TD1

```
pandoc -s --toc README.md --css=./github-pandoc.css -o README.html
```

lscpu

lscpu donne des infos utiles sur le processeur : nb core, taille de cache :

```
cedrick@cedrick:~$ lscpu
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          39 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   12
  On-line CPU(s) list:    0-11
Vendor ID:                GenuineIntel
  Model name:             Intel(R) Core(TM) i7-9850H CPU @ 2.60GHz
    CPU family:           6
    Model:                158
    Thread(s) per core:   2
    Core(s) per socket:   6
    Socket(s):            1
    Stepping:             13
    CPU max MHz:          4600,0000
    CPU min MHz:          800,0000
    BogoMIPS:             5199.98
    Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge m
                          ca cmov pat pse36 clflush dts acpi mmx fxsr sse
sse2 s
                          s ht tm pbe syscall nx pdpe1gb rdtscp lm
constant_tsc
                          art arch_perfmon pebs bts rep_good nopl xtopology
nons
                          top_tsc cpuid aperfmperf pni pclmulqdq dtes64
monitor
                          ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr
pdcm p
                          cid sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_tim
                          er aes xsave avx f16c rdrand lahf_lm abm
3dnowprefetch
                           cpuid_fault epb ssbd ibrs ibpb stibp
ibrs_enhanced tp
                          r_shadow flexpriority ept vpid ept_ad fsgsbase
tsc_adj
                          ust sgx bmi1 avx2 smep bmi2 erms invpcid mpx
rdseed ad
                          x smap clflushopt intel_pt xsaveopt xsavec
xgetbv1 xsa
```

```
                           ves dtherm ida arat pln pts hwp hwp_notify
hwp_act_win
                           dow hwp_epp vnmi sgx_lc md_clear flush_l1d
arch_capabi
                           lities
Virtualization features:
  Virtualization:          VT-x
Caches (sum of all):
  L1d:                     192 KiB (6 instances)
  L1i:                     192 KiB (6 instances)
  L2:                      1,5 MiB (6 instances)
  L3:                      12 MiB (1 instance)
NUMA:
  NUMA node(s):            1
  NUMA node0 CPU(s):       0-11
Vulnerabilities:
  Gather data sampling:    Mitigation; Microcode
  Itlb multihit:           KVM: Mitigation: VMX disabled
  L1tf:                    Not affected
  Mds:                     Not affected
  Meltdown:                Not affected
  Mmio stale data:         Mitigation; Clear CPU buffers; SMT vulnerable
  Reg file data sampling: Not affected
  Retbleed:                Mitigation; Enhanced IBRS
  Spec rstack overflow:    Not affected
  Spec store bypass:       Mitigation; Speculative Store Bypass disabled via
prct
                           l
  Spectre v1:              Mitigation; usercopy/swapgs barriers and __user
pointe
                           r sanitization
  Spectre v2:              Mitigation; Enhanced / Automatic IBRS; IBPB
conditiona
                           l; RSB filling; PBRSB-eIBRS SW sequence; BHI SW
loop,
                           KVM SW loop
  Srbds:                   Mitigation; Microcode
  Tsx async abort:         Mitigation; TSX disabled
```

