TD3 : Optimisation des Boucles dans un Algorithme de Multiplication de Matrices

NOM de l'étudiant ➔ Adel Bouzidi

Introduction

Dans ce travail pratique (TP), nous avons travaillé sur l'optimisation des boucles dans un algorithme de multiplication de matrices. L'objectif principal est d'analyser l'impact des permutations des boucles sur la performance d'un algorithme et de vérifier si ces changements conduisent à une amélioration des performances en termes de cycles d'horloge nécessaires pour l'exécution.

EXO 1 :

On a :

```
adel@adel:~/Documents/TD3_DB/libmcblas002$ ll
total 48
drwxr-xr-x 10 adel adel 4096 mars  22 17:13 ./
drwxr-xr-x  3 adel adel 4096 mars  21 14:43 ../
-rwxr-xr-x  1 adel adel 1828 mars  10 16:00 CMakeLists.txt*
drwxr-xr-x  2 adel adel 4096 mars  21 14:43 mcblas001/
drwxr-xr-x  2 adel adel 4096 mars  21 14:43 mcblas002/
drwxr-xr-x  2 adel adel 4096 mars  21 14:43 mcblas003/
drwxr-xr-x  2 adel adel 4096 mars  21 14:43 mcblas004/
drwxr-xr-x  2 adel adel 4096 mars  21 14:43 mcblas005/
drwxr-xr-x  2 adel adel 4096 mars  21 14:43 mcblas006/
drwxr-xr-x  2 adel adel 4096 mars  21 14:43 mcblas007/
-rwxr-xr-x  1 adel adel  532 mars  22 14:00 README.org*
drwxr-xr-x  3 adel adel 4096 mars  21 14:43 tests/
```

EXO 2:

- ❖ 2.1 Analyse de Dépendance
  - ✓ Qu'est-ce qu'une dépendance de données ?
  - ✓ Application aux permutations de boucles.
  - ✓ Cas d'étude: ssgemm dans mcblas001
  - ✓ Conclusion : Permutations possibles
- ❖ 2.2 Application des permutations et évaluation
  - ✓ Cas 1 : Code de base
  - ✓ Résultat global des versions
  - ✓ Permutation 1 : (r, d, c)
  - ✓ Permutation 2 : (d, r, c)
  - ✓ Permutation 3 : (c, r, d)
  - ✓ Permutation 4 : (c, d, r)
  - ✓ Permutation 5 : (d, c, r)
  - ✓

Q- 2.1 Analyse de dépendance:

Introduction:

Qu'est-ce qu'une dépendance de données dans une boucle ?

Une dépendance de données se produit lorsqu'une itération d'une boucle dépend des résultats d'une autre itération.
Il existe trois types de dépendances :

1-Dépendance vraie (Flow Dependence)
2-Dépendance anti-dépendance (Anti-dependence)
3-Dépendance de sortie (Output Dependence)

Exemples concrets de dépendances:

1. Dépendance vraie (Flow Dependence) =>
Une itération lit une valeur qui a été modifiée par une itération précédente. Par exemple:

```
for (int i = 1; i < 10; i++) {
    A[i] = A[i - 1] + 2;
}
```

-A[i] dépend de A[i - 1] (valeur de l'itération précédente).
-Chaque itération **doit attendre** que la précédente soit terminée pour fonctionner correctement.
-Il **n'est pas possible d'exécuter ces itérations en parallèle**.

Solution possible :
Si l'on souhaite optimiser cela, une **technique de pré-calcul** pourrait être appliquée, mais en général, ce type de dépendance empêche la parallélisation.

2.Dépendance anti-dépendance (Anti-dependence) =>
Une itération écrit une valeur qui sera lue par une itération suivante. Par exemple :

```
for (int i = 0; i < 10; i++) {
    A[i] = B[i] + 2;
    B[i] = A[i] * 3;
}
```

-A[i] est calculé avec B[i], puis **B[i] est modifié**.
-Si les instructions étaient inversées, **le programme donnerait un résultat différent**.

Solution possible :
Utiliser des **variables temporaires** pour éviter l'écrasement de B[i].

```
//solution possible
for (int i = 0; i < 10; i++) {
    float temp = B[i] + 2; // Stockage temporaire
    B[i] = temp * 3;
    A[i] = temp;
}
//Maintenant, les opérations peuvent être parallélisées.
```

3.Dépendance de sortie (Output Dependence) =>
Deux itérations écrivent sur la même variable.

```
for (int i = 0; i < 10; i++) {
    A[0] = A[0] + i;
}
```
-Toutes les itérations modifient A[0] !
-Cela **crée une dépendance entre chaque itération**.

Solution possible :
Utiliser une variable locale et **réaliser l'accumulation après la boucle**.

```
//solution possible
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum += i;
}
A[0] = sum;
//Maintenant, les itérations n'ont plus de dépendance entre elles,
//et la boucle peut être optimisée pour le parallélisme.
```

Application aux permutations de boucles:
_Si une boucle **n'a pas de dépendance vraie**, nous pouvons **changer son ordre** sans affecter le résultat.
_Dans ssgemm, il faut tester différentes **permutations** et voir laquelle optimise **les accès mémoire et la vitesse d'exécution**.

Qu'est-ce qu'une dépendance de données dans un nid de boucles :
Une dépendance de données se produit lorsqu'une itération d'une boucle lit ou écrit une valeur qui dépend d'une autre itération.
Dans une fonction comme ssgemm, nous avons trois boucles imbriquées (r, c, d).
Chacune d'elles manipule la matrice C qui stocke le résultat.

Une dépendance existe si :
_Une **itération dépend des valeurs modifiées par une autre**.
_Une **réorganisation des boucles entraîne un changement du résultat final**.

Analyse des dépendances dans notre TP :

Commençons par ssgemm qui se trouve dans **mcblas001/mcblas3.cpp** :

```
void ssgemm(
    int m, int n, int k,
    float alpha,
    float ** A,
    float ** B,
    float beta,
    float ** C)
{
    // std::cout << "************************************" << std::endl;
    // std::cout << "   WARNING: Not Yet Implemented..." << std::endl;
    // std::cout << "************************************" << std::endl;

    int r, c, d;

    for (r = 0;  r < m; ++r) // Parcourt les lignes de C
    {
        for (c = 0;  c < n; ++c) // Parcourt les colonnes de C
        {
            for (d = 0;  d < m; ++d)    //Accumulation (sommation)
            {
                C[r][c] = beta * C[r][c] + alpha * A[r][d] * B[d][c];
            }
        }
    }
```

-C[r][c] est **modifié plusieurs fois** dans la boucle d.
-Il **n'y a pas de dépendance entre différentes valeurs de r et c**, mais uniquement dans d.

Analyse des dépendances possibles :

-Dépendance sur C[r][c]
Chaque itération de d lit et met à jour C[r][c]. Il n'y a pas de conflit car d est toujours lu et modifié sur la même cellule C[r][c].
-Dépendance sur A[r][d] et B[d][c]
Il n'y a aucune dépendance entre différentes valeurs de r et c, donc nous pouvons réorganiser les boucles.

Permutations de boucles possibles:

Nous pouvons essayer les permutations suivantes :

Permutation :	Ordre des boucles :
(r, c, d) (original)	for r → for c → for d
(r, d, c)	for r → for d → for c
(d, r, c)	for d → for r → for c
(d, c, r)	for d→ for c → for r
(c, r, d)	for c → for r → for d
(c, d, r)	for c → for d → for r

Q-2.2 : Application des permutations et évaluation: (dans mcblas001/mcblas3.cpp)

Cas 1: (version de base (sans modification dans le code)):

Compilation et Exécution des Tests:

```
adel@adel:~/Documents/TD3_DB/libmcblas002/build$ ll
total 8
drwxrwxr-x  2 adel adel 4096 mars  22 17:16 ./
drwxr-xr-x 11 adel adel 4096 mars  22 17:16 ../
adel@adel:~/Documents/TD3_DB/libmcblas002/build$ cmake ../
-- The CXX compiler identification is GNU 11.4.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/adel/Documents/TD3_DB/libmcblas002/build
adel@adel:~/Documents/TD3_DB/libmcblas002/build$
```

Cela va générer les fichiers de compilation à partir de CMakeLists.txt
Exécution des tests : (lance la compilation et génère les fichiers exécutables et les bibliothèques nécessaires à l'exécution des test).
Cela donne:

```
[ 94%] Generating data/Rtst00402_1000.data
[ 94%] Built target Rtst00402_1000
[ 95%] Generating data/Rtst00502_0400.data
[ 95%] Built target Rtst00502_0400
[ 96%] Generating data/Rtst00502_0800.data
[ 96%] Built target Rtst00502_0800
[ 97%] Generating data/Rtst00502_1000.data
[ 97%] Built target Rtst00502_1000
[ 98%] Generating data/Rtst00602_0400.data
[ 98%] Built target Rtst00602_0400
[100%] Generating data/Rtst00602_0800.data
[100%] Built target Rtst00602_0800
[100%] Built target O2_RUNS
[100%] Built target ALL_RUNS
[100%] Built target BALLS
```

make GALL : générer les graphiques de performance : Cela a généré plusieurs fichiers de données tels que gr_0400.data, gr_0800.data, etc. Ces fichiers contiennent les cycles d'horloge pour différentes tailles de matrices (400, 800, 1000), et ont servi à créer les graphiques.

