Analyse des performances de multiplication matricielle

Q1

Pour `dim = 1023`, le temps de calcul est de 2.71s, et les MFlops sont de 788.96.
Pour `dim = 1024`, le temps de calcul est de 7.20s, et les MFlops sont de 298.063.
Pour `dim = 1025`, le temps de calcul est de 2.92s, et les MFlops sont de 738.335.

Explication: Il s'agit d'un gros problème de cache. Les nombres sont stockés de manière contiguë en RAM, et la lecture par le cache se fait par puissances de 2 (8 ou 16, par exemple). Par conséquent, pour accéder aux différentes valeurs successivement, il faut faire un "saut de 1024" entre chaque opération. Comme ces sauts se font en lisant des valeurs à des adresses multiples de 2^n (par exemple 8 ou 16), l'adressage dans le cache devient modulaire par 2^n, ce qui empêche une bonne exploitation des lignes de cache, écrasant à chaque fois les valeurs précédemment stockées.

Q2

Nous sommes passés à 3859 MFlops. Le changement d'ordre d'accès par le cache a dû faire en sorte que les valeurs soient lues successivement, plutôt que de faire des sauts de 1024.

Q3

On observe d'abord une augmentation linéaire des MFlops, puis une baisse d'efficacité, atteignant un plafond de 25000 MFlops avec 16 threads utilisés.

Nombre de threads	**MFlops**
1	3660
2	7240
3	8662
4	12341
5	12782
6	13849

Nombre de threads	MFlops
7	18700
8	18400
9	19600
10	20180
11	22000
12	22340
13	22773
14	23000
15	25900
16	24600

Ce plafonnement des performances semble dû à la capacité du cache processeur. Dans mon cas, le processeur dispose de 12 Mo de cache, et les matrices calculées ont une dimension de 1024x1024, soit environ 8 Mo chacune (matrice de doubles, chaque élément étant stocké sur 8 octets). Cela monopolise 24 Mo de mémoire, sans compter les opérations supplémentaires. Le processeur doit alors effectuer des demandes d'accès à la RAM, qui est beaucoup plus lente que le cache.

D'après la documentation de mon processeur, il est possible d'optimiser les performances en effectuant les calculs exclusivement dans le cache, où les temps d'accès sont bien plus rapides (Cache L3 : 20ns, RAM : 80ns).

Q4

L'idée serait de partitionner les matrices en blocs plus petits, afin de calculer les résultats par blocs, de sorte que toutes les données restent dans le cache sans jamais accéder à la RAM.

Q5

Pour un cache de 12 Mo, il serait optimal de calculer des matrices d'environ 2 Mo, soit des matrices de 512x512, et de les stocker ensuite.

Taille des blocs	MFlops
32	3502
64	3979
128	4175
256	4098
512	4700

Il semble que l'optimisation des performances soit meilleure avec des matrices de taille 512x512.

Q6

Lorsque l'on n'alloue qu'un seul thread, le processus devient plus lent en raison de l'ajout d'une étape de création des sous-blocs matriciels.

Q7

En parallélisant avec 2 threads, il apparaît que la taille de bloc la plus optimale est de 512.

Taille des blocs	**MFlops**
32	6566
64	6743
128	7637
256	7451
512	8100

Il semble que nous ne puissions pas allouer plus de n threads si $n \times \text{szBlock} > 1024$, sous peine d'erreur numérique. Nous allons donc paralléliser avec 4 threads et utiliser des tailles de blocs jusqu'à 256.

Taille des blocs	**MFlops**
32	11575
64	11630
128	13022
256	11548

Pour 8 threads et des tailles de blocs ≤ 128 :

Taille des blocs	**MFlops**
32	15000
64	15800
128	18242

Pour 16 threads et des tailles de blocs ≤ 64 :

Taille des blocs	**MFlops**
32	18826
64	24000

Q8

BLAS ne rivalise pas beaucoup plus, probablement à cause d'un problème interne à ma machine, puisque chez un collègue, BLAS atteint les 100000 MFlops.