OS202

TD Numéro 1

Valentin Quoniam-Barre - Eric Baudet

Janvier 2022

EXERCICE 1 : Produit Matrice-Matrice

Q1.

Nous avons modifié le main de *TestProductMatrix.cpp* pour qu'il renvoie tous les temps de calcul de la dimension 0 à 1030, avec une simple boucle for (Pour des soucis de durée, on a calculé uniquement pour les dimensions paires). Nous avons ensuite fait un graphique sur Python pour observer les résultats.

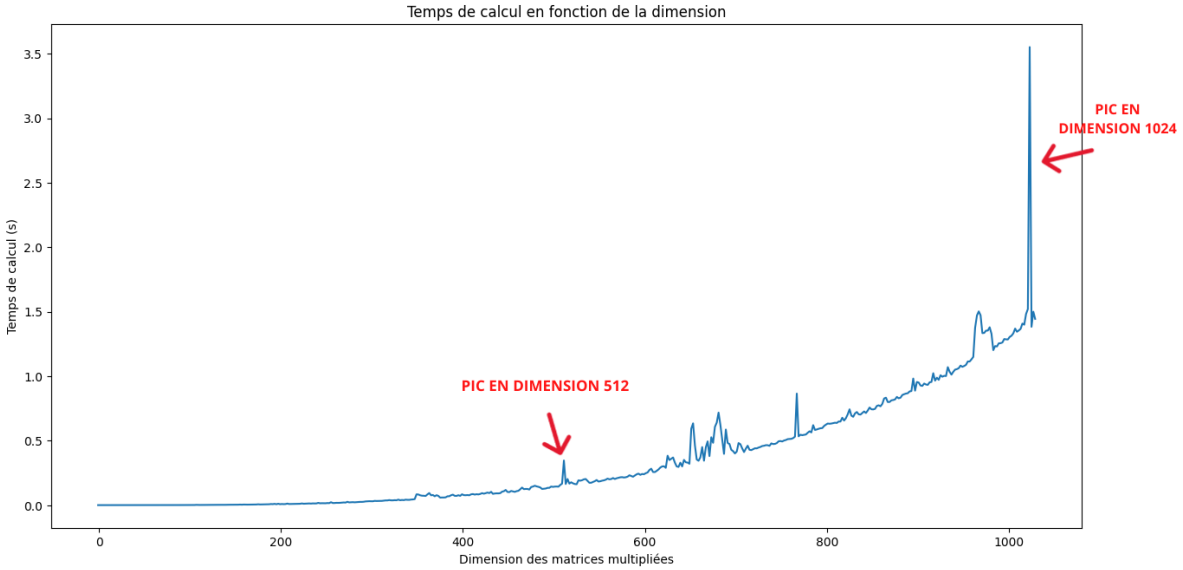

FIGURE 1 – Temps de calcul de produit de matrices

On observe un grand pic anormal en dimension 1024, mais aussi des pics en dimension 512 et 768. On n'analysera pas ici les anomalies autour de la dimension 700 et de 970.

Pour comprendre l'origine de ces pics, il faut s'intéresser au fonctionnement de la **mémoire cache**. Cette mémoire d'accès rapide est plus petite que la RAM et est découpée en lignes de 64 octets. Sur l'ordinateur de Valentin, voici la taille du cache :

```
Caches (sum of all):
  L1d:                    192 KiB (4 instances)
  L1i:                    128 KiB (4 instances)
  L2:                     5 MiB (4 instances)
  L3:                     8 MiB (1 instance)
```

FIGURE 2 – Taille des différents caches via lscpu

Ici, on utilise uniquement le cache L1i pour traiter les instructions, et L1d pour traiter les données. On a 4 instances de chaque cache, donc il faut prendre le quart des tailles données. Pour chaque CPU, L1d a alors une taille de 48KiB, et L1i a une taille de 32KiB (Avec $1KiB = 2^{10}$ octets) Tout calcul fait, L1d contient **768 lignes** et L1i contient **512 lignes**.

Ensuite, l'adressage des adresses mémoire depuis la RAM sur le cache se fait de manière *associative*. Chaque adresse RAM ne peut aller que sur une seule ligne de cache. Comme la RAM est plus grande que le cache, quand toutes les adresses ont été attribuées à leur ligne de cache, on repart de 0 pour les adresses suivantes. La répartition dans le cache se fait donc via un principe de modulo.

