Sorbonne Université

Rapport de projet

Implantations efficaces de calculs sur les polynômes à une variable : FFT

Pierre Lin
Enzo Roaldes
DM-IM 2022 L2

Encadrant : V. Neiger
Responsable : N. Benabbou

20 Janvier 2022 — 11 Juillet 2022

Table des matières

Introduction **2**

1 Algorithmes Naïf et de Karatsuba **3**

1.1 Implémentation 3

1.2 Comparaison - Naïf/Karatsuba 3

2 Fast Fourier Transform (FFT) **4**

2.1 Multiplication de polynômes par évaluation-interpolation 4

2.2 Évalution d'un polynôme (DFT) 5

2.3 Implémentation optimisée de la FFT sur le corps $\mathbb{Z}/p\mathbb{Z}$ 6

2.3.1 Utilisation des unsigned int (Uint) et choix de p 6

2.3.2 Opérations modulo p 6

2.3.3 Optimisation de la DFT 7

3 Implémentation de la vectorisation dans la FFT avec AVX2 **9**

3.1 Opérations modulo p avec AVX2 9

3.2 Comparaison de temps des opérations avec et sans AVX2 10

3.3 Implémentation dans la DFT 10

3.4 Multiplication des polynômes par FFT 12

3.5 Améliorations 13

4 Temps Finaux **15**

Introduction

Les polynômes sont un objet mathématique fondamental, qui apparaît de manière omniprésente dans de nombreux problèmes et applications provenant de contextes scientifiques variés. Par conséquent, les calculs sur les polynômes (addition, multiplication, division, PGCD, solutions d'équations...) forment une composante essentielle de solutions algorithmiques et logicielles conçues pour répondre aux besoins de ces contextes d'utilisation.

Les sciences du numérique fournissent un champ naturel d'applications pour ce type de calculs, par exemple concernant la cryptologie et les codes correcteurs d'erreurs. Par ailleurs, ces domaines demandent souvent de résoudre des instances dont la taille est particulièrement importante. Il est donc essentiel de disposer d'implantations haute-performance pour ce type de calculs. Ces implantations doivent se concentrer sur les meilleurs algorithmes connus, et exploiter au mieux les techniques les plus récentes disponibles via les processeurs modernes. [1]

Ces opérations fondamentales sur les polynômes sont en réalité partout et réalisées un nombre incalculable de fois par seconde par nos ordinateurs. La moindre amélioration de performance pour réaliser ces opérations n'en sera donc que démultipliée par leur utilisation continue.

Dans le cadre de l'UE LU2IN013, nous avons réalisé un projet sur l'optimisation de la multiplication de deux polynômes à une variable dont les coefficients appartiennent à $\mathbb{Z}/p\mathbb{Z}$.

Pour ce faire, nous nous intéressons à plusieurs type d'algorithmes pour la multiplication, en particulier : l'algorithme naïf, de Karatsuba et de multiplication par Fast Fourier Transform (FFT).

Nous avons tout d'abord débuté nos expérimentations sur Python avant de nous tourner vers l'utilisation d'un langage de plus bas niveau, nécessaire pour améliorer la vitesse d'exécution d'un programme, le langage C.

Nous exploiterons aussi des techniques de "vectorisation" qui permettent d'effectuer certains calculs en parallèle sur des vecteurs de données.

Vous pouvez retrouver notre code ici : *https://github.com/PiAir2/IN013_Projet*

1. Tiré de l'introduction du sujet numéro 7.

1 Algorithmes Naïf et de Karatsuba

1.1 Implémentation

Pour commencer, nous avons réalisé un algorithme simple (l'algorithme naïf) de multiplication de deux polynômes P et Q de degrés n. Cet algorithme consiste à développer le produit terme à terme, comme nous le ferions à la main, c'est-à-dire que nous écrivons :

$$R(X) = PQ(X) = \sum_{i=0}^{n} \sum_{j=0}^{n} p_i q_j X^{i+j}$$

où $p_0, \ldots, p_n \; et \; q_0, \ldots, q_n$ sont les coefficients respectifs de P et Q. Ici, à chaque tour de boucle, il faut appliquer un modulo p sur les opérations de produits et de sommes pour que les coefficients restent dans $\mathbb{Z}/p\mathbb{Z}$.
De par sa simplicité, cet algorithme nous permet de vérifier les résultats de nos futurs algorithmes plus performants mais plus complexes à implémenter.

Par la suite, grâce aux différentes sources trouvées, nous avons implémenté l'algorithme de Karatsuba. Nous avons appliqué l'algorithme se trouvant sur la source [2] sans oublier d'appliquer les modulos lors des différentes opérations élémentaires ($+$, $-$, $*$, ...).

1.2 Comparaison - Naïf/Karatsuba

Afin de pouvoir comparer nos différents algorithmes, nous allons parler dans toute la suite de notre rapport de complexité algorithmique, cette complexité est calculée en fonction du nombre d'opérations dans $\mathbb{Z}/p\mathbb{Z}$ effectuée par nos algorithmes. De plus, sauf indication contraire, les temps donnés seront en secondes.

