), \text{ avec } u_{kj}(x) \text{ de degré au plus 3.}$$

Multiplication de polynômes Left-to-Right Comb W, 64 bits

$$\text{Soit } a(x) = \sum_{i=0}^{63} \sum_{j=0}^{t-1} x^{64j} \cdot a_{i+64j} \cdot x^i, \text{ avec } t = \frac{\lceil m-1 \rceil}{64}$$

$$= \sum_{k=0}^{15} \sum_{j=0}^{t-1} x^{4k+64j} \cdot u_{kj}(x), \text{ avec } u_{kj}(x) \text{ de degré au plus 3.}$$

$$\text{Alors } c(x) = a(x) \cdot b(x) = \sum_{k=0}^{15} \sum_{j=0}^{t-1} x^{4k+64j} \cdot u_{kj}(x) \cdot b(x)$$

Multiplication de polynômes Left-to-Right Comb W, 64 bits

$$\text{Soit } a(x) = \sum_{i=0}^{63} \sum_{j=0}^{t-1} x^{64j} \cdot a_{i+64j} \cdot x^i, \text{ avec } t = \frac{\lceil m-1 \rceil}{64}$$

$$= \sum_{k=0}^{15} \sum_{j=0}^{t-1} x^{4k+64j} \cdot u_{kj}(x), \text{ avec } u_{kj}(x) \text{ de degré au plus 3.}$$

$$\text{Alors } c(x) = a(x) \cdot b(x) = \sum_{k=0}^{15} \sum_{j=0}^{t-1} x^{4k+64j} \cdot u_{kj}(x) \cdot b(x)$$

1: Calcule $B_u(x) = u(x) \cdot b(x)$ pour tous polynômes $u(x)$ de degré au plus 3.
2: $C \leftarrow 0$
3: **for** k de 15 à 0 // 64 bits $= 16 \times 4$. **do**
4: **for** j de 0 à $t-1$ // tableaux de t mots de 64 bits. **do**
5: Soit $u = (u_3, u_2, u_1, u_0)$, où u_i est le bit $(4k+i)$ de $A[j]$. Ajoute B_u et $C\{j\}$.
6: **end for**
7: **if** $k \neq 0$ **then**
8: $C \leftarrow C \cdot x^4$.
9: **end if**
10: **end for**
11: **return** (C).

Produits élémentaires : PCLMULQDQ 128 bits (__m128i)

Pour multiplier deux polynômes de degré au plus 63 (avec les processeurs Intel i3, i5, ou i7) :

- Multiplication "Carry-less" : L-R Comb W (64 bits) devient

```
__m128i x = _mm_load_si128( (__m128i const *) &A);
__m128i y = _mm_load_si128( (__m128i const *) &B);
__m128i z = _mm_clmulepi64_si128 (x, y, 0);

C[0] = _mm_extract_epi64(z,0);
C[1] = _mm_extract_epi64(z,1);
```

Optimisation AB, AC

- Pour additionner deux points en coordonnées mixtes (Lopez-Dahab), nous avons :

$$\begin{cases} F = X_3 + X_2.Z_3, \\ \\ G = X_3 + Y_2.Z_3. \end{cases}$$

  F et G sont deux paramètres dans le calcul de Y_3.
- Le principe : la combinaison du calcul de $X_2 \cdot Z_3$ et de $Y_2 \cdot Z_3$ dans l'algorithme L-R Comb W, permet de ne calculer qu'une seule table des polynômes $u_i \cdot Z_3$, où u_i sont de degré au plus 3.

Optimisation $AB + CD$ L-R Comb W 64 bits

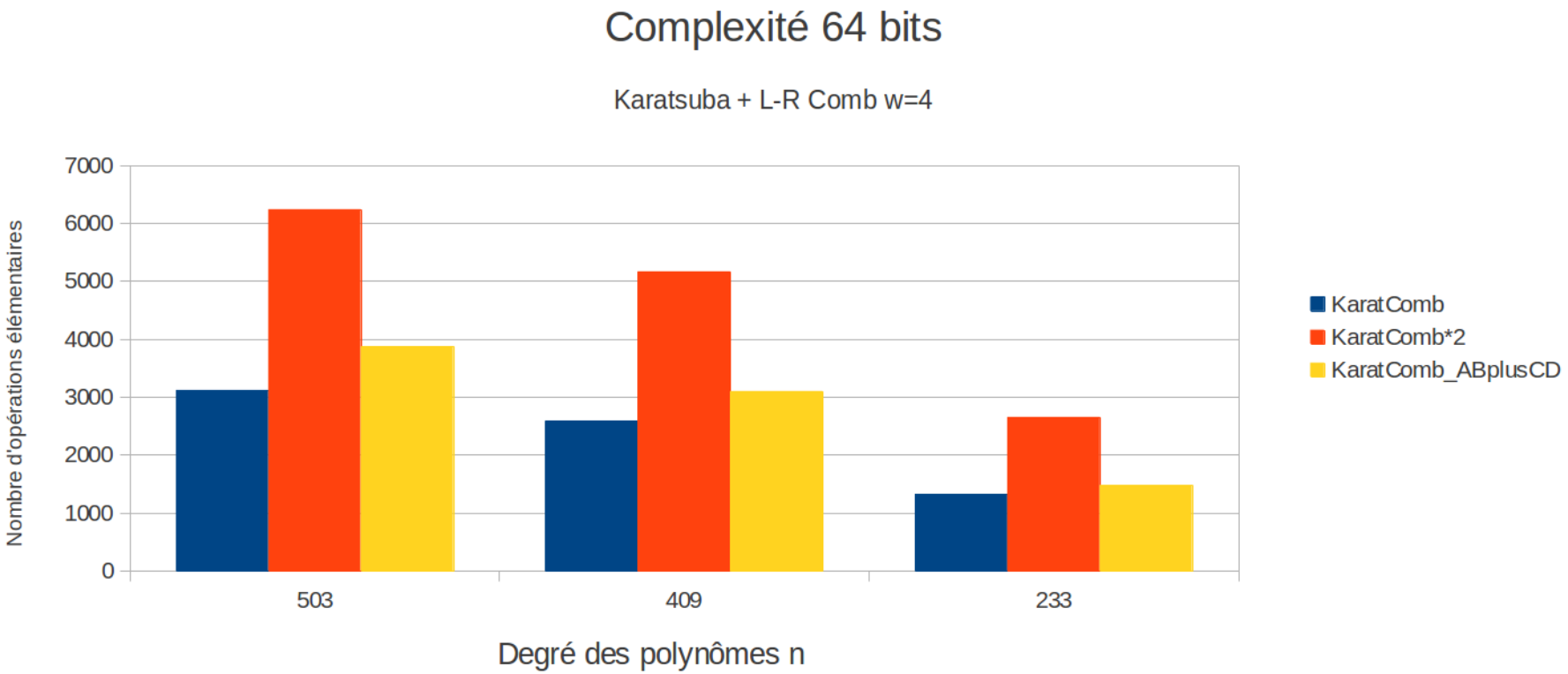
Require: Polynômes binaires $a(x)$, $b(x)$, $c(x)$, $d(x)$ de degré au plus 63.
Ensure: $e(x) = a(x) \cdot b(x) + c(x) \cdot d(x)$.
1: Calcul de $B_u(x) = u(x) \cdot b(x)$ pour tous polynômes $u(x)$ de degré au plus 3.
2: Calcul de $D_u(x) = u(x) \cdot d(x)$ pour tous polynômes $u(x)$ de degré au plus 3.
3: $E \leftarrow 0$
4: **for** k de 15 à 1 **do**
5: Soit $u = (u_3, u_2, u_1, u_0)$, où u_i est le bit $(4k + i)$ de A.
6: Soit $v = (v_3, v_2, v_1, v_0)$, où v_i est le bit $(4k + i)$ de C.
7: $E \leftarrow E \oplus B_u \oplus D_v$.
8: $E \leftarrow E \cdot x^4$.
9: **end for**
10: Soit $u = (u_3, u_2, u_1, u_0)$, où u_i est le bit i of A.
11: Soit $v = (v_3, v_2, v_1, v_0)$, où v_i est le bit i de C.
12: $E \leftarrow E \oplus B_u \oplus D_v$.
13: **return** (E).

Optimisation $AB + CD$ L-R Comb W 64 bits

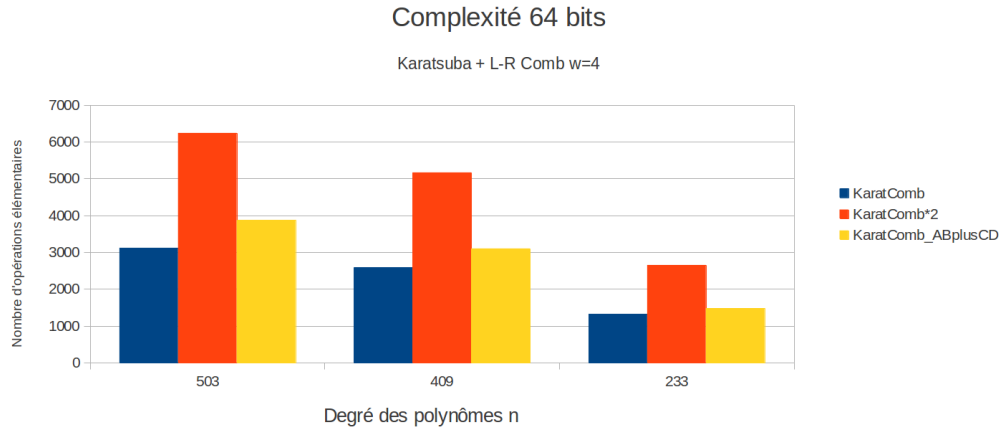
Require: Polynômes binaires $a(x)$, $b(x)$, $c(x)$, $d(x)$ de degré au plus 63.
Ensure: $e(x) = a(x) \cdot b(x) + c(x) \cdot d(x)$.
1: Calcul de $B_u(x) = u(x) \cdot b(x)$ pour tous polynômes $u(x)$ de degré au plus 3.
2: Calcul de $D_u(x) = u(x) \cdot d(x)$ pour tous polynômes $u(x)$ de degré au plus 3.
3: $E \leftarrow 0$
4: **for** k de 15 à 1 **do**
5: Soit $u = (u_3, u_2, u_1, u_0)$, où u_i est le bit $(4k + i)$ de A.
6: Soit $v = (v_3, v_2, v_1, v_0)$, où v_i est le bit $(4k + i)$ de C.
7: $E \leftarrow E \oplus B_u \oplus D_v$.
8: $E \leftarrow E \cdot x^4$.
9: **end for**
10: Soit $u = (u_3, u_2, u_1, u_0)$, où u_i est le bit i of A.
11: Soit $v = (v_3, v_2, v_1, v_0)$, où v_i est le bit i de C.
12: $E \leftarrow E \oplus B_u \oplus D_v$.
13: **return** (E).