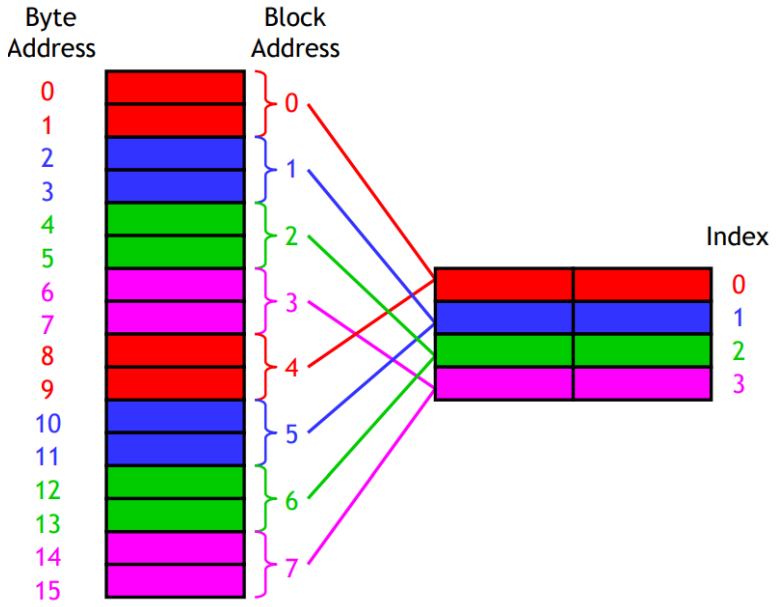

Figure 3 – Exemple d'adressage pour un cache de 4 lignes de 2 adresses

On voit donc que si on utilise seulement les adresses 2 et 3, 10 et 11, 18 et 19, on n'utilisera une seule ligne de cache (Et les autres ne serviront à rien !). **C'est le grand défaut de l'associativité du cache** et c'est exactement ce qu'il se passe dans notre cas.

En effet, pour calculer chaque coefficient de la matrice résultat, notre programme de calcul récupère une ligne de la matrice A et une colonne de la matrice B. Mais les adresses mémoires des coefficients sont stockées dans l'ordre des lignes des matrices. Par exemple, pour une matrice de taille 512, 2 coefficients d'une même colonne ont donc des adresses séparées d'un multiple de 512.

On a donc pas de problème pour récupérer les coefficients des lignes. Par contre si notre matrice est de taille 512 (ce qui est autant que le nombre de lignes de cache de L1i), **tous les coefficients d'une même colonne seront situés sur la même ligne de cache !** La mémoire n'est pas utilisée intelligemment et on perd alors beaucoup de temps.

Comme 1024 est multiple de 512, on a le même problème en dimension 1024. Pour ce qui est de la dimension 768, c'est au niveau du traitement des données (et non des instructions,comme pour 512 et 1024) que l'on aura un problème de cache. En effet, 768 est le nombre de lignes de L1d.

Q2.

On a vu qu'il faut éviter le plus possible de parcourir les colonnes. Or, en regardant ProdMat-Mat.cpp, on voit que k et j sont trouvés en parcourant les colonnes des matrices. De plus, on lit j mais on écrit k, ce qui est plus coûteux en temps. On va donc mettre **la boucle k en 1er, puis la boucle j et enfin la boucle i.**

```
void prodSubBlocks(int iRowBlkA, int iColBlkB, int iColBlkA, int szBlock,
                   const Matrix& A, const Matrix& B, Matrix& C) {
  for (int k = iColBlkA; k < std::min(A.nbCols, iColBlkA + szBlock); k++)
    for (int j = iColBlkB; j < std::min(B.nbCols, iColBlkB + szBlock); j++)
      for (int i = iRowBlkA; i < std::min(A.nbRows, iRowBlkA + szBlock); ++i)

        C(i, j) += A(i, k) * B(k, j);
}
```

FIGURE 4 – Boucles de calcul modifiées

En dimension 1024, on passe de 8 secondes à 1 seconde de temps de calcul. C'est une amélioration énorme pour une modification qui paraît mineure.

Q3.

Nous avons observé les temps de calcul en utilisant de 1 à 11 threads. Nous nous sommes remis en boucles "basiques" : i -j -k. Voici les résultats ci-dessous.