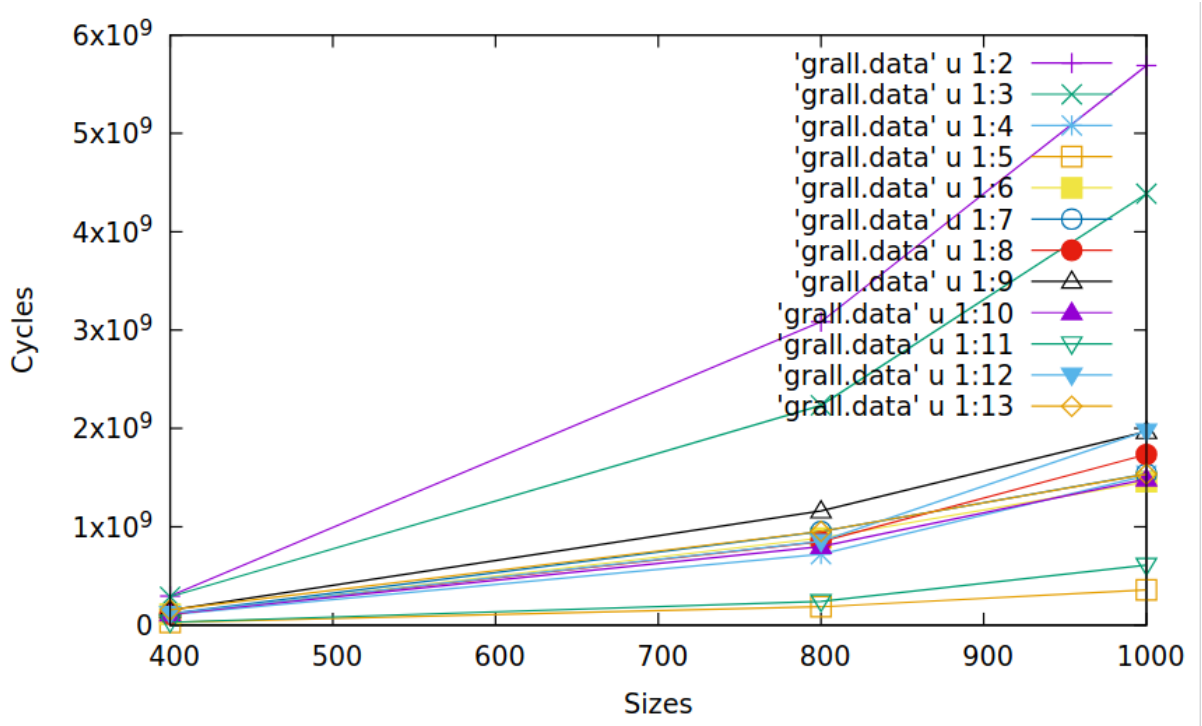
```
adel@adel:~/Documents/TP3_DB_LAST/libmcblas002/build$ make GALL
[ 33%] Generating graphs/gr_1000.data
[ 33%] Built target TTgraphs_gr_1000
[ 33%] Generating graphs/gr_0400.data
[ 33%] Built target TTgraphs_gr_0400
[ 66%] Generating graphs/gr_0800.data
[ 66%] Built target TTgraphs_gr_0800
[100%] Generating graphs/grall.data
[100%] Built target TTGRALL
[100%] Built target GALL
```

Les résultats de la commande make GALL indiquent que les **données des tests** ont été générées avec succès et agrégées dans des fichiers situés dans graphs/.

Pour générer les graphiques sous forme de fichiers PDF :

```
adel@adel:~/Documents/TP3_DB_LAST/libmcblas002/build/tests/runs/graphs$ make -f grph.mk
gnuplot gpltv123456.plt > grv123456.pdf
adel@adel:~/Documents/TP3_DB_LAST/libmcblas002/build/tests/runs/graphs$
```

Cela donne =>

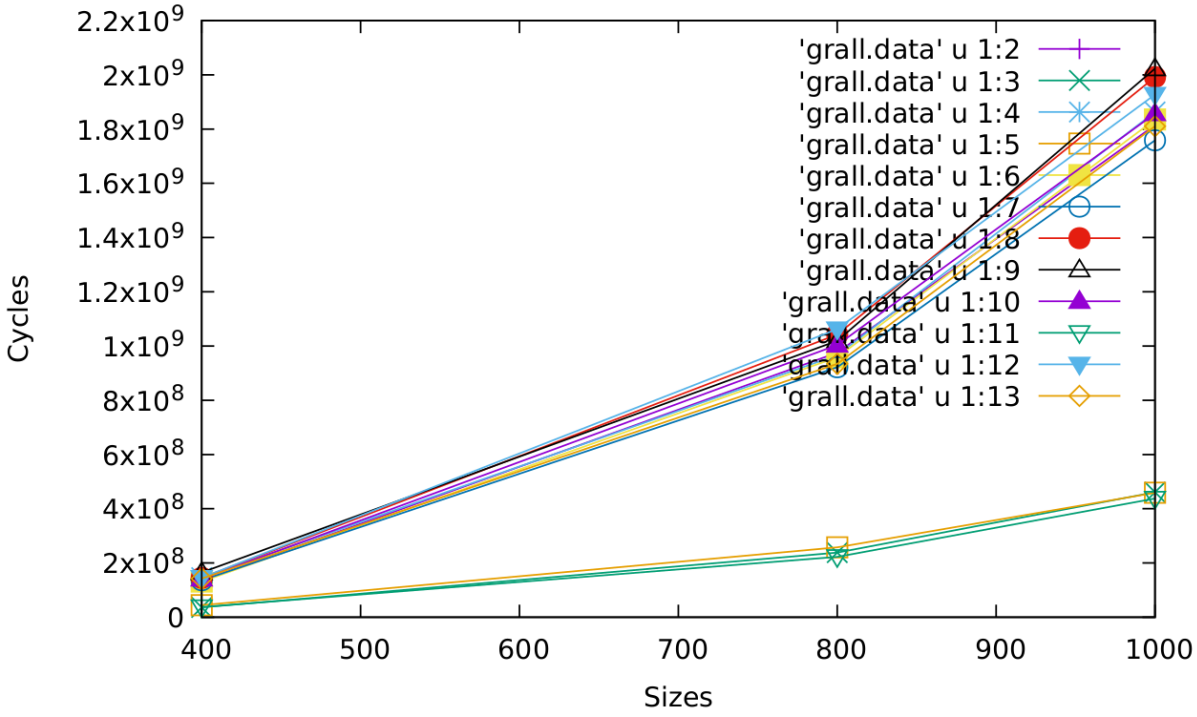


_Le graphe regroupe toutes les versions en une seule visualisation, et chaque courbe correspond à une implémentation spécifique de ssgemm.
_Chaque ligne dans le graphe représente une version différente de ssgemm, provenant des différents fichiers mcblas3.cpp situés dans les sous-répertoires mcblas001, mcblas002,…..
_Ces différentes versions ont été compilées et exécutées avec les tests (tst001.cpp, tst002.cpp, etc.), et les résultats ont été enregistrés dans les fichiers de données utilisés pour générer le graphe.
_Certaines versions sont plus optimisées que d'autres et auront donc un nombre de cycles CPU plus bas pour les mêmes tailles de matrice.
_Certaines versions utilisent des techniques comme le blocking pour optimiser l'utilisation du cache, tandis que d'autres utilisent des itérations simples.
_Lorsque nous avons exécuté cmake et make, la compilation a généré des versions optimisées de chaque ssgemm avec différentes options d'optimisation (-O2 et -O3), c'est à cause de ça que le graphe ne représente pas seulement les 7 versions de ssgemm issues des fichiers mcblas3.cpp.