La complexité de l'algorithme naïf est en $O(n^2)$ et celui de Karatsuba en $O(n^{log_2(3)}) \simeq O(n^{1.58})$. Voici les temps d'exécution de ces deux algorithmes :

Degré de P et Q	Naïf	Karatsuba
2^{15}	0.429183	0.085549
2^{16}	1.761229	0.242966
2^{17}	6.875559	0.645986
2^{18}	28.165627	2.023236

Tableau 1 – *Temps de l'algorithme naïf et de l'algorithme de Karatsuba.*

Ici, nous observons que lorsque nous multiplions le degré du polynôme par 2 il y a environ un facteur $2^2 = 4$ pour le temps d'exécution de l'algorithme naïf, contre seulement $2^{1.58} \simeq 3$ pour Karatsuba. Ces temps sont bien cohérents avec les complexités que nous avons citées.

Par ailleurs, nous n'avons pas vraiment cherché à optimiser l'algorithme de Karatsuba car il existe l'algorithme de multiplication par FFT qui est encore plus rapide.

2 Fast Fourier Transform (FFT)

2.1 Multiplication de polynômes par évaluation-interpolation

Une fois nos deux premières implémentations terminées, nous avons cherché à implémenter un troisième algorithme encore plus performant que celui de Karatsuba pour des polynômes de degrés supérieurs à quelques centaines (selon les implémentations). Le nom de cet algorithme est la FFT. L'algorithme se base sur le théorème suivant :

Théorème 1. ***(Interpolation de Lagrange) :*** *Soient $a_1, \ldots, a_n$ et $b_1, \ldots, b_n$ dans $\mathbb{Z}/p\mathbb{Z}$ (avec les a_i deux à deux distincts). Alors il existe un unique polynôme P de degré $< n$ tel que :*

$$\forall i \in \{1, ..., n\},\ P(a_i) = b_i.$$

En effet, l'idée est d'évaluer les polynômes P et Q en n points spécifiques. Ces n points sont les puissances successives d'une racine principale n-ième de l'unité du corps $\mathbb{Z}/p\mathbb{Z}$. Puis, il faut faire le produit de ces évaluations et retrouver l'unique polynôme $R = PQ$ à partir des valeurs du produit. Pour pouvoir retrouver PQ par cette méthode, grâce au *Théorème 1*, il suffit que $n > deg(PQ)$.
Pour appliquer cet algorithme nous nous sommes servis de la source [1].

Algorithme de multiplication par FFT

Entrée. P et Q deux polynômes, n une puissance de 2 avec $n > deg(PQ)$ et ω une racine principale n-ième de l'unité.
Sortie. $R = PQ$
Algorithme.

1. *Pré-calcul.* Calculer les puissances $\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1}$.
2. *Évaluation (DFT).* Calculer les valeurs :
   $Eval(P) = (P(\omega^0), \ldots, P(\omega^{n-1}))$; $Eval(Q) = (Q(\omega^0), \ldots, Q(\omega^{n-1}))$.
3. *Produit point à point.* $Eval(R) = (PQ(\omega^0), \ldots, PQ(\omega^{n-1}))$.
4. *Interpolation.* Nous retrouvons R grâce à $Eval(R)$.

Nous pouvons voir que la complexité des étapes de pré-calcul et de produit point à point est en $O(n)$. Notre problème est désormais de voir comment effectuer les étapes d'évaluation et d'interpolation rapidement.

2.2 Évalution d'un polynôme (DFT)

L'algorithme d'évaluation d'un polynôme est la deuxième étape de la multiplication de polynômes par FFT. Plus communément appelé Discrete Fourier Transform (DFT), il se base sur le principe de diviser pour régner. Soient $n = 2^k$, $P(X) = p_n X^n + \ldots + p_1 X + p_0$ et P_0, P_1 les polynômes composés des coefficients de rang respectivement pair et impair de P, c'est-à-dire : $P_0(X) = p_n X^{n/2} + \ldots + p_2 X + p_0$ *et* $P_1(X) = p_{n-1} X^{(n-2)/2} + \ldots + p_3 X + p_1$.
Nous avons alors que $P(X) = P_0(X^2) + X P_1(X^2)$.
La DFT se base sur cette propriété. En effet, il est clair que, avec une telle décomposition, nous pouvons faire un algorithme récursif où le degré est divisé par deux à chaque occurrence, tout en mettant au carré les points où nous évaluons le polynôme.

Algorithme DFT

Entrée. $P = p_0 + \ldots + p_{n-1} X^{n-1}$, n une puissance de 2 et le tableau $[1, \omega, \ldots, \omega^{n-1}]$ où ω est une racine n-ième principale de l'unité.
Sortie. $P(1), \ldots, P(\omega^{n-1})$.
Algorithme.

