LU2IN013 - Rapport de projet

LIN Clément - XIAO Carine - XU Audic

PIMA 2021/2022

1 Introduction

Notre projet porte sur les algorithmes et le calcul haute performance pour l'algèbre linéaire. Plus précisément, il porte sur le calcul matriciel car en effet, il s'agit d'une composante fondamentale de nombreux problèmes scientifiques. Par ailleurs, ses déclinaisons algorithmiques et logicielles sont intensivement utilisées dans les sciences en général, mais aussi tout particulièrement dans celles du numériques comme en cryptologie, imagerie, etc ... Il est donc essentiel de disposer d'implémentations à hautes performances pour ces types de calculs où l'on s'attend à ce qu'elles exploitent au mieux les opérateurs arithmétiques.

L'objectif sera de commencer avec des algorithmes naïfs. Il faudra alors les étudier et identifier les surcoûts afin d'intégrer différentes techniques d'optimisations. Le but étant d'améliorer leurs comportements pratiques.

Chaque implémentation est lancée sur le gpu-4 de Sorbonne Université possédant 40 coeurs et 1152 Go de RAM. Pour chacune d'elles, nous ajouterons le code et un graphique comparant son temps d'exécution en fonction de la taille de la matrice avec celle de la version naïve. Les gains et pertes de temps en pourcentages sont calculés en fonction du temps d'exécution de la version naïve. Par exemple, si la version naïve met 100 secondes pour des matrices d'une certaine taille, et si l'optimisation apportée à cette dernière réduit le temps d'exécution à 75 secondes, alors le gain est de 25 %.

Nous remercions Mohab Safey El Din, responsable des masters SFPN et HPC à Sorbonne Université, de nous avoir encadré tout au long du projet.

2 Bibliothèques externes

Tout au long du projet, on considère A et B deux matrices de taille $n \in \mathbb{N}$. Notons respectivement $(a_{i,j})_{1 \leq i,j \leq n}$ et $(b_{i,j})_{1 \leq i,j \leq n}$ les entrées des matrices. Afin d'avoir un premier aperçu de la différence de complexité temporelle entre un algorithme naïf et ceux de bibliothèques spécialisées, nous avons installé les bibliothèques externes FLINT et NTL. En effet, toutes deux sont des bibliothèques rapides pour la théorie des nombres, et sont notamment utilisées en cryptologie. FLINT est une bibliothèque `C` tandis que NTL est une bibliothèque `C++`.

Tout d'abord, nous avons multiplié de façon naïve des matrices à coefficient entier int_32. Ensuite avec FLINT où les matrices sont à coefficient mp_limb_t, et avec NTL où les coefficients sont zz_p. Dans les trois cas, nous travaillons dans l'ensemble $\frac{\mathbb{Z}}{p\mathbb{Z}}$ avec p le premier nombre premier après 2^{30}, à savoir 1073741827.

Pour la version naïve, pour tout i, j $\in [\![1, n]\!]$, on a : $a_{i,j} < 2^{31}$ et $b_{i,j} < 2^{31}$ donc $a_{i,j} \times b_{i,j} < 2^{62}$. Il n'y a pas de problème en travaillant avec du int_64, mais avec des coefficients int_32, nous avons dû passer par une variable temporaire de 64 bits. Voici le code des trois fonctions implémentées.

NTL : (C1)

```
void initi(mat_zz_p &mat1, mat_zz_p &mat2, long n){
    mat1.SetDims(n, n);
    mat2.SetDims(n, n);
    random(mat1, n, n);
    random(mat2, n, n);
}

int main(int argc,char **argv){
    long p = 1073741827;
    zz_p::init(p);
    mat_zz_p mat1;
    mat_zz_p mat2;
    mat_zz_p mat3;

    initi(mat1, mat2, atol(argv[1]));
    mul(mat3, mat1, mat2); // multiplication

    clear(mat1);
    clear(mat2);
    clear(mat3);

    return 0;
}
```

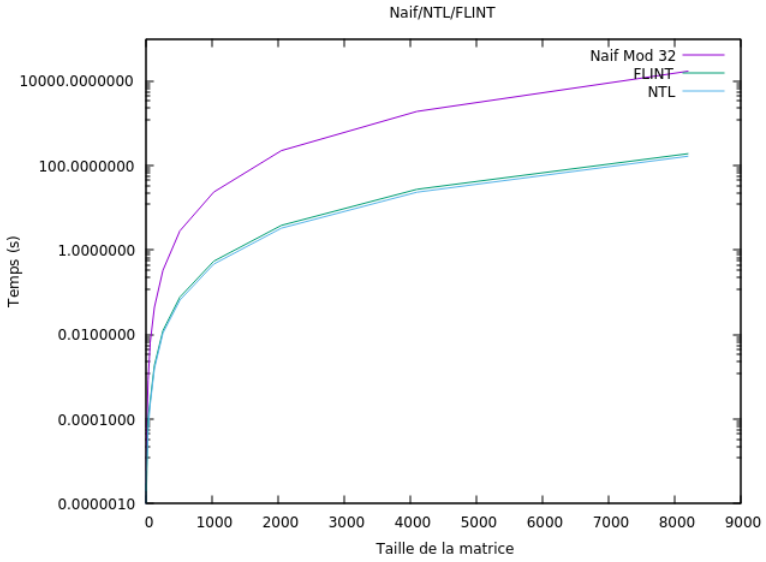
FLINT : (C2)

```
void init(nmod_mat_t mat1, nmod_mat_t mat2, slong taille, mp_limb_t mod){
    FLINT_TEST_INIT(state);
    nmod_mat_init(mat1, taille, taille, mod);
    nmod_mat_init(mat2, taille, taille, mod);
    nmod_mat_randtest(mat1, state);
    nmod_mat_randtest(mat2, state);
}

int main(int argc, char** argv){
    mp_limb_t n = 1073741827;
    nmod_mat_t A;
    nmod_mat_t B;
    nmod_mat_t C;

    nmod_mat_init(C, atoi(argv[1]), atoi(argv[1]), n);
    init(A, B, taille, n);
    nmod_mat_mul(C, A, B); // multiplication

    nmod_mat_clear(A);
    nmod_mat_clear(B);
    nmod_mat_clear(C);

    return 0;
}
```