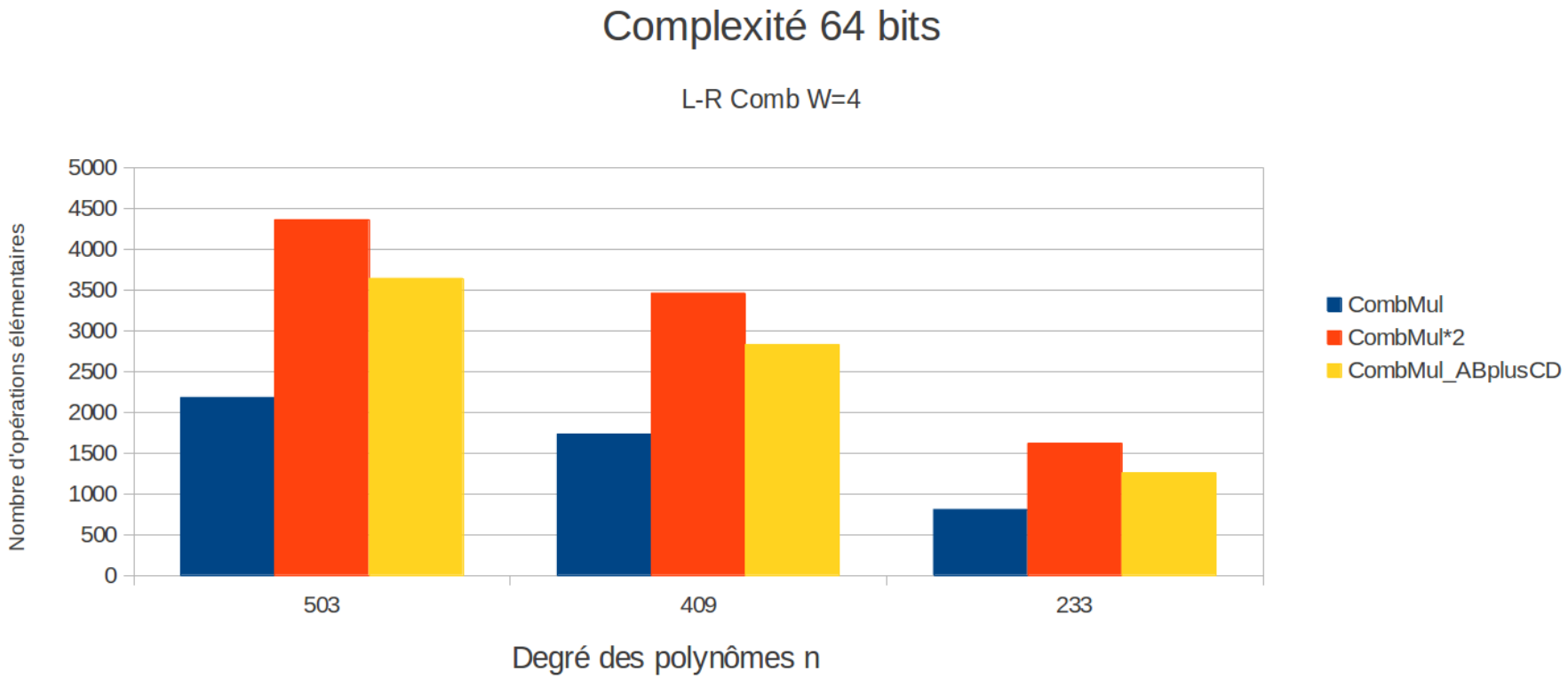
Avec l'Algorithme de Karatsuba : une seule reconstruction finale au lieu de deux.

Optimisations AB, AC et $AB + CD$: bilan de complexité

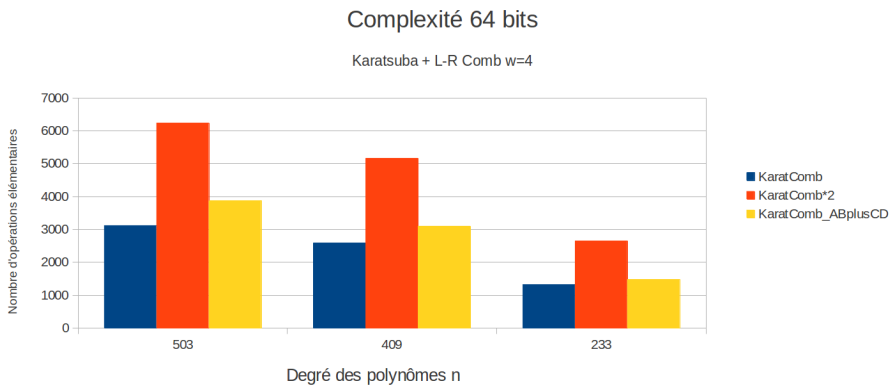


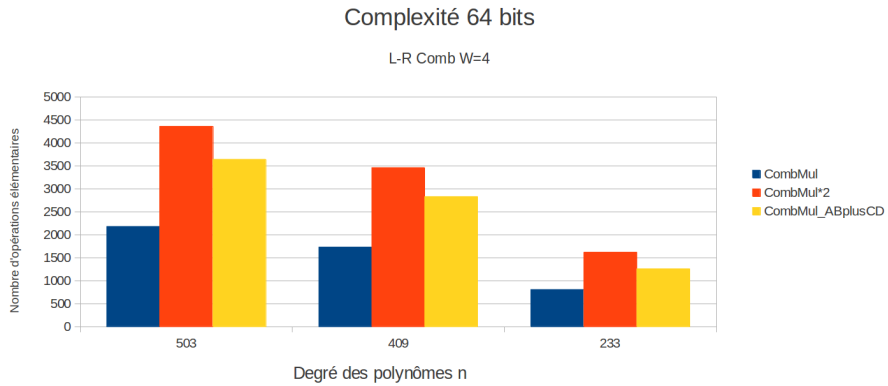
Optimisations AB, AC et $AB + CD$: bilan de complexité

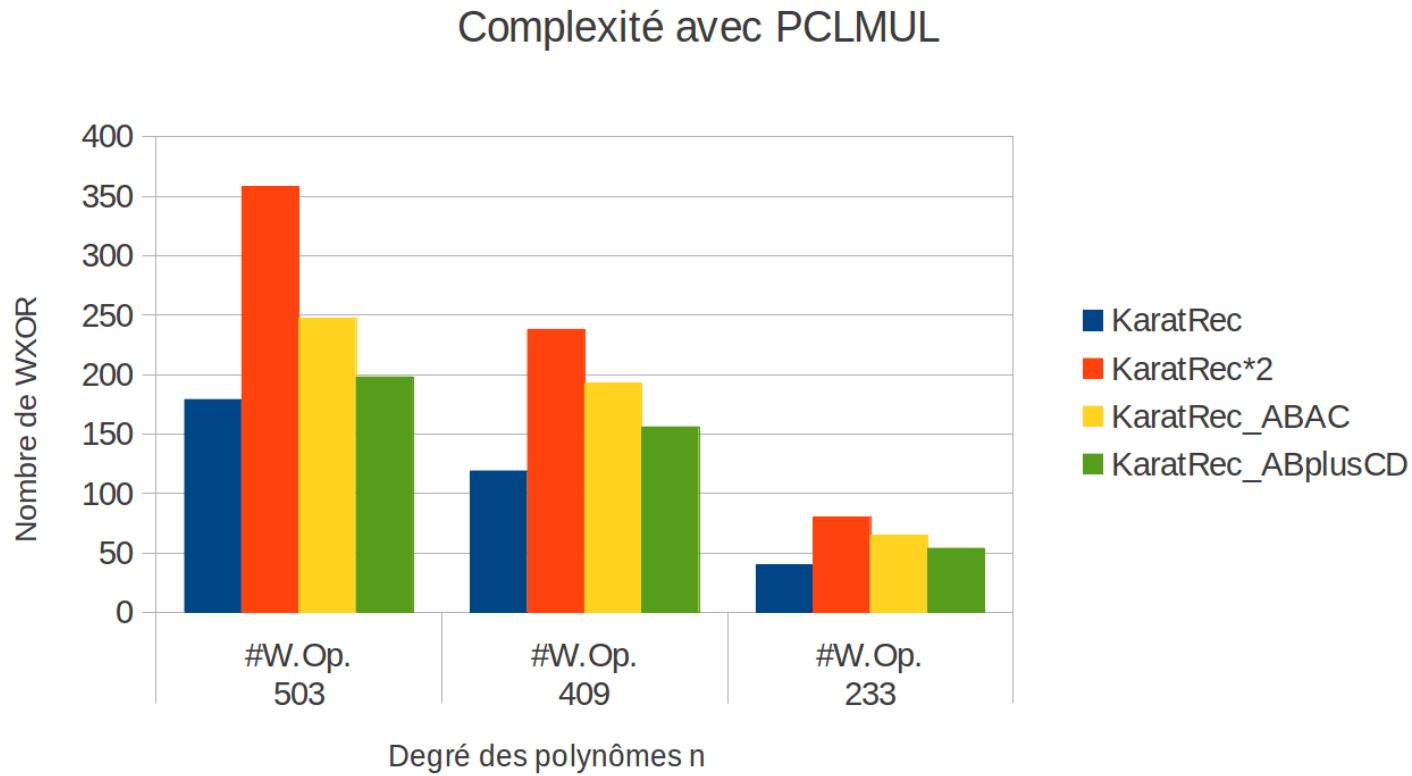




Optimisations AB, AC et $AB + CD$: bilan de complexité







Algorithmes et Implémentations Optimisées

Principaux Paramètres

Le *NIST* recommande les valeurs de m et des différents paramètres.

- L'équation de la courbe est $E : y^2 + xy = x^3 + x^2 + b$;

Principaux Paramètres

Le *NIST* recommande les valeurs de m et des différents paramètres.

- L'équation de la courbe est $E : y^2 + xy = x^3 + x^2 + b$;
- On considère les deux corps binaires $\mathbb{F}_{2^{233}}$ défini avec :

$$f(x) = x^{233} + x^{74} + 1$$

et $\mathbb{F}_{2^{409}}$ défini avec :

$$f(x) = x^{409} + x^{87} + 1$$

Principaux Paramètres

Le *NIST* recommande les valeurs de m et des différents paramètres.

