

Rapport de TP : Optimisation des Performances à Travers la Vectorisation

Nom de l'étudiant : **Adel Bouzidi**

Exercice 1: On a :

```
adel@adel:~/Téléchargements/TD4-20250324/libmcblas003$ ll
total 56
drwxr-xr-x 12 adel adel 4096 mars 18 08:47 ./
drwxrwxr-x 3 adel adel 4096 mars 24 22:27 ../
-rw-r--r-- 1 adel adel 2021 mars 18 08:47 CMakeLists.txt
drwxr-xr-x 2 adel adel 4096 mars 11 15:13 mcblas001/
drwxr-xr-x 2 adel adel 4096 mars 11 15:13 mcblas002/
drwxr-xr-x 2 adel adel 4096 mars 11 15:13 mcblas003/
drwxr-xr-x 2 adel adel 4096 mars 11 15:13 mcblas004/
drwxr-xr-x 2 adel adel 4096 mars 11 15:13 mcblas005/
drwxr-xr-x 2 adel adel 4096 mars 11 15:13 mcblas006/
drwxr-xr-x 2 adel adel 4096 mars 11 15:13 mcblas007/
drwxr-xr-x 2 adel adel 4096 mars 11 22:45 mcblas008/
drwxr-xr-x 2 adel adel 4096 mars 13 11:24 mcblas009/
-rw-r--r-- 1 adel adel 579 mars 18 08:46 README.org
drwxr-xr-x 3 adel adel 4096 mars 13 15:10 tests/
```

Exercice 2

Utiliser le fichier README.org afin de compiler les sources

compiler les sources :

1. Créer le répertoire build :

```
mkdir build
cd build
```

2. Exécuter CMake pour générer les fichiers de build :

```
cmake ../
```

3. Compiler les sources :

```
make
```

```

adel@adel:~/Téléchargements/TD4-20250324/libmcblas003$ mkdir build
adel@adel:~/Téléchargements/TD4-20250324/libmcblas003$ cd build
adel@adel:~/Téléchargements/TD4-20250324/libmcblas003/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/Téléchargements/TD4-20250324/libmcblas003/build

```

make donne: (la prochaine fois make make clean, make RALL, make GALL pour réexécuter)

```

[ 90%] Generating data/tst00702_0400.data
[ 90%] Built target tst00702_0400
[ 90%] Generating data/tst00702_0512.data
[ 90%] Built target tst00702_0512
[ 91%] Generating data/tst00702_0800.data
[ 91%] Built target tst00702_0800
[ 92%] Generating data/tst00702_1000.data
[ 92%] Built target tst00702_1000
[ 94%] Generating data/tst00802_0400.data
[ 94%] Built target tst00802_0400
[ 95%] Generating data/tst00802_0512.data
[ 95%] Built target tst00802_0512
[ 95%] Generating data/tst00802_0800.data
[ 95%] Built target tst00802_0800
[ 96%] Generating data/tst00802_1000.data
[ 96%] Built target tst00802_1000
[ 97%] Generating data/tst00902_0400.data
[ 97%] Built target tst00902_0400
[ 98%] Generating data/tst00902_0512.data
[ 98%] Built target tst00902_0512
[100%] Generating data/tst00902_0800.data
[100%] Built target tst00902_0800
[100%] Built target O2_RUNS
[100%] Built target ALL_RUNS
[100%] Built target RALL
adel@adel:~/Téléchargements/TD4-20250324/libmcblas003/build$ ^C

```

Exercice 3 : Utilisation des outils d'analyse de performance

Q- 3.1.1. Compter avec les compteurs

Modifier les scripts CMakeList.txt afin d'utiliser l'outil Linux Perf avec la commande perf stat :

Etape à suivre :

1. gedit tests/runs/CMakeLists.txt
2. Décommenter/modifier cette ligne : (la ligne 20)


```

=====
Matrices after ssgemm:
[
  [ 1, 1, 1, 1, 1, 1, 1, 1]
  [ 1, 1, 1, 1, 1, 1, 1, 1]
  [ 1, 1, 1, 1, 1, 1, 1, 1]
  [ 1, 1, 1, 1, 1, 1, 1, 1]
  [ 1, 1, 1, 1, 1, 1, 1, 1]
  [ 1, 1, 1, 1, 1, 1, 1, 1]
  [ 1, 1, 1, 1, 1, 1, 1, 1]
  [ 1, 1, 1, 1, 1, 1, 1, 1]
]

Performance counter stats for './tst00102':

          3,64 msec task-clock                #    0,769 CPUs utilized
             0      context-switches         #    0,000 /sec
             0      cpu-migrations           #    0,000 /sec
            130     page-faults              #   35,748 K/sec
       5 786 659     cycles                   #    1,591 GHz
       5 415 414     instructions             #    0,94  insn per cycle
       945 253      branches                 #   259,928 M/sec
        37 904      branch-misses            #    4,01% of all branches

0,004731529 seconds time elapsed

0,002962000 seconds user
0,001974000 seconds sys

adel@adel:~/Téléchargements/TD4-20250324/libmcbblas003/build/tests$

