Master Calcul Haute Performance et Simulation

Rapport

Multiplication matricielle et DC

Réalisé par :

Dekkal Dyhia

Moussi Nassima

Année universitaire : 2021/2022

0.1 Introduction

Le produit de matrices est une opération qui intervient souvent dans les calculs informatiques et numériques.C'est typiquement le genre d'opérations assez lourd à mener. Ce n'est pas sa complexité qui est embarrassante, mais son aspect répétitif dès que la taille de la matrice devient importante. Pour économiser un peu de temps de calcul **Volker Strassen** a imaginé l'algorithme appelé algorithme de strassen qui s'appuie sur le principe de divide and conquer

0.2 Approche divide and conquer :

La méthode diviser pour régner est une approche de résolution de problème qui consiste à décomposer un problème complexe en sous-problèmes plus facile à traiter afin de résoudre le problème initial.
Principe : Le paradigme diviser pour régner est souvent utilisé pour trouver une solution optimale à un problème donnée qui doit être décomposable, cette décomposition pouvant être récursive, il s'agit d'une approche de haut en bas également appelée approche descendante.
Paradigme "diviser pour régner" :
Il donne lieu à trois étapes à chaque niveau de récursivité :

1. **Diviser** le problème en un certain nombre de sous-problèmes.
2. **Régner** sur le sous-problème en les résolvant récursivement ou, si la taille d'un sous-problème est assez réduite, le résoudre directement.
3. **Combiner** les solutions des sous-problèmes en une solution complète du problème initial.

Exemple : Supposons dans ce qui suit que n est une puissance exacte de 2. Décomposons les matrices A,B et C en sous-matrices de taille$n/2 \times n/2$.L'équation $C = A \times B$ peut alors se réecrire : $\begin{bmatrix} r & s \\ t & u \end{bmatrix}$ $= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & g \\ f & h \end{bmatrix}$

En developpant cette équation ,nous obtenons : $\begin{cases} r = ae + bf \\ s = ag + bh \\ t = ce + df \\ u = cg + dh \end{cases}$

Chacune de ces quatre opérations correspond à deux multiplications de matrices carrées de taille n/2 et une addition de telles matrices. L'addition des matrices carrées de taille n/2 étant en$\theta(n$š) à partir de ses équations, on peut aisément dériver un algorithme diviser pour régner dont la complexité est donnée par la récurrence :

$T(n) = 8T(n/2) + \theta(n$š)

0.3 Multiplication de matrices :

L'aspect répétitif de l'approche classique à trois boucles imbriquées pose le problème de temps de calcul lorsque la taille de la matrice devient importante. **Exemple :** Soient = (a_{ij}) t = $(b_{ij}$ deux matrices carrées de taille *ntimesn* la multiplication de A et B est définie par $C = A \times B$ avec : $c_{ij} = \sum_{k=1}^{n} a_{ij} b_{kj}$

0.4 Multiplication naïve de matrice :

Algorithme :

Algorithm 1: Algorithme naive

Input: A et B deux matrices $M_{n,n}(k)$
Output: $R = A \times B$

```
1 for i =1 à n do do
2     for j= 1 à n do do
3         R[i,j] =0
4         for k=1 à n faire do
5             R[i,j]= R[i,j]+ A[i,k]×B[k, j]
```

0.5 algorithme de strassen

L'algorithme de Strassen est un algorithme diviser pour régner qui n'effectue que 7 multiplications de matrices (contrairement à 8 dans l'algorithme précédent)mais qui effectue plus d'additions et de soustraction de matrices : ceci est sans conséquence, une addition de matrices étant gratuite par rapport au cout d'une multiplication. ci-dessous l'algorithme de strassen :

Strassen :

Entrée : Deux matrices A,X $\in M_n(k)$, avec n = 2^k.

Sortie : Le produit AX.

— Si $n = 1$, renvoyer AX.

Décomposer $A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$ *et* X= $\begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$

avec a, b, c, d et x, y, z, t dans $M_{n/2}(k)$

— Calculer récurssivement les produits :

$M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$

$M_2 = (A_{21} + A_{22}) \times B_{11}$

$M_3 = A_{11} \times (B_{12} - B_{22})$

$M_4 = A_{22} \times (B_{21} - B_{11})$

$M_5 = (A_{11} + A_{12}) \times B_{22}$

$M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$

$M_7 = (A_{12} - A_{22}) \times (B_{21} + B22)$

— Calculer les sommes :

$C_{11} = M_1 + M_4 - M_5 + M_7$

$C_{12} = M_3 + M_5$

$C_{21} = M_2 + M_4$

$C_{22} = M_1 - M_2 + M_3 + M_6$

— Renvoyer

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

Exemple :