- L'équation de la courbe est $E : y^2 + xy = x^3 + x^2 + b$;
- On considère les deux corps binaires $\mathbb{F}_{2^{233}}$ défini avec :

$$f(x) = x^{233} + x^{74} + 1$$

et $\mathbb{F}_{2^{409}}$ défini avec :

$$f(x) = x^{409} + x^{87} + 1$$

- Les polynômes binaires se représentent sous la forme de tableaux hexadécimaux.

Principaux Paramètres

Le *NIST* recommande les valeurs de m et des différents paramètres.

- L'équation de la courbe est $E : y^2 + xy = x^3 + x^2 + b$;
- On considère les deux corps binaires $\mathbb{F}_{2^{233}}$ défini avec :

$$f(x) = x^{233} + x^{74} + 1$$

et $\mathbb{F}_{2^{409}}$ défini avec :

$$f(x) = x^{409} + x^{87} + 1$$

- Les polynômes binaires se représentent sous la forme de tableaux hexadécimaux.
- Le point générateur est $P = (G_x, G_y)$. Exemple, pour $m = 233$:

$$\begin{cases} \texttt{Gx = 0fa c9dfcbac 8313bb21 39f1bb75 5fef65bc 391f8b36 f8f8eb73 71fd558b} \\ \texttt{Gy = 100 6a08a419 03350678 e58528be bf8a0bef f867a7ca 36716f7e 01f81052} \end{cases}$$

Plate-formes d'expérimentations :

- Codes écrits en langage C (gcc 4.6.1 x86_64 compiler);
- Système d'Exploitation Linux, distribution UBUNTU (11.10).

Plate-formes			
		Dell Insp. 15R	*Dell Optiplex 990*
Processeur Intel 64 bits	Core 2 Duo	Core i5	Core i7
nombre de cœurs	2	4	8
Mémoire cache de niveau L3	2Mo	3Mo	8 Mo
Mémoire cache de niveau L1	32 ko	32 ko	32 ko
Multiplication *Carry-less* (`PCLMULQDQ`)	NON	OUI	OUI
Registres Multimedia (variables de type `__m128i`)	OUI	OUI	OUI

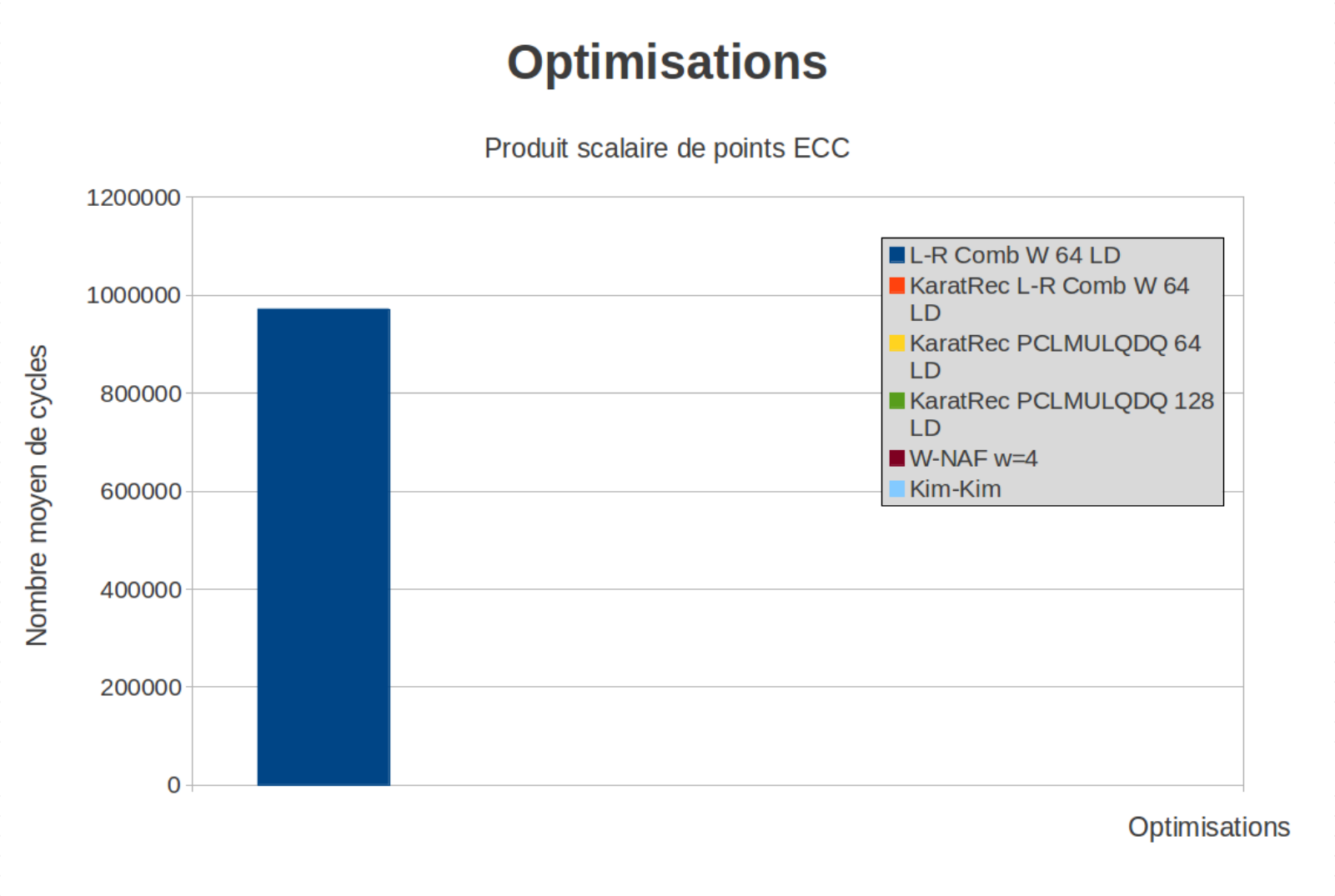
Algorithmes et Implémentations Optimisées

Implémentations, Expérimentations : Atteindre l'Etat de l'Art

Une implémentation optimisée du produit scalaire se présente comme suit :

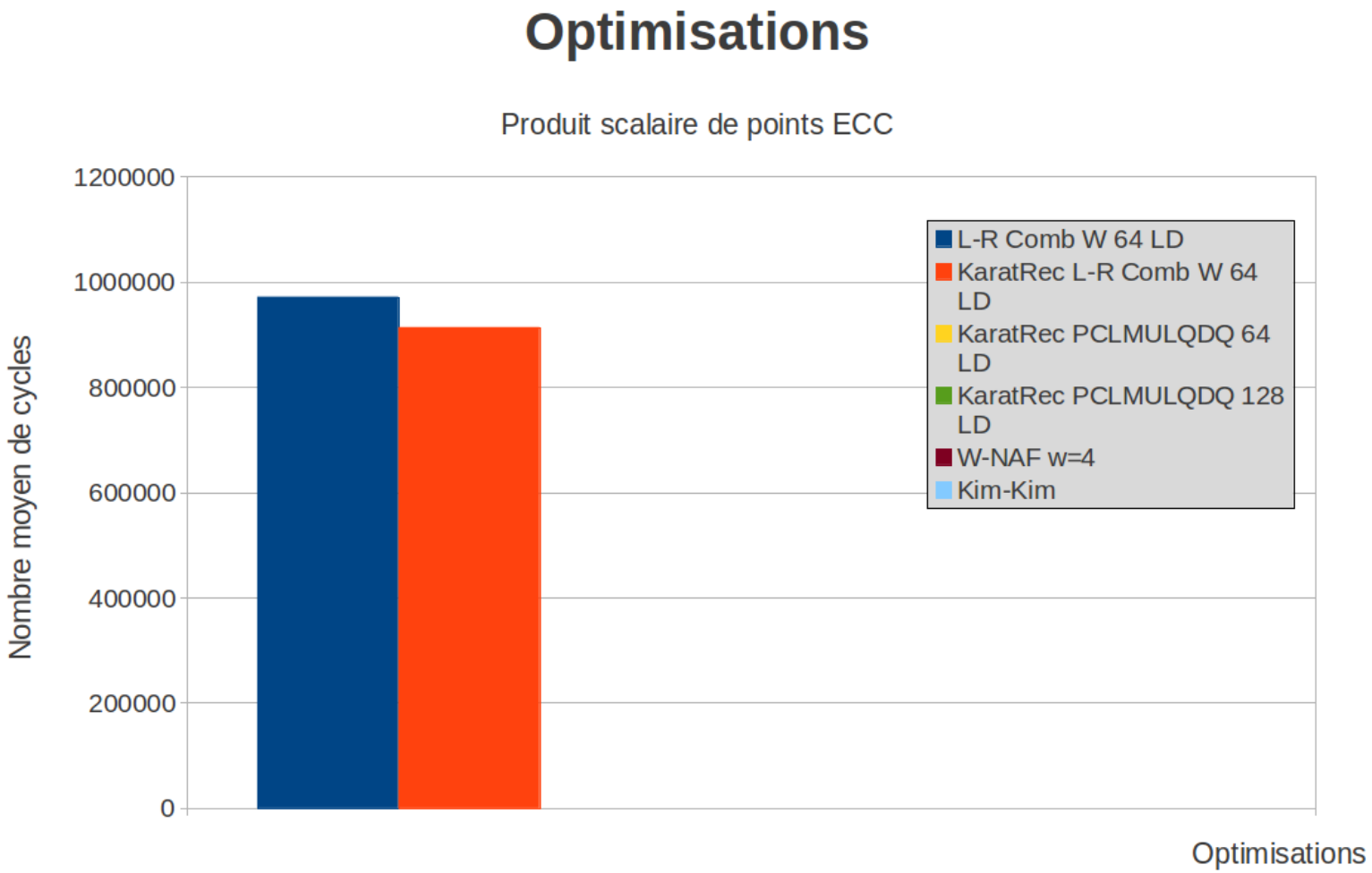
- La meilleure multiplication de polynômes : Karatsuba avec multiplication "carry-less" `PCLMULQDQ` ;
- Utilisation des registres multimédias pour encoder les polynômes (type `__m128i`) ;
- Représentation d'entiers *W-NAF* avec $w = 4$;
- Inversion de polynôme optimisée (calculs de carrés multiples tabulés) ;
- Doublement et addition de points de courbe elliptique Kim-Kim ;

Expérimentations : Atteindre l'Etat de l'Art (i5)