- Permutation 1 : (r, d, c):

```
for (r = 0;  r < m; ++r) {
    for (d = 0;  d < m; ++d) {
        for (c = 0;  c < n; ++c) {
            C[r][c] = beta * C[r][c] + alpha * A[r][d] * B[d][c];
        }
    }
}
```

==> cela donne



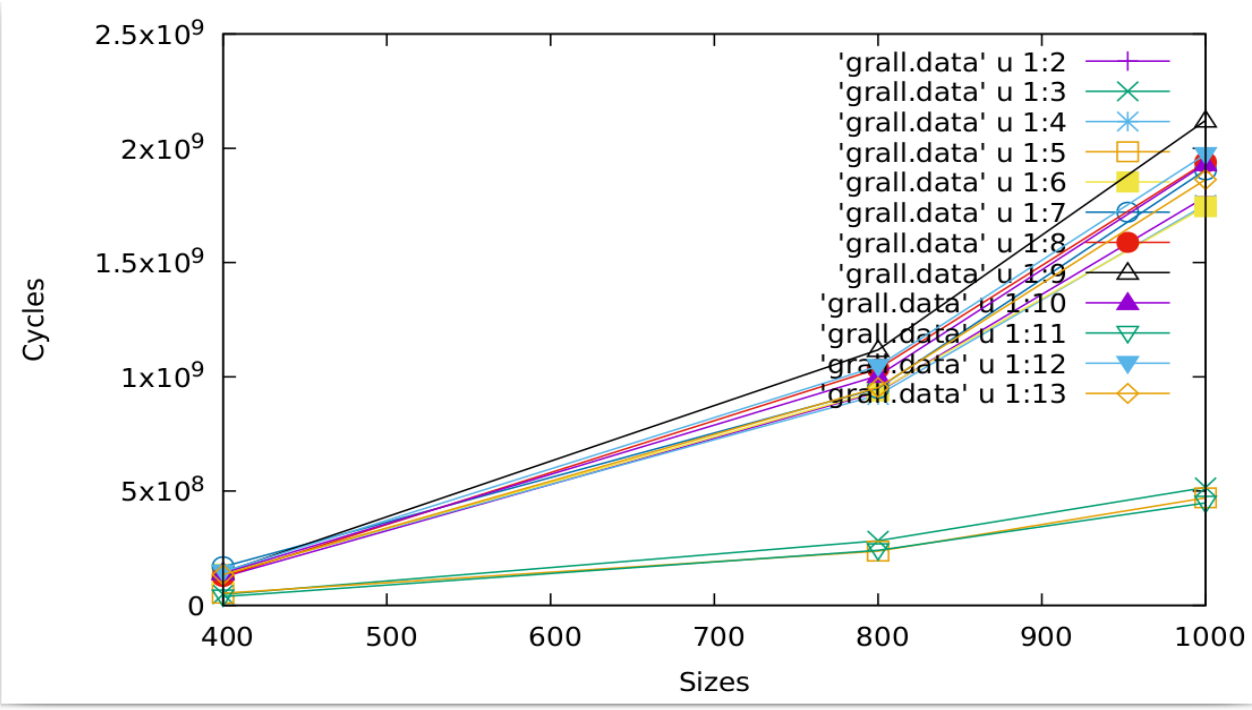
Résultats :
Les performances sont globalement bonnes et stables. Le nombre de cycles reste modéré même pour les grandes tailles (800 et 1000), ce qui indique une certaine efficacité.

Explication :
L'ordre (r, d, c) permet une bonne exploitation du cache pour l'écriture dans la matrice C et un accès cohérent à A et B, ce qui limite les défauts mémoire. Cette structure s'avère équilibrée entre simplicité et performance.

- Permutation 2 : (d, r, c):

```
for (d = 0;  d < m; ++d) {
    for (r = 0;  r < m; ++r) {
        for (c = 0;  c < n; ++c) {
            C[r][c] = beta * C[r][c] + alpha * A[r][d] * B[d][c];
        }
    }
}
//Objectif : Modifier l'ordre d'accès à B pour voir l'impact sur la mémoire cache
```

==> cela donne:
Résultats : Performances encore plus mauvaises par rapport à (r, d, c).



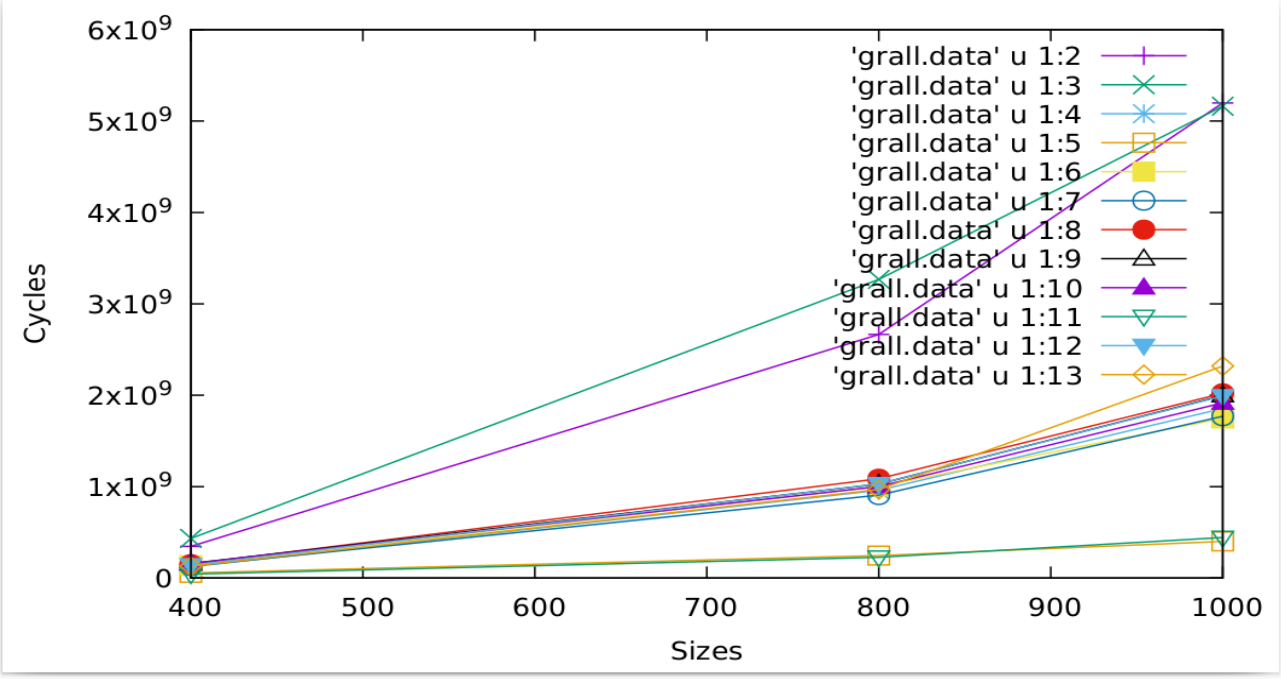
Résultats :
Cette permutation affiche les pires performances globales.

Explication :
L'ordre d'accès mémoire (d, r, c) perturbe la localité spatiale, ce qui entraîne une forte latence due aux nombreux défauts de cache.

-Permutation 3 : (c, r, d):

```
for (c = 0;  c < n; ++c) {
    for (r = 0;  r < m; ++r) {
        for (d = 0;  d < m; ++d) {
            C[r][c] = beta * C[r][c] + alpha * A[r][d] * B[d][c];
        }
    }
}
//Objectif : Améliorer la localité spatiale de C.
```

==> cela donne:



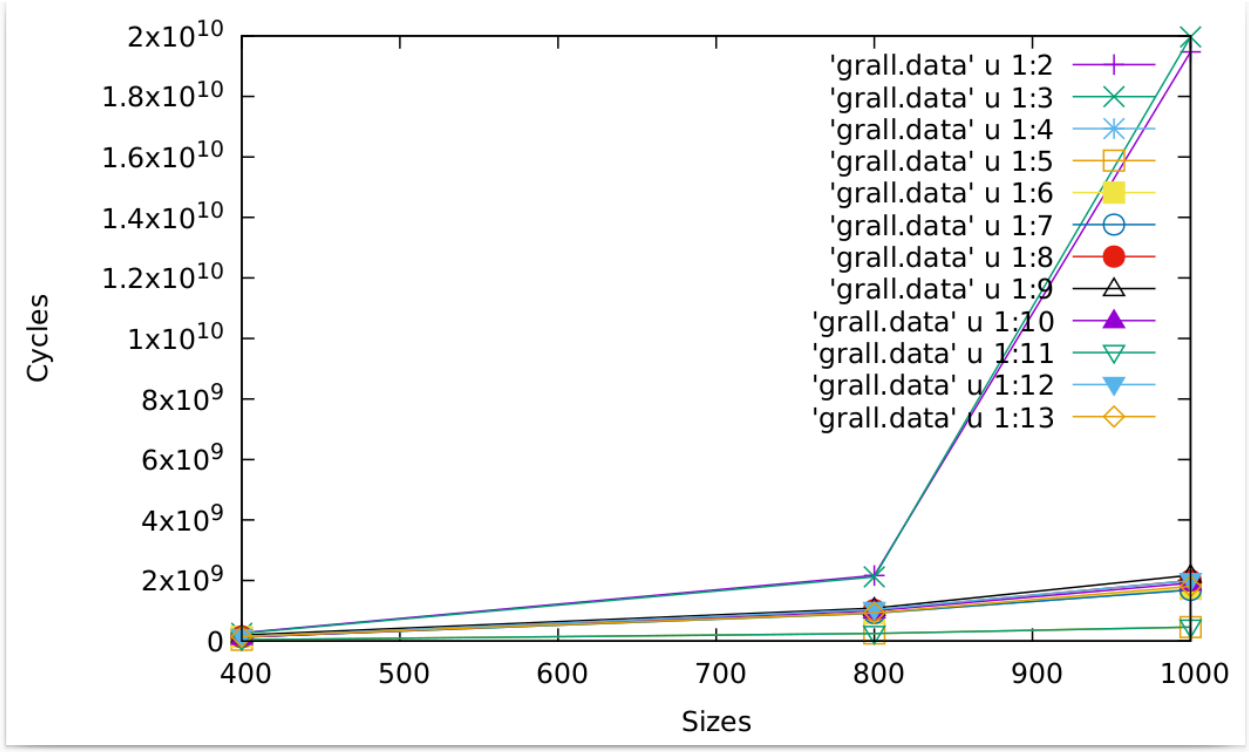
_Résultats :
Temps d'exécution amélioré par rapport aux deux permutations précédentes.