Produit matrice-matrice

Effet de la taille de la matrice

n	MFlops
1024 (origine)	304.761
1023	282.849
1025	283.735

Expliquons les résultats obtenus

```
On remarque que le temps de calcul est meilleur pour des matrices de
dimension 1024, par rapport à des matrices de dimension 1023 ou 1025.
On constate que l'exécution du produit matriciel s'effectue plus rapidement
pour une taille de 1024 par rapport aux autres dimensions testées.
Celà peut s'expliquer sur 2 points:

- Alignement mémoire optimal: La dimension 1024 est une puissance de 2, ce
qui permet aux données de la matrice d'être correctement alignées en
mémoire. Cet alignement favorise une meilleure exploitation des blocs de
mémoire cache (L1, L2, L3), réduisant ainsi les accès à la mémoire
principale qui sont plus lents.
- Efficacité de la hiérarchie cache : Un alignement optimal signifie que le
cache est utilisé de manière plus efficace, avec un taux de cache hit
élevé. Cela réduit considérablement le temps d'accès aux données pendant
l'exécution des boucles imbriquées du produit matriciel.

La dimension 1024 permet donc à ces optimisations de s'appliquer de manière
optimale, contrairement aux dimensions 1023 ou 1025 qui ne correspondent
pas aussi bien aux structures de la mémoire.

En somme, Le résultat observé s'explique principalement par l'alignement
mémoire et l'utilisation efficace de la hiérarchie cache du processeur. En
effet, 1024 étant une puissance de 2, la taille des matrices permet une
organisation en mémoire qui correspond mieux aux blocs utilisés par le
cache. Cela minimise les temps d'accès et améliore la gestion de la mémoire
cache, contrairement aux dimensions 1023 ou 1025 où l'alignement n'est pas
optimal, induisant des accès plus lents et une gestion moins efficace du
cache.
```

Permutation des boucles

Expliquons comment est compilé le code (ligne de make ou de gcc) : on aura besoin de savoir l'optim, les paramètres, etc. Par exemple :

`make TestProduct.exe && ./TestProduct.exe 1024`

Pour compiler et exécuter le programme, on utilise :

```
make TestProductMatrix.exe && ./TestProductMatrix.exe 1024
```

1. **Compilation (`make TestProductMatrix.exe`)**
    - Utilise `g++` avec les options :
        - `-O2` : Optimisation standard (bon équilibre entre vitesse et stabilité).
        - `-O3` : Optimisation agressive (fusion de boucles, vectorisation...).
        - `-march=native` : Exploite les instructions spécifiques au processeur.
        - `-Wall` : Active les avertissements pour détecter d'éventuelles erreurs.
2. **Exécution (`./TestProductMatrix.exe 1024`)**

- Lance le programme avec une matrice 1024×1024 pour mesurer les performances.

Si besoin de débogage :

```
make DEBUG=yes TestProductMatrix.exe && ./TestProductMatrix.exe 1024
```

(utilise `-O0` pour désactiver les optimisations et `-g` pour le débogage).

ordre	time	MFlops	MFlops(n=2048)
i,j,k	2.12581	1010.2	486.144
j,i,k	2.16959	989.809	402.17
i,k,j(origine)	7.04645	304.761	217.261
k,i,j	7.33375	292.822	146.79
j,k,i	0.814834	2635.49	2350.9
k,j,i	0.609918	3520.94	2793.11

Discutons les résultats :

Dans le cours, il est expliqué que l'accès à la mémoire est un facteur clé dans l'optimisation du calcul parallèle.

- Pour un accès mémoire efficace, il est préférable d'accéder aux données stockées en mémoire de manière séquentielle.
- L'accès aux données en cache est beaucoup plus rapide que l'accès à la RAM. Le **cache L1 et L2** doivent être utilisés au maximum.

L'algorithme naïf (i, j, k) accède aux éléments de `B(k, j)` de façon non contiguë, ce qui cause un mauvais cache hit rate et entraîne des accès mémoire coûteux.

Ainsi, l'ordre de boucle optimale est `(k, j, i)` car :

1. On a un accès mémoire optimisé :
    - `A(i, k)` est parcouru colonne par colonne, ce qui est efficace car A est stockée en mémoire ligne par ligne.
    - `B(k, j)` est accédé de manière séquentielle sur les colonnes, ce qui améliore la localité spatiale et réduit les accès mémoire coûteux.
    - `C(i, j)` est mis à jour de manière contiguë dans la mémoire.
2. On se sert du Blocage pour exploiter le cache :
    - La boucle est découpée en sous-blocs de taille `szBlock`, ce qui permet de réduire la quantité de données chargées en cache simultanément.
    - Ce blocage des calculs améliore le taux de réutilisation des données sans éviction prématurée des caches L1 et L2.

3. Ce type d'algorithme se prête bien à une parallélisation avec OpenMP, car chaque thread peut traiter un bloc indépendant, minimisant ainsi les conflits de mémoire.