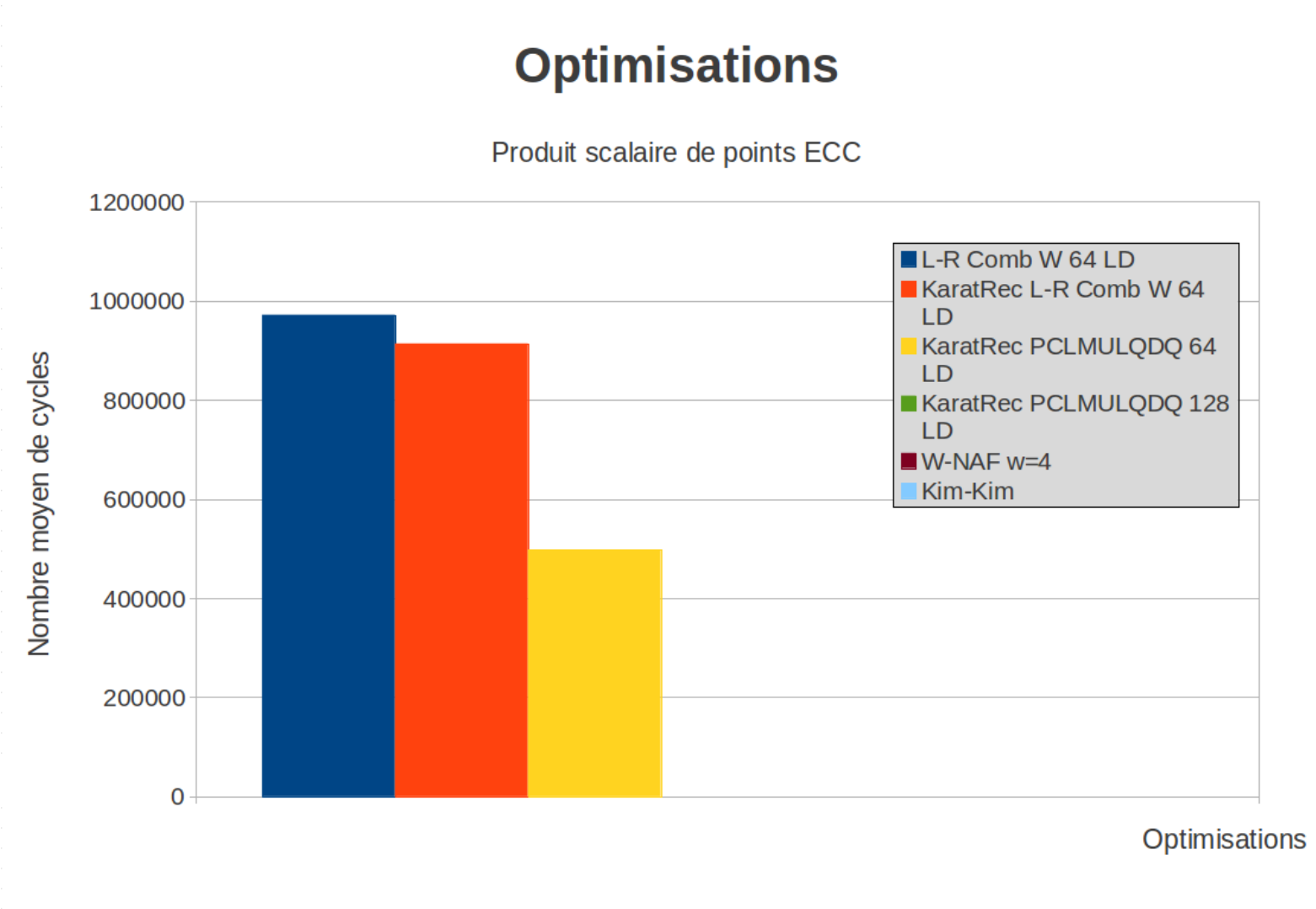
$$m = 233$$

Expérimentations : Atteindre l'Etat de l'Art (i5)



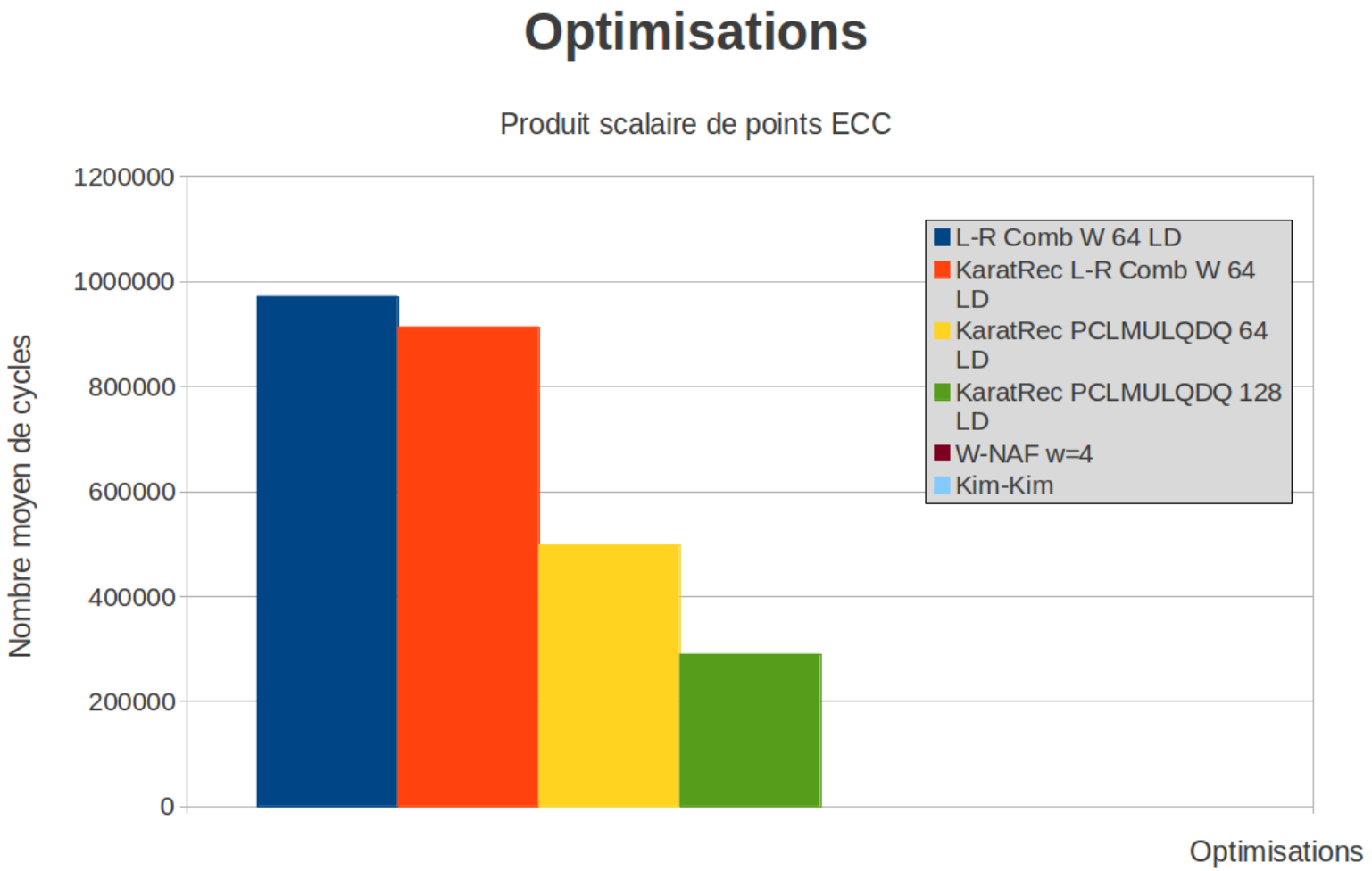
KaratRec L-R Comb W : -5,96%

Expérimentations : Atteindre l'Etat de l'Art (i5)



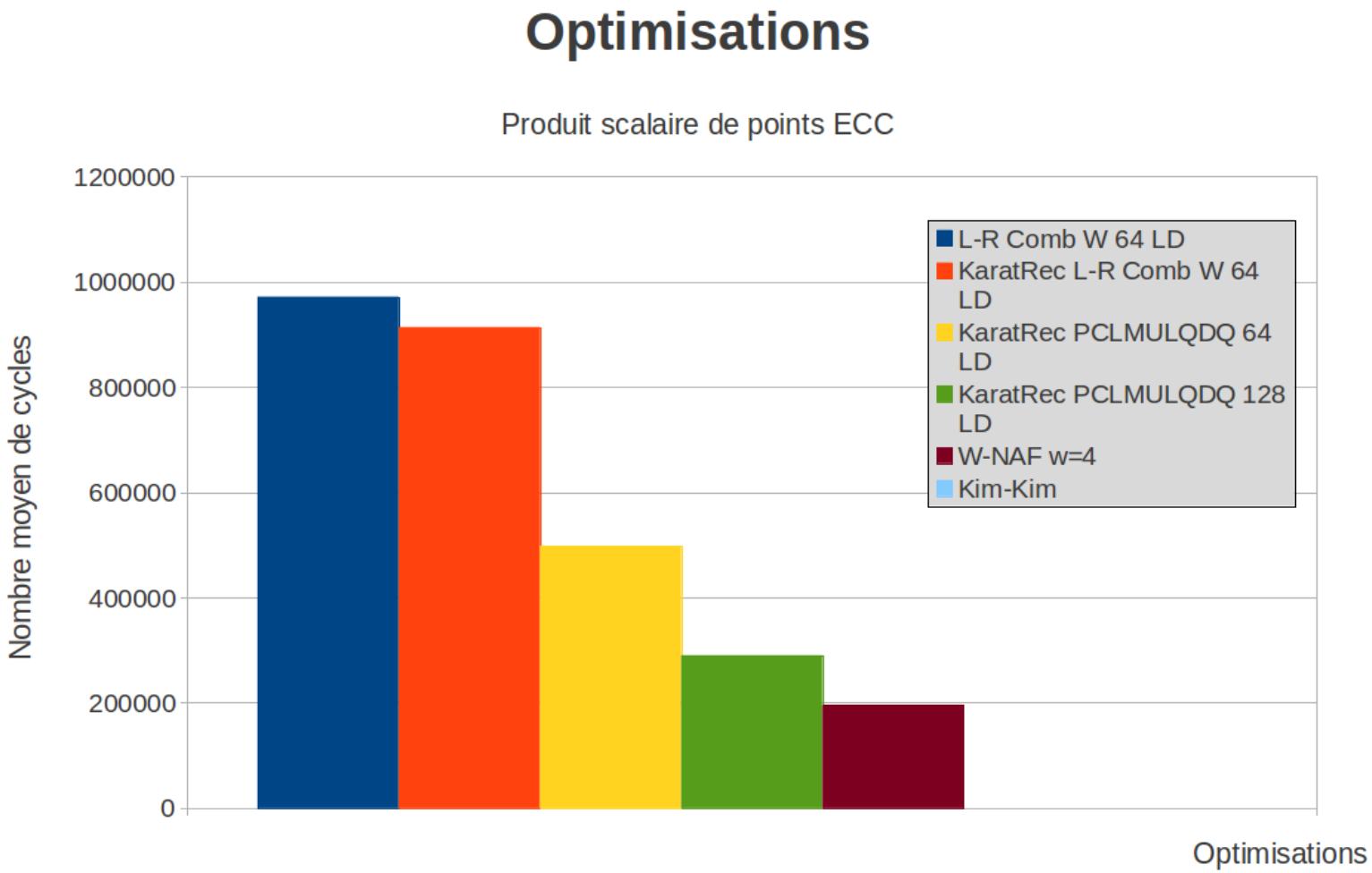
KaratRec `PCLMULQDQ` : -45,5%

Expérimentations : Atteindre l'Etat de l'Art (i5)