1. Si $n = 1$, renvoyer p_0.
2. Sinon, soit $k = n/2$. Calculer :

$$R_0(X) = \sum_{i=0}^{k-1} (p_i + p_{i+k}) X^i$$

$$R_1(X) = \sum_{i=0}^{k-1} (p_i - p_{i+k}) \omega^i X^i$$

3. Calculer récursivement $R_0(1), R_0(\omega^2), \ldots, R_0((\omega^2)^{k-1})$ et $R_1(1), R_1(\omega^2), \ldots, R_1((\omega^2)^{k-1})$.
4. Renvoyer $R_0(1), R_1(1), R_0(\omega^2), R_1(\omega^2), \ldots, R_0((\omega^2)^{k-1}), R_1((\omega^2)^{k-1})$.

Nous voyons dans la deuxième étape de l'algorithme qu'il est nécessaire de faire une boucle allant de 0 à $\frac{n}{2} - 1$ pour les polynômes R_0 et R_1. Cette opération s'effectue en $O(n)$. De plus, le degré étant divisé par deux à chaque tour de récursion, la complexité de la DFT est donc en $O(n\ log_2(n))$.

Par ailleurs, nous savons démontrer que pour l'étape d'interpolation, il suffit essentiellement de réutiliser l'algorithme de DFT mais en appelant la fonction avec les coefficients $Eval(R) = Eval(PQ)$, et, avec l'inverse des racines principales n-ième de l'unité, c'est-à-dire, $\omega^0, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}$. Finalement, la complexité de la FFT est en $O(n\ log_2(n))$.

2.3 Implémentation optimisée de la FFT sur le corps $\mathbb{Z}/p\mathbb{Z}$

2.3.1 Utilisation des unsigned int (Uint) et choix de p

Comme nous travaillons sur $\mathbb{Z}/p\mathbb{Z}$, les coefficients des polynômes sont tous positifs. Cela nous permet de travailler avec des $unsigned\ int\ (Uint)$ et ainsi avec des entiers plus grands (jusqu'à $2^{32}-1$ au lieu de $2^{31}-1$ avec les int). Notre nombre premier valant initialement $p = 2013265921$ ($2^{30} < p < 2^{31}$), les $Uint$ nous permettaient de pouvoir stocker la somme de deux entiers a et b entre 2^{30} et $p-1$, alors que ce n'était possible avec les int car :
$a + b \geq 2 \times 2^{30} = 2^{31}$.

Malheureusement lors de l'implémentation de la vectorisation avec AVX2 dans notre code, nous avons constaté qu'AVX2 ne permettait que de stocker des int, c'est pourquoi nous avons décidé de prendre un nombre premier plus petit, $p = 754974721$ qui est entre 2^{29} et 2^{30}.

De plus, le choix de p doit être judicieux pour pouvoir trouver des racines principales n-ième de l'unité. Effectivement, ici $p = 754974721 = 1 + 2^{24} * 3^2 * 5$, une racine primitive $p-1$-ième de l'unité est $r = 11$, c'est-à-dire que $11^{p-1} \ \% \ p = 1$ et que $11^k \ \% \ p \neq 1$ pour tout k dans $\{1, \ldots, p-2\}$. Nous posons $ordre = p-1$, l'ordre de la racine. Ainsi, pour trouver une racine principale n-ième de l'unité pour la FFT, nous avons la formule $r\_principale = r^{ordre/n} \ \% \ p$.

Nous remarquons aussi dans cette formule qu'il faut que $ordre$ soit divisible par n, avec n une puissance de 2. C'est pourquoi la décomposition de p doit contenir une grande puissance de 2 pour multiplier des polynômes de degré conséquent : ici, $ordre = p-1 = 2^{24} * 3^2 * 5$. Nous comprenons donc qu'avec ce nombre premier muni de la racine $r = 11$, nous pouvons multiplier au plus des polynômes de degrés 2^{23} (2^{24} étant le degré maximum du polynôme résultat de la multiplication).

2.3.2 Opérations modulo p

Nous avons décidé de faire des fonctions d'addition, de soustraction et de multiplication de deux nombres avec modulo. Comme l'instruction $\%$ prend environ 10 fois plus de temps [2] que les instructions $+$ et $-$, nous avons utilisé la propriété suivante pour optimiser l'addition modulo :

$$(a+b) \ \% \ p = \begin{cases} a+b-p & si\ a+b \geq p \\ a+b & sinon \end{cases}$$

De même pour la soustraction nous utilisons :

$$(a-b) \ \% \ p = \begin{cases} p-(b-a) & si\ b > a \\ a-b & sinon \end{cases}$$

2. Temps observé sur nos machines.

Pour la multiplication, nous avons tout de même utilisé l'opération $\%$ tout en castant la multiplication en $long$ car la multiplication de deux coefficients peut facilement dépasser le nombre maximum des $Uint$. Il existe cependant des algorithmes plus efficaces pour faire le modulo dans la multiplication comme la réduction de Barrett que nous allons utiliser dans la partie 3.1.