Ainsi, on peut faire une Comparaison de nos résultats pour différents types de permutation

Ordre des boucles	**Accès à A(i,k)**	**Accès à B(k,j)**	**Accès à C(i,j)**	**Performance (basée sur MFlops)**
`i, j, k` (naïf)	Séquentiel	Non séquentiel	Séquentiel	Mauvais (1010.2 MFlops)
`j, i, k`	Non séquentiel	Non séquentiel	Séquentiel	Mauvais (989.809 MFlops)
`i, k, j`	Séquentiel	Séquentiel	Non séquentiel	Médiocre (304.761 MFlops)
`k, i, j`	Séquentiel	Non séquentiel	Séquentiel	Médiocre (292.822 MFlops)
`j, k, i`	Non séquentiel	Séquentiel	Séquentiel	Très bon (2635.49 MFlops)
`k, j, i` (optimal)	**Séquentiel**	**Séquentiel**	**Séquentiel**	**Excellent (3520.94 MFlops)**

OMP sur la meilleure boucle

`make TestProduct.exe && OMP_NUM_THREADS=8 ./TestProduct.exe 1024`

```
//Code considéré: ProdMatMat.cpp

#include <algorithm>
#include <cassert>
#include <iostream>
#include <thread>
#if defined(_OPENMP)
#include <omp.h>
#endif
#include "ProdMatMat.hpp"

namespace {
void prodSubBlocks(int iRowBlkA, int iColBlkB, int iColBlkA, int szBlock,
                   const Matrix& A, const Matrix& B, Matrix& C) {
    // Ordre k -> j -> i pour optimiser l'accès mémoire
    #pragma omp parallel
    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);
```

```
}
const int szBlock = 32; // Taille de bloc adaptée au cache
}  // namespace

Matrix operator*(const Matrix& A, const Matrix& B) {
  Matrix C(A.nbRows, B.nbCols, 0.0);

  // Parallélisation sur les blocs de lignes de A (iRowBlkA)
  #pragma omp parallel for
  for (int iRowBlkA = 0; iRowBlkA < A.nbRows; iRowBlkA += szBlock) {
    // Parcours des blocs colonnes de B (j)
    for (int iColBlkB = 0; iColBlkB < B.nbCols; iColBlkB += szBlock) {
      // Parcours des blocs colonnes de A (k)
      for (int iColBlkA = 0; iColBlkA < A.nbCols; iColBlkA += szBlock) {
        prodSubBlocks(iRowBlkA, iColBlkB, iColBlkA, szBlock, A, B, C);
      }
    }
  }

  return C;
}
```

OMP_NUM	MFlops	MFlops(n=2048)	MFlops(n=512)	MFlops(n=4096)
1	3388.86	2765.99	3138.42	2357.51
2	3385.66	4968.66	5838.37	4418.62
3	3370.12	7271.2	7212.75	6284.18
4	3354.99	9314.27	10447.2	8168.77
5	3459.52	12062.8	10356.5	9725.05
6	3207.41	13177.6	9298.74	11452.7
7	3468.93	11470.7	9421.53	10137.1
8	3396.06	12610.9	11959.9	10993.8

Tracer les courbes de speedup (pour chaque valeur de n), discuter les résultats.

OMP_NUM	T	T(n=2048)	T(n=512)	T(n=4096)
1	0.633688	6.21111	0.0855321	58.2984
2	0.634289	3.45764	0.0459778	31.1045
3	0.637213	2.36273	0.0372168	21.8706
4	0.640087	1.84447	0.0256945	16.8249
5	0.620745	1.4242	0.0259196	14.1325
6	0.669539	1.30372	0.0288679	12.0006

OMP_NUM	T	T(n=2048)	T(n=512)	T(n=4096)
7	0.619062	1.49772	0.0284917	13.5581
8	0.632346	1.3623	0.0224446	12.5014