Soient les deux matrices A et B suivantes et C matrice résultante :

$A = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix} \times$ B$= \begin{bmatrix} 6 & 5 \\ 1 & 2 \end{bmatrix} = C = \begin{bmatrix} 26 & 24 \\ 19 & 17 \end{bmatrix}$

$M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) = 40$

$M_2 = (A_{21} + A_{22}) \times B_{11} = 24$

$M_3 = A_{11} \times (B_{12} - B_{22}) = 12$

$M_4 = A_{22} \times (B_{21} - B_{11}) = -5$

$M_5 = (A_{11} + A_{12}) \times B_{22} = 12$

$M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12}) = -11$

$M_7 = (A_{12} - A_{22}) \times (B_{21} + B22) = 3$

$C_{11} = M_1 + M_4 - M_5 + M_7 = 26$

$C_{12} = M_3 + M_5 = 24$

$C_{21} = M_2 + M_4 = 19$

$C_{22} = M_1 - M_2 + M_3 + M_6 = 17$

0.6 Complexité algorithmique

1. Complexité algorithmique dans le cas classique Multiplication Naive

$$T_n = O(n^3) \tag{1}$$

2. Complexité algorithmique dans le cas divide and conquer :

$$T(n) = 8 \times T(n/2) + \theta(n^2)$$

(2)

Amélioration de l'efficacité de la multiplication(Strassen) :

$$T(n) = 7 \times T(n/2) + \theta(n^2)$$

(3)

0.7 travail demandé :

Le travail demandé était d'implémenter en python l'algorithme de multiplication de matrices naïf ainsi que l'algorithme de Strassen. Le but est de comparer les performances des deux algorithmes. Obtenir un graphique du temps d'exécution en fonction de la taille des matrices permettra de comparer facilement les deux algorithmes.

0.8 Développement du projet :

Le projet s'est découpé en plusieurs modules :

1. naïf
2. strassen
3. test

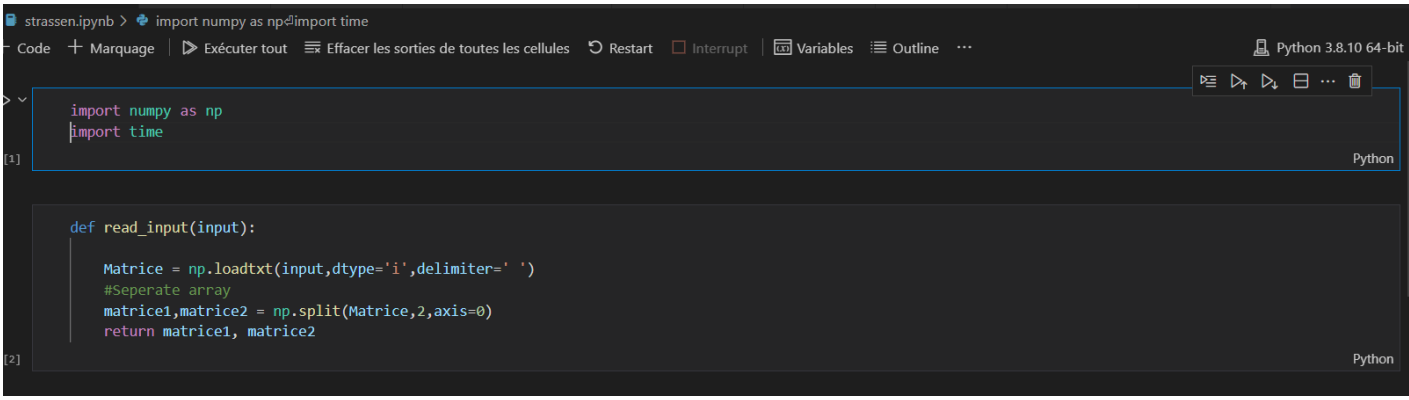
Chronologiquement, voici les différentes étapes de développement :

1. Implémentation d'une fonction de création de grandes matrices de taille n avec des coefficients aléatoires qui seront sauvegardés dans un fichier
2. Implémentation de l'algorithme naïf sur le fichier input.txt qui contient les coefficients des deux matrices créés, en sortie on aura le fichier output.txt qui contiendra la matrice résultante
   Dans cette fonction on a mesuré le temps de calcul
3. Implémentation de l'algorithme strassen sur le fichier input.txt qui contient les coefficients des deux matrices créés, en sortie on aura le fichier output.txt qui contiendra la matrice résultante
   Dans cette fonction, on a mesuré le temps de calcul

Voici en images les étapes :