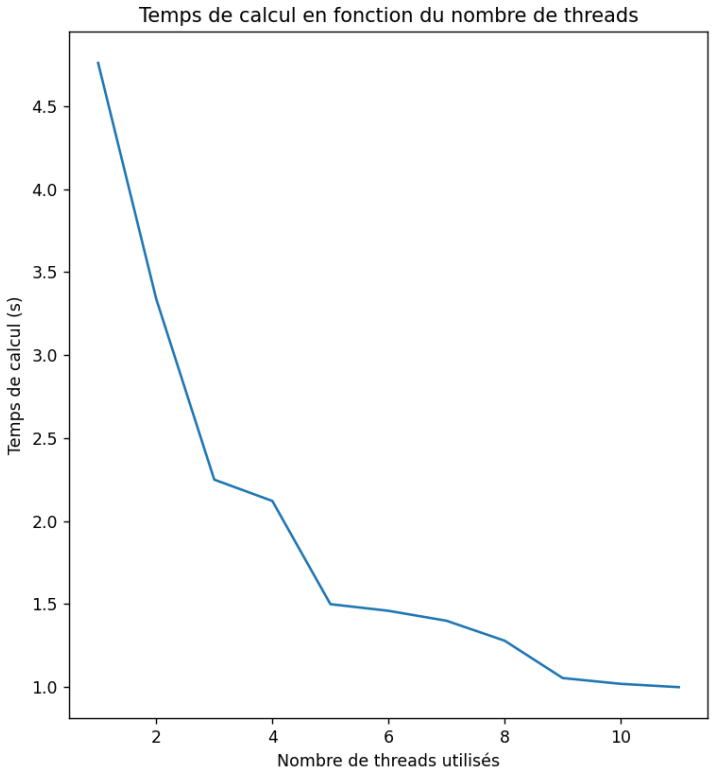

FIGURE 5 – Temps de calcul pour différents nombres de threads

On voit bien que le temps de calcul diminue quand on augmente la parallélisation. Si on parallélise trop néanmoins, le temps de calcul va augmenter.

Q4.

On ne sait pas quel nombre de thread à utiliser est optimal. Par ailleurs, calculer les coefficients d'une matrice un par un est long. On devrait plutôt calculer par bloc, pour avoir un meilleur accès au cache.

Q5.

Comme indiqué dans l'énoncé, on se remet en configuration k-j-i. Nous avons modifié le main de TestProductMatrix.cpp pour qu'il calcule tous les temps de calcul selon des tailles de sous-blocs de 1 à 128 (On a aussi du modifier ProdMatMat.cpp car l'opérateur * ne prend que 2 arguments, et on en voulait 3 - le 3ème étant la taille des blocs, qui devient une variable). Voici le résultat.

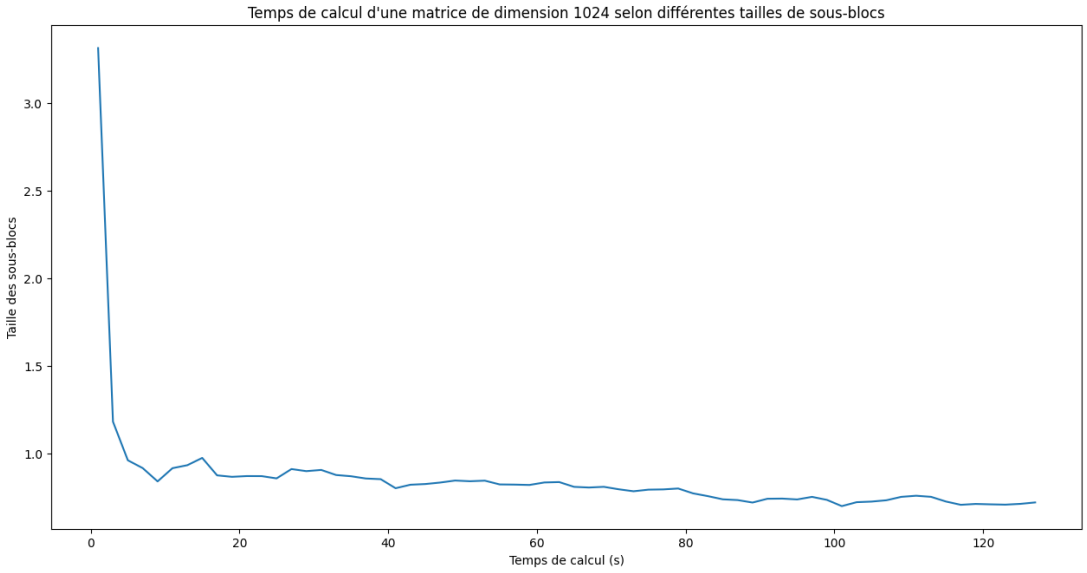

FIGURE 6 – Temps de calcul pour différentes tailles de sous-blocs

On observe une nette décroissance rapide du temps de calcul, puis une stabilisation. Pour un compromis puissance - temps de calcul, prendre des sous-blocs de dimension 32 semble assez pertinent. On pourrait prendre nimporte quelle autre taille plus grand évidemment, mais les modifications seront mineures.