Voici les tables de speedup pour chaque valeur de n, calculées comme $$ \text{Speedup} = \frac{T_{\text{seq}}}{T_{\text{par}}} \quad \text{où} \quad T_{\text{seq}} = T(\text{OMP_NUM}=1). $$

---

a. **Speedup pour (n = 2048)**

OMP_NUM	Speedup
1	1.000
2	1.797
3	2.629
4	3.367
5	4.362
6	4.764
7	4.148
8	4.559

---

b. **Speedup pour (n = 512)**

OMP_NUM	Speedup
1	1.000
2	1.860
3	2.299
4	3.329
5	3.300
6	2.963
7	3.002
8	3.812

---

c. **Speedup pour (n = 4096)**

OMP_NUM	Speedup
1	1.000

OMP_NUM	Speedup
2	1.874
3	2.666
4	3.465
5	4.125
6	4.858
7	4.300
8	4.663

---

4. **Courbes de Speedup**

- **Axe X** : Nombre de threads (`OMP_NUM`).
- **Axe Y** : Speedup (logique linéaire).
- **Courbes** : Une par valeur de (n).

Discussion des résultats

- Impact de la taille du problème (n):
    - **(n = 4096)** : Meilleur scaling grâce à un grain de calcul plus gros. Le speedup atteint **4.86** avec 6 threads (efficacité ≈ 81%).
    - **(n = 2048)** : Scaling correct mais moins efficace que (n = 4096). Pic à **4.76** avec 6 threads.
    - **(n = 512)** : Scaling médiocre. Le speedup chute dès 5 threads, remonte à 8 threads (**3.81**), signe de surcoût parallèle (surcharge de gestion des threads pour un petit problème).
- Efficacité des threads supplémentaires :
    - Pour (n = 4096), chaque thread supplémentaire améliore le speedup jusqu'à 6 threads.
    - Pour (n = 512), au-delà de 4 threads, les gains sont erratiques (surparallelisme).
- Valeurs de MFlops : Les MFlops suivent le speedup. Par exemple, pour (n = 4096), les MFlops passent de **2357.51** (1 thread) à **11452.7** (6 threads), confirmant une accélération réelle.

Conclusion

- **Grands (n)** : Le parallélisme est efficace (speedup proche de 5 pour 8 threads).
- **Petits (n)** : Le surcoût des threads domine (speedup limité à ≈3.8).
- **Optimal** : Utiliser 4–6 threads pour (n = 2048/4096), et 4–8 threads pour (n = 512) (selon les ressources disponibles). Ainsi, en augmentant le nombre de threads, le temps d'exécution devrait diminuer, indiquant une amélioration des performances. Cependant, il peut y avoir un plateau ou une légère augmentation du temps à partir d'un certain nombre de threads en raison de la surcharge de gestion des threads et de la contention pour les ressources.

 Image Description

Réponse à la question 4:

Il est possible d'améliorer le résultat obtenu car :

- Des algorithmes comme la multiplication par blocs permettent de réduire le temps d'accès à la mémoire et d'améliorer les performances.
- Une meilleure répartition des tâches entre les threads peut réduire la contention et maximiser l'utilisation des ressources.
- Avec des bibliothèques comme BLAS exploitent des techniques avancées et des optimisations spécifiques au matériel, offrant des performances supérieures.