_Explication :
Un meilleur ordonnancement des accès mémoire permet une meilleure exploitation du cache, bien que ce ne soit pas encore optimal.

-Permutation 4 : (c, d, r):

```
for (c = 0;  c < n; ++c) {
    for (d = 0;  d < m; ++d) {
        for (r = 0;  r < m; ++r) {
            C[r][c] = beta * C[r][c] + alpha * A[r][d] * B[d][c];
        }
    }
}
//Objectif : Changer la structure d'accès aux matrices pour voir
// si cela améliore l'utilisation des caches.
```

==> cela donne:



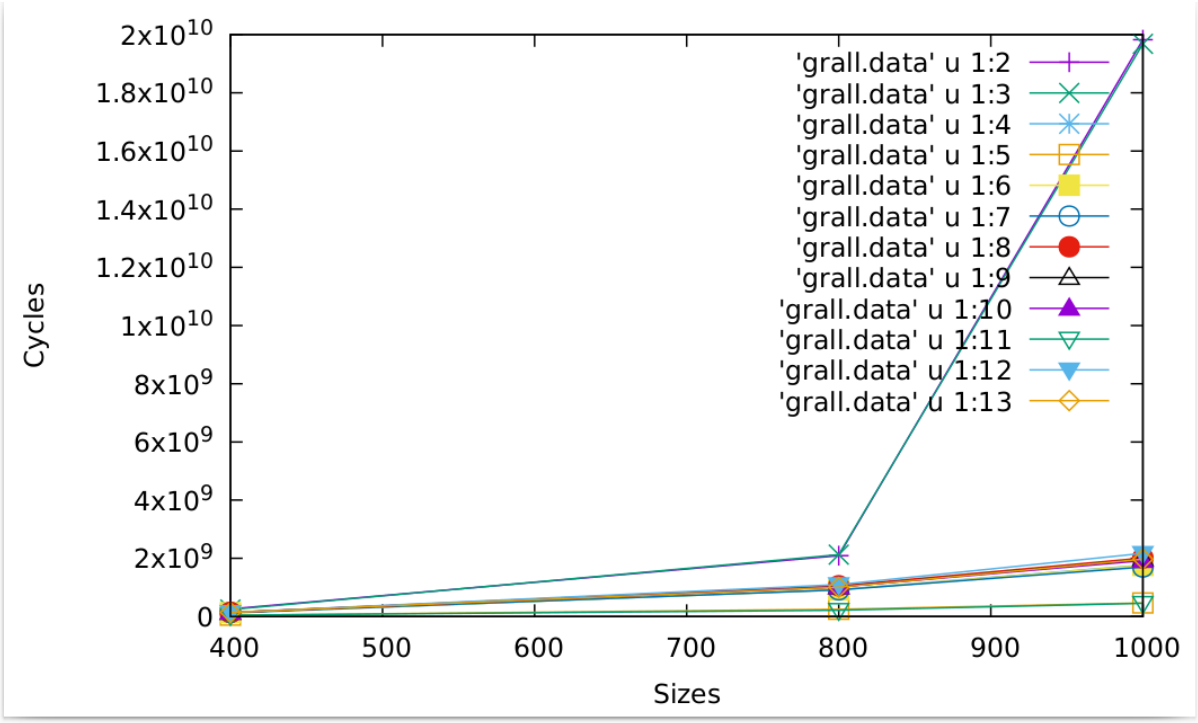
Résultats :
On observe une amélioration notable par rapport à d'autres permutations, surtout pour les grandes tailles.

Explication : L'ordre (c, d, r) améliore la localité spatiale des données, ce qui réduit les défauts de cache et optimise le parcours mémoire un peu.

-Permutation 5 : (d, c, r):

```
for (d = 0;  d < m; ++d) {
    for (c = 0;  c < n; ++c) {
        for (r = 0;  r < m; ++r) {
            C[r][c] = beta * C[r][c] + alpha * A[r][d] * B[d][c];
        }
    }
}
//Objectif : Changer la structure d'accès aux matrices pour voir
// si cela améliore l'utilisation des caches.
```

==> cela donne:



Résultats :
Les performances sont globalement correctes, mais inférieures à celles observées avec certaines autres permutations, surtout pour les grandes tailles.

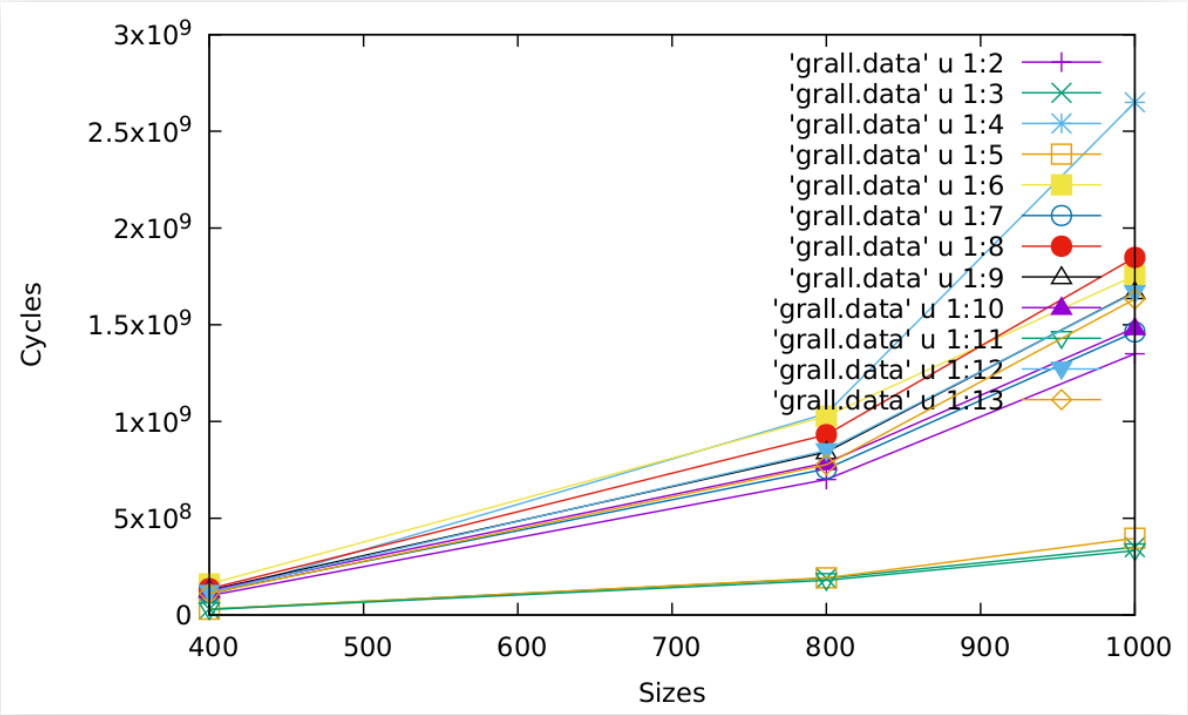
Explication :
Bien que l'ordre (d, c, r) permette un accès séquentiel à la matrice B et une certaine régularité dans les écritures sur C, l'accès non optimal à A (ligne par ligne à l'intérieur d'une boucle peu favorable) dégradee l'efficacité du cache.

Conclusion :

Parmi toutes les permutations testées, la permutation **(r, d, c)** c'est la **plus performante**, en particulier pour les grandes tailles de matrices (800 et 1000).
Elle présente des **temps d'exécution plus faibles et des courbes plus stables**, ce qui reflète une **meilleure gestion de la mémoire cache** et une **localité d'accès optimale**.
Ces résultats confirment que **l'ordre des boucles a un impact significatif sur les performances**, et que (r, d, c) est la **structure à privilégier** pour le reste du TP.