2.3.3 Optimisation de la DFT

Nous avons d'abord fait une première version fonctionnelle mais non optimisée de la DFT (voir la fonction $eval\_malloc()$) avec beaucoup de malloc inutiles. Par exemple, nous avons malloc, à chaque tour de récursion dans la DFT, les tableaux des polynômes R_0 et R_1 de l'algorithme. Nous avons alors décidé de faire un profilage de cette fonction dont voici le résultat :

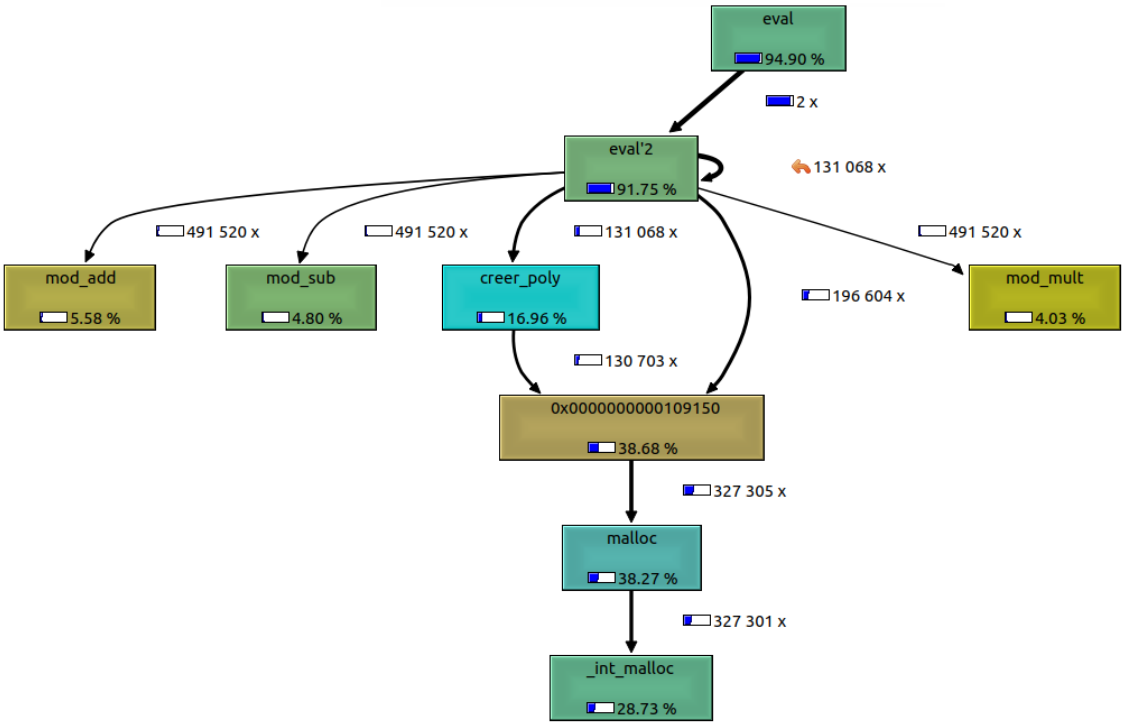

Figure 1 – *Profilage de notre première version de la DFT.*

Constatant que les malloc représentent plus de 40% du temps d'exécution de notre fonction, notre priorité a été de réduire le nombre de malloc.

Nous avons d'abord remarqué que nous pouvions enlever le malloc pour le tableau $racines\_bis$. En effet, ce tableau sert à stocker les valeurs des cases $0, 2, \ldots, n$ du tableau $racines$, il était donc possible de se passer du tableau $racines\_bis$ en donnant en paramètre un pas, pas_rac, que nous multiplions par deux à chaque tour.

Concernant le malloc des tableaux de R_0 et R_1 de tailles $\frac{n}{2}$, nous avons pu les enlever en stockant les coefficients de R_0 dans la première moitié du tableau de P (de taille n) et ceux de R_1 dans la seconde moitié. Ceci étant grâce au fait que nous n'ayons plus besoin de P aux prochains tours de récursion. Comme espéré, la nouvelle fonction optimisée (fonction $eval()$) est plus rapide que la première version :

n	$eval\_malloc()$	$eval()$
2^{21}	0.769839	0.216310
2^{22}	1.454626	0.561354
2^{23}	2.825178	1.398087
2^{24}	5.641160	3.615043

Tableau 2 – *Comparaison des temps d'exécution entre l'ancienne version de notre DFT ($eval\_malloc()$) et la nouvelle version ($eval()$).*