1. Lecture dU fichier crée de la matrice

2. Strassen function :




```python
def strassen(matrice1,matrice2):
    n = len(matrice1)
    if n == 1:
        return matrice1 * matrice2
    else:
        a11 = matrice1[:int(len(matrice1)/2),:int(len(matrice1)/2)]
        a12 = matrice1[:int(len(matrice1)/2),int(len(matrice1)/2):]
        a21 = matrice1[int(len(matrice1)/2):,:int(len(matrice1)/2)]
        a22 = matrice1[int(len(matrice1)/2):,int(len(matrice1)/2):]

        b11 = matrice2[:int(len(matrice2)/2),:int(len(matrice2)/2)]
        b12 = matrice2[:int(len(matrice2)/2),int(len(matrice2)/2):]
        b21 = matrice2[int(len(matrice2)/2):,:int(len(matrice2)/2)]
        b22 = matrice2[int(len(matrice2)/2):,int(len(matrice2)/2):]

        S1 = b12 - b22
        S2 = a11 + a12
        S3 = a21 + a22
        S4 = b21 - b11
        S5 = a11 + a22
        S6 = b11 + b22
        S7 = a12 - a22
        S8 = b21 + b22
        S9 = a11 - a21
        S10 = b11 + b12

        P1 = strassen(a11,S1)
        P2 = strassen(S2,b22)
        P3 = strassen(S3,b11)
        P4 = strassen(a22,S4)
        P5 = strassen(S5,S6)
        P6 = strassen(S7,S8)
        P7 = strassen(S9,S10)

        c11 = P5 +P4 -P2 +P6
```

3. Temps de calcul Strassen

```python
if __name__ == "__main__":
    matrice1,matrice2 = read_input('input.txt')
    start_time = time.time()
    output = strassen(matrice1,matrice2)
    print("TIME TAKEN: ",time.time()-start_time)
    save_ouput(output)
```