Produit par blocs

```
make TestProduct.exe && ./TestProduct.exe 1024
```

```
//Code considéré

#include <algorithm>
#include <cassert>
#include <iostream>
#include <thread>
#if defined(_OPENMP)
#include <omp.h>
#endif
#include "ProdMatMat.hpp"

namespace {
void prodSubBlocks(int iRowBlkA, int iColBlkB, int iColBlkA, int szBlock,
                   const Matrix& A, const Matrix& B, Matrix& C) {
    const int iEnd = std::min(A.nbRows, iRowBlkA + szBlock);
    const int kEnd = std::min(A.nbCols, iColBlkA + szBlock);
    const int jEnd = std::min(B.nbCols, iColBlkB + szBlock);

    // Optimisation mémoire avec pré-chargement des blocs
    for (int k = iColBlkA; k < kEnd; ++k) {
        for (int j = iColBlkB; j < jEnd; ++j) {
            const double b_kj = B(k, j); // Pré-chargement car accès
multiple à B
            for (int i = iRowBlkA; i < iEnd; ++i) {
                C(i, j) += A(i, k) * b_kj;
            }
        }
    }
}

}  // namespace

Matrix operator*(const Matrix& A, const Matrix& B) {
  Matrix C(A.nbRows, B.nbCols, 0.0);
```

```
  // Paramètre clé à optimiser selon votre architecture
  constexpr int szBlock = 1024; // Valeur initiale à tester

  // Parcours des blocs dans l'ordre optimal pour la localité mémoire
  for (int iRowBlkA = 0; iRowBlkA < A.nbRows; iRowBlkA += szBlock)
    for (int iColBlkA = 0; iColBlkA < A.nbCols; iColBlkA += szBlock)
      for (int iColBlkB = 0; iColBlkB < B.nbCols; iColBlkB += szBlock)
        prodSubBlocks(iRowBlkA, iColBlkB, iColBlkA, szBlock, A, B, C);

  return C;
}
```

szBlock	MFlops	MFlops(n=2048)	MFlops(n=512)	MFlops(n=4096)
origine (=max)				
32	3460.02	3245.1	3053.83	2600.71
64	3833.69	3429.22	3122.58	3260.02
128	3997.53	4026.67	3198.56	3445.53
256	3732.56	3376.74	3119.95	3421.45
512	3802.35	3560.99	3139.71	3740.98
1024	3873.44	3622.03	3394.57	3570.16

Discutons les résultats. Le choix de la taille de bloc (`szBlock`) influence directement l'utilisation des caches (L1, L2, L3) du processeur. Pour des petites matrices, une petite taille de bloc (ex : 32 ou 64) permet de tenir les données dans le cache L1($n \leq 1024$), rapide mais limité en taille. Pour des matrices plus grandes, des blocs plus gros (128 ou 512) exploitent mieux les caches L2($1024 < n \leq 2048$) et L3($n > 2048$), réduisant les accès à la mémoire principale, plus lente. Les résultats montrent que **szBlock=128** est optimal pour des matrices de taille moyenne (n=1024-2048), tandis que **szBlock=512** est meilleur pour les très grandes matrices (n=4096), car il utilise efficacement le cache L3. Ainsi, on peut résumer comme ceci:

- **szBlock=128** : Meilleur compromis pour n=1024-2048 (≈4000 MFlops).
- **szBlock=512** : Optimal pour n=4096 (≈3740 MFlops).
  En conclusion, on pourra utiliser **szBlock=128** par défaut, et passez à **szBlock=512** pour $n \geq 4096$.

Bloc + OMP

```
//Code considéré

#include <algorithm>
#include <cassert>
#include <iostream>
#if defined(_OPENMP)
#include <omp.h>
#endif
#include "ProdMatMat.hpp"
```

```
namespace {
void prodSubBlocks(int iRowBlkA, int iColBlkB, int iColBlkA, int szBlock,
                   const Matrix& A, const Matrix& B, Matrix& C) {
    const int iEnd = std::min(A.nbRows, iRowBlkA + szBlock);
    const int kEnd = std::min(A.nbCols, iColBlkA + szBlock);
    const int jEnd = std::min(B.nbCols, iColBlkB + szBlock);

    // Optimisation mémoire avec pré-chargement des blocs
    for (int k = iColBlkA; k < kEnd; ++k) {
        for (int j = iColBlkB; j < jEnd; ++j) {
            const double b_kj = B(k, j); // Pré-chargement car accès
multiple à B
            #pragma omp simd // Vectorisation forcée
            for (int i = iRowBlkA; i < iEnd; ++i) {
                C(i, j) += A(i, k) * b_kj;
            }
        }
    }
}

// Taille de bloc adaptée à la hiérarchie mémoire de mon pd
constexpr int szBlock = 256; // Optimisé pour L2 cache (1.5MB/core)
}  // namespace

Matrix operator*(const Matrix& A, const Matrix& B) {
  Matrix C(A.nbRows, B.nbCols, 0.0);

  #pragma omp parallel for schedule(dynamic) // Charge dynamique pour
déséquilibres
  for (int iRowBlkA = 0; iRowBlkA < A.nbRows; iRowBlkA += szBlock) {
    for (int iColBlkA = 0; iColBlkA < A.nbCols; iColBlkA += szBlock) {
      for (int iColBlkB = 0; iColBlkB < B.nbCols; iColBlkB += szBlock) {
        prodSubBlocks(iRowBlkA, iColBlkB, iColBlkA, szBlock, A, B, C);
      }
    }
  }

  return C;
}
```