3 Implémentation de la vectorisation dans la FFT avec AVX2

Pour que notre FFT soit encore plus rapide, nous avons eu recours à la vectorisation avec AVX2. AVX2 nous permet de faire un certain nombre d'opérations (addition, soustraction...) beaucoup plus rapidement qu'avec les opérateurs de base du processeur que utilisons en C ($+$, $-$, $*$, ...). En effet, AVX2 permet de créer un type de variable ($\_\_m256i$) pouvant stocker 8 entiers de 32 bits (à l'image d'un tableau), ce qui convient parfaitement à la structure des polynômes. AVX2 pouvant réaliser une opération élémentaire sur deux $\_\_m256i$ aussi vite qu'une opération élémentaire sur deux int en C, il est donc possible d'aller jusqu'à 8 fois plus vite avec AVX2. Voici un exemple de l'addition de deux $\_\_m256i$ en AVX2 :

```
__m256i a = _mm256_set_epi32(1, 2, 3, 4, 5, 6, 7, 8);
__m256i b = _mm256_set_epi32(1, 0, 1, 0, 1, 0, 1, 0);
__m256i x = _mm256_add_epi32(a, b);
// x == [2, 2, 4, 4, 6, 6, 8, 8]
```

Toutes les fonctions utilisées peuvent être retrouvées à l'adresse suivante :
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX,AVX2

3.1 Opérations modulo p avec AVX2

Pour des raisons de lisibilité nous allons illustrer cette partie avec des exemples de tableaux de taille 2.

Le problème avec AVX2 est qu'il n'y a pas d'opération modulo, de plus, comme nous opérons directement sur 8 entiers en même temps, il est difficile de faire le modulo comme dans la partie 2.3.2. Par exemple, pour l'addition supposons que :
$p = 7,\ \overrightarrow{a} = [6, 2],\ \overrightarrow{b} = [4, 1]$ et $\overrightarrow{x} = \overrightarrow{a} + \overrightarrow{b}$
Nous avons $\overrightarrow{x}[0] = 10$ et $\overrightarrow{x}[1] = 3$, nous devrions donc retirer p pour faire le modulo sur $\overrightarrow{x}[0]$, mais alors nous aurions $\overrightarrow{x}[1] = 3 - 7 = -4$, ce qui n'est pas le résultat attendu.

Pour faire l'addition et la soustraction modulo, il faut savoir qu'AVX2 nous permet de prendre le minimum "positif" sur chaque case de deux $\_\_m256i$ (tous les nombres négatifs vont être converti en $Uint$), par exemple :
$min\_pos([1, 0],\ [0, -10]) = min([1, 0],\ [0, \text{UINT\_MAX}-10]) = [0, 0]$ et non $[0, -10]$.

Pour l'addition, avec $\overrightarrow{x} = \overrightarrow{a} + \overrightarrow{b}$, nous faisons l'opération $min\_pos(\overrightarrow{x},\ \overrightarrow{x} - \overrightarrow{p})$ où $\overrightarrow{p}$ est un $\_\_m256i$ ne contenant que des p. Ceci permet bien de faire le modulo car, si $\overrightarrow{a}[i] + \overrightarrow{b}[i] = \overrightarrow{x}[i] \geq p$, alors l'appel de la fonction retourne $\overrightarrow{x}[i] - p$. Sinon, nous avons $\overrightarrow{a}[i] + \overrightarrow{b}[i] = \overrightarrow{x}[i] < p$, donc $\overrightarrow{x}[i] - p < 0$, et après conversion en $Uint$, $\overrightarrow{x}[i] - p$ sera plus grand que $\overrightarrow{x}$[i] et donc l'opération retournera $\overrightarrow{x}$[i]. Ceci correspond bien aux différents cas vus dans la partie 2.3.2.
C'est la même logique pour la soustraction $\overrightarrow{x} = \overrightarrow{a} - \overrightarrow{b}$, nous prenons $min\_pos(\overrightarrow{x},\ \overrightarrow{x} + \overrightarrow{p})$.

Pour la multiplication, nous avons utilisé l'algorithme de réduction de Barrett, nous nous sommes inspirés de la source [3] qui propose une implémentation de cet algorithme avec vectorisation sur des entiers 16 bits.

3.2 Comparaison de temps des opérations avec et sans AVX2

Dans cette partie, nous allons comparer les temps d'exécution pour les opérations d'addition, de soustraction et de multiplication modulo p sans AVX et avec AVX. Ces fonctions sont dans le fichier $vect.c$ et sont respectivement : $vect\_mod\_add()$, $vect\_mod\_sub()$ et $vect\_mod\_mult()$. Pour les comparer, nous avons utilisé ces fonctions sur deux tableaux de taille 8 000 000.

Opération	Sans AVX2	Avec AVX2
Addition	0.012610	0.006697
Soustraction	0.014247	0.007139
Multiplication	0.015053	0.007497

Tableau 3 – *Temps des opérations d'addition, soustraction et multiplication sans AVX2 et avec AVX2.*

Ici, nous gagnons un facteur environ 2 grâce à AVX2. Il ne nous reste plus qu'à implémenter ces fonctions dans la DFT.