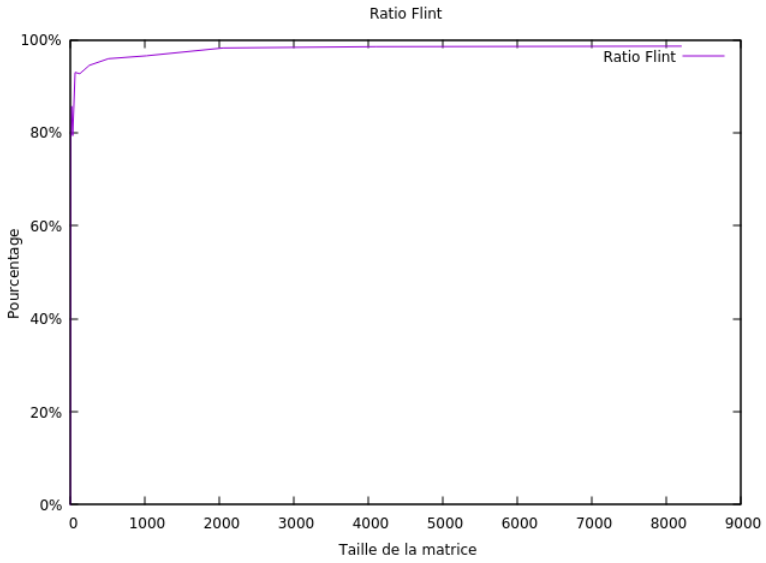
Naïve int_32 : (C3)

```
u_int32_t** produit_matrice32_mod(u_int32_t** mat1, u_int32_t** mat2, int taille, long p){
    u_int32_t** mat = alloue_matrice32(taille);

    for (int i = 0; i < taille; i++){
        for (int j = 0; j < taille; j++){
            for (int k = 0; k < taille; k++){
                u_int64_t stock_modulo = (((u_int64_t) mat1[i][k]) * mat2[k][j]) % p;
                mat[i][j] = (mat[i][j] + stock_modulo) % p;
            }
        }
    }
    return mat;
}
```

Voici le graphe de comparaison construit à partir des codes (C1), (C2) et (C3) :



Nous voyons qu'entre FLINT et NTL, la différence est minime. Alors que la différence avec la version naïve est significative, voici un autre graphique tracé à partir du gain de temps de FLINT en fonction de la taille de la matrice :



Comment explique-t-on cette différence de temps ? Afin d'y répondre, nous allons étudier cette version naïve et implémenter différentes optimisations pour y pallier.

3 Étude de l'algorithme naïf

On rappelle qu'on travaille avec les matrices A et B de taille $n \in \mathbb{N}$, telles que $A \times B = C$. Notons respectivement $(a_{i,j})_{1\leq i,j\leq n}$, $(b_{i,j})_{1\leq i,j\leq n}$ et $(c_{i,j})_{1\leq i,j\leq n}$ les coefficients des matrices. Sauf mention contraire, ces derniers sont tous dans $\frac{\mathbb{Z}}{p\mathbb{Z}}$ avec $p = 1073741827$.

3.1 Complexité théorique

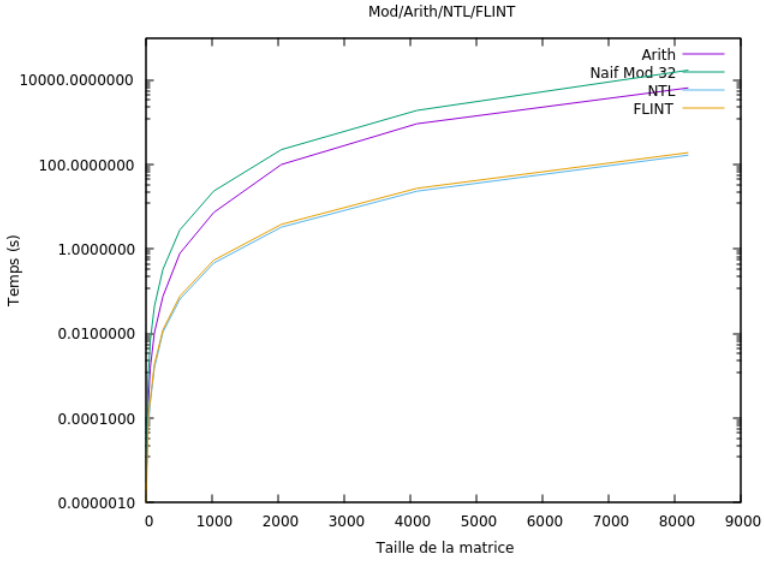
En calculant théoriquement le nombre d'opérations pour multiplier A et B de taille n, on trouve : $2n^3 - n^2$. De plus, pour effectuer 2 000 000 000 opérations sans déplacement dans la mémoire, la machine met environ 3.21 secondes à les effectuer, voici le code :

(C4)

```
void effectue_operations(int taille){
    u_int32_t a = (u_int32_t)rand();

    for (int i = 0; i < taille; i++){
        for (int j = 0; j < taille; j++){
            for (int k = 0; k < taille; k++){
                a += a*a;
            }
        }
    }
}
```

Or pour deux matrices de taille 1024 à coefficient int_32, le nombre d'opérations à effectuer est de 2 146 435 072, ce qui devrait correspondre à 3.44 secondes. Pourtant, avec les données précédentes, nous voyons que le temps mis est de 23.5 secondes.

Nous verrons mieux la différence sur un graphique. Ajoutons la courbe "Arith" qui fait le nombre d'opérations théorique en fonction de la taille des matrices, aux comparaisons que nous avons faites précédemment. Graphique construit à partir des codes (C1), (C2), (C3) et (C4) :



Nous voyons que l'écart entre les courbes "Arith" et "Mod" est tout de même conséquent. À quoi est dûe cette différence ? D'une part, la localité des données, qui est souvent négligée dans les analyses de complexité, peut jouer un rôle assez conséquent. D'autre part, travailler dans $\frac{\mathbb{Z}}{p\mathbb{Z}}$ est plus intéressant mais aussi plus coûteux.

3.2 Localité des données

Dans la partie précédente, nous avons arbitrairement choisi de prendre des matrices à coefficients entiers int_32, mais il existe aussi des entiers int_16 et int_64. Néanmoins, dans le cadre du projet, nous travaillons avec des matrices dont les coefficients peuvent dépasser 16 bits, c'est pourquoi nous allons uniquement travailler avec des int_32 et int_64.

Afin d'observer l'impact de la localité sur la multiplication de matrices avec des int_32 et int_64, nous avons réécrit une version qui effectue les mêmes opérations (dont le modulo) mais sans déplacement mémoire :

Naïve int_64 : (C5)

```
u_int64_t** produit_matrice64_mod(u_int64_t** mat1, u_int64_t** mat2, int taille, long p){
    u_int64_t** mat = alloue_matrice64(taille);

    for (int i = 0; i < taille; i++){
        for (int j = 0; j < taille; j++){
            for (int k = 0; k < taille; k++){
                mat[i][j] += (mat1[i][k] * mat2[k][j]) % p;
            }
            mat[i][j] = mat[i][j] % p;
        }
    }

    return mat;
}
```