```

Pour évaluer les performances des différentes versions de nos algorithmes, nous avons utilisé l'outil perf stat. Celui-ci a été intégré dans les scripts CMakeLists.txt du dossier tests/runs/. grâce à cela, l'exécution de make RALL permet automatiquement de générer les données de performance pour tous les exécutables.

il y a **10 exécutables** en total, car chaque test (tst00x) existe en deux versions : O2, O3

Comparaison O2 Vs O3 :

Base	Cycles_O2	Cycles_O3	Instr O2	Instr O3	IPC O2	IPC O3
001	5786659	4773933	5415414	5434467	0,94	1,14
002	4894626	4894626	5424950	5424950	1,11	1,11
007	4746891	7204510	5463469	5481960	1,15	0,76
008	5722125	4768270	5462199	5449026	0,95	1,14
009	5298643	5580349	5483573	5485349	1,03	0,98
Base	Branch Misses_O2	Branch Misses_O3	Cycles_Improvement	IPC_Improvement_%	BranchMiss_Reduction	
001	37364	19496	17,51038034	21,27655974	47,82143239	
002	19279	19279				
007	19958	19918	-51,77324313	-33,91030435	0,200420884	
008	37750	19740	16,6695939	20	47,70860927	
009	26757	33770	-5,315658789	-4,854368932	-26,20996375	

Les Données issues de la commande précédente :

Compteur	Valeur	Explication détaillée
task-clock	3.45 ms	Temps total passé par le CPU pour exécuter ce programme. Si ce chiffre est bas, cela signifie une exécution rapide (ici, c'est très court, ce qui est normal pour Size=8)
cycles	6.38 M	Nombre total de cycles d'horloge utilisés. C'est utile pour comparer entre exécutions avec différentes optimisations.
instructions	5.4 M	Nombre d'instructions réellement exécutées.
IPC	0.85	Instructions Per Cycle = instructions / cycles. C'est un indicateur clé d'efficacité d'utilisation du CPU.
branches	944 K	Nombre d'instructions de branchement (boucles, if, etc).
branch-misses	20.6 K (2.19%)	Nombre (et taux) de mauvaises prédictions de branchement. Un taux de 2% est bon.

1. Q- 3.1.1. Compter avec les compteurs (sur /mcblas001/mcblas3.cpp) :

Après avoir recompiler (cmake.. && make RALL) on fait :

```
perf stat ./tests/tst00102 1000
perf stat ./tests/tst00103 1000
```

```
adel@adel:~/Téléchargements/TD4-20250324/libmcblas003/build$ perf stat ./tests/tst00102 1000
1000
4747057696

Performance counter stats for './tests/tst00102 1000':

      2 263,26 msec task-clock           #    1,000 CPUs utilized
          6      context-switches       #    2,651 /sec
          1      cpu-migrations         #    0,442 /sec
          3 073      page-faults        #    1,358 K/sec
      8 158 592 390      cycles          #    3,605 GHz
      9 058 040 910      instructions    #    1,11 insn per cycle
      1 010 097 307      branches        # 446,301 M/sec
          1 073 704      branch-misses   #    0,11% of all branches

      2,263921027 seconds time elapsed

      2,254907000 seconds user
      0,008999000 seconds sys
```

```

adel@adel:~/Téléchargements/TD4-20250324/libmcbias003/build$ perf stat ./tests/tst00103 1000
1000
4456798350

Performance counter stats for './tests/tst00103 1000':

      2 124,29 msec task-clock                #    1,000 CPUs utilized
           17      context-switches          #    8,003 /sec
           2      cpu-migrations              #    0,941 /sec
          3 075      page-faults              #    1,448 K/sec
      8 062 895 908      cycles                #    3,796 GHz
      9 057 613 395      instructions          #    1,12 insn per cycle
      1 010 035 182      branches              # 475,469 M/sec
          1 079 385      branch-misses        #    0,11% of all branches

      2,125226624 seconds time elapsed

      2,115828000 seconds user
      0,008999000 seconds sys

```

Résumé des résultats obtenus avec `perf stat` :

1. Temps d'exécution (task-clock)

- O2 : 2263,26 ms
- O3 : 2124,29 ms

Analyse : Le temps d'exécution est réduit de 6% avec -O3.