3.3 Implémentation dans la DFT

Nous voyons que dans l'algorithme de DFT, nous devons faire des opérations d'addition ($p_i + p_{i+k}$), de soustraction et de multiplication ($(p_i - p_{i+k}) * \omega^i$).
L'implémentation de l'addition et de la soustraction vectorisée dans la DFT est plutôt directe, nous faisons une boucle dont l'indice augmente de 8 en 8 en passant les tableaux de taille 8 correspondants dans les fonctions de vectorisation.

De plus, nous pouvons remarquer que l'addition et la soustraction se font avec les mêmes coefficients (p_i et p_{i+k}). Nous voyons aussi que dans les fonctions $vect\_mod\_add()$ et $vect\_mod\_sub()$, nous devons charger les tableaux en $\_\_m256i$ à chaque fois (avec la fonction $\_mm256\_loadu\_si256()$). Pour optimiser ces chargements, nous avons donc créé la fonction $vect\_mod\_add\_sub\_eval()$ qui permet de faire l'addition et la soustraction en ne chargeant qu'une fois les deux tableaux. Le code commenté de cette fonction est disponible sur la prochaine page.

```
void vect_mod_add_sub_eval(Uint *res_add, Uint *res_sub, Uint *tab1,
                           Uint *tab2) {
    __m256i x, result;
    // Chargement du tableau avec que des p
    __m256i p = _mm256_set1_epi32(NB_P);
    // Chargement de 8 cases successives dans tab1
    __m256i a = _mm256_loadu_si256((__m256i *) tab1);
    // Chargement de 8 cases successives dans tab2
    __m256i b = _mm256_loadu_si256((__m256i *) tab2);

    // Addition modulo p
    // x = a + b
    x = _mm256_add_epi32(a, b);
    // min_pos(x, x - p) = (a+b)%p
    result = _mm256_min_epu32(x, _mm256_sub_epi32(x, p));
    // Stockage du resultat dans res_add
    _mm256_storeu_si256((__m256i *) res_add, result);

    // Soustraction modulo p
    // x = a - b
    x = _mm256_sub_epi32(a, b);
    // min_pos(x, x + p) = (a-b)%p
    result = _mm256_min_epu32(x, _mm256_add_epi32(x, p));
    // Stockage du resultat dans res_sub
    _mm256_storeu_si256((__m256i *) res_sub, result);
}
```

Désormais, pour la multiplication $(p_i - p_{i+k}) * \omega^i$, le principal problème que nous avons rencontré est le pas pas_rac pour le tableau $racines$. En effet, nous voyons que dans la fonction sans AVX2 : $eval()$, dans la première boucle for, nous devons accéder à la case $i * pas\_rac$. Donc en travaillant sur des tableaux de taille 8 avec AVX2, il faudrait que nous accédions aux cases $(i+0) * pas\_rac, (i+1) * pas\_rac, \ldots, (i+7) * pas\_rac$. Malheureusement notre fonction $\_mm256\_loadu\_si256()$ ne nous permet que de charger 8 cases à la suite (les cases $i+0, i+1, \ldots, i+7$). Cependant, nous avons trouvé la fonction $\_mm256\_i32gather\_epi32()$ qui prend en paramètre deux tableaux $\_\_m256i$: tab et $indices$ où $indices$ contient les indices des éléments que nous voulons avoir dans tab, la fonction prend aussi un entier $scale$ qui représente la taille en octet des éléments dans tab. Ce qui fait qu'ici, tab correspond au tableau $racines$, $indices$ au tableau $[(i+0) * pas\_rac,\ (i+1) * pas\_rac,\ \ldots,\ (i+7) * pas\_rac]$ et $scale$ à 4 (taille d'un int).

Pour charger les cases voulues du tableau $racines$, il nous suffit alors de créer $indices$ grâce aux tableaux :
$\overrightarrow{i} = [i,\ \ldots,\ i]$, $\overrightarrow{u} = [0, 1, 2, 3, 4, 5, 6, 7]$ et $\overrightarrow{pas} = [pas\_rac,\ \ldots,\ pas\_rac]$.
Ce qui nous donne $indices = (\overrightarrow{i} + \overrightarrow{u}) * \overrightarrow{pas}$.

Voici donc les temps obtenus après ces implémentations :

n	DFT Sans AVX2	DFT Avec AVX2
2^{20}	0.060076	0.056392
2^{21}	0.175610	0.147659
2^{22}	0.380583	0.333849
2^{23}	0.975016	0.775812
2^{24}	2.470365	2.090264

Tableau 4 – *Temps de la DFT sans AVX2 et avec AVX2.*

Nous pouvons voir que le facteur de temps entre nos versions de la DFT n'est pas énorme, nous allons voir dans la partie 3.5 que nous pouvons faire bien mieux.

3.4 Multiplication des polynômes par FFT

L'étape 2 de l'algorithme de multiplication de deux polynômes par FFT étant terminée, nous allons devoir implémenter les étapes 3 et 4 de cet algorithme.