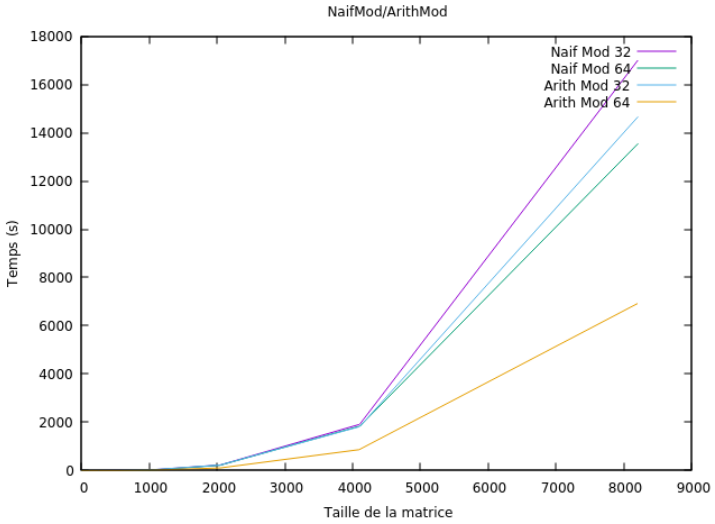
Simulation de produits matriciel int_32 : (C6)

```
void effectue_operations_mod32(int taille, long p){
    u_int32_t a = (u_int32_t)rand();

    for (int i = 0; i < taille; i++){
        for (int j = 0; j < taille; j++){
            for (int k = 0; k < taille; k++){
                a += ((u_int64_t) a) * a % p;
                a = a % p;
            }
        }
    }
}
```

Simulation de produits matriciel int_64 : (C7)

```
void effectue_operations_mod64(int taille, long p){
    u_int64_t a = (u_int64_t)rand();

    for (int i = 0; i < taille; i++){
        for (int j = 0; j < taille; j++){
            for (int k = 0; k < taille; k++){
                a += a*a % p;
            }
        a = a % p;
        }
    }
```

Pour voir la différence de temps, nous avons tracé le graphique suivant avec les codes (C3), (C5), (C6) et (C7) :



De ces observations, nous en déduisons que les déplacements mémoires ne sont pas du tout négligeables.

L'impact de la localité des données n'est pas invisible et devrait donc être utilisé intelligemment.

3.3 Modulo

Il est certain que de travailler dans $\frac{\mathbb{Z}}{p\mathbb{Z}}$ demande un certain coût. La question est de savoir de combien. Nous avons vu précédemment des codes calculant la multiplication de matrice int32_t et int_64 avec modulo. Il nous faut donc celles sans :

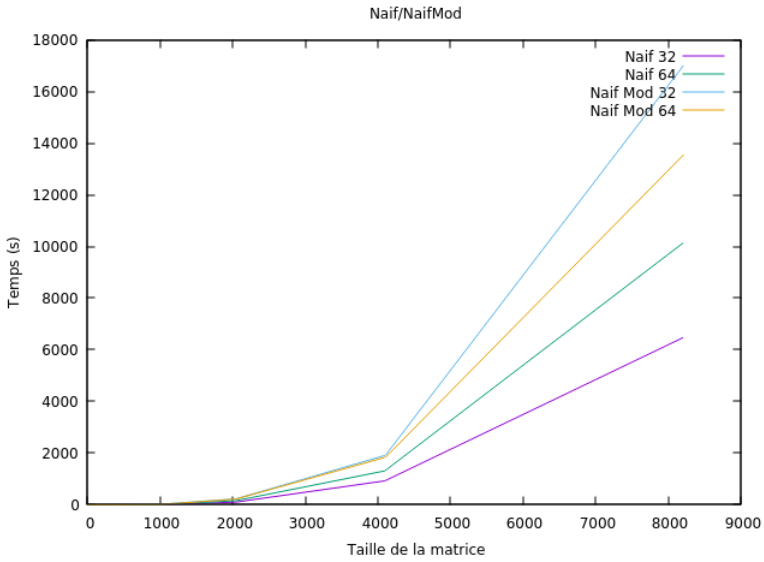
Naïf int_32 sans modulo : (C8)

```
u_int32_t** produit_matrice32(u_int32_t** mat1, u_int32_t** mat2, int taille){
    u_int32_t** mat = alloue_matrice32(taille);

    for (int i = 0; i < taille; i++){
        for (int j = 0; j < taille; j++){
            for (int k = 0; k < taille; k++){
                mat[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
    return mat;
}
```

Naïf int_32 sans modulo : (C9)

```
u_int64_t** produit_matrice64(u_int64_t** mat1, u_int64_t** mat2, int taille){
    u_int64_t** mat = alloue_matrice64(taille);

    for (int i = 0; i < taille; i++){
        for (int j = 0; j < taille; j++){
            for (int k = 0; k < taille; k++){
                mat[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
    return mat;
}
```

Voici le graphe comparant les quatre codes cités, à savoir (C3), (C6), (C8) et (C9) :



Maintenant que nous avons identifié ce qui est coûteux : les déplacements et les modulos, essayons de les optimiser le mieux possible.

4 Optimisations

Nous reprenons les notations de la partie précédente : on cherche à calculer $A \times B = C$ où les matrices sont de taille $n \in \mathbb{N}$ et les coefficients sont respectivement notés $(a_{i,j})_{1 \leq i,j \leq n}$, $(b_{i,j})_{1 \leq i,j \leq n}$ et $(c_{i,j})_{1 \leq i,j \leq n}$. Ces derniers sont tous dans $\frac{\mathbb{Z}}{p\mathbb{Z}}$ avec p = 1073741827. Nous n'optimiserons que la version naïve avec des coefficients int_64 puisque d'une part nous travaillons avec des machines 64-bits, et d'autre part pour des raisons de simplicités puisque le but ici est de présenter méthodes utilisées. Le lecteur pourra, s'il en a envie, transposer ces méthodes sur des matrices int_32.

4.1 Localité des données et Cache

Le principe de localité des données est la constatation du comportement d'un programme. On a d'une part, la localité temporelle qui est le fait qu'un programme qui demande à manipuler une information a de très grandes chances de la manipuler à nouveau peu de temps après. D'autre part, on a la localité spatiale qui est le fait qu'un programme qui demande à manipuler une information a de très grandes chances de manipuler les informations adjacentes. Ce qui introduit la notion de cache dans nos systèmes d'exploitations modernes. Celle-ci est caractérisée par une petite mémoire dans le CPU qui va stocker une copie des informations que le principe de localité aura jugé utile. Cela nous a conduit à effectuer deux optimisations : le produit matriciel par transposée et le produit matriciel par bloc.

4.1.1 Produit matriciel avec la transposée

La première optimisation est l'utilisation de la transposée. En effet, le programme effectuant le produit matriciel demande à manipuler toute une ligne et toute une colonne. Or, la localité des données favorise les appels mémoires sur les lignes. Donc, nous avons décidé d'implémenter un produit matriciel où l'on transpose la deuxième matrice puis l'on modifie le produit $a_{i,k} \times b_{k,j}$ en $a_{i,k} \times b_{j,k}$ pour $i, j, k \in [\![1, n]\!]$.