2. Cycles processeur

- O2 : 8 158 592 390
- O3 : 8 062 895 908

Analyse : Légère réduction du nombre total de cycles avec -O3 (1,2%).

3. Instructions exécutées

- O2 : 9 058 040 910
- O3 : 9 057 613 395

Analyse : Pratiquement identique.

4. Instructions par cycle (IPC)

- O2 : 1,11
- O3 : 1,12

Analyse : Très légère amélioration avec -O3.

5. Branches et erreurs de branchement

Branches :

- O2 : 1 010 097 307
- O3 : 1 010 035 182

Ratés :

- O2 : 1 073 704 (0,11%)
- O3 : 1 079 385 (0,11%)

Analyse : Équivalent pour les deux versions.

6. Utilisation CPU

— CPU utilisé : 1,000 pour les deux versions

Temps utilisateur + système :

— O2 : 2,263 s

— O3 : 2,125 s

Alors :

La version -O3 montre une exécution plus rapide, avec un nombre de cycles légèrement inférieur et un IPC un peu meilleur, bien que le nombre d'instructions reste presque identique.

Le gain reste modéré, ce qui est typique des programmes déjà bien structurés et qui ne présentent pas de goulet d'étranglement majeur facilement optimisable.

Q-3.1.2 : Utiliser perf record dans les scripts CMake:

Commentaire et décommentaire dans les Cmake Principale :

Commentons : #set(PERFTOOL "perf stat")

Décommentons : set(PERFTOOL "perf record -g -F 999")

Ensuite, relançons les commandes :

Make clean

Cmake ..