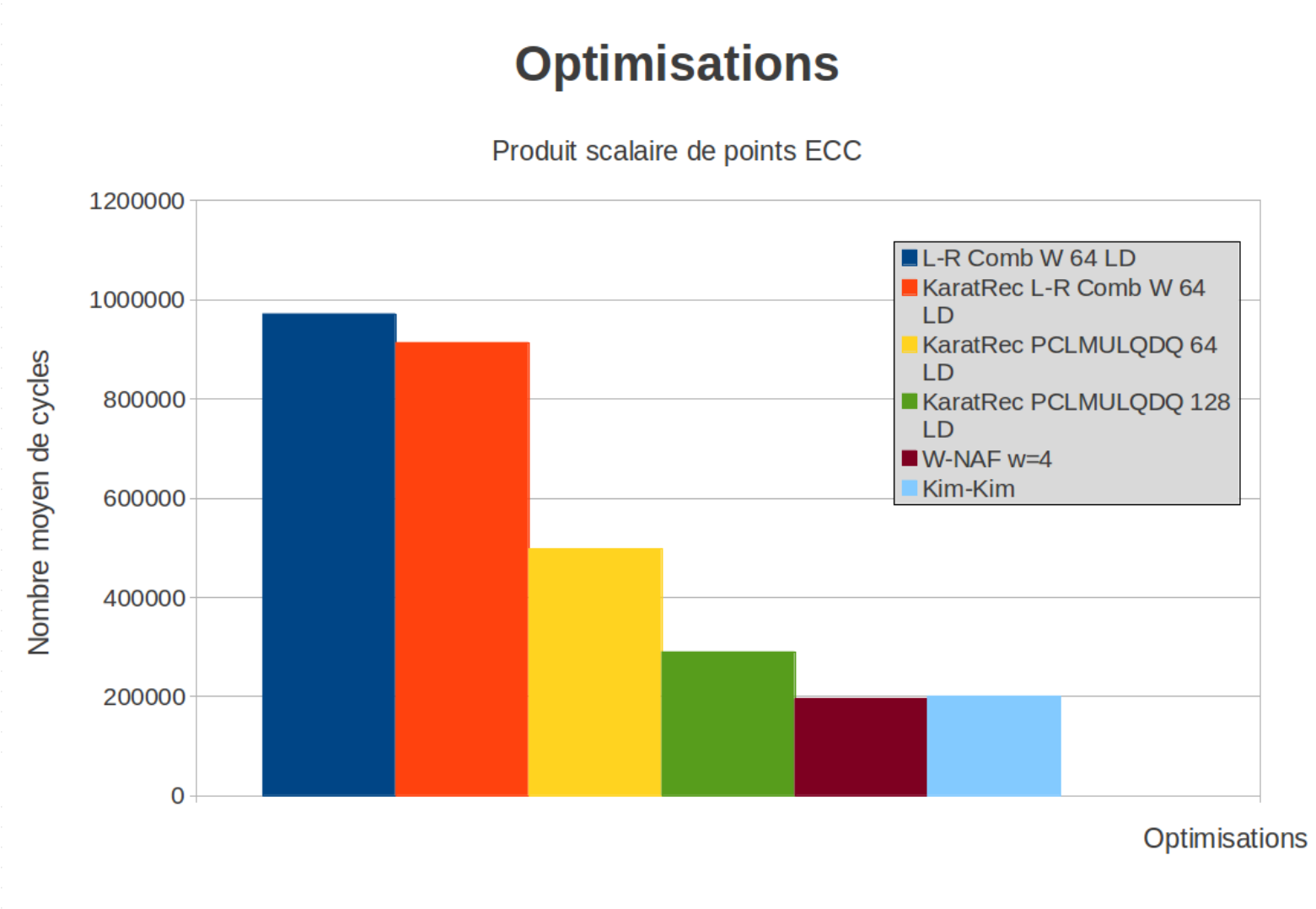
`__m128i` : -41,8%

Expérimentations : Atteindre l'Etat de l'Art (i5)



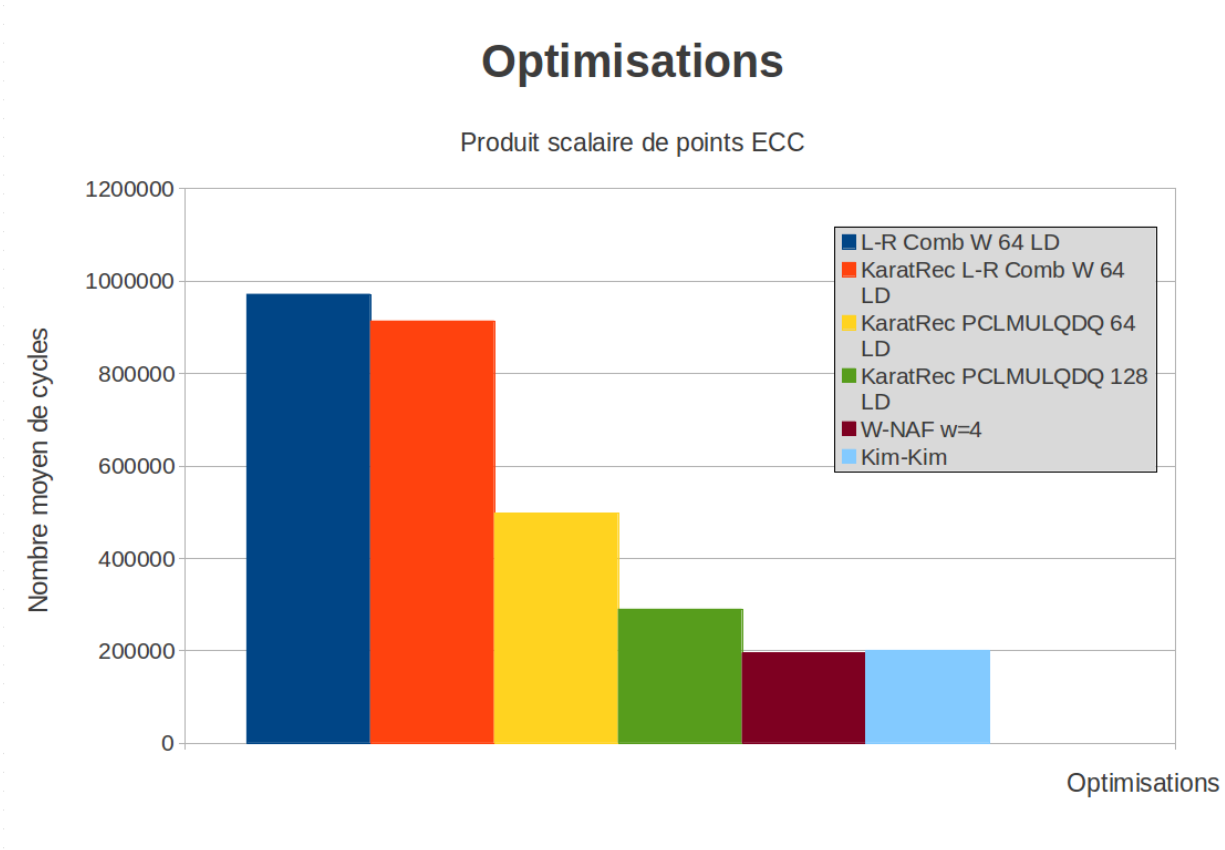
$W\text{-}NAF \quad w = 4$: -32,7%

Expérimentations : Atteindre l'Etat de l'Art (i5)



Opérations Kim-Kim : +3,13%

Expérimentations : Atteindre l'Etat de l'Art (i5)