Transposée : (C10)

```c
u_int64_t** produit_matrice_transposee_mod64(u_int64_t** mat1, u_int64_t** mat2, int taille, long p){
    u_int64_t** mat = alloue_matrice64(taille);
    u_int64_t** mat_t = transposee_matrice64(mat2, taille); // Transpose la matrice mat2

    for (int i = 0; i < taille; i++){
        for (int j = 0; j < taille; j++){
            for (int k = 0; k < taille; k++){
                mat[i][j] += (mat1[i][k] * mat_t[j][k]) % p;
            }
            mat[i][j] = mat[i][j] % p;
        }
    }

    return mat;
}
```

Démonstration. Notons $(t_{i,j})_{1\leq i,j\leq n}$ les coefficients de la transposée de B.
En appliquant notre produit avec la transposée, on a :

$$\forall i,j \in [\![1,n]\!], c_{i,j} = (\sum_{k=0}^{n}(a_{i,k} \times t_{j,k} \mod p) \mod p).$$

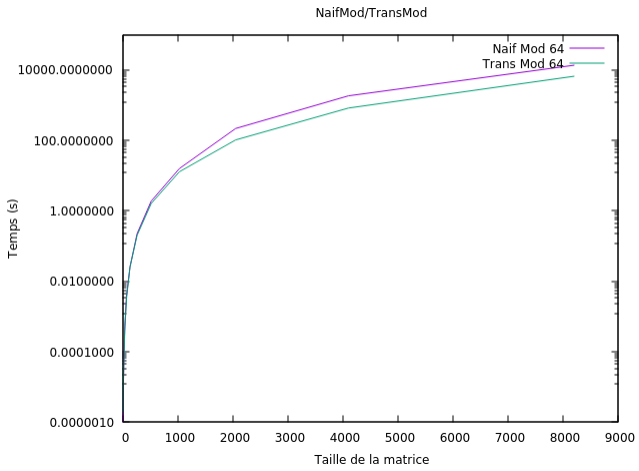
Or par définition de la transposée, on a pour tout $i,j \in [\![1,n]\!]$, l'égalité $b_{i,j} = t_{j,i}$.
Ainsi on a :

$$\forall i,j \in [\![1,n]\!], c_{i,j} = (\sum_{k=0}^{n}(a_{i,k} \times b_{k,j} \mod p) \mod p),$$

qui est le produit matriciel naïf.

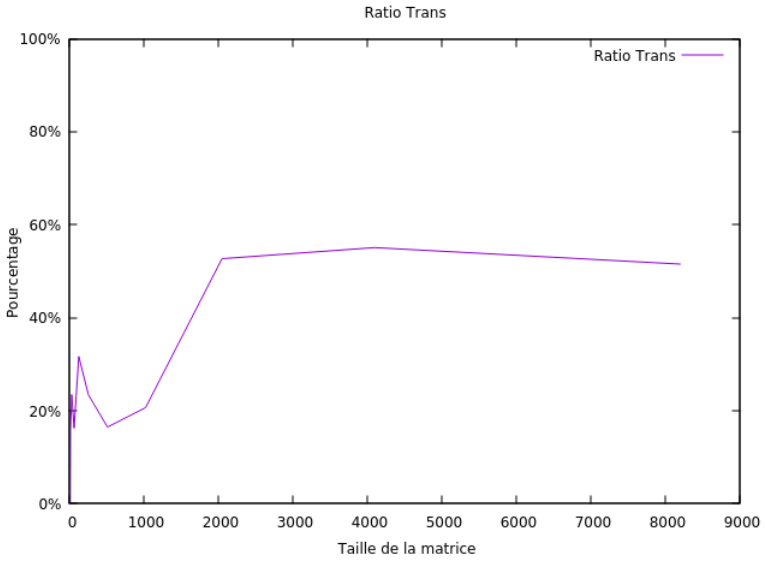
□

Voici le graphe de comparaison entre (C5) et (C10) :



Nous verrons mieux l'optimisation apportée par la transposée en traçant de nouveau un graphique, du

gain de temps en fonction de la taille des matrices.



4.1.2 Produit matriciel par bloc

La seconde optimisation est l'utilisation de blocs. En effet, lors d'une manipulation sur une zone mémoire, le cache aura seulement en copie les zones adjacentes les plus proches. Donc nous avons décidé d'implémenter un produit matriciel où l'on découpe les matrices de manière conceptuelle en plusieurs blocs afin d'exploiter le cache au maximum.
NB : La taille du bloc est déterminée par l'architecture du CPU, on prendra des blocs de taille 64.

Par blocs : (C11)

```
u_int64_t** blocking_algorithm64_mod(u_int64_t** mat1, u_int64_t** mat2, int taille, long p){
    u_int64_t** mat = alloue_matrice64(taille);
    int blocksize = 64;

    for (int ii = 0; ii < taille; ii += blocksize){
        for (int jj = 0; jj < taille; jj += blocksize){
            for (int kk = 0; kk < taille; kk += blocksize){

                for (int i = ii; i < ii + blocksize && i < taille; i++){
                    for (int j = jj; j < jj + blocksize && j < taille; j++){
                        for (int k = kk; k < kk + blocksize && k < taille; k++){
                            mat[i][j] += (mat1[i][k] * mat2[k][j]) % p;
                        }
                    }
                }
            }
        }
    }

    for (int i = 0; i < taille; i++){
        for(int j = 0; j < taille; j++){
            mat[i][j] = mat[i][j] % p;
        }
    }

    return mat;
}
```

Démonstration. Soit $b \in [\![1, n]\!]$ correspondant à la taille des blocs.
En décomposant par la division euclidienne l'entier n, on a : $n = b \times q + r$ où q est le quotient et r le reste.
On peut écrire A, B et C sous forme de matrice en blocs :

$$A = \begin{bmatrix} A_{1,1} & \cdots & A_{1,q} \\ \vdots & \ddots & \vdots \\ A_{q,1} & \cdots & A_{q,q} \end{bmatrix}, \ B = \begin{bmatrix} B_{1,1} & \cdots & B_{1,q} \\ \vdots & \ddots & \vdots \\ B_{q,1} & \cdots & B_{q,q} \end{bmatrix} \text{ et } C = \begin{bmatrix} C_{1,1} & \cdots & C_{1,q} \\ \vdots & \ddots & \vdots \\ C_{q,1} & \cdots & C_{q,q} \end{bmatrix}$$

où la taille des blocs est donnée par $X \in \{A, B, C\}$:
Si $r = 0$,

$$X_{i,j} \in \mathcal{M}_b(\mathbb{R}), \forall i, j \in [\![1, q]\!]$$

Si $r \neq 0$,

$$X_{i,j} \in \begin{cases} \mathcal{M}_b(\mathbb{R}) & \text{si } i \leq q-1 \text{ et } j \leq q-1 \\ \mathcal{M}_r(\mathbb{R}) & \text{si } i = q \text{ et } j = q \\ \mathcal{M}_{b,r}(\mathbb{R}) & \text{si } i \leq q-1 \text{ et } j = q \\ \mathcal{M}_{r,b}(\mathbb{R}) & \text{si } i = q \text{ et } j = q \end{cases}$$

On effectue le produit par blocs,

$$\forall i, j \in [\![1, n]\!], C_{i,j} = \sum_{k=0}^{q} A_{i,k} \times B_{k,j}$$

En effet, le produit ci-dessus est bien défini.
Et pour calculer un coefficient $c_{i,j}$ quelconque, il suffit de trouver à quel bloc il appartient.

