

Commande lscpu:

```
rbk@rbk-X550VX:~/Desktop/ln203/tp3$ lscpu
Architecture:                x86_64
CPU op-mode(s):              32-bit, 64-bit
Byte Order:                  Little Endian
Address sizes:               39 bits physical, 48 bits virtual
CPU(s):                      8
On-line CPU(s) list:        0-7
Thread(s) per core:          2
Core(s) per socket:          4
Socket(s):                   1
NUMA node(s):                1
Vendor ID:                   GenuineIntel
CPU family:                   6
Model:                       94
Model name:                  Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
Stepping:                    3
CPU MHz:                     2600.000
CPU max MHz:                 3500.0000
CPU min MHz:                 800.0000
BogoMIPS:                    5199.98
Virtualization:              VT-x
L1d cache:                   128 KiB
L1i cache:                   128 KiB
L2 cache:                    1 MiB
L3 cache:                    6 MiB
NUMA node0 CPU(s):          0-7
Vulnerability Itlb multihit: KVM: Mitigation: VMX disabled
Vulnerability Itlb:          Mitigation; PTE Inversion; VMX conditional cache flushes, SMT vulnerable
Vulnerability Mds:           Mitigation; Clear CPU buffers; SMT vulnerable
Vulnerability Meltdown:      Mitigation; PTI
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl and seccomp
Vulnerability Spectre v1:    Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:    Mitigation; Full generic retpoline, IBPB conditional, IBRS_FW, STIBP conditional, RSB filling
Vulnerability Srbds:         Mitigation; Microcode
```

En utilisant la commande lscpu, mon pc a 8 coeur physiques et virtuelles, en fait, il s'agit de deux threads par coeur physique et la somme total des coers physiques est 4.

Le nombre de socket est le nombre d'emplacement de processeur du pc.

Exercice1:

Q1)

Code sur github

Q2)

Pour le temps de calcul du produit scalaire seulement on a:

*pour le code séquentiel:

T=0.78

En exécutant le code parallèle (OpenMP) avec différents nombre de thread on a le tableau suivant:

Nombre de threads	Speed up =ts/tp
2	$0.78/0.38=2.05$
4	$0.78/0.2=3.9$
6	$0.78/0.184=4.23$
8	$0.78/0.180=4.33$

Commentaire:

On remarque que si on augmente le nombre de thread le temps de calcul augmentre legerement de même pour le speed up et que le temps de calcul parallele est legrement plus petit que le temps de calcul du code séquentiel.

Q3) j'ai essayé d'exécuter le fichier "dotproduct_thread.cpp" en modifiant le nombre de threads dans le code et j'ai obtenu le tableau suivant:

Nombre de threads	Speed up
4	$0.78/1.49=0.52$
8	$0.78/1.29=0.4$

Q4)

En fait , le temps de calcul en parallele avec OpenMP est très petit devant celui utilisé avec les threads.

Q5)

Tout en utilisant deux méthodes de prallélisation , le temps de calcul reste proche de celui du code séquentiel ce qui n'est pas souhaitable.

Donc ,il s'agit de 'memory bounding" c'est à dire le temps de calcul et d'exécution est limité par le temps d'accès aux données de la mémoire. Puisque le nombre d'opérations est égale au nombre d'accès a la mémoire.

Exercice 2:

Q1) Ce qu'on peut remarquer dans cet exercice est que le nombre d'accès a la mémoire($3 \times \text{blocksize}^2$) qui est inférieure au nombre d'operations($2 \times \text{blocksize}^3$).Il s'agit de CPU bound.

dimension	temps	Mflops
1023	1.561	1371.27
1024	4.08	526.058
1025	1.562	1390

Commentaire:

On remarque le temps pour dim=1024 est plus log par rapport aux deux autres dimensions, on peut explique ça par le fait que le nombre de ligne=1024 est multiple de nombre de lignes caches c ad que les données seront stockés dans la même partie d'une ligne de cache. Ce qui n'est pas le cas pour dim=1023 et 1025.

D'ou ,l'execution dure plus pour dim=1024.

Q2)

Pour n=1024

Permutation des boucles	temps	Mflops
I,j,k	3.83	560.653
J,ik	6.30	340.5
I,k,j	17.73	121.079
K,I,j	18.32	117.211
J,k,i	0.95	2250.15
K,j,i	1.08	1981.55

COMMENTAIRE:

On remarque que le temps de calcul le optimale est réalisé lorsque i est au bas niveau ce qui est approuvé par le cours .

On peut expliquer cela par le fait que travailler en ligne en bas niveau nous permet de stocker les données sur les différents lignes de caches pour éviter un retard d'exécution.

Q3)

En utilisant la boucle j,k,l(dont le temps de calcul est le plus court) en parallélisant le code avec Open MP on obtient le tableau suivant :

Pour n=1024

Nombre de threads	temps	Mflops
2	0.93	2305
4	0.92	2315
8	0.87	2463

Q4)on peut améliorer notre code par passage au calcul par bloc de matrice qui nous permet une meilleure exploitation des caches .

Q5) code sur github

En testant le code pour différentes valeurs de szblock(taille du block) , on remarque que pour szblock=510 , on atteint un maximum de MFlops qui est égale à peu près à 3000.91.

EN fait d'après la commande lscpu, on remarque que la capacité de mémoire cache L3 est 6MB==(à peu près)6100k ,d'autre part pour les 3 matrices A ,B et C on a , $510^2 \times 3 \times 8 (\text{sizeof double}) = 6200K$ qui peuvent alors être stockés dans L3.

Q6)

Pour la boucle jki

dimension	temps	MFlops
1023	0.73	2923.5
1024	0.72	2948.44
1025	0.74	2845.64

Malgré l'augmentation de la taille (dimension) de la matrice il n'y a pas de grand changement concernant le temps et MFlops ce qui se différencie de la première question.

Q7) code sur github

Pour la dim 1024 j'ai essayer diffents nombre de threads sur le code par block

Nombre de thread (open MP)	temps	MFlops
2	0.89	2394
4	0.88	2424.91
8	0.89	2396