TP2 de MSSELLATI Noam et MOREL Florian

```
pandoc -s --toc tp2.md --css=./github-pandoc.css -o tp1.html
```

lscpu

```
Architecture :                          x86_64
Mode(s) opératoire(s) des processeurs : 32-bit, 64-bit
Boutisme :                              Little Endian
Address sizes:                          39 bits physical, 48 bits virtual
Processeur(s) :                         4
Liste de processeur(s) en ligne :       0-3
Thread(s) par cœur :                    1
Cœur(s) par socket :                    4
Socket(s) :                             1
Nœud(s) NUMA :                          1
Identifiant constructeur :              GenuineIntel
Famille de processeur :                 6
Modèle :                                126
Nom de modèle :                         Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GH
                                        z
Révision :                              5
Vitesse du processeur en MHz :          1190.400
BogoMIPS :                              2380.80
Constructeur d'hyperviseur :            KVM
Type de virtualisation :                complet
Cache L1d :                             192 KiB
Cache L1i :                             128 KiB
Cache L2 :                              2 MiB
Cache L3 :                              24 MiB
Nœud NUMA 0 de processeur(s) :          0-3
Vulnerability Itlb multihit:            KVM: Mitigation: VMX unsupported
Vulnerability L1tf:                     Not affected
Vulnerability Mds:                      Not affected
Vulnerability Meltdown:                 Not affected
Vulnerability Mmio stale data:          Vulnerable: Clear CPU buffers attempted,
                                         no microcode; SMT Host state unknown
Vulnerability Retbleed:                 Vulnerable
Vulnerability Spec store bypass:        Vulnerable
Vulnerability Spectre v1:               Mitigation; usercopy/swapgs barriers and
                                         __user pointer sanitization
Vulnerability Spectre v2:               Mitigation; Retpolines, STIBP disabled,
                                        RSB filling, PBRSB-eIBRS Not affected
Vulnerability Srbds:                    Unknown: Dependent on hypervisor status
Vulnerability Tsx async abort:          Not affected
Drapaux :                               fpu vme de pse tsc msr pae mce cx8 apic
```

```
sep mtrr pge mca cmov pat pse36 clflush
mmx fxsr sse sse2 ht syscall nx rdtscp l
m constant_tsc rep_good nopl xtopology n
onstop_tsc cpuid tsc_known_freq pni pclm
ulqdq ssse3 cx16 pcid sse4_1 sse4_2 x2ap
ic movbe popcnt aes xsave avx rdrand hyp
ervisor lahf_lm abm 3dnowprefetch invpci
d_single fsgsbase avx2 invpcid rdseed cl
flushopt md_clear flush_l1d arch_capabil
ities
```

Des infos utiles s'y trouvent : nb core, taille de cache

EXERCICE 1 Produit matrice-matrice

Q1.

On remarque que le temps de calcul du produit de deux matrices est anormalement long lorsque les deux matrices à multiplier sont de taille 1024, 2048, 4096... Ceci s'explique par le fait que nos processeurs fonctionnent en mode "Associative memory cache". Les lignes de caches étant de taille 1024, à chaque opération nous allons solliciter la même ligne de cache. Il faudra alors la recherger à chaque fois avec les valeurs qui nous intéressent. Dans ce cas ci, le cpu fonctionne comme si le cache était beaucoup plus petit ce qui entraine ce temps de calcul exceptionnellement grand.

Q2. Permutation des boucles

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

ordre	time(n=1024)	MFlops(n=1024)	MFlops(n=2048)
i,j,k (origine)	4.46023 s	481.474	331.614
j,i,k	4.72208 s	454.775	270.837
i,k,j	9.01586 s	238.189	83.1927
k,i,j	8.32321 s	258.012	82.7571
j,k,i	1.08502 s	1979.2	1967.08
k,j,i	1.07867 s	1990.86	1883.8

On remarque que les deux dernières combinaisons sont les plus rapides en temps de calcul. Ceci est dû au fait que la dernière boucle for agit sur les lignes de A. En effet, les matrices sont codés de façon à stocker deux valeurs voisines sur une colonne à la suite dans la mémoire. La ligne de cache en cours de lecture pour le calcul sera alors entièrement lue avant de recharger une nouvelle ligne de cache.