Tout d'abord, nous pouvons voir que l'étape 3 se résume en une boucle de n multiplications. De plus, grâce à notre algorithme de multiplication modulo avec AVX2, nous pouvons vectoriser cet étape facilement.

Désormais, pour l'étape 4, nous avions vu dans la partie 2.2 qu'il suffisait essentiellement de réutiliser la DFT avec l'inverse des racines principales. En effet, pour faire cet étape nous devons :

- Calculer les puissances successives de l'inverse de la racine principale : $\omega^0, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}$.
- Appliquer la DFT à $Eval(R)$ avec ces inverses.
- Diviser les coefficients obtenus par la DFT par n (c'est-à-dire multiplier par l'inverse de n sur $\mathbb{Z}/p\mathbb{Z}$).

Ces étapes demandent de calculer l'inverse d'un nombre sur $\mathbb{Z}/p\mathbb{Z}$. Nous avons donc implémenté un algorithme inversant un nombre a sur le corps $\mathbb{Z}/p\mathbb{Z}$ en résolvant l'équation $au + pv = 1$ (nous cherchons à trouver u et v), cet algorithme est en fait l'algorithme d'Euclide étendu.
Premièrement, u et v existent, en effet, p étant un nombre premier, nous pouvons en déduire que a et p sont premier entre eux. Donc, par le théorème de Bézout, il existe deux entiers relatifs u et v tels que $au + pv = 1$.
De plus, u est l'inverse de a sur $\mathbb{Z}/p\mathbb{Z}$. En effet, supposons que $au + pv = 1$, comme nous travaillons sur $\mathbb{Z}/p\mathbb{Z}$, nous avons :
$au + pv = (au + pv)\%p = (au)\%p + (pv)\%p = (au)\%p = 1$.
C'est-à-dire que $au = 1$ sur $\mathbb{Z}/p\mathbb{Z}$, u est donc bien l'inverse de a par définition de l'inverse.

Enfin, u étant un entier relatif, il peut être négatif, nous devons donc ajouter p dans ce cas.

En sachant tout cela, nous avons pu implémenter l'algorithme grâce au pseudo-code donné sur ce site.

Avec cet algorithme, nous avons alors pu terminer l'étape 4. Par ailleurs, pour l'étape de division par n, cela était le même concept que pour l'étape 3 de la FFT, nous avons donc pu la vectoriser aussi. Ceci conclut donc l'implémentation de la multiplication de polynômes par FFT.

Après avoir vérifié que notre multiplication par la FFT (avec et sans AVX2) nous donnait des résultats corrects en les confrontant à ceux obtenus avec l'algorithme naïf, nous avons comparé les temps obtenus avec et sans AVX2 :

n	FFT Sans AVX2	FFT Avec AVX2
2^{20}	0.158404	0.150727
2^{21}	0.503089	0.437402
2^{22}	1.240285	1.075995
2^{23}	3.028623	2.317684
2^{24}	7.108149	5.847869

Tableau 5 – *Temps de la multiplication par FFT sans AVX2 et avec AVX2.*

Ici, la DFT représente la majeure partie du temps d'exécution de la FFT. En effet, dans l'algorithme de FFT nous devons exécuter trois DFT, et nous observons un facteur 3 entre les temps des tableaux 4 et 5. C'est pourquoi, tout comme dans le tableau 4, le facteur de temps observé dans le tableau 5 n'est pas très élevé.
Cela montre bien que les étapes en $O(n)$ de la FFT sont négligeables par rapport à celles en $O(n\ log_2(n))$.

3.5 Améliorations

Afin d'améliorer les performances de notre FFT, nous avons décidé de faire quelques tests pour voir quelles opérations nous prenaient le plus de temps dans notre DFT. Nous avons vu que c'était l'étape de multiplication qui prenait énormément de temps, et plus spécifiquement lorsque nous voulons chercher dans le tableau $racines$ les cases voulues. Après réflexion, nous avons repensé à notre fonction $eval\_malloc()$ où nous faisons un $malloc$ à chaque tour pour créer le tableau $racines\_bis$ où toutes les racines que nous utilisons se suivent. Nous avons remis cette mécanique et nous nous sommes débarrassé du pas, pas_rac. Nous avons alors pu utiliser la fonction de multiplication avec AVX2 sans avoir à chercher des indices spécifiques dans le tableau $racines$.

Après tests, cela nous a fait gagner beaucoup de temps pour des degrés grands, les voici :

n	FFT Sans AVX2	FFT Avec AVX2 (Version 2)
2^{20}	0.158404	0.180472 [3]
2^{21}	0.503089	0.355293
2^{22}	1.240285	0.706925
2^{23}	3.028623	1.504402
2^{24}	7.108149	2.939824

Tableau 6 – *Temps de la multiplication par FFT sans AVX2 et avec AVX2 (Deuxième version).*

Pour finir, nous avons essayé de chercher des librairies faisant aussi la FFT (ou DFT, c'est surtout cet algorithme qui importe) vectorisée avec AVX2 sur $\mathbb{Z}/p\mathbb{Z}$ pour voir quelles sont les meilleures implémentations que nous pouvons avoir à ce jour. La première page que nous avons trouvée était la librairie NTL (codée en C++) qui propose cela : *https://libntl.org/*.