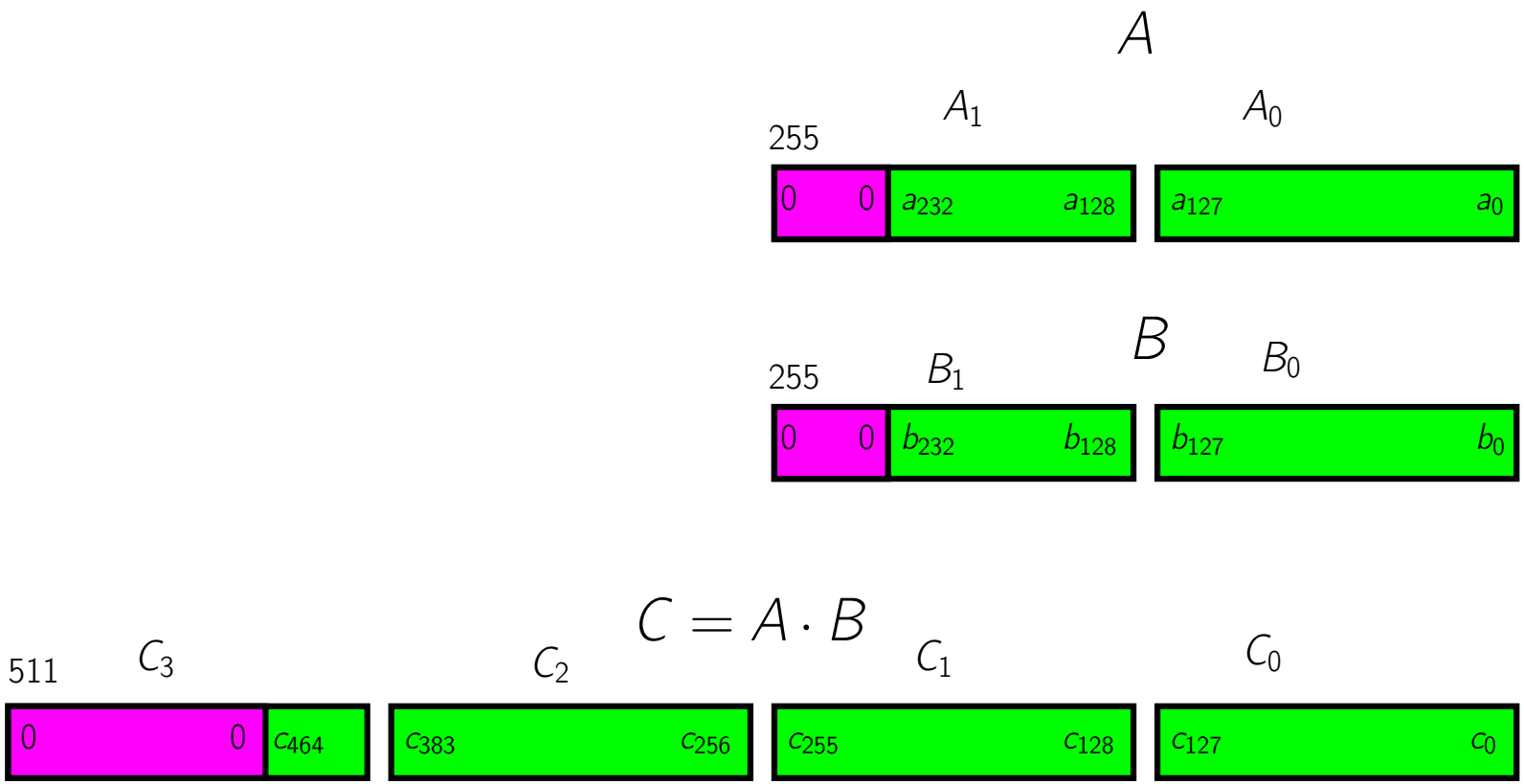


Comparaison : Nombre de Cycles Moyens (2000 exécutions)			
	Etat de l'Art (i5) [1]	nos résultats (i5)	nos résultats (i7)
W-NAF Double-and-add $m = 233$	216700	201312 (7.10%)	233850
W-NAF Double-and-add $m = 409$	871600	787385 (9.66%)	912795

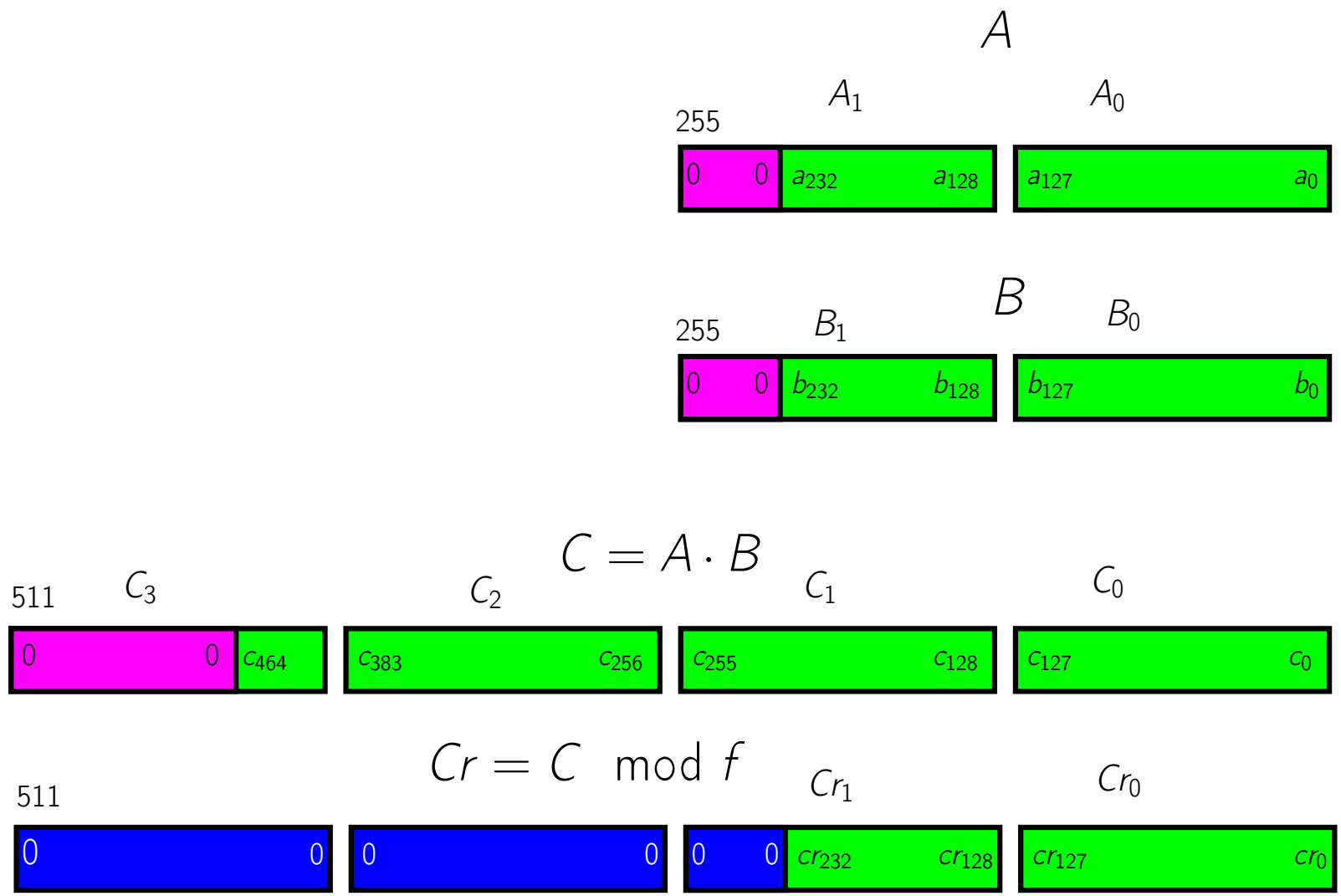
[1] selon J. Taverne et al., CHES 2011.