Q6.

Nous avons ici un temps minimal de l'ordre de 500ms. Pour un calcul "scalaire", on était autour de la seconde. On a donc un temps de calcul divisé par 2 ! Cette différence est dûe au temps d'accès en mémoire aux coefficients k et j. En effet, Pour chaque k, on ne va chercher que 32 coefficients j et non 1024.

Q7.

On a parallélisé le calcul **par bloc** et non pas par coefficient, car cela prendrait un temps démentiel. On obtient un temps encore plus faible : 0.2 secondes.

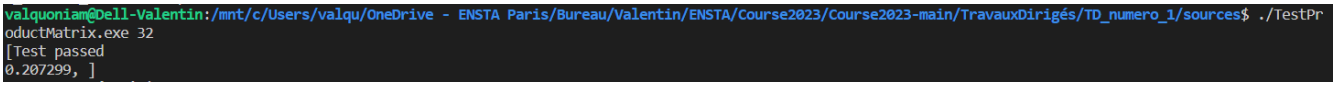

FIGURE 7 – Résultat de la parallélisation par blocs de taille 32

C'est un résultat cohérent : On allie 2 méthodes qui accélèrent les calculs. Notre calcul bénéficie à la fois de la parallélisation et des avantages des blocs. On a donc un code "super" rapide !

Q8.

Avec Blas, on obtient un temps de calcul de 0.15 secondes. C'est encore plus faible que le résultat de la Q7. Ce programme optimisé est clairement la meilleure version de toutes. On est passé d'un calcul de 5 secondes au début du TP à un calcul de moins de 200ms.

EXERCICE 2.1 : Circulation d'un jeton dans un anneau

Voici le programme :

```
nbp = 10  #valeur arbitraire à fixer

def circulation_jeton(jeton,n) :
    if n == 0 :
        if jeton == 0 :
            jeton = 1
            circulation_jeton(jeton,n+1)
        else :
            print(jeton)
            return 0
    elif n == nbp :
        jeton+=1
        circulation_jeton(jeton,0)
    else :
        jeton+=1
        circulation_jeton(jeton,n+1)
    return 1
```

Ce programme fait circuler un jeton dans un anneau de taille npb à partir du rang n.

EXERCICE 2.2 : Calcul très approché de pi

Voici le programme :

```
    import numpy as np
import numpy.random as rd

def calc_pi_s() :  #version séquentielle
    n = 10000 #valeur arbitraire du nombre de points générés dans le carré
    L = []  #liste qui va contenir nos n points
    q = 0   #nombre de points dans le carré qui sont aussi dans le cercle unité

    for i in range(n) :
        x = (-1)**rd.randint(1,2) * rd.rand()
        y = (-1)**rd.randint(1,2) * rd.rand()
        L.append([x,y])

        if np.sqrt(x**2 + y**2) <= 1 :
            q+=1
    r = q/n
    #print(4*r)
    return 4*r


#### version avec parallélisation MPI
import time

def random_coordonnee() :
    return (-1)**rd.randint(1,2) * rd.rand()
def ajoute_liste(elem,L) :
    L.append(elem)
    return L
def est_dans_cercle(X,q):
    if np.sqrt(X[0]**2 + X[1]**2) <= 1 :
        q+=1
    return q

def calc_pi_p() :  #version parallèle
    n = 10000 #valeur arbitraire du nombre de points générés dans le carré
    L = []  #liste qui va contenir nos n points
    q = 0   #nombre de points dans le carré qui sont aussi dans le cercle unité

    for i in range(n) :
        x = random_coordonnee()
        y = random_coordonnee()
        ajoute_liste([x,y],L)
        q = est_dans_cercle([x,y],q)

    r = q/n
    #print(4*r)
    return 4*r

##Mesure du temps :
S = 0
P = 0
for i in range(1000):
    deb_s = time.perf_counter()
    calc_pi_s()
    fin_s = time.perf_counter()
    S+=(fin_s-deb_s)/1000

    deb_p=time.perf_counter()
    calc_pi_p()
    fin_p = time.perf_counter()
    P+=(fin_p - deb_p)/100

print(S,P)
```