□

Proposition Soient $b, n \in \mathbb{N}^*$ et q, r le quotient et le reste de $n \mod b$.
Pour tout $i, j \in [\![1, n]\!]$, il existe $ii, jj \in [\![1, q]\!]$ tel que

$$\begin{cases} ii \times b \leq i < (ii+1) \times b \\ jj \times b \leq j < (jj+1) \times b \end{cases}$$

Démonstration. Soit $i, j \in [\![1, n]\!]$
Prenons $ii = \lfloor \frac{i}{b} \rfloor$ et $jj = \lfloor \frac{j}{b} \rfloor$.
Par les propriétés simples de la partie entière, on a :

$$\begin{cases} \lfloor \frac{i}{b} \rfloor \leq \frac{i}{b} < \lfloor \frac{i}{b} \rfloor + 1 \\ \lfloor \frac{j}{b} \rfloor \leq \frac{j}{b} < \lfloor \frac{j}{b} \rfloor + 1 \end{cases}$$

C'est-à-dire,

$$\begin{cases} ii \leq \frac{i}{b} < ii + 1 \\ jj \leq \frac{j}{b} < jj + 1 \end{cases}$$

$$\begin{cases} ii \times b \leq i < (ii+1) \times b \\ jj \times b \leq j < (jj+1) \times b \end{cases}$$

□

Exemple Soient $n = 3$, $b = 2$ et $A, B \in \mathcal{M}_3(\mathbb{R})$ définie par

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} B = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

Alors $r = 1$ et on a

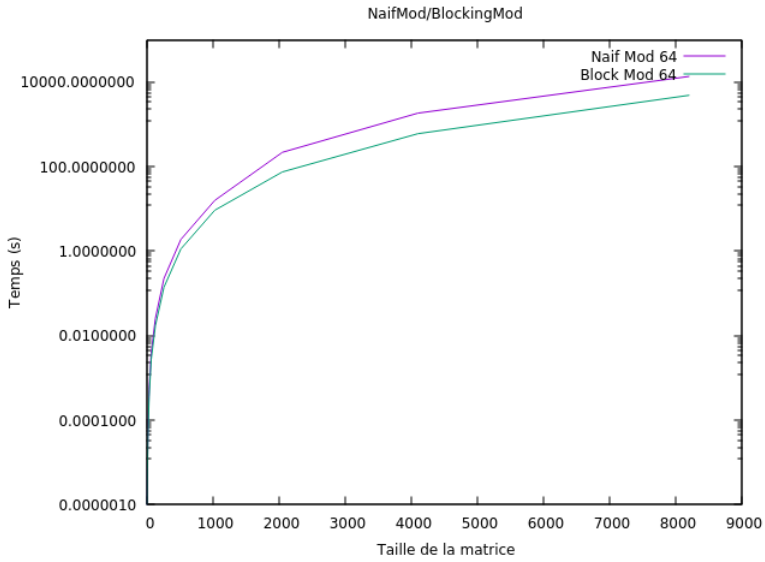
$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

Avec

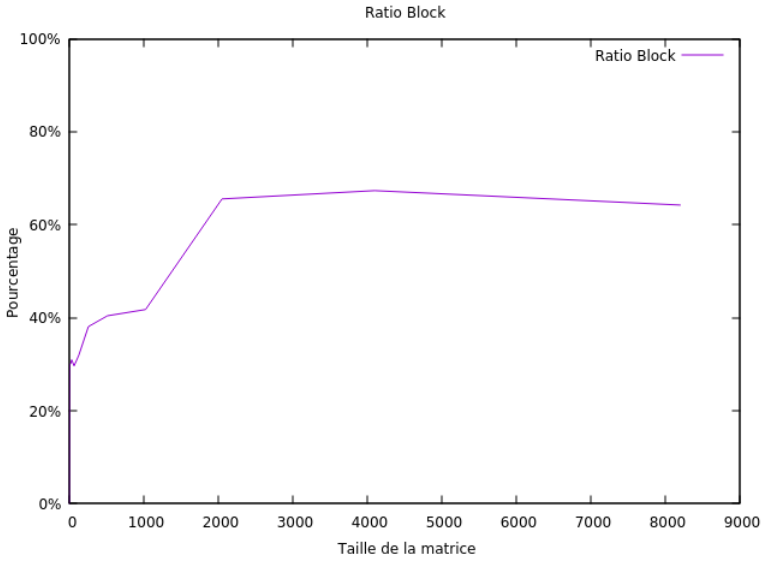
$$A_{1,1} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} A_{1,2} = \begin{bmatrix} a_{1,3} \\ a_{2,3} \end{bmatrix} A_{2,1} = \begin{bmatrix} a_{3,1} & a_{3,2} \end{bmatrix} A_{2,2} = \begin{bmatrix} a_{3,3} \end{bmatrix}$$

$$B_{1,1} = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} B_{1,2} = \begin{bmatrix} b_{1,3} \\ b_{2,3} \end{bmatrix} B_{2,1} = \begin{bmatrix} b_{3,1} & b_{3,2} \end{bmatrix} B_{2,2} = \begin{bmatrix} b_{3,3} \end{bmatrix}$$

Voici la comparaison entre (C5) et (C11) :



Et le graphique sur le gain de temps :



4.2 Modulo

Concernant le modulo, il y a deux optimisations à faire : d'une part la quantité, d'autre part la qualité.

4.2.1 Décomposition des termes

Concernant la quantité : pour deux matrices de tailles n, notre algorithme utilise $n^2(n+1)$ fois le modulo. Pour réduire ce nombre, nous avons pensé à décomposer chaque multiplication entre les coefficients des deux matrices.

Décomposition : (C12)

```
u_int64_t** produit_matrice_decomp64(u_int64_t** mat1, u_int64_t** mat2, int taille, long mod){
    u_int64_t** mat = alloue_matrice64(taille);

    for (int i = 0; i < taille; i++){
        for (int j = 0; j < taille; j++){
            u_int64_t h = 0;
            u_int64_t l = 0;
            for (int k = 0; k < taille; k++){
                u_int64_t chiffre = mat1[i][k] * mat2[k][j];
                h += chiffre >> 32;
                l += chiffre - ((chiffre >> 32) << 32) ;
            }
            u_int64_t n = 1 ;
            mat[i][j] = ((h % mod) * (n<<32) + (l % mod)) % mod;
        }
    }

    return mat;
}
```

Démonstration. On a : $\forall (i,j) \in [\![1,n]\!]^2, c_{i,j} = (\sum_{k=1}^{n}((a_{i,k} \times b_{k,j}) \mod p)) \mod p$.