Make RALL

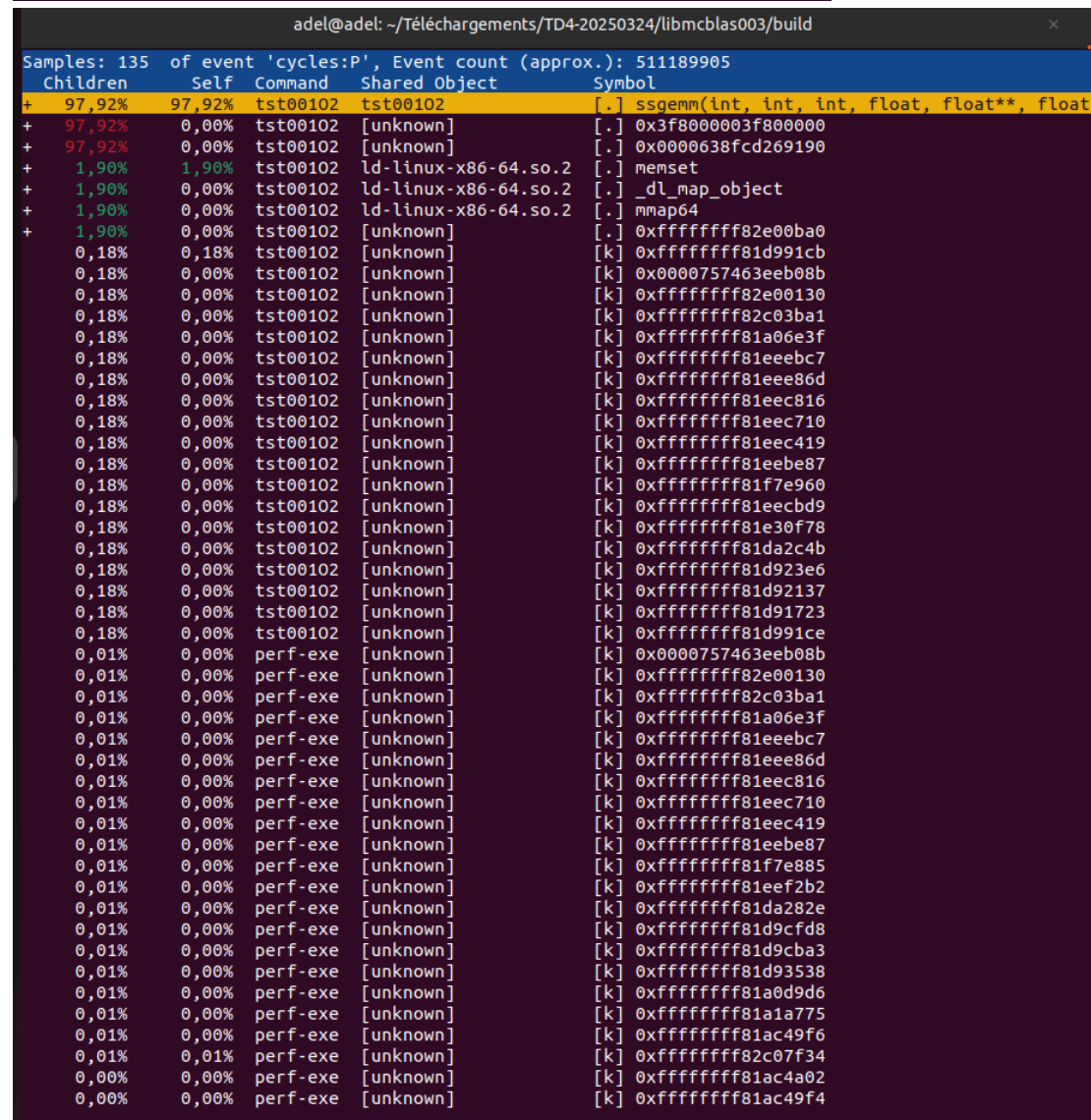
Puis, Vérifions la présence des fichiers perf.data dans tests/runs/data/.

```
adel@adel:~/Téléchargements/TD4-20250324/libmcblas003/build$ cd tests/runs/data/
adel@adel:~/Téléchargements/TD4-20250324/libmcblas003/build/tests/runs/data$ ls -lh _perf_*
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00102_0400.data
-rw-rw-r-- 1 adel adel 977 mars 24 23:47 _perf_tst00102_0512.data
-rw-rw-r-- 1 adel adel 981 mars 24 23:47 _perf_tst00102_0800.data
-rw-rw-r-- 1 adel adel 985 mars 24 23:47 _perf_tst00102_1000.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00103_0400.data
-rw-rw-r-- 1 adel adel 977 mars 24 23:47 _perf_tst00103_0512.data
-rw-rw-r-- 1 adel adel 981 mars 24 23:47 _perf_tst00103_0800.data
-rw-rw-r-- 1 adel adel 985 mars 24 23:47 _perf_tst00103_1000.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00202_0400.data
-rw-rw-r-- 1 adel adel 975 mars 24 23:47 _perf_tst00202_0512.data
-rw-rw-r-- 1 adel adel 979 mars 24 23:47 _perf_tst00202_0800.data
-rw-rw-r-- 1 adel adel 985 mars 24 23:47 _perf_tst00202_1000.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00203_0400.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00203_0512.data
-rw-rw-r-- 1 adel adel 977 mars 24 23:47 _perf_tst00203_0800.data
-rw-rw-r-- 1 adel adel 979 mars 24 23:47 _perf_tst00203_1000.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00702_0400.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00702_0512.data
-rw-rw-r-- 1 adel adel 977 mars 24 23:47 _perf_tst00702_0800.data
-rw-rw-r-- 1 adel adel 979 mars 24 23:47 _perf_tst00702_1000.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00703_0400.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00703_0512.data
-rw-rw-r-- 1 adel adel 977 mars 24 23:47 _perf_tst00703_0800.data
-rw-rw-r-- 1 adel adel 979 mars 24 23:47 _perf_tst00703_1000.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00802_0400.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00802_0512.data
-rw-rw-r-- 1 adel adel 977 mars 24 23:47 _perf_tst00802_0800.data
-rw-rw-r-- 1 adel adel 977 mars 24 23:47 _perf_tst00802_1000.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00803_0400.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00803_0512.data
-rw-rw-r-- 1 adel adel 977 mars 24 23:47 _perf_tst00803_0800.data
-rw-rw-r-- 1 adel adel 979 mars 24 23:47 _perf_tst00803_1000.data
-rw-rw-r-- 1 adel adel 973 mars 24 23:47 _perf_tst00902_0400.data
```


Ensuite : On peut vérifier un de ces fichiers avec :

```
data$ perf report -i _perf_tst00102_0400.data
```