Algorithmes et Implémentations Optimisées

Very Lazy Reduction ($m = 233$, `__m128i`) : le juste nécessaire

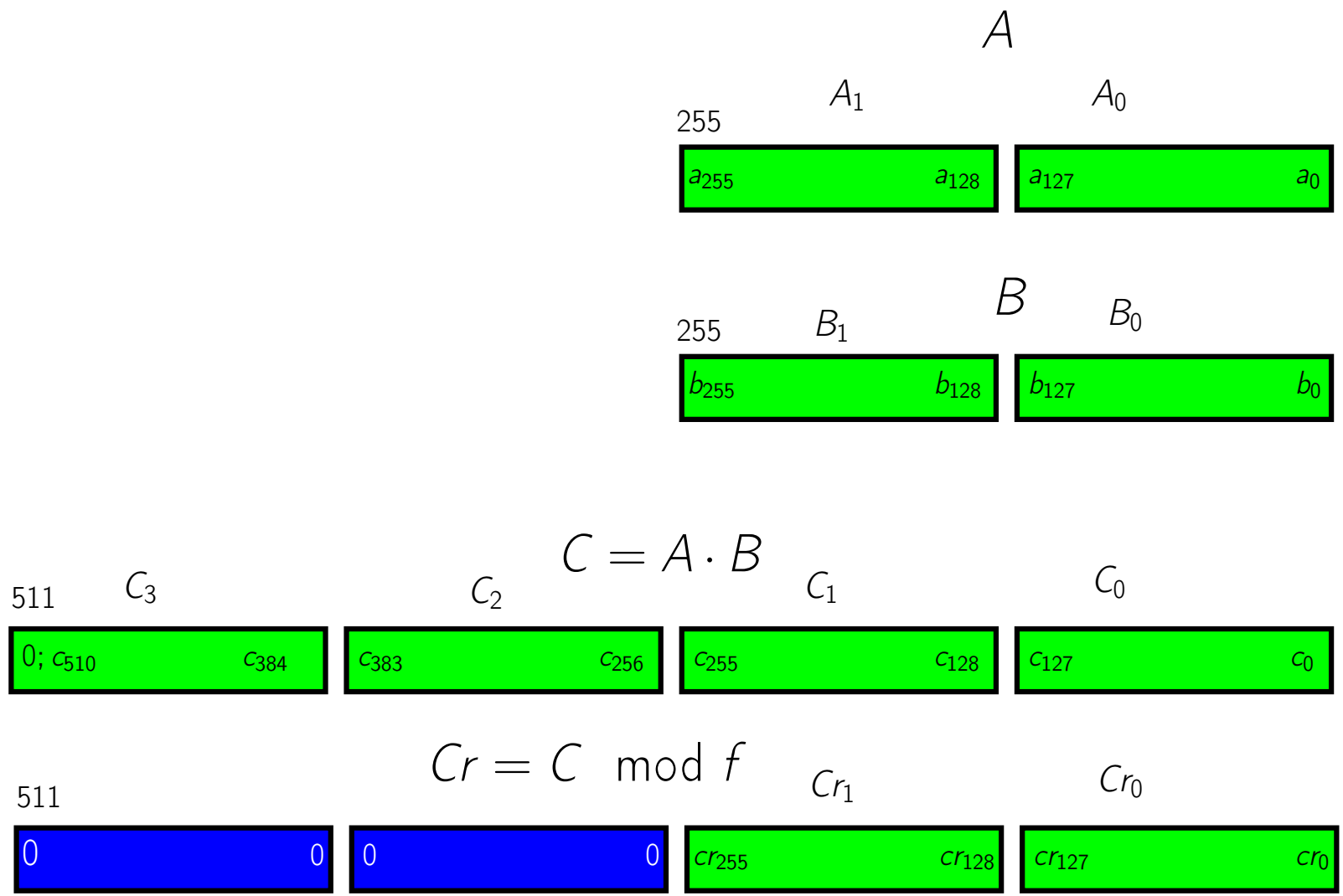


Very Lazy Reduction ($m = 233$, `__m128i`) : le juste nécessaire



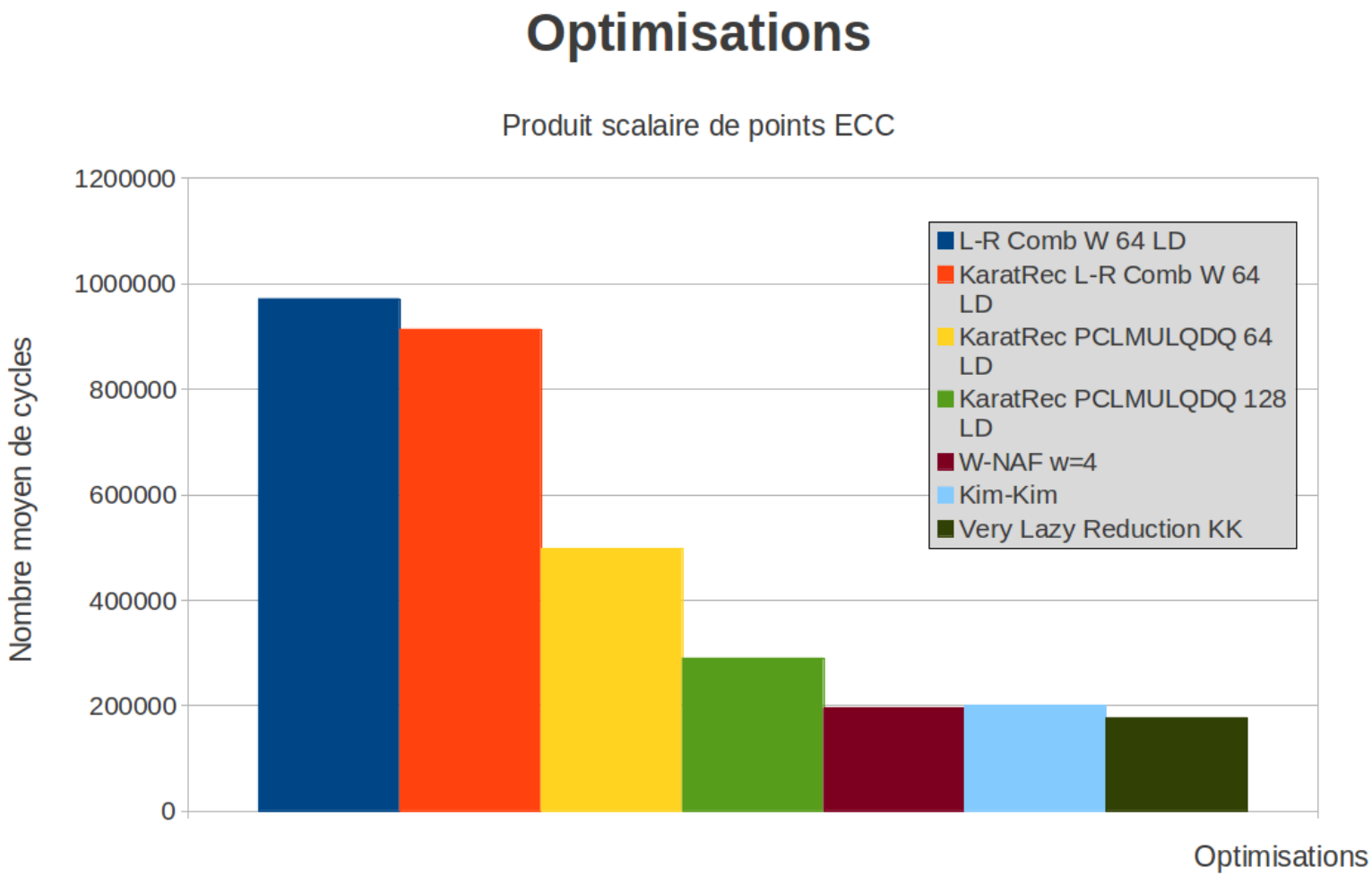
Reduction of C_3, C_2 and C_1.

Very Lazy Reduction ($m = 233$, `__m128i`) : le juste nécessaire



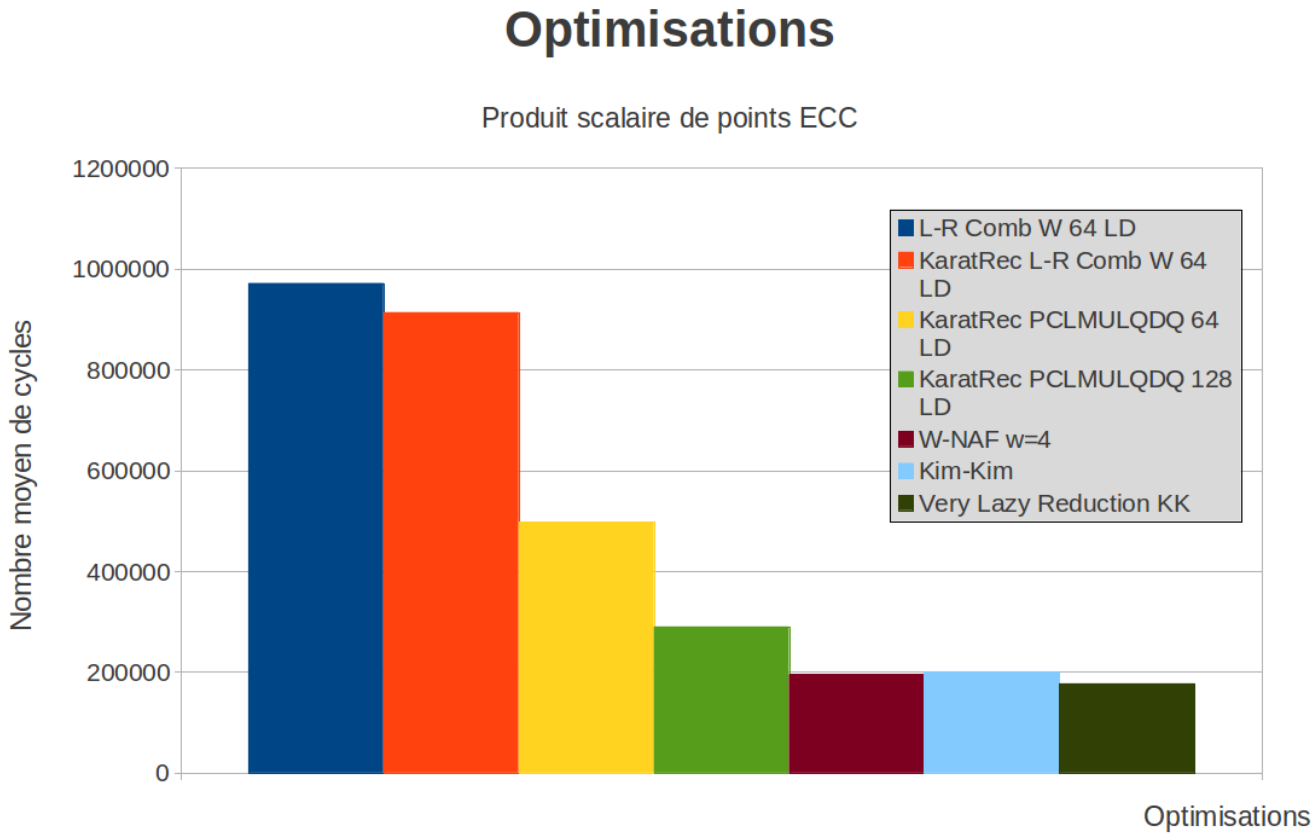
Reduction of C_3 and C_2 only.

Very Lazy Reduction ($m = 233$, `__m128i`) : Performances des Implémentations.



Very Lazy reduction sur *W-NAF* (i5) : -11,9%

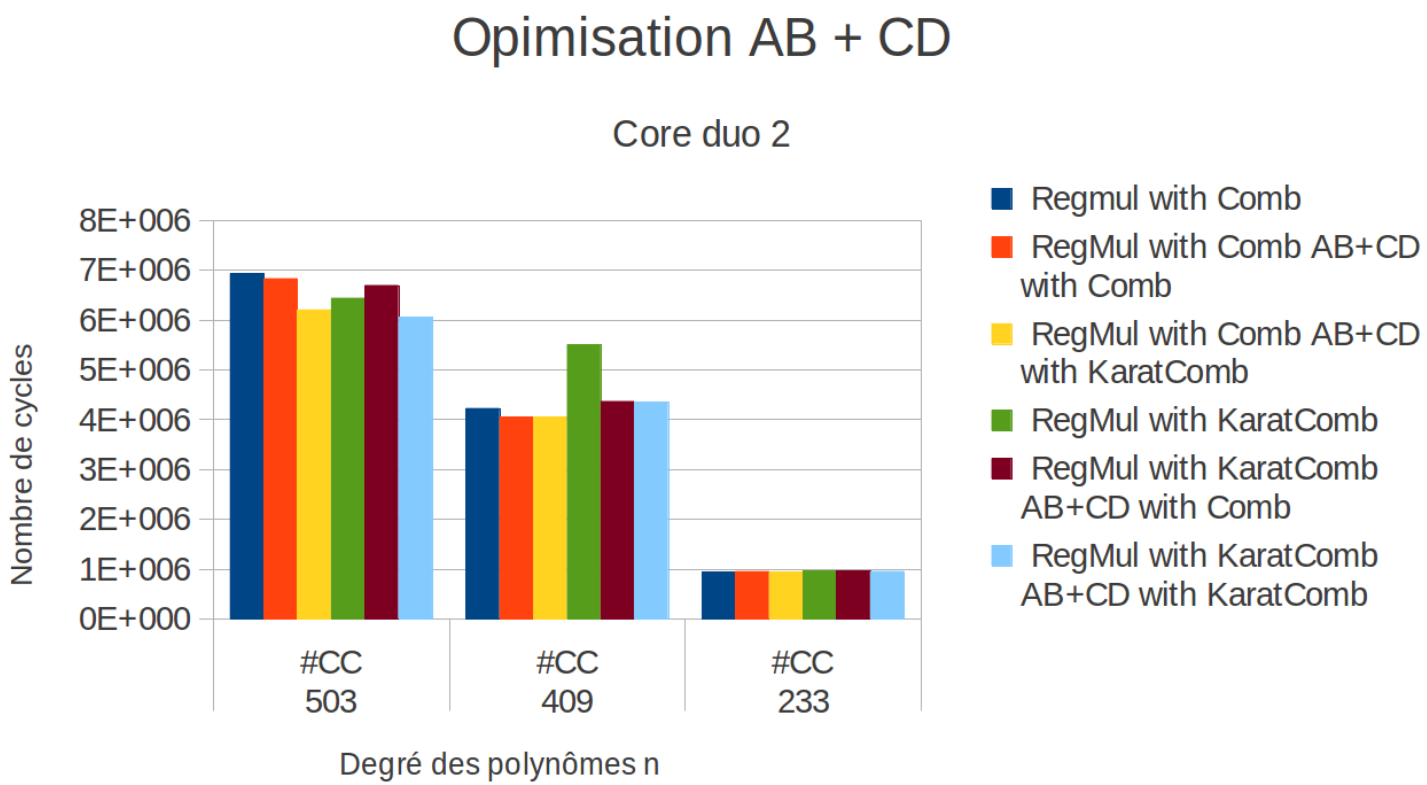
Very Lazy Reduction ($m = 233$, `__m128i`) : Performances des Implémentations.



