TD1 par Joseph Mouscadet

La machine sur laquelle le code a été exécuté présente les caractéristiques suivantes :

```
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) i7-8750H CPU @ 2.20GHz
    CPU family:          6
    Model:               158
    Thread(s) per core:  2
    Core(s) per socket:  6
    Socket(s):           1
    Stepping:            10
    CPU max MHz:         4100,0000
    CPU min MHz:         800,0000
```

```
Caches (sum of all):
  L1d:                   192 KiB (6 instances)
  L1i:                   192 KiB (6 instances)
  L2:                    1,5 MiB (6 instances)
  L3:                    9 MiB (1 instance)
```

Produit Matrice-Matrice

Mesures et première optimisation

Ordre des boucles	1023	1024	1025
ijk	1.17s (1025 MFlops)	2.64s (810 MFlops)	1.17s (1840 MFlops)
jki	0.9s (2400 MFlops)	0.9s (2400 MFlops)	0.9s (2400 MFlops)
kji	0.65s (3280 MFlops)	0.65s (3260 MFlops)	0.67s (3180 MFlops)

On voit que pour l'ordre de boucle par défaut, la dimension 1024 est la plus lente. Cela est du au fait que le cache est organisé par lignes de 1024 octets, ce qui est exactement égal à la dimension des colonnes de la matrice. Les colonnes sont donc séparées sur les lignes de cache et cela rallonge les temps d'accès mémoire, ce qui est le principal goulot d'étranglement.
Concernant l'ordre des boucles, comme le produit fait apparaître le calcul suivant :

$$C_{ij} \leftarrow C_{ij} + A_{ik} * B_{kj}$$

On voit que si on n'a pas i comme boucle la plus interne, on ne peut pas accéder aux coefficients de C et de A de manière contiguë ce qui fait perdre beaucoup de temps. La boucle i la plus interne est cohérente avec mes résultats, mais mettre la boucle k au milieu devrait donner des meilleurs résultats car cela permet d'accéder également aux éléments de B de manière contiguë. De plus, la différence entre l'ordre ijk et les autres devrait être plus marquée. Cela est probablement dû à la grande taille du cache de mon CPU, comme vous me l'avez dit. Cela est également peut-être du au mode "Power Saver" de mon ordinateur au lieu du mode "Performance". De toute manière, on voit qu'il est important de bien comprendre le fonctionnement de la mémoire car de simples optimisations peuvent nous faire gagner beaucoup de temps.

Première parallélisation

Voici mes résultats :

NUM_THREADS	Temps	Accélération
Aucune	0.65s (3200 MFlops)	----------
1	0.12s (17760 MFlops)	81%
2	0.12s (17750 MFlops)	81%
4	0.12s (17820 MFlops)	81%
8	0.12s (17600 MFlops)	81%

Ce n'est pas logique d'avoir de tels résultats, on devrait voir une accélération croissante avec l'augmentation du nombre de threads mais qui s'amenuisent à mesure que le nombre de threads augmente.

Deuxième optimisation

Modification effectuée :

```
Matrix operator*(const Matrix& A, const Matrix& B) {
  Matrix C(A.nbRows, B.nbCols, 0.0);
  for (int jb=0; jb < B.nbCols; jb += szBlock )
      for (int kb=0; kb<A.nbCols; kb += szBlock )
          for(int ib=0; ib<A.nbRows; ib += szBlock)
              prodSubBlocks(ib, jb, kb, szBlock, A, B, C);
  return C;
}
```

Résultats :

Block size	Temps
32	0.63s (3350 MFlops)
64	0.65s (3250 MFlops)
128	0.62s (3410 MFlops)
256	0.59s (3630 MFlops)
512	0.58s (3700 MFlops)

On voit donc que l'on a une légère amélioration par rapport au produit scalaire classique mais l'amélioration n'est pas flagrante. La bande-passante mémoire n'est donc sûrement pas le goulot d'étranglement.

Parallélisation du produit par blocs

Modification effectuée :

```
Matrix operator*(const Matrix& A, const Matrix& B) {
  Matrix C(A.nbRows, B.nbCols, 0.0);
  # pragma omp parallel for collapse(2)
  for (int jb=0; jb < B.nbCols; jb += szBlock )
      for (int kb=0; kb<A.nbCols; kb += szBlock )
          for(int ib=0; ib<A.nbRows; ib += szBlock)
              prodSubBlocks(ib, jb, kb, szBlock, A, B, C);
  return C;
}
```

Résultats :

Block size	Temps
32	0.13s (16300 MFlops)
64	0.14s (14600 MFlops)

On voit donc que l'on est aussi performant que la parallélisation simple, donc la gestion par blocs n'a pas beaucoup d'effet lorsque le calcul est déjà parallélisé. De plus, sur ma machine, prendre une taille de bloc de 128 ou supérieure mène à un plantage du programme en raison d'erreurs de calcul.

Comparaison avec BLAS

En comparant avec BLAS, j'obtiens un résultat de quasiment 100 000 MFlops (99 730 MFlops), cette librairie est donc bien plus performante et il nous reste donc beaucoup d'optimisations à trouver. Il est donc quasiment toujours plus pertinent d'utiliser une librairie

existante pour leurs algorithmes très optimisés, sauf si l'on est dans un cas extrêmement particulier où l'on peut faire mieux qu'elles.