=>



Samples: 135 of event 'cycles:P', Event count (approx.): 511189905				
Children	Self	Command	Shared Object	Symbol
+ 97,92%	97,92%	tst00102	tst00102	[.] ssgemv(int, int, int, float, float**, float*
+ 97,92%	0,00%	tst00102	[unknown]	[.] 0x3f8000003f800000
+ 97,92%	0,00%	tst00102	[unknown]	[.] 0x0000638fcd269190
+ 1,90%	1,90%	tst00102	ld-linux-x86-64.so.2	[.] memset
+ 1,90%	0,00%	tst00102	ld-linux-x86-64.so.2	[.] _dl_map_object
+ 1,90%	0,00%	tst00102	ld-linux-x86-64.so.2	[.] mmap64
+ 1,90%	0,00%	tst00102	[unknown]	[.] 0xffffffff82e00ba0
0,18%	0,18%	tst00102	[unknown]	[k] 0xffffffff81d991cb
0,18%	0,00%	tst00102	[unknown]	[k] 0x0000757463eeb08b
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff82e00130
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff82c03ba1
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81a06e3f
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81eeebc7
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81eee86d
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81eec816
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81eec710
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81eec419
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81eebe87
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81f7e960
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81eecbd9
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81e30f78
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81da2c4b
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81d923e6
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81d92137
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81d91723
0,18%	0,00%	tst00102	[unknown]	[k] 0xffffffff81d991ce
0,01%	0,00%	perf-exe	[unknown]	[k] 0x0000757463eeb08b
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff82e00130
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff82c03ba1
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81a06e3f
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81eeebc7
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81eee86d
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81eec816
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81eec710
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81eec419
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81eebe87
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81f7e885
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81eef2b2
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81da282e
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81d9cfd8
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81d9c3a3
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81d93538
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81a0d9d6
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81a1a775
0,01%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81ac49f6
0,01%	0,01%	perf-exe	[unknown]	[k] 0xffffffff82c07f34
0,00%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81ac4a02
0,00%	0,00%	perf-exe	[unknown]	[k] 0xffffffff81ac49f4

Cela lance un rapport interactif basé sur le fichier de données de profiling généré précédemment via perf record. Ce fichier enregistre les cycles processeur consommés pendant l'exécution de tst00102 avec paramètre de taille 400:

Analyse de la sortie perf report

La vue montre la répartition des cycles processeur consommés par les différentes fonctions exécutées.

Ligne principale :

```
+ 97.92% tst00102 [.] ssgemm(int, int, int, float, float**, float**, ...)
```

Cela signifie que 97.92% du temps CPU a été passé dans la fonction `ssgemm`.

Détails :

- `ssgemm` = Single-precision general matrix-matrix multiplication
- Donc, c'est la multiplication de matrices qui prend presque tout le temps d'exécution. Ce comportement est attendu dans un test de benchmark.

Remarque : pour la question (Q- 3.1.3. Visualiser les données) j'ai visualiser les sortie sous format texte en parallèle avec les réponse précédente de l'exercice

EXO 4:

Introduction :

Application de la Vectorisation et Analyse des Performances

La **vectorisation** est une technique d'optimisation permettant d'exécuter plusieurs opérations en parallèle sur des données multiples en utilisant des instructions SIMD. Elle permet ainsi de réduire le nombre d'instructions nécessaires et d'améliorer les performances des programmes, notamment pour des opérations lourdes comme la multiplication matricielle. Cette technique est particulièrement utile dans les contextes de calcul scientifique et de traitement de grandes quantités de données.

Il existe plusieurs façons d'appliquer la vectorisation:

1. Vectorisation explicite dans le code :

La vectorisation dans le code peut être réalisée de deux manières principales : par **pragmas** ou par **intrinsic SIMD**. La **vectorisation par pragmas** consiste à utiliser des directives spécifiques dans le code source, telles que `#pragma omp simd`, qui permettent au compilateur de générer du code vectorisé automatiquement. Ces pragmas indiquent au compilateur de paralléliser certaines boucles ou opérations, souvent sans modification manuelle du code. D'autre part, les **intrinsic SIMD** sont des fonctions fournies par le compilateur, telles que celles de la bibliothèque `<immintrin.h>`, qui permettent d'exploiter directement les capacités SIMD des processeurs. Ces intrinsics permettent de charger, manipuler et stocker plusieurs valeurs en parallèle dans des registres vectoriels spécifiques (comme les registres AVX2), offrant un contrôle plus fin sur l'optimisation des performances. Contrairement aux pragmas, l'utilisation des intrinsic SIMD nécessite de modifier explicitement le code, mais offre généralement de meilleures performances et une plus grande flexibilité.

2. Vectorisation au niveau de la compilation :

En activant des options de compilation dans le fichier CMakeLists.txt, comme -O3 -march=native -ftree-vectorize, permettant au compilateur d'appliquer la vectorisation sur l'ensemble du programme. Cela permet de tirer parti des capacités du compilateur pour optimiser automatiquement le code pour les architectures spécifiques.

Plan:

Q- 4.1 Modification du "Blocking"

- 1.présentation des codes**
- 2.vérifier les performance avec un code de blocking simple.**
- 3. Comparaison entre les deux versions (code sans et avec vectorisation)**

Q- 4.2 Variation des paramètres

- 4.Comparaison entre les niveaux d'optimisation -O2 et -O3** lors de la compilation du code vectorisé
- 5.Variation de la taille des blocs** : Finalement, je vais tester différentes tailles de blocs pour trouver la meilleure taille en termes de performances. La taille des blocs influence l'efficacité de la vectorisation et des caches du processeur, et il est important de l'optimiser pour chaque configuration.