Soient i, j, k $\in [\![1,n]\!]$, on peut écrire $a_{i,k} \times b_{k,j} = h_k \times 2^{32} + l_k$ où $0 \leq h_k < 2^{30}$ et $0 \leq l_k < 2^{32}$. En faisant de même pour chaque terme de la somme $c_{i,j}$, on construit les familles $(h_k)_{1\leq k\leq n}$ et $(l_k)_{1\leq k\leq n}$. On note maintenant $h = \sum_{k=1}^{n} h_k$ et $l = \sum_{k=1}^{n} l_k$.

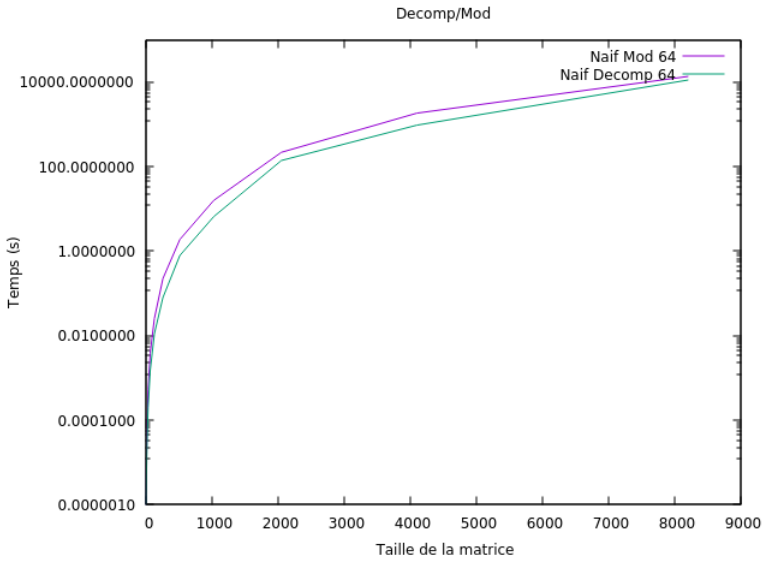
D'où le fait que :

$$\begin{aligned}
\forall (i,j) \in [\![1,n]\!]^2 : c_{i,j} &= (\sum_{k=1}^{n}((a_{i,k} \times b_{k,j}) \mod p)) \mod p \\
&= (\sum_{k=1}^{n}((h_k \times 2^{32} + l_k) \mod p)) \mod p \\
&= ((h \mod p) \times 2^{32} + (l \mod p)) \mod p.
\end{aligned}$$

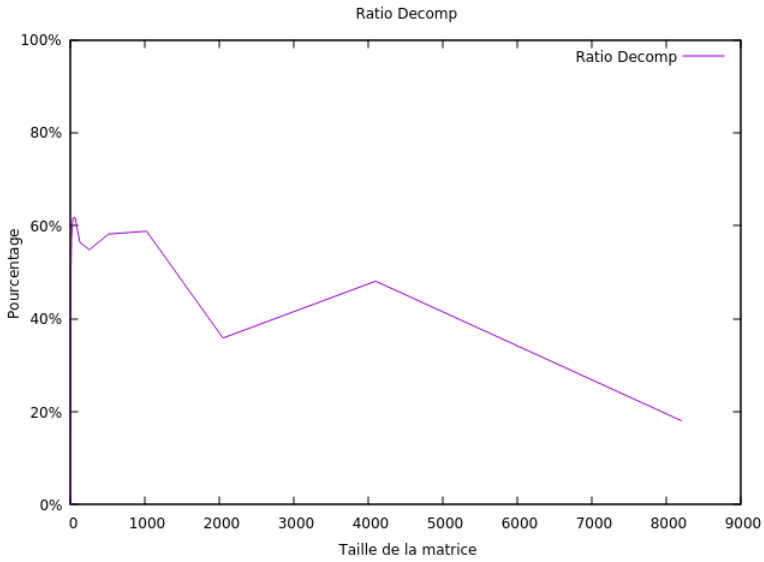
Ainsi, pour i, j $\in [\![1,n]\!]$, nous n'avons plus que trois modulos à faire pour obtenir $c_{i,j}$ alors qu'il nous en fallait $n+1$. Plus largement, pour multiplier deux matrices de taille n entre elles, nous sommes passés de $n^2(n+1)$ modulos à faire à $3n^2$.

Concernant la pratique et le fait d'implémenter cette méthode, supposons que ce soit sur une machine 64 bits. Alors pour $n < 2^{32}$, $0 \leq h < 2^{62}$ et $0 \leq l < 2^{64}$. L'implémentation ne pose donc pas de problème pour la multiplication de matrice de taille inférieure à 2^{32}. □

Voici le graphe comparant (C5) et (C12) :



Et le graphique sur le gain de temps :



4.2.2 Réduction de Barrett

Concernant la qualité : l'opération modulo fourni par le langage `C` est une opération lente et coûteuse. En effet, pour calculer a mod p, nous pouvons écrire $a = q \times p + r$ où $q \in \mathbb{Z}$ et $r \in [\![0, p-1]\!]$. Le modulo renvoie r et pour cela, il nous faut q $= \lfloor \frac{a}{p} \rfloor$. C'est justement cette division qui est coûteuse. Pour remédier à ce problème, nous avons eu recours à la réduction de Barrett, dont le principe est d'approximer $\frac{a}{p}$ avec des multiplications et des divisions de puissance de deux qui sont des opérations gratuites sur machine.

Réduction de Barrett : (C13)

```
u_int32_t reduire(const u_int64_t a, const u_int32_t p){
   const u_int32_t s = 30;
   const u_int32_t t = 32;

   const u_int64_t b = a >> s;
   const u_int64_t c = (b * 4294967284) >> t;
   int64_t d = a - c * p;
   while(d >= p){
     d -= p;
   }
   return d;
}
```

Propriété : La fonction est correcte et renvoie a mod p. De plus, le nombre d'itérations de la boucle est au plus de h, où $h = \lfloor \frac{2^s}{2^{r-1}} + \frac{\alpha 2^r}{2^{s+t}} \rfloor$.