Appliquant la permutation (r,d,c) sur /mcblas002/mcblas3 :

cela donne :



Après avoir appliqué la permutation optimale (r, d, c) dans le fichier /mcblas002/mcblas3.cpp, on observe une légère amélioration des performances. Comparée à la version (où on appliqué r,d,c juste sur mcblas001/mcblas3.cpp), cette version semble mieux exploiter la hiérarchie mémoire.

Exercice 3. Blocage de boucles:

- ❖ **Introduction au blocage**
- ✓ **C'est quoi le blocage**
- ✓ **Objectifs du blocage**
- ✓ **Comparaison : version classique vs version bloquée**

- ❖ **Implémentation du blocage sur mcblas005**

- ✓ **Choix du fichier d'étude d'analyse**
- ✓ **Implémentation du code avec blocage**
- ✓ **Compilation et génération des données**
- ✓ **Blocage avec différentes tailles de blocs :**
  - ➢ Taille 16
  - ➢ Taille 32
  - ➢ Taille 64
  - ➢ Taille 128
- ✓ **Interprétation comparative des résultats**

- ❖ **Uniformisation du blocage avec la meilleure permutation trouvée**

- ✓ **Objectif de l'uniformisation**
- ✓ **Résultat observé**

- ✓ **Conclusion générale sur le bloking**

Introduction:

Le blocage (tiling) est une technique d'optimisation très utilisée pour améliorer la performance des algorithmes manipulant de grandes matrices, notamment pour la multiplication de matrices comme ssgemm.

Le but est de :

- ➢ Réduire les accès mémoire couteux
- ➢ Exploiter le cache de manière efficace
- ➢ Améliorer la localité spatiale et temporelle

Pratiquement. Au lieu de parcourir les matrices ligne par ligne ou colonne par colonne, on découpe les matrices en petits blocs carrés ou rectangulaires, et on effectue les calculs bloc par bloc. Autrement dit, L'idée du blocage est de diviser les grandes boucles en petites sous-boucles, appelées blocs, pour traiter une portion de la matrice à la fois

Version classique (sans blocage):

```
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      C[i][j] += A[i][k] * B[k][j];
```

Déroulement:

-Pour chaque ligne i de A
-Pour chaque colonne j de B
-Pour chaque élément k, on multiplie A[i][k] avec B[k][j] et on l'ajoute à C[i][j]

Inconvénients :

-Accès mémoire **non local** → mauvais pour les **caches**
-Les matrices sont **trop grandes** pour tenir dans le cache → ralentissements

Version avec **blocage (tiling):**

```
for (int ii = 0; ii < N; ii += Tr)
  for (int jj = 0; jj < N; jj += Tc)
    for (int kk = 0; kk < N; kk += Td)
      for (int i = ii; i < min(ii+Tr, N); ++i)
        for (int j = jj; j < min(jj+Tc, N); ++j)
          for (int k = kk; k < min(kk+Td, N); ++k)
            C[i][j] += A[i][k] * B[k][j];
```

Déroulement bloc par bloc :

Variable:	**Rôle:**

Tr	nombre de lignes par bloc
Tc	nombre de colonnes par bloc
Td	profondeur k du bloc

Grâce au blocage :

_On travaille dans de **petits morceaux**.
_On gardes mieux les **données en cache**, donc **plus rapide**.

Q- 3.1 Analyse de dépendance

On a :

Dossiers	**Technique utilisée**	**Blocage appliqué deja ?**	**Commentaire**
mcblas001	Boucle simple (non optimisée)	Non	Utilisé pour tester les permutations
mcblas002	Boucle simple (non optimisée)	Non	Utilisé aussi pour tester les permutations
mcblas003	Blocage classique	Oui	Avec blocs Tr, Tc, Td définis
mcblas004	Blocage avec autres tailles	Oui	Blocage avec Tr=16, Tc=64, Td=32
mcblas005	Boucle simple	Non	**on peux tester le blocage ici**
mcblas006	Blocage (tiling) avec accès en 1D	OUI	blocage efficace!

choisir mcblas005 pour analyser les dépendance et tester le blocage par la suite.

=> Parce que c'est le seul fichier parmi les versions restantes qui n'a pas encore de blocage.

Étape 1 – Activer le code bloqué (commenté) et commente le bloc non-bloqué :

```
6   #define A(r,c) A[r*k+c]
7   #define B(r,c) B[r*n+c]
8   #define C(r,c) C[r*n+c]
9
10  void ssgemm(
11      int m, int n, int k,
12      float alpha,
13      float *A,
14      float *B,
15      float beta,
16      float *C)
17  {
18      // std::cout << "************************************" << std::endl;
19      // std::cout << "   WARNING: Not Yet Implemented..." << std::endl;
20      // std::cout << "************************************" << std::endl;
21
22      int r, c, d;
23      int rr, cc, dd;
24      const int Tr = 32;
25      const int Tc = 32;
26      const int Td = 32;
```

mcblas3.cpp

C: > Users > adelb > Documents > debo_profil_c > pro_para > TP_MPI_no_solution > TD3_DB > libmcblas002 > mcblas005 >

```
17  {
28      for (dd = 0;  dd < k; dd += Td)
29      {
30          for (rr = 0;  rr < m; rr += Tr)
31          {
32              for (cc = 0;  cc < m; cc += Tc)
33              {
34                  for (r = rr;  r < std::min(rr+Tr ,m); ++r)
35                  {
36                      for (c = cc;  c < std::min(cc+Tc, n); ++c)
37                      {
38                          float t=0.0;
39                          for (d = dd;  d < std::min(dd+Td, k); ++d)
40                          {
41                              //t = t + alpha * A[r*n+d] * B[d*n+c];
42                              t = t + alpha * A(r,d) * B(d,c);
43                          }
44                          C(r,c) = C(r,c) * beta + t;
45                      }
46                  }
47              }
48          }
49      }
```

L'analyse de dépendance a pour objectif d'examiner s'il existe des liens de dépendance entre les itérations des boucles imbriquées d'un programme. Ces dépendances, si elles existent, peuvent empêcher certaines transformations de boucles comme le parallélisme, les permutations ou le blocage (tiling).
Dans le cas de notre fonction ssgemm, on cherche à déterminer si le blocage des boucles de calcul (c'est-à-dire diviser les boucles en petits blocs traités indépendamment) est possible sans modifier le résultat final.

Le blocage (ou tiling) consiste à découper les grandes boucles en petites portions, appelées blocs, que l'on parcourt avec une boucle supplémentaire. Exemple typique :
Les tailles de blocs Tr, Tc et Td définissent respectivement :

- -le nombre de lignes (r),
- -de colonnes (c),
- -et de profondeur (d) dans chaque sous-bloc.

Même si l'algorithme de base est **correct**, son exécution sur des matrices de grande taille souffre souvent de **mauvaise localité mémoire**.

Cela signifie que les accès aux éléments des matrices ne suivent pas toujours un schéma **optimisé pour le cache**.

-Le blocage permet :

- _D'améliorer la localité spatiale (accès à des éléments proches),
- _D'améliorer la localité temporelle (réutiliser rapidement les données),
- _De réduire le nombre de cache misses (accès lents à la RAM),
- _Et donc d'améliorer les performances d'exécution sur des grandes matrices.

Maintenant que nous avons confirmé que le blocage est autorisé par l'analyse de dépendance, il est utile de se poser les questions suivante :

-Est ce que le blocking il va vraiment améliorer les performance sur ma machine ??
-Quelle taille de bloc choisir ??

Q- 3.2 Blocking

On a : (apres avoir appliquer le blocking sur mcblas005 en enlevant les commantaire)