1. Présentation des codes :

Dans ce projet, deux versions du code de multiplication de matrices seront utilisées : mcblas002 contient l'implémentation avec blocking uniquement, sans vectorisation. mcblas001 contient une version de mcblas002/mcblas3.cpp optimisée par vectorisation SIMD, tout en conservant le blocking. (Remarque : à la base mcblas2 contient un code simple, j'ai collé le code de mcblas004/mcblas3.cpp dans lequel, ensuite j'ai appliqué la vectorisation sur ce dernier code de blocking , ensuite je l'ai collé dans mcblas3.cpp de mcblas001)

mcblas002/mcblas3.cpp :

```

1 #include <mcblas3.hpp>
2 #include <iostream>
3 /*
4  * Routines of cblas level 3
5  */
6
7 void ssgemm(
8     int m, int n, int k,
9     float alpha,
10    float ** A,
11    float ** B,
12    float beta,
13    float ** C)
14 {
15     // std::cout << "*****" << std::endl;
16     // std::cout << "    WARNING: Not Yet Implemented..." << std::endl;
17     // std::cout << "*****" << std::endl;
18
19     int r, c, d;
20     int rr, cc, dd;
21     const int Tr = 16;
22     const int Tc = 64;
23     const int Td = 32;
24
25     for (dd = 0; dd < m; dd += Td)
26     {
27         for (cc = 0; cc < m; cc += Tc)
28         {
29             for (rr = 0; rr < m; rr += Tr)
30             {
31                 for (c = cc; c < std::min(cc+Tc, n); ++c)
32                 {
33                     for (r = rr; r < std::min(rr+Tr, m); ++r)
34                     {
35                         float t=0.0;
36                         for (d = dd; d < std::min(dd+Td, m); ++d)
37                         {
38                             t = t + alpha * A[r][d] * B[d][c];
39                         }
40                         C[r][c] = C[r][c] * beta + t;
41                     }
42                 }
43             }
44         }
45     }
46 }
47 }

```

mcblas001/mcblas3.cpp :

```

1 #include <immintrin.h>
2 #include <iostream>
3 #include <algorithm>
4
5 void ssgemm(
6     int m, int n, int k,
7     float alpha,
8     float ** A,
9     float ** B,
10    float beta,
11    float ** C)
12 {
13     int r, c, d;
14     int rr, cc, dd;
15     const int Tr = 64;
16     const int Tc = 64;
17     const int Td = 64;
18
19     for (dd = 0; dd < k; dd += Td)
20     {
21         for (cc = 0; cc < n; cc += Tc)
22         {
23             for (rr = 0; rr < m; rr += Tr)

```

```

22 {
23     for (rr = 0; rr < m; rr += Tr)
24     {
25         for (c = cc; c < std::min(cc + Tc, n); ++c)
26         {
27             for (r = rr; r < std::min(rr + Tr, m); ++r)
28             {
29                 __m256 t8 = _mm256_setzero_ps(); // accumulateur SIMD
30                 int d_vec = dd;
31
32                 // partie vectorisée
33                 for (; d_vec <= std::min(dd + Td, k) - 8; d_vec += 8)
34                 {
35                     __m256 a8 = _mm256_loadu_ps(&A[r][d_vec]); // A[r][d_vec..d_vec+7]
36                     __m256 b8 = _mm256_set_ps(
37                         B[d_vec + 7][c], B[d_vec + 6][c],
38                         B[d_vec + 5][c], B[d_vec + 4][c],
39                         B[d_vec + 3][c], B[d_vec + 2][c],
40                         B[d_vec + 1][c], B[d_vec + 0][c]); // colonne B en scalaires
41                     __m256 mul = _mm256_mul_ps(a8, b8);
42                     t8 = _mm256_add_ps(t8, mul);
43                 }
44
45                 // réduction SIMD vers scalaire
46                 float tmp[8];
47                 _mm256_storeu_ps(tmp, t8);
48                 float t = tmp[0] + tmp[1] + tmp[2] + tmp[3] +
49                     tmp[4] + tmp[5] + tmp[6] + tmp[7];
50
51                 // reste scalaire (si Td pas multiple de 8)
52                 for (; d_vec < std::min(dd + Td, k); ++d_vec)
53                 {
54                     t += A[r][d_vec] * B[d_vec][c];
55                 }
56
57                 // mise à jour finale de C
58                 C[r][c] = C[r][c] * beta + alpha * t;
59             }
60         }
61     }
62 }
63 }
64 }

```

2.vérifier les performance avec un code de blocking simple (mcblas001/mcblas3.cpp)