OMP sur la meilleure boucle

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

OMP_NUM	MFlops	MFlops(n=2048)	MFlops(n=512)	MFlops(n=4096)
1	2097.89	2045.69	2397.93	??
2	3560.14			
3	5182.75			
4	6344.09			
5	6101.28			
6	5915.58			
7	6197.21			
8	6380.2			

Sans surprise, lorsqu'on augmente le nombre de threads, le temps de calcul diminue et donc le nombre de MFlops augmente. Cependant, à partir de 5 threads, le temps de calcul stagne ou même diminue. Ceci est dû au fait que le lscpu affiche 4 processeurs disponibles. les threads supérieurs à 5 seront alors executés sur des processeurs déjà occupés par un autre thread ce qui n'augmentera pas la vitesse de calcul global.

Produit par blocs

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

szBlock	MFlops	MFlops(n=2048)	MFlops(n=512)	MFlops(n=4096)
origine (=max)	1953.53	2117.03	2325.79	1934.4
32	1618.79	1544.14	1587.55	?
64	1919.54	1782.43	1960.27	?
128	2035.24	2043.32	2113.84	?
256	2075.03	2208.49	2121.09	?
512	2124.74	2186.29	x	?
1024	x	1794.92	x	?

Pour la meilleure efficacité possible, il faut que les bloques puissent entrer en cache. On remarque que c'est pour une taille de bloque de 512 que le MFlops est le plus élevé.

Bloc + OMP

szBlock	OMP_NUM	MFlops	MFlops(n=2048)	MFlops(n=512)	MFlops(n=4096)
A.nbCols	1	2059.35	2059.63	2543.18	2076.88
512	8	7445.27	7085.16	6248.28	6840.28

szBlock	OMP_NUM	MFlops	MFlops(n=2048)	MFlops(n=512)	MFlops(n=4096)
Speed-up		362 %	344 %	246 %	329 %

On remarque que la double optimisation : calcul par bloque + parallélisation permet d'augmenter d'environ 320 % le MFlops. Ce qui permet de diminuer considérablement le temps de calcul de n'importe quelle multiplication de matrices.

Comparaison with BLAS

Méthode	MFlops	MFlops(n=2048)	MFlops(n=512)	MFlops(n=4096)
calcul par bloc + parallélisation	7445.27	7085.16	6248.28	6840.28
blas	31828.2	44834.9	45406.3	41594.5

La bibliothèque blas permet une optimisation beaucoup plus poussé du calcul matrice matrice. Son MFlops est nécessairement plus important. Nos deux actions pour optimisées ne sont alors pas suffisant pour optimiser au maximum ce calcul.

Exercice 2.1

Nous avons tenté de faire marcher l'anneau mais nous avons eu un soucis de boucle infini. De plus en utilisant mpirun pour python, le nombre de coeurs ne peut etre superieur à 1. Pour ce qui est du code il est assez explicite : on recoit un emessage du precdent (left) que l'on incremente et l'on stock dans i, puis on renevoi au prochain (right) donc rank+1. on arrete lorsque l'on a fait un tour complet (d'où le if).

EXERCICE 2.2

Nous n'avons pas réussi à exécuter notre programme. L'objectif était de d'abord créer un thread maître (rank==0) qui distribue le travail du calcul de pi à deux autres threads (rank 1 et 2). Le thread maître récupère ces valeurs et en fait la moyenne.

Cependant, nous rencontrons des problèmes lors de l'envoie des données d'un thread à un autre. La dimension des données ne coincide pas. (erreur : [[58567,0],0] ORTE_ERROR_LOG: Data unpack had inadequate space in file util/show_help.c at line 513)

Par la suite, nous voulions utiliser les fonctions MPI_Isend et MPI_Irecv pour réellement parlléliser. Dans le code qui est actuellement écrit, les threads

s'exécutent de façon séquentielle.