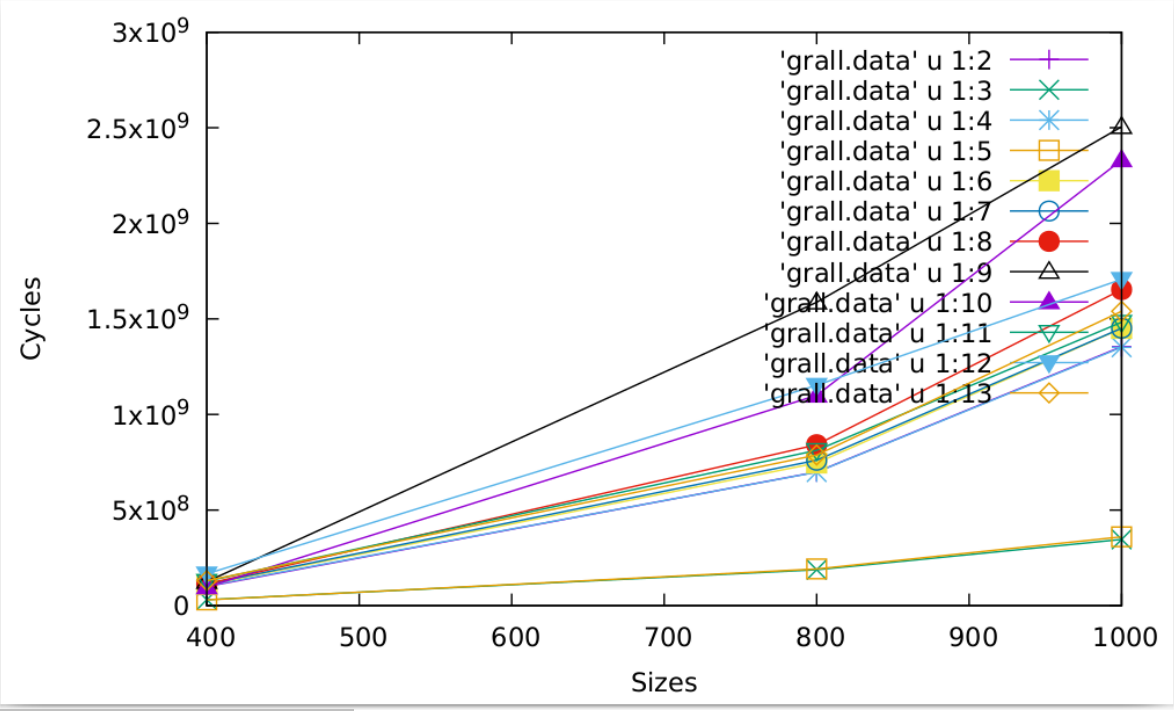
```
 7 #define A(r,c) A[r*k+c]
 8 #define B(r,c) B[r*n+c]
 9 #define C(r,c) C[r*n+c]
10
11 void ssgemm(
12     int m, int n, int k,
13     float alpha,
14     float *A,
15     float *B,
16     float beta,
17     float *C)
18 {
19
20         // std::cout << "**************************************" << std::endl;
21         // std::cout << "   WARNING: Not Yet Implemented..." << std::endl;
22         // std::cout << "**************************************" << std::endl;
23
24          int r, c, d;
25          int rr, cc, dd;
26          const int Tr = 32;
27          const int Tc = 32;
28          const int Td = 32;
29
30
31          for (dd = 0;  dd < k; dd += Td)
32          {
33                 for (rr = 0;  rr < m; rr += Tr)
34                 {
35                         for (cc = 0;  cc < m; cc += Tc)
36                         {
37                                 for (r = rr;  r < std::min(rr+Tr ,m); ++r)
38                                 {
39                                         for (c = cc;  c < std::min(cc+Tc, n); ++c)
40                                         {
41                                                 float t=0.0;
42                                                 for (d = dd;  d < std::min(dd+Td, k); ++d)
43                                                 {
44                                                         //t = t + alpha * A[r*n+d] * B[d*n+c];
45                                                         t = t + alpha * A(r,d) * B(d,c);
46                                                 }
47                                                 C(r,c) = C(r,c) * beta + t;
48                                         }
49                                 }
50                         }
51                 }
52         }
```

Recompiler et tester :

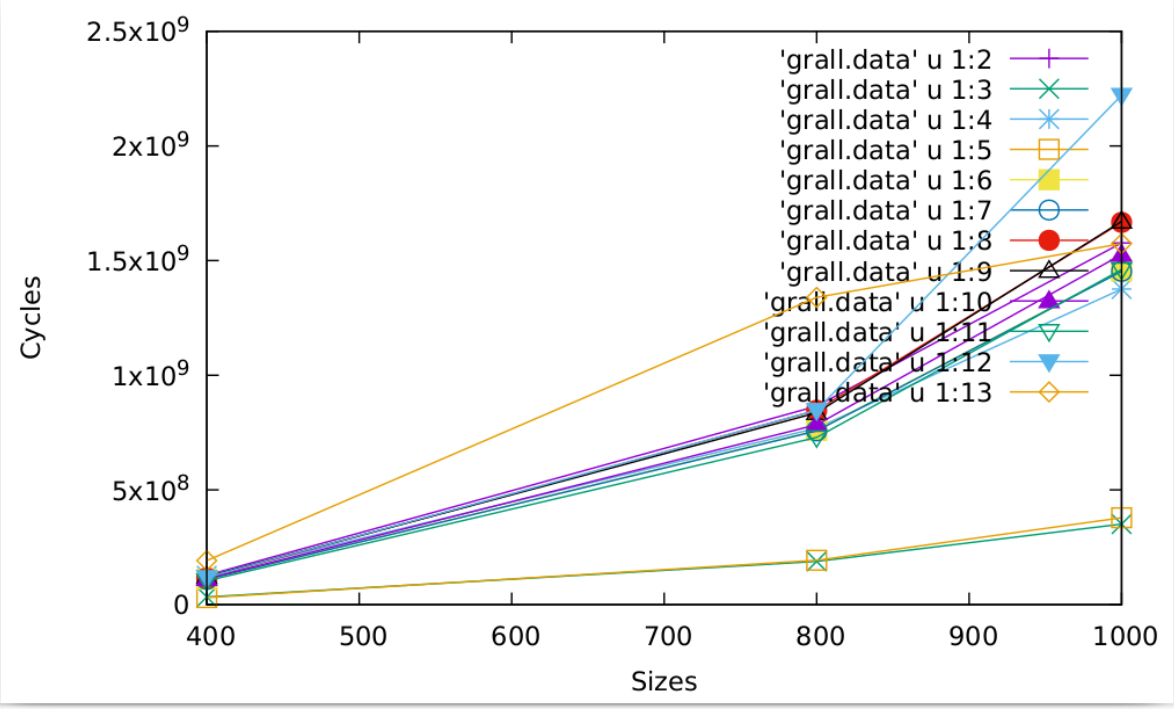
```
make
make GALL
cd tests/runs/graphs
make -f grph.mk
```

appliquer le blocking (dans /mcblas005/mcblas3.cpp) avec différentes tailles de bloques en conservant la permutation (r,d,c) sur mcblas001/mcblas3.cpp et mcblas002/mcblas3.cpp:

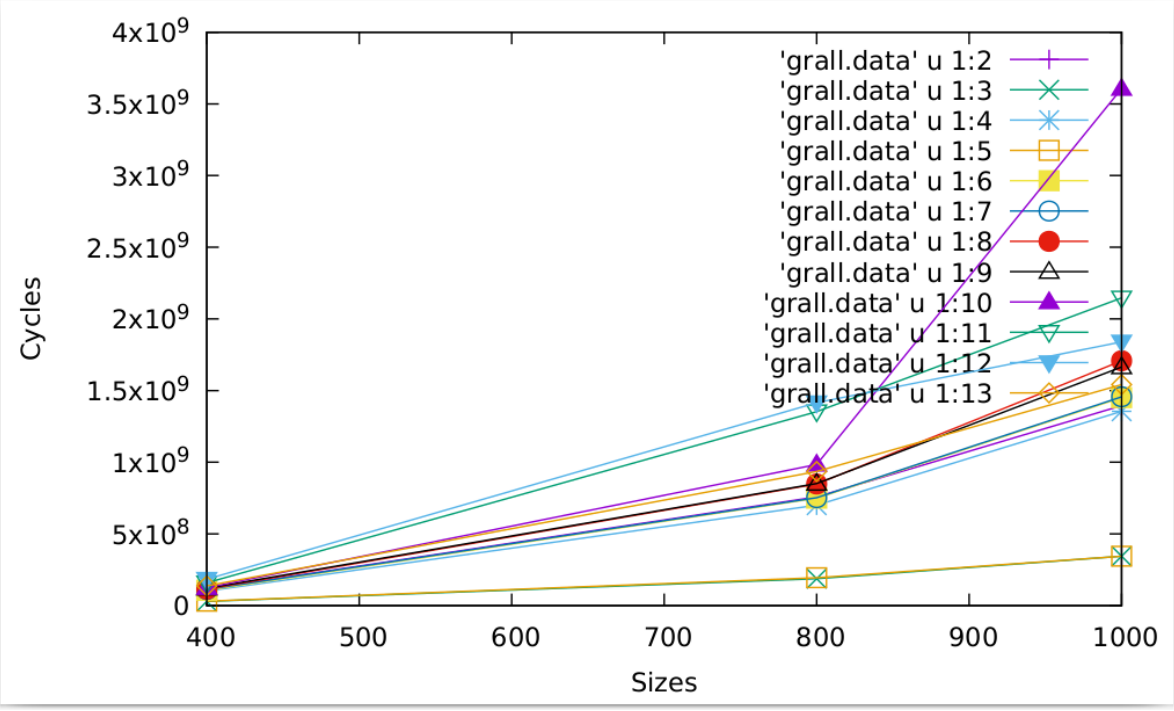
Taille du bloque = 16=>



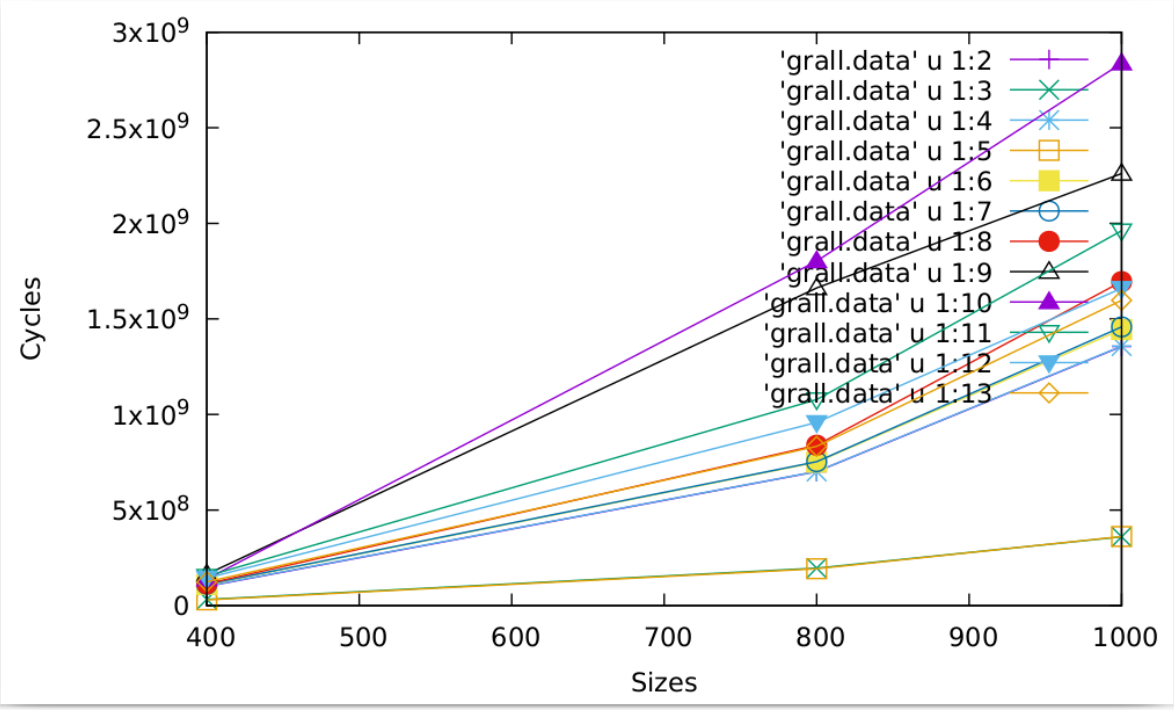
Taille du bloque = 32=>



Taille du bloque = 64=>



Taille du bloque = 128=>



Analysons les résultats :

Bloc 16 :

- Plusieurs courbes dépassent 2.5×10^9 cycles pour la taille 1000.
- *Performances instables*, avec une dispersion entre versions => peu optimal.

Bloc 32 :

- la majorité des courbes sont moins hautes que pour bloc 16.
- Meilleur compromis entre stabilité et performance => Bon.

Bloc 64 :

- Quelques versions affichent encore des pics marqués (jusqu'à 4×10^9).
- Performances pas toujours meilleures que bloc 32 → Moins cohérent.

Bloc 128 :

- Résultats globalement moins bons que bloc 32 dans ce nouveau contexte.
- Certaines courbes montent au-dessus de 2.5×10^9
-

Alors :

La taille de bloc 32 offre les meilleures performances dans ce contexte :

_ Moins de cycles en moyenne pour les grandes tailles,

Remarque :

Lorsqu'on souhaite évaluer l'impact d'un paramètre comme la taille de bloc (par exemple 16, 32, 64, 128), il est essentiel de l'appliquer à l'ensemble des scripts de blocking du TP (/mcblas003, /mcblas004, /mcblas005, /mcblas006, etc.) et non uniquement à un seul (/mcblas005), pour les raisons suivantes :

-Chaque ligne dans le graphe représente une version (un script) différente, mais on sait pas avec certitude à quel fichier correspond chaque ligne.

-Le comportement du cache n'est pas déterministe :

- Deux exécutions d'un même script peuvent produire des résultats légèrement différents.
- Une très légère amélioration dans un fichier peut passer inaperçue ou au contraire sembler pire à cause de variations aléatoires liées au système.

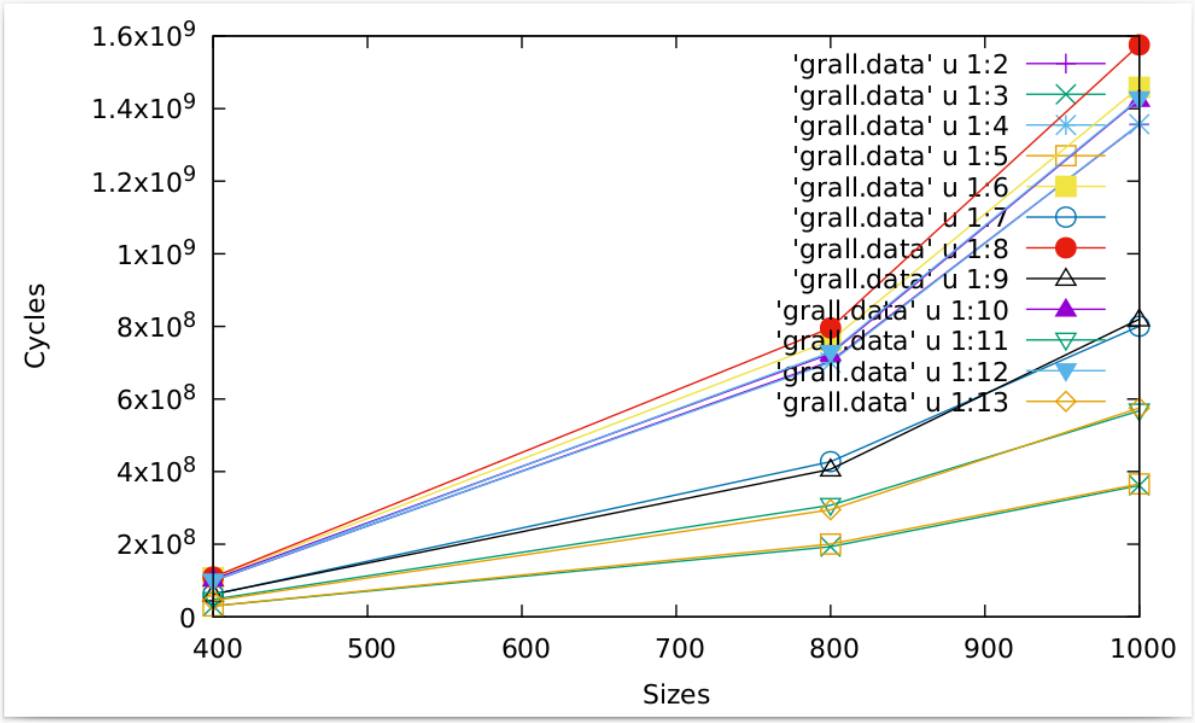
-Modifier un seul script et observer le graphe global (où les autres scripts ne changent pas) n'est pas suffisant pour isoler l'impact de la modification.

Pour valider rigoureusement qu'une taille de bloc est réellement meilleure qu'une autre, dans le contexte de notre TP il faut :

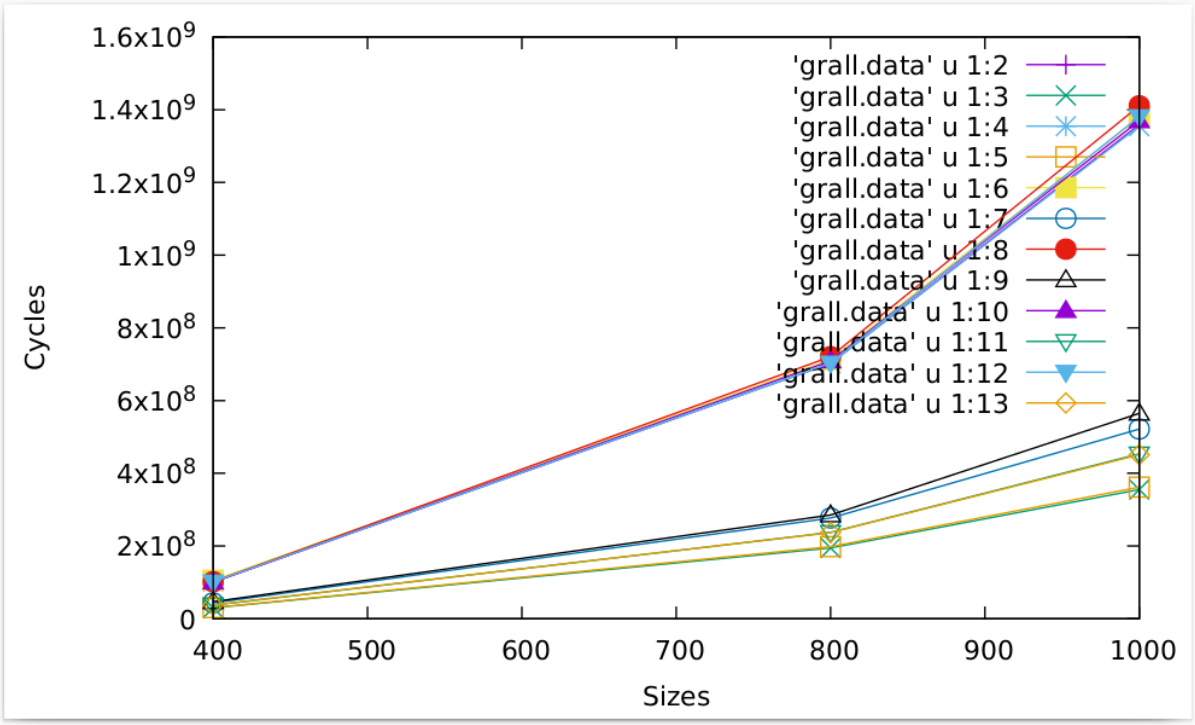
- Uniformiser cette taille de bloc sur tous les scripts de blocking.
- Puis générer des nouveaux graphes comparatifs, permettants une comparaison équitable.