Donc, deux versions du code ont été comparées :mcblas002 (exécutable tst00203) : version avec blocking uniquement, sans vectorisation.

```

adel@adel:~/Téléchargements/TD4-20250324/TD4_LAST/build$ ./tests/tst00203 1000
1000 3804709068
adel@adel:~/Téléchargements/TD4-20250324/TD4_LAST/build$ ./tests/tst00203 2048
2048 20287439840
adel@adel:~/Téléchargements/TD4-20250324/TD4_LAST/build$ ./tests/tst00103 1000
1000 2358812834
adel@adel:~/Téléchargements/TD4-20250324/TD4_LAST/build$ ./tests/tst00103 2048
2048 15018365816
adel@adel:~/Téléchargements/TD4-20250324/TD4_LAST/build$

```

```
adel@adel:~/Téléchargements/TD4-20250324/TD4_LAST/build$ perf stat ./tests/tst00203 1000
1000
3471170714
```

Performance counter stats for './tests/tst00203 1000':

1 668,47 msec	task-clock	#	0,999 CPUs utilized
3	context-switches	#	1,798 /sec
0	cpu-migrations	#	0,000 /sec
3 075	page-faults	#	1,843 K/sec
3 055 010 922	cycles	#	1,831 GHz
6 537 455 971	instructions	#	2,14 insn per cycle
1 074 558 801	branches	#	644,037 M/sec
2 170 003	branch-misses	#	0,20% of all branches

1,669321227 seconds time elapsed

1,658478000 seconds user

0,010996000 seconds sys

```
adel@adel:~/Téléchargements/TD4-20250324/TD4_LAST/build$ perf stat ./tests/tst00203 2048
2048
17790856358
```

Performance counter stats for './tests/tst00203 2048':

8 517,96 msec	task-clock	#	1,000 CPUs utilized
54	context-switches	#	6,340 /sec
3	cpu-migrations	#	0,352 /sec
12 450	page-faults	#	1,462 K/sec
27 896 131 790	cycles	#	3,275 GHz
55 843 483 404	instructions	#	2,00 insn per cycle
9 178 355 720	branches	#	1,078 G/sec
17 543 147	branch-misses	#	0,19% of all branches

8,519448838 seconds time elapsed

8,466137000 seconds user

0,052994000 seconds sys

```
adel@adel:~/Téléchargements/TD4-20250324/TD4_LAST/build$ perf stat ./tests/tst00103 1000
1000
2296157418
```

Performance counter stats for './tests/tst00103 1000':

1 104,24 msec	task-clock	#	0,999 CPUs utilized
4	context-switches	#	3,622 /sec
1	cpu-migrations	#	0,906 /sec
3 073	page-faults	#	2,783 K/sec
1 985 557 061	cycles	#	1,798 GHz
3 718 272 188	instructions	#	1,87 insn per cycle
181 235 920	branches	#	164,127 M/sec
310 461	branch-misses	#	0,17% of all branches

1,105169276 seconds time elapsed

1,093226000 seconds user

0,011991000 seconds sys

```
adel@adel:~/Téléchargements/TD4-20250324/TD4_LAST/build$ perf stat ./tests/tst00103 2048
2048
12749586194

Performance counter stats for './tests/tst00103 2048':

        6 091,96 msec task-clock                #    1,000 CPUs utilized
              55      context-switches         #    9,028 /sec
               3      cpu-migrations            #    0,492 /sec
              12 449      page-faults           #    2,044 K/sec
    19 850 444 307      cycles                   #    3,258 GHz
    31 604 309 445      instructions             #    1,59 insn per cycle
     1 510 640 731      branches                 # 247,973 M/sec
       2 304 998      branch-misses             #    0,15% of all branches

        6,094222522 seconds time elapsed

        6,061479000 seconds user
        0,031002000 seconds sys
```

3. Comparaison entre les deux versions (code sans et avec vectorisation)

Les deux codes ont été exécutés avec des tailles de matrices 1000 et 2048, et les résultats suivants ont été obtenus :

La version vectorisée permet donc un gain de performance significatif.

La version vectorisée permet donc un gain de performance significatif :

- Pour 1000x1000, le temps d'exécution est réduit d'environ 38%.
- Pour 2048x2048, le gain est d'environ 25%.

Ces gains sont confirmés par les mesures détaillées avec perf :

Exemple avec taille 1000 :

- Sans vectorisation (tst00203) :
 - Temps : 1.66 s
 - Instructions par cycle (IPC) : 2.14
- Avec vectorisation (tst00103) :
 - Temps : 1.10 s
 - IPC : 1.87

Q-4.2 Variation des paramètres:

Dans cette section, nous cherchons à affiner les performances du code vectorisé en étudiant l'impact de différents paramètres d'optimisation

Nous procédons en deux étapes :

Comparaison entre les niveaux d'optimisation -O2 et -O3 lors de la compilation du code vectorisé. Cette première analyse permet d'évaluer dans quelle mesure le niveau -O3 améliore les performances par rapport à -O2. **Une fois le meilleur niveau de compilation identifié**, nous allons varier les tailles des blocs (Tr, Tc, Td) dans le code vectorisé compilé avec ce niveau. L'objectif est de trouver les valeurs de

blocking qui maximisent l'efficacité du calcul, en équilibrant la charge mémoire et la vectorisation.