szBlock	OMP_NUM	MFlops	MFlops(n=2048)	MFlops(n=512)	MFlops(n=4096)
1024	1	7066.88	6499.97	6666.74	6362.83
1024	8	7067.53	7486.49	5816.2	4377.09
512	1	7636.49	6852.32	6196.14	7292.27
512	8	12194.4	18900	5908.53	11243.2

Discutons les résultats.

Les résultats montrent l'impact de la parallélisation (OpenMP) et de la taille de bloc (szBlock) sur les performances. Avec szBlock=512, la parallélisation sur 8 threads améliore significativement les

performances pour n=2048 et n=4096, grâce à une meilleure utilisation des caches et des cœurs CPU. Cependant, pour n=512, les performances chutent avec 8 threads, probablement à cause du surcoût de gestion des threads pour une petite taille de problème. La taille de bloc szBlock=512 est plus efficace que szBlock=1024, car elle exploite mieux les caches L2/L3 sans dépasser leur capacité.

Meilleure configuration :

n=512 : szBlock=512, 1 thread (6196 MFlops).

n=2048 : szBlock=512, 8 threads (18900 MFlops).

n=4096 : szBlock=512, 8 threads (11243 MFlops).

Conclusion :

```
Utilisez szBlock=512 pour maximiser les performances.

Activez la parallélisation (8 threads) pour n ≥ 2048, mais on doit la
désactiver pour n=512.
```

Comparaison avec BLAS, Eigen et numpy

Comparons les performances avec un calcul similaire utilisant les bibliothèques d'algèbre linéaire BLAS, Eigen et/ou numpy.

Les bibliothèques optimisées comme BLAS, Eigen ou numpy exploitent des techniques avancées (vectorisation SIMD, gestion fine des caches, parallélisme multicœur) pour maximiser les performances. En comparaison, une implémentation manuelle (même optimisée) atteint rarement leur efficacité, sauf dans des cas particuliers : AInsi, on remarque que, pour de petites matrices : Notre code peut rivaliser ces bibliothèques à très petite échelle (ex : n ≤ 1024), car le surcoût d’appel des fonctions BLAS/numpy domine. (les valeurs des MFLOPs sont légèrement élevées pour ./test_product_matrice blas (3578 MFLOPs) pour n=1024, ainsi notre code optimisé par bloc a un rapport de temps quasiment 3 fois supérieur à celui fait avec Blas, mais en règle générale, blas domine pour des matrices assez grandes)

Cependant, BLAS/Eigen et numpy (via OpenBLAS/MKL) devraient être 2 à 10× plus rapides pour n ≥ 2048, grâce à Une exploitation optimale des instructions processeur (AVX, FMA).

Conclusion :

- Pour les grandes matrices, il est préférable de privilégier BLAS/Eigen/numpy.
- Pour les petites tailles ou besoins spécifiques (contraintes mémoire, matrices structurées), une implémentation manuelle peut rivaliser, mais cela reste rare et dépend de l’optimisation matérielle.

Tips

```
    env
    OMP_NUM_THREADS=4 ./produitMatriceMatrice.exe
```

```
    $ for i in $(seq 1 4); do elap=$(OMP_NUM_THREADS=$i
./TestProductOmp.exe|grep "Temps CPU"|cut -d " " -f 7); echo -e
"$i\t$elap"; done > timers.out
```