Démonstration. Pour $i, j, k \in [\![1, n]\!]$, on pose $a = a_{i,k} \times b_{k,j}, n = t = 32, p = 1073741827$, $\alpha = 2^{n-1} = 2^{31}$, s = 30, h = 2 , r = m = 31 et $q = \lfloor \frac{2^{s+t}}{p} \rfloor = \lfloor \frac{2^{62}}{p} \rfloor = 4294967284$.
On a que $a < \alpha p$ par hypothèse, d'où :

$$\begin{aligned} bq = \lfloor \frac{a}{2^{30}} \rfloor \times \lfloor \frac{2^{62}}{p} \rfloor &\leq \lfloor \frac{a \times \lfloor \frac{2^{62}}{p} \rfloor}{2^{30}} \rfloor \\ &< \lfloor \frac{\alpha \times p \times \lfloor \frac{2^{62}}{p} \rfloor}{2^{30}} \rfloor \\ &\leq \lfloor \frac{\alpha \times p \times \frac{2^{62}}{p}}{2^{30}} \rfloor \\ &\leq \alpha \times 2^{32} = 2^{n-1} \times 2^n \\ &< 2^{2n}. \end{aligned}$$

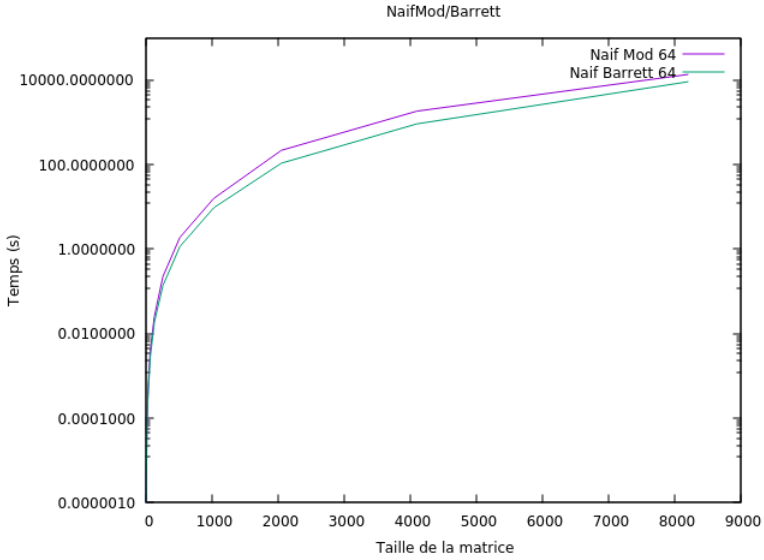
On a donc que bq tient sur $2n$ bits. De plus, $c = \lfloor \frac{bq}{2^t} \rfloor < \lfloor \frac{\alpha \times 2^t}{2^t} \rfloor = \alpha$.
D'où c $< \alpha$ et on a également $cp < \alpha p < 2^{62}$. Ainsi cp tient sur 2n bits également.

On a l'égalité suivante : $c - \lfloor \frac{a}{p} \rfloor = (\lfloor \frac{bq}{2^t} \rfloor - \frac{bq}{2^t}) + \frac{q}{2^t}(b - \frac{a}{2^s}) + \frac{a}{2^{s+t}}(q - \frac{2^{s+t}}{p}) + (\frac{a}{p} - \lfloor \frac{a}{p} \rfloor)$,
que le lecteur pourra vérifier par un simple développement.
Dans le deuxième et troisième terme de la somme, on remplace b et q par leurs expressions respectives, pour avoir :
$c - \lfloor \frac{a}{p} \rfloor = (\lfloor \frac{bq}{2^t} \rfloor - \frac{bq}{2^t}) + \frac{q}{2^t}(\lfloor \frac{a}{2^s} \rfloor - \frac{a}{2^s}) + \frac{a}{2^{s+t}}(\lfloor \frac{2^{s+t}}{p} \rfloor - \frac{2^{s+t}}{p}) + (\frac{a}{p} - \lfloor \frac{a}{p} \rfloor)$.
On obtient ainsi les deux inégalités suivantes en majorant par 0 ou 1 (resp. minorant par -1 ou 0) chaque termes : $-1 - \frac{q}{2^t} - \frac{a}{2^{s+t}} < c - \lfloor \frac{a}{p} \rfloor < 1$ $(*)$
De plus, on a $s \geq 1$ alors, on peut alors montrer que $\frac{q}{2^t} + \frac{a}{2^{s+t}} \leq h$.
En effet, on a :

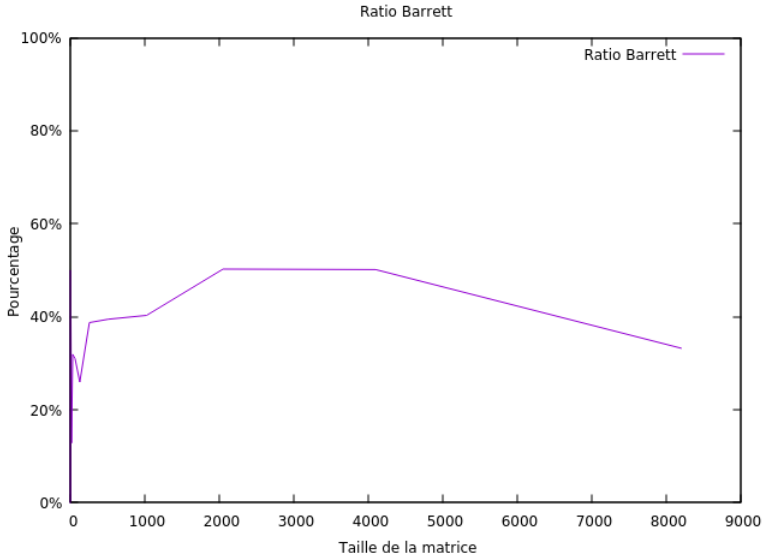
$$\begin{aligned} \frac{q}{2^t} + \frac{a}{2^{s+t}} &\leq \frac{\lfloor \frac{2^{s+t}}{p} \rfloor}{2^t} + \frac{\alpha 2^r}{2^{s+t}} \\ &\leq \frac{2^{s+t}}{2^t p} + \frac{\alpha 2^r}{2^{s+t}} \\ &\leq \frac{2^s}{p} + \frac{\alpha 2^r}{2^{s+t}} \\ &\leq \frac{2^s}{2^{r-1}} + \frac{\alpha 2^r}{2^{s+t}} \\ &\leq \lceil \frac{2^s}{2^{r-1}} + \frac{\alpha 2^r}{2^{s+t}} \rceil = h \end{aligned}$$

Où on a utilisé le fait que $\frac{1}{p} < \frac{1}{2^{r-1}}$ (c'est à dire $\frac{1}{2^{30}}$) dans la 4ème inégalité.
En utilisant l'inégalité que nous venons de montrer combiner à $(*)$ et le fait que $c - \lfloor \frac{a}{p} \rfloor$ est entier, on obtient : $-h \leq c - \lfloor \frac{a}{p} \rfloor \leq 0$. On en conclut que c est une approximation de $\lfloor \frac{a}{p} \rfloor$, que h est le nombre maximal de tour de la boucle, et que la fonction est valide. □

Voici le graphique pour (C5) et (C13) :



Et le graphique sur le gain de temps :