On a alors:

```
adel@adel:~/Téléchargements/TD4-20250324/TD4_LAST/build$ perf stat ./tests/tst00102 4096
4096
171845091788

Performance counter stats for './tests/tst00102 4096':

      81 554,91 msec task-clock                #    0,999 CPUs utilized
         2 328      context-switches          #    28,545 /sec
         219      cpu-migrations              #     2,685 /sec
        49 349      page-faults               #   605,102 /sec
    183 167 289 880 cycles                    #     2,246 GHz
    252 718 155 407 instructions              #     1,38 insn per cycle
     12 048 778 433 branches                  #   147,738 M/sec
        22 894 765 branch-misses              #    0,19% of all branches

      81,669144438 seconds time elapsed

      81,366936000 seconds user
       0,175937000 seconds sys

adel@adel:~/Téléchargements/TD4-20250324/TD4_LAST/build$ perf stat ./tests/tst00103 4096
4096
107470538814

Performance counter stats for './tests/tst00103 4096':

      51 091,74 msec task-clock                #    1,000 CPUs utilized
         147      context-switches          #     2,877 /sec
          4      cpu-migrations              #     0,078 /sec
        49 349      page-faults               #   965,890 /sec
    160 240 281 465 cycles                    #     3,136 GHz
    252 252 985 493 instructions              #     1,57 insn per cycle
     11 976 762 630 branches                  #   234,417 M/sec
        18 135 623 branch-misses              #    0,15% of all branches

      51,095972769 seconds time elapsed

      50,991118000 seconds user
       0,100992000 seconds sys
```

Comparaison des performances – tst00102 VS tst00103 :

Critère	tst00102	tst00103	Observation
Temps d'exécution	81,67 s	51,10 s	-03 est 37\% plus rapide
Cycles processeur	183,2 milliards	160,2 milliards	Moins de cycles pour -03
Instructions	252,7 milliards	252,3 milliards	Pratiquement équivalent
Instructions / cyc	1,38	1,57	Meilleure efficacité avec -03
Branches	12,05 milliards	11,98 milliards	Similaire
Branch-misses	0,19%	0,15%	Léger avantage pour -03
Fréquence CPU m	2,25 GHz	3,13 GHz	Le CPU booste mieux sous -03

- Le passage de -02 à -03 offre **un** gain massif de temps (30 s).
- Le nombre total d'instructions reste identique, mais la version -03 est bien plus efficace, avec un IPC de 1,57 vs 1,38.

- Le processeur tourne aussi à une fréquence moyenne plus élevée avec -O3 (grâce au code plus "dense" et optimisé).

-Varier les tailles des blocs (Tr, Tc, Td) dans le code vectorisé compilé avec le niveau O3:

Après avoir confirmé l'efficacité de la compilation avec l'option -O3, nous étudions maintenant l'influence des tailles de blocs (Tr, Tc, Td) sur les performances du code vectorisé.

Trois configurations de tailles de blocs ont été testées :

Bloc 64×64×64 / Bloc 32×32×32 / Bloc 16×16×16

Les résultats de perf stat :

Bloc (Tr/Tc/Td)	Temps (s)	Instructions	IPC	Branch-misses (%)
64 / 64 / 64	51,1	252,3 G	1,57	0,15%
32 / 32 / 32	63,23	297,9 G	1,65	0,46%
16 / 16 / 16	72,24	390,3 G	1,99	0,09%

Performance globale : La configuration 64×64×64 offre les meilleures performances, avec un temps d'exécution de 51,10 secondes, contre 63,32 s pour 32×32×32 et 72,25 s pour 16×16×16.

Instructions exécutées : Plus les blocs sont petits, plus le nombre total d'instructions augmente significativement (de 252 milliards à 390 milliards).

Efficacité CPU (IPC) : L'IPC (instructions par cycle) augmente avec des blocs plus petits, car les boucles internes deviennent plus simples et le CPU peut les exécuter plus rapidement.

Branches et erreurs de branchement :

Les petits blocs génèrent beaucoup plus de branches, mais avec un taux d'échec très bas. Le meilleur taux est atteint avec 16×16×16 (0,09 %), mais cela n'améliore pas les performances globales.