Python

```
TIME TAKEN:  24816.340373516083
```

4. Matrice Résultante :

output.txt

```
2.686833000000000000e+06 2.671287000000000000e+06 2.680486000000000000e+06 2.599706000000000000e+06 2.616512000000000000e+06 2.630170000000000000e
2.668444000000000000e+06 2.616899000000000000e+06 2.566379000000000000e+06 2.490375000000000000e+06 2.468973000000000000e+06 2.510712000000000000e
2.704795000000000000e+06 2.632809000000000000e+06 2.644013000000000000e+06 2.594870000000000000e+06 2.584997000000000000e+06 2.632909000000000000e
2.671034000000000000e+06 2.642502000000000000e+06 2.538185000000000000e+06 2.556948000000000000e+06 2.513212000000000000e+06 2.520702000000000000e
2.597999000000000000e+06 2.634900000000000000e+06 2.582964000000000000e+06 2.503289000000000000e+06 2.458016000000000000e+06 2.502932000000000000e
2.668232000000000000e+06 2.621567000000000000e+06 2.598643000000000000e+06 2.600388000000000000e+06 2.586260000000000000e+06 2.530184000000000000e
2.571645000000000000e+06 2.517067000000000000e+06 2.528494000000000000e+06 2.466401000000000000e+06 2.465698000000000000e+06 2.497145000000000000e
2.676707000000000000e+06 2.662470000000000000e+06 2.588016000000000000e+06 2.610868000000000000e+06 2.555450000000000000e+06 2.639889000000000000e
2.739478000000000000e+06 2.644920000000000000e+06 2.634125000000000000e+06 2.562085000000000000e+06 2.547793000000000000e+06 2.593252000000000000e
2.713796000000000000e+06 2.649659000000000000e+06 2.661148000000000000e+06 2.619242000000000000e+06 2.589728000000000000e+06 2.618123000000000000e
2.711359000000000000e+06 2.735987000000000000e+06 2.717453000000000000e+06 2.706881000000000000e+06 2.651428000000000000e+06 2.654422000000000000e
2.638121000000000000e+06 2.651120000000000000e+06 2.619136000000000000e+06 2.593576000000000000e+06 2.548065000000000000e+06 2.571599000000000000e
2.652574000000000000e+06 2.577590000000000000e+06 2.619378000000000000e+06 2.506983000000000000e+06 2.544300000000000000e+06 2.589997000000000000e
2.611393000000000000e+06 2.532846000000000000e+06 2.548765000000000000e+06 2.458897000000000000e+06 2.516280000000000000e+06 2.496713000000000000e
2.601033000000000000e+06 2.639239000000000000e+06 2.553897000000000000e+06 2.557392000000000000e+06 2.510051000000000000e+06 2.496126000000000000e
2.655832000000000000e+06 2.621122000000000000e+06 2.572596000000000000e+06 2.492883000000000000e+06 2.527783000000000000e+06 2.536413000000000000e
2.637994000000000000e+06 2.621135000000000000e+06 2.582350000000000000e+06 2.570747000000000000e+06 2.547514000000000000e+06 2.562082000000000000e
2.629854000000000000e+06 2.548269000000000000e+06 2.590318000000000000e+06 2.611616000000000000e+06 2.390985000000000000e+06 2.562473000000000000e
2.606289000000000000e+06 2.626821000000000000e+06 2.643237000000000000e+06 2.521633000000000000e+06 2.523875000000000000e+06 2.552909000000000000e
2.676684000000000000e+06 2.590433000000000000e+06 2.646993000000000000e+06 2.515584000000000000e+06 2.522077000000000000e+06 2.586947000000000000e
2.616070000000000000e+06 2.610663000000000000e+06 2.565799000000000000e+06 2.560909000000000000e+06 2.509431000000000000e+06 2.523739000000000000e
2.684744000000000000e+06 2.642814000000000000e+06 2.611258000000000000e+06 2.549251000000000000e+06 2.567772000000000000e+06 2.572863000000000000e
2.664637000000000000e+06 2.637646000000000000e+06 2.627287000000000000e+06 2.565976000000000000e+06 2.571283000000000000e+06 2.608774000000000000e
2.662516000000000000e+06 2.625467000000000000e+06 2.668229000000000000e+06 2.570799000000000000e+06 2.577899000000000000e+06 2.617159000000000000e
2.595943000000000000e+06 2.558518000000000000e+06 2.537884000000000000e+06 2.472045000000000000e+06 2.526802000000000000e+06 2.445092000000000000e
2.635726000000000000e+06 2.610227000000000000e+06 2.564968000000000000e+06 2.478606000000000000e+06 2.511487000000000000e+06 2.521819000000000000e
2.587386000000000000e+06 2.596682000000000000e+06 2.535651000000000000e+06 2.461492000000000000e+06 2.473915000000000000e+06 2.546935000000000000e
2.578570000000000000e+06 2.566703000000000000e+06 2.577037000000000000e+06 2.473965000000000000e+06 2.519894000000000000e+06 2.528511000000000000e
2.701779000000000000e+06 2.671536000000000000e+06 2.647217000000000000e+06 2.570421000000000000e+06 2.560166000000000000e+06 2.588043000000000000e
2.661188000000000000e+06 2.662598000000000000e+06 2.625719000000000000e+06 2.586728000000000000e+06 2.568057000000000000e+06 2.564495000000000000e
2.739521000000000000e+06 2.712727000000000000e+06 2.687100000000000000e+06 2.601834000000000000e+06 2.657738000000000000e+06 2.653326000000000000e
2.676178000000000000e+06 2.598438000000000000e+06 2.562125000000000000e+06 2.529074000000000000e+06 2.500278000000000000e+06 2.554157000000000000e
2.631651000000000000e+06 2.574685000000000000e+06 2.547505000000000000e+06 2.516568000000000000e+06 2.488680000000000000e+06 2.542703000000000000e
2.613273000000000000e+06 2.621085000000000000e+06 2.559364000000000000e+06 2.484196000000000000e+06 2.505090000000000000e+06 2.539407000000000000e
2.639748000000000000e+06 2.530844000000000000e+06 2.566888000000000000e+06 2.547862000000000000e+06 2.486785000000000000e+06 2.503151000000000000e
2.578794000000000000e+06 2.524625000000000000e+06 2.529560000000000000e+06 2.436640000000000000e+06 2.468532000000000000e+06 2.508790000000000000e
2.629279000000000000e+06 2.555342000000000000e+06 2.536646000000000000e+06 2.470162000000000000e+06 2.490782000000000000e+06 2.483915000000000000e
2.579542000000000000e+06 2.564919000000000000e+06 2.470481000000000000e+06 2.519784000000000000e+06 2.474479000000000000e+06 2.499202000000000000e
```

De Même Pour naïf ce que vous allez voir sur les fichiers python joignés

Nous avons fait des tests sur les deux fonctions naïf et strassen, de même en générant des matrices aléatoires qui sont peu grandes que les deux précédentes

comme nous avons testé avec des matrices plus petites voir inférieur à 200

et en faisant la comparaison, on constate que strassen est bien meilleurs pour les matrices qui sont de taille supérieure à 200 tandis que naïf est bien meilleur que strassen avec des matrices de taille inférieure à 200

pour nos futures perspectives en vue de développer notre algorithme, nous envisageons de faire une hybridation des deux fonctions question de tirer profit des avantages de chacunes .

et comme on a utilisé un fichier de matrices de grandes tailles, nous avons exécuté notre programme sur le centre de calcul muso@LR ce que nous constatons qu'il y avait une certaine rapidité au niveau des calculs qui implique un temps réduit par rapport aux performances de nos machines .