Comparaison : Nombre de Cycles Moyens (2000 exécutions)			
	Etat de l'Art (i5) [2]	nos résultats (i5)	nos résultats (i7)
W-NAF Double-and-add $m = 233$, $w = 4$	216700	177419 (18.13%)	210195 (3.002%)

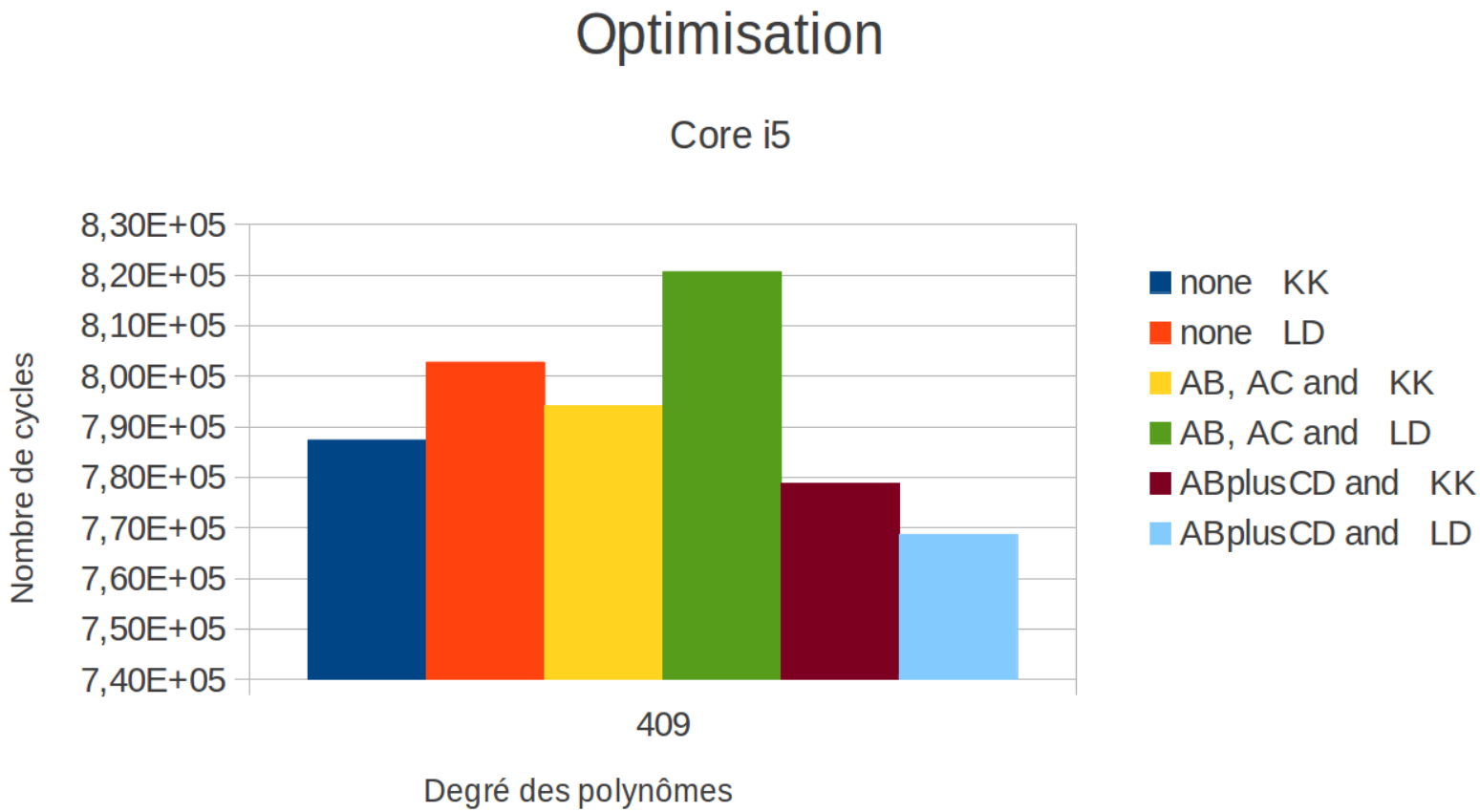
[2] selon J. Taverne et al., CHES 2011.

Optimisation AB, AC et $AB + CD$, Principaux Résultats :



- $m = 233$: Regmul with Comb reste le meilleur.
- $m = 409$: Regmul with Comb and AB+CD with Comb est 4,1% meilleure que Regmul with Comb.
- $m = 503$: Regmul with Karat-Comb and AB+CD with Karat-Comb est 5,9% meilleure que Regmul with Karat-Comb.

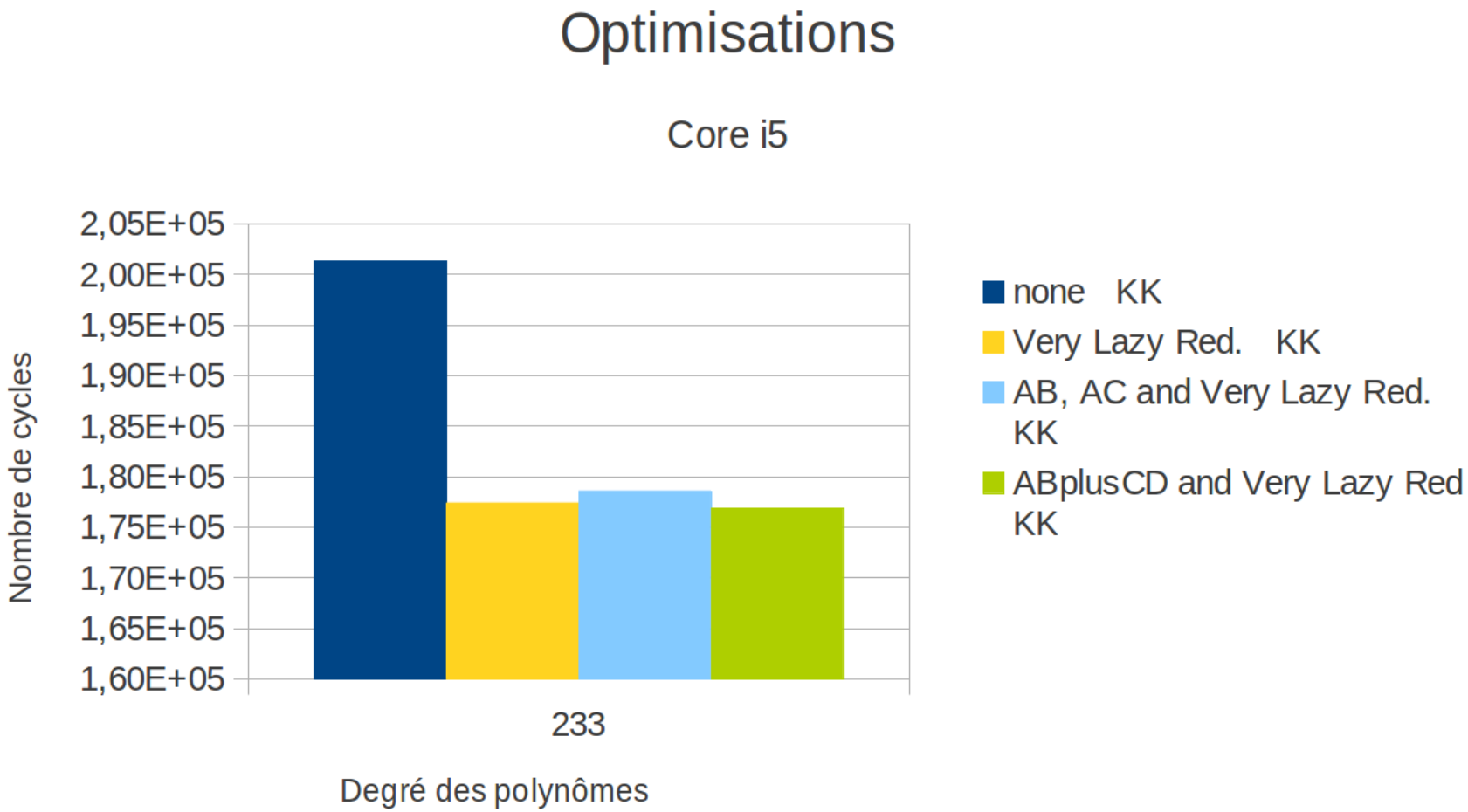
Optimisation AB, AC et $AB + CD$, Principaux Résultats :



- AB, AC : L'optimisation n'apporte rien.
- $AB + CD$: Meilleure configuration avec opérations LD (768711 cycles soit 2,4% de mieux que none KK).

Toutes versions avec Very Lazy Reduction.

Optimisation AB, AC et $AB + CD$, Principaux Résultats :



- AB+CD and Very Lazy Reduction KK : -0,3% /Very Lazy Red. KK)

Toutes versions avec Very Lazy Reduction.

Algorithmes et Implémentations Optimisées

Conclusion et Perspectives :

- Nous avons atteint l'état de l'art en matière de performance sur les produits scalaires ;
- Nous avons testé deux optimisations :
  - AB, AC ;
  - $AB + CD$;

Conclusion et Perspectives :

- Nous avons atteint l'état de l'art en matière de performance sur les produits scalaires ;
- Nous avons testé deux optimisations :
  - AB, AC ;
  - $AB + CD$;

Des gains sont observés dans certaines conditions :
$AB + CD, m = 409, m = 503.$

Conclusion et Perspectives :

- Nous avons atteint l'état de l'art en matière de performance sur les produits scalaires ;
- Nous avons testé deux optimisations :
  - AB, AC ;
  - $AB + CD$;

Des gains sont observés dans certaines conditions :
$AB + CD$, $m = 409$, $m = 503$.

Les performances ne dépendent pas seulement des algorithmes, mais aussi :

- Des types de variables choisis (64 ou 128 bits) ;
- Du compilateur ;
- Du processeur (tailles des mémoires cache, pipelining...).

Nous avons exploré une partie de la combinatoire des possibilités.

Je vous remercie de votre attention,

et suis à l'écoute de vos questions ?