appliquer la meme taille du bloque dans tous les script de blocking dans notre TP avec différentes tailles de bloques en appliquant aussi la permutation (r,d,c) sur tous les scripts mcblas3.cpp :

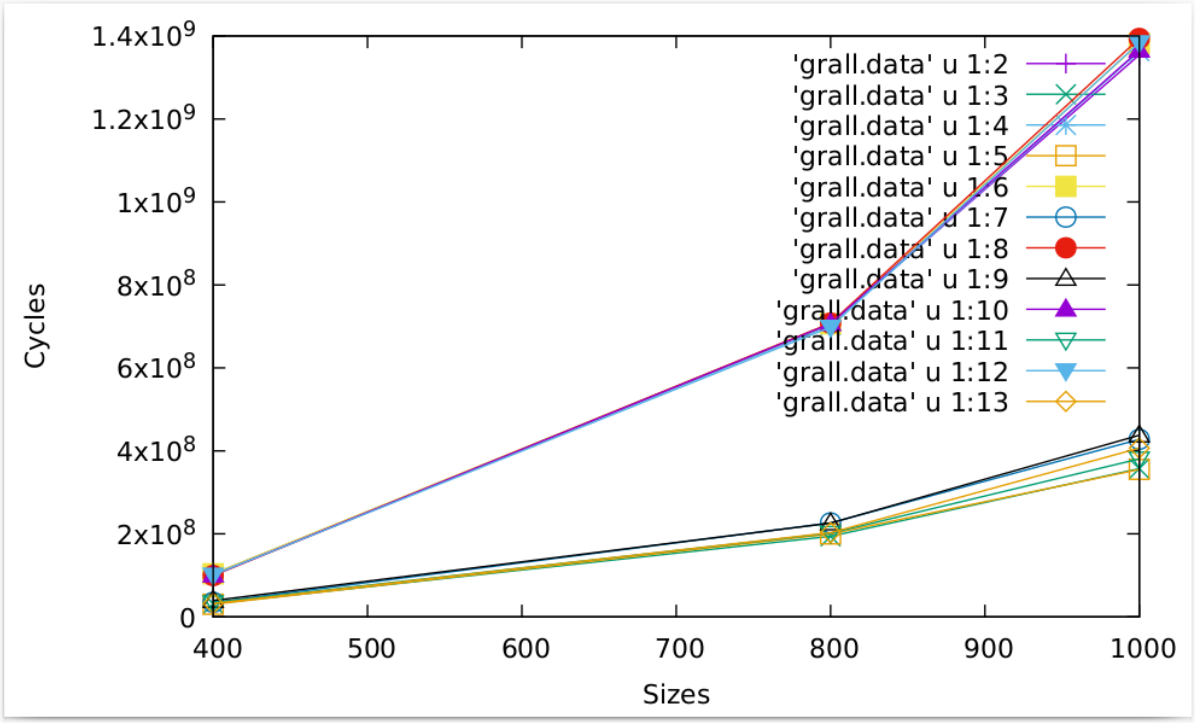
Taille du bloque uniformisée = 16 avec permutation r,d,c =>



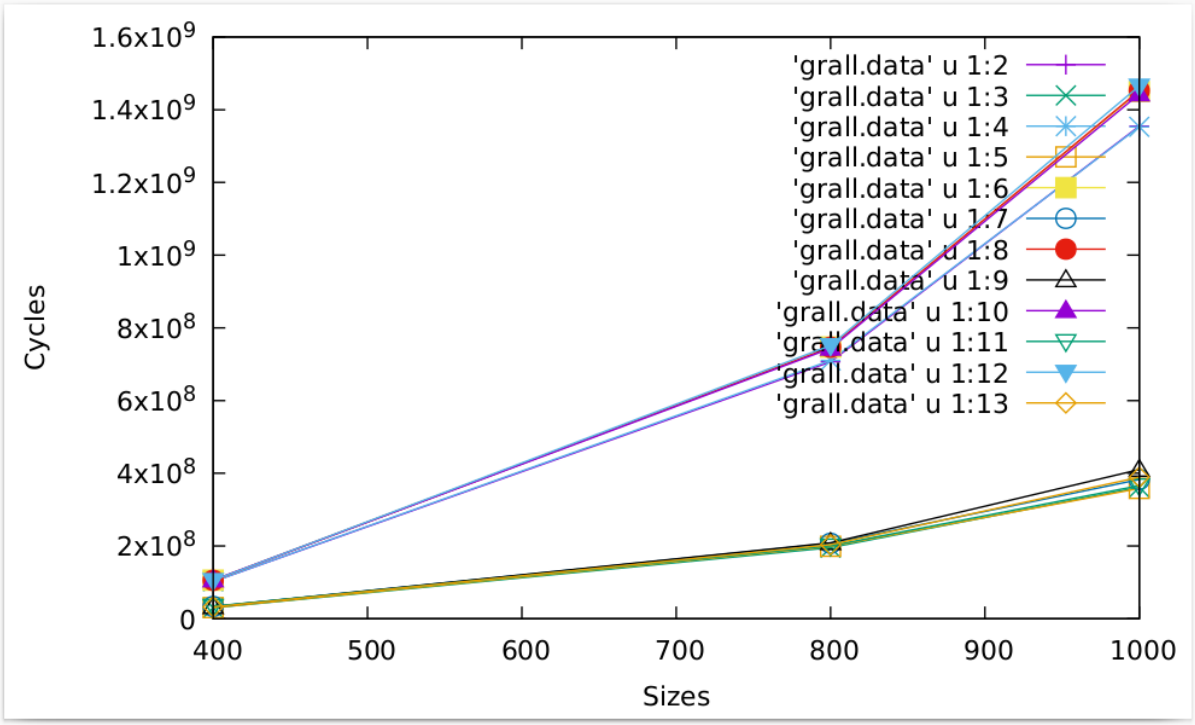
Taille du bloque uniformisée = 32 avec permutation r,d,c =>



Taille du bloque uniformisée = 64 avec permutation r,d,c =>



Taille du bloque uniformisée = 128 avec permutation r,d,c =>



Analysons les résutats :

Bloc 16 :

- Performances globalement moyennes.
- Quelques courbes montent au-delà de 1.5×10^9, avec dispersion visible.

Bloc 32 :

- Beaucoup de courbes très rapprochées, surtout vers la fin (taille 1000).
- Les cycles sont légèrement inférieurs à ceux du bloc 16.
- Amélioration claire par rapport à 16.

Bloc 64 :

- Les cycles sont les plus bas pour toutes les tailles (400 à 1000).
- Très peu de dispersion, pas de pics anormaux.
- => Excellente stabilité et performance globale.

Bloc 128 :

- Courbes un peu plus dispersées qu'avec le bloc 64.
- Quelques versions montent un peu plus haut à taille 1000.
- bon, mais légèrement moins optimal que 64.

Alors : La taille de bloc 64 est la meilleure dans ce cas :

- Cycles plus bas et réguliers pour presque toutes les versions,
- Pas de pics extrêmes et dispersion,

Conclusion :

L'uniformisation du blocage sur l'ensemble des scripts, combinée à l'application de la permutation optimale (r, d, c), a permis d'effectuer une évaluation plus fiable et plus homogène de l'impact de la taille des blocs sur les performances globales.

_La taille de bloc optimale dans ce TP est 64, lorsqu'elle est uniformisée dans tous les scripts mcblas3.cpp, avec la permutation (r, d, c) appliquée.

Elle offre:

-Des performances homogènes entre toutes les versions,

-Des temps d'exécution plus bas, en particulier pour les matrices de taille 800 et 1000,

-Une excellente gestion du cache et peu de dispersion entre les courbes.