Après installation de la librairie sur notre ordinateur, nous avons pu exécuter la DFT que propose NTL pour plusieurs degrés. Pour cela, nous avons pris un nombre premier de taille 27 bits (le notre n'étant pas disponible) et avons exécuté leur DFT pour les degrés disponibles. Voici un tableau comparant directement nos temps et ceux de NTL :

n	No AVX2 (NTL)	No AVX2	AVX2 (NTL)	AVX2 [4]
2^{18}	0.00297	0.01017	0.00182	0.00944
2^{20}	0.01517	0.06008	0.01113	0.06168
2^{22}	0.08613	0.38058	0.07253	0.21169

Tableau 7 – *Comparaison de nos DFT à celles de NTL.*

Nous pouvons dans un premier temps remarquer que sans AVX2 il y a un facteur croissant entre nos deux DFT (allant de 3 à 5). Notre DFT est donc encore largement optimisable mais il nous faudrait beaucoup plus de compétences pour arriver aux performances que propose NTL. NTL est, par ailleurs, la bibliothèque de référence pour la DFT, bénéficiant depuis des années de l'expertise de pointe de la recherche à ce sujet. C'est pourquoi le facteur est si important.

Dans un second temps, nous pouvons remarquer que plus le degré augmente, plus le temps gagné par NTL avec AVX2 diminue. Il est possible que cela soit dû au fait que leur DFT sans AVX2 est très optimisée, ce qui rend la vectorisation ardue.

En revanche, nous observons le comportement inverse pour nos DFT. Cela est dû au fait que nos opérations étaient plus simple à vectoriser. C'est pourquoi, avec AVX2, nous obtenons un facteur décroissant (allant de 5 à 3).

3. Pour les degrés inférieurs à 2^{20} les performances de la deuxième version sont moins bonnes. Il serait donc intéressant de distinguer, selon le degré, quelle version utiliser.

4. Pour les temps AVX2, nous avons pris le meilleur temps entre notre première et deuxième version de la DFT.

4 Temps Finaux

Voici un tableau comportant l'ensemble des résultats que nous avons obtenus grâce à nos différents algorithmes. Concernant les algorithmes naïf et de Karatsuba, nous avons calculé les temps théoriques pour les degrés supérieurs à 2^{18}.

n	Naïf	Karatsuba	FFT Sans AVX2	FFT Avec AVX2 V2
2^{18}	28.17	2.02	0.037	0.045
2^{19}	1.2×10^2	6.07	0.085	0.094
2^{20}	4.6×10^2	18.21	0.16	0.18
2^{21}	1.8×10^3	54.63	0.50	0.36
2^{22}	7.2×10^3	1.6×10^2	1.24	0.71
2^{23}	2.8×10^4	4.8×10^2	3.03	1.50
2^{24}	1.1×10^5	1.4×10^3	7.11	2.94

Tableau 8 – *Comparaison finale de tous nos algorithmes implémentés.*

Nous pouvons constater que la multiplication de deux polynômes par FFT est largement plus rapide qu'avec les algorithmes naïf et de Karatsuba. Cependant, il faut faire attention au fait que la FFT sur $\mathbb{Z}/p\mathbb{Z}$ n'est pas utilisable pour des degrés très grands (selon p), donc cet algorithme a aussi ses limites.

Les temps que nous obtenons peuvent être améliorés, comme nous l'avons vu avec la librairie NTL. Aussi, à l'avenir, de nouvelles méthodes de calcul plus performantes pourraient faire leur apparition, ce qui permettrait l'avènement d'algorithmes encore plus efficaces.
Toutefois, cette course à l'optimisation arrivera-t-elle un jour à terme ou bien sera-t-elle dépassée par l'arrivée de nouvelles technologies comme les ordinateurs quantiques ?

Références

[1] Bostan A., Chyzak F., Giusti M., Lebreton R., Lecerf G., Salvy B., and Schost E. Algorithmes Efficaces en Calcul Formel. Technical report, Université Paris Diderot and Écoles Normales Supérieures de Cachan et de Paris and École Polytechnique, Décembre 2018.

[2] Werner B. and Schost E. Informatique Tronc Commun TD2 Récursivité 2 : l'Algorithme de Karatsuba. Technical report, École Polytechnique, Novembre 1999.

[3] Van Der Hoeven J., Lecerf G., and Quintin G. Modular SIMD Arithmetic in Mathemagix. Technical report, CNRS and École Polytechnique and Laboratoire LIX Université de Limoges and Laboratoire XLIM, Août 2016.