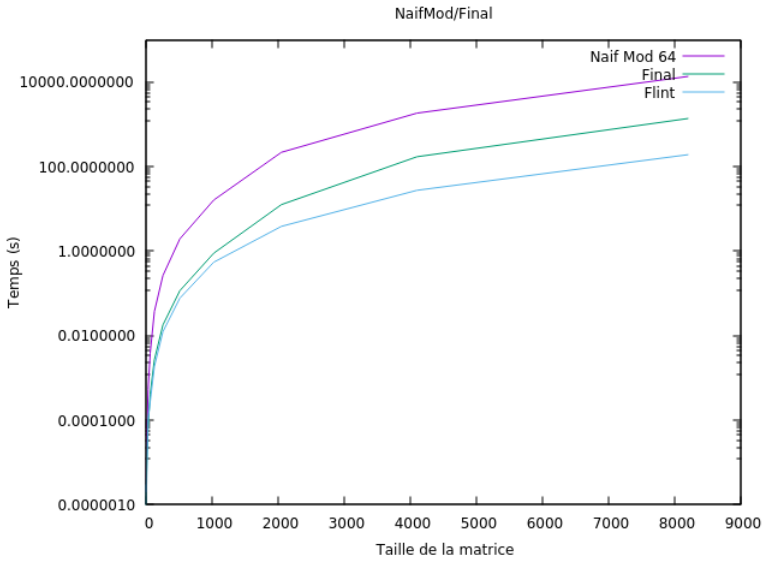
4.3 Résultat

Maintenant que nous avons vu ces méthodes d'optimisations qui, implémentées individuellement, apportent toutes une amélioration du temps d'exécution, nous les avons ensuite fusionnées dans un code unique.

Transposée - Bloc - Décomp - Barrett : (C14)

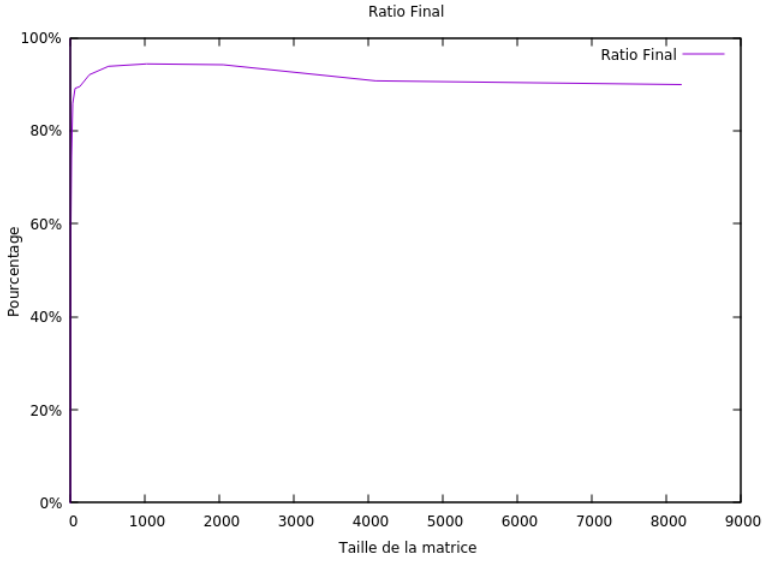
```
u_int64_t** produit_matrice_blocking_trans_barrett_decomp_64(u_int64_t** mat1, u_int64_t** mat2,
    int taille, long mod){
    u_int64_t** mat = alloue_matrice64(taille);
    u_int64_t** mat_t = transposee_matrice64(mat2, taille);
    int blocksize = 64;

    for (int ii = 0; ii < taille; ii += blocksize){
        for (int jj = 0; jj < taille; jj += blocksize){
            for (int kk = 0; kk < taille; kk += blocksize){

                for (int i = ii; i < ii + blocksize && i < taille; i++){
                    for (int j = jj; j < jj + blocksize && j < taille; j++){
                        u_int64_t h = 0;
                        u_int64_t l = 0;
                        for (int k = kk; k < kk + blocksize && k < taille; k++){
                            u_int64_t chiffre = mat1[i][k] * mat_t[j][k];
                            h += chiffre >> 32;
                            l += chiffre - ((chiffre >> 32) << 32) ;
                        }
                        u_int64_t n = 1;
                        h = reduire(h, mod);
                        l = reduire(l, mod);
                        mat[i][j] += reduire(h * (n<<32) + l, mod);
                    }
                }

            }
        }
    }

    for (int i = 0; i < taille; i++){
        for(int j = 0; j < taille; j++){
            mat[i][j] = reduire(mat[i][j], mod);
        }
    }

    return mat;
}
```

Finalement, comparons une nouvelle fois avec flint pour voir nos progrès. Voici le graphique avec (C2), (C5) et (C14) :



Nous voyons qu'en dessous de la taille 512, notre algorithme est très proche de flint. Mais au dessus de cette taille, la différence de temps s'accroît. Après avoir regardé le code de FLINT, nous avons remarqué que celui-ci inclut l'algorithme de Strassen, ce que nous avons pas.

Voici le graphique sur le gain de temps :



Conclusion

Grâce ce projet que nous avons eu la chance de réaliser, nous en tirons plusieurs bénéfices. Tout d'abord, étant un projet de groupe, cela nous a permis de nous familiariser avec des outils collaboratifs tel que *git*. Ensuite, nous avons tous eu notre premier contact avec LaTeX. et *Beamer* : objet que nous utiliserons souvent à l'avenir.
Concernant la méthode de travail, cela nous a appris à entreprendre une démarche scientifique. En effet, nous avons commencé à comparer théorie et pratique, et à partir de cela, nous avons entamé les recherches afin de trouver réponses à nos questions.
En plus de cela, ce projet nous a permis de nous initier au calcul haute performance, une composante extrêmement importante dans la science moderne, que se soit dans le stockage, le traitement ou l'analyse de donnée. Ce qui nous a notamment montré un aperçu du master HPC.

Références

[FLINT]

[1] https://www.flintlib.org/

[NTL]

[2] https://libntl.org/

[Localité des données]

[3] http://www-int.impmc.upmc.fr/impmc/Enseignement/ye/informatique/systemes/chap5/55.html

[4] https://fr.wikipedia.org/wiki/Principe_de_localit%C3%A9_(informatique)

[5] https://boowiki.info/art/les-theories-de-l-informatique/principe-de-localisation-informatique.html

[6] https://www.youtube.com/watch?v=bcBOyAlU2m4&t=403s

[7] https://www.udacity.com/course/high-performance-computer-architecture--ud007

[Blocking Algorithm]

[8] https://sites.cs.ucsb.edu/~gilbert/cs140resources/slides/cs140-06-matmul.pdf

[9] https://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf

[10] https://malithjayaweera.com/2020/07/blocked-matrix-multiplication/

[11] http://dc-vis.irb.hr/repository/2014/Optimal%20Block%20Size%20for%20Matrix%20Multiplication%20Using%20Blocking.pdf

[12] https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm#:~:text=Block%20matrix%20multiplication.,is%20assigned%20to%20one%20processor.

[Shift Operator]

[13] https://www.geeksforgeeks.org/left-shift-right-shift-operators-c-cpp/

[Réduction de Barrett]

[14] https://fr.wikipedia.org/wiki/R%C3%A9duction_de_Barrett

[15] https://arxiv.org/ftp/arxiv/papers/1407/1407.3383.pdf