# MN – TP n°1

## I – Présentation de la machine de tests

Pour ce travail la machine de tests est un Apple MacBook Air de mi 2012. L'ordinateur dispose de 8Go de RAM, et d'un CPU dont les spécifications sont les suivantes :

| | | | |
|---|---|---|---|
| Introduction Date: | June 11, 2012 | Discontinued Date: | June 10, 2013 |
| Processors: | 1 (2 Cores) | Architecture: | 64-Bit |
| Geekbench 2 (32): | 6831 | Geekbench 2 (64): | 7644 |
| Geekbench 3 (32): | 2564 | Geekbench 3 (32): | 5181 |
| Geekbench 3 (64): | 2865 | Geekbench 3 (64): | 5877 |
| Processor Speed: | 2.0 GHz | Processor Type: | Core i7 (I7-3667U) |
| Details: | This model is powered by a 22 nm, 64-bit Intel Mobile Core i7 "Ivy Bridge" (I5-3667U) processor which includes two independent processor "cores" on a single silicon chip. Each core has a dedicated 256k level 2 cache, shares 4 MB of level 3 cache, and has an integrated memory controller (dual channel).<br><br>This system also supports "Turbo Boost 2.0" -- which "automatically increases the speed of the active cores" to improve performance when needed (up to 3.2 GHz for this model) -- and "Hyper Threading" -- which allows the system to recognize four total "cores" or "threads" (two real and two virtual).<br><br>Also see: How much faster are the custom processor configured "Mid-2012" MacBook Air models than the stock models? Is the extra performance worth the extra cost? | | |
| Turbo Boost: | 3.2 GHz | Custom Speeds: | N/A |
| Processor Upgrade: | Soldered | FPU: | Integrated |
| System Bus Speed: | 5 GT/s* | Cache Bus Speed: | 2.0 GHz (Built-in) |
| ROM/Firmware Type: | EFI | EFI Architecture: | 64-Bit |
| L1 Cache: | 32k/32k x2 | L2/L3 Cache: | 256k x2, 4 MB (on chip) |
| RAM Type: | DDR3L SDRAM* | Min. RAM Speed: | 1600 MHz |

## II – Présentation du travail

Le travail se décompose en 6 fichiers. On y retrouve notamment :

- un Makefile qui s'utilise de la manière suivante make pour compiler le programme et make clear pour supprimer les fichiers générés à la compilation
- vecteur.h comportant les spécifications des fonctions sur les vecteurs
- vecteur.c comportant le corps des fonctions sur les vecteurs
- libvect.a la librairie des vecteurs
- test_vect.c le fichier contenant le programme de test
- gen_aleat.c le fichier contenant le programme qui permet de générer les fichiers contenant les composantes des vecteurs.

Les codes sources contiennent tous les détails, mais ne sont pas commentés.

Lors de la compilation deux exécutables vont être générés :

Gen_aleat et test_vect

Gen_aleat s'utilise de la manière suivante : ./gen_aleat [nombre de composantes]
Il va alors créer un fichier test_vect.txt contenant tous les vecteurs.

Test_vect s'utilise de la manière suivante : ./test_vect[d/f (double/float)] [s/b (gros ou petit vecteur pour le cache)] [a/m/s (addition/multiplication/produit scalaire)].

## III – Protocole des tests

Afin de tester le travail réalisé nous allons suivre le protocle suivant :

- compiler en utilisant l'optimisation O, O2, O3 de gcc
- utiliser gen_aleat pour fabriquer notre fichier de vecteurs
- exécuter 100000 fois l'opération sélectionner afin d'avoir une meilleure vue des résultats
- monitorer le temps d'éxécution a l'aide de time
- monitorer l'utilisation CPU à l'aide de top

## IV – Résultats des tests

Sans optimisation voici le résultat de la commande :

```
time test_vect d b s < test_vect.txt
real 0m35.536s
user 0m35.161s
sys  0m0.205s

time test_vect d s s < test_vect.txt
real 0m0.372s
user 0m0.359s
sys  0m0.006s

time test_vect f s s < test_vect.txt
real 0m0.392s
user 0m0.383s
sys  0m0.005s

time test_vect f b s < test_vect.txt
real 0m37.746s
user 0m37.488s
sys  0m0.118s
```

Déjà nous notons un large gain de temps lorsque l'on passe de vecteurs à 1000 composantes à des vecteurs à 100000 composantes et qui par conséquent ne rentrent plus en cache. Le gain de temps est d'environ 100 fois plus rapide quand les données sont stockées en cache !

De plus les double prennent deux fois plus de temps à s'éxécuter que les floats. Normal, il y'a le double de données à traiter.

Passons désormais aux tests avec optimisation O3 :

```
time test_vect d b s < test_vect.txt
real0m10.766s
user0m10.580s
sys  0m0.077s

time test_vect d s s < test_vect.txt
real0m0.121s
user0m0.114s
sys  0m0.005s

time test_vect f s s < test_vect.txt
real0m0.120s
user0m0.112s
sys  0m0.005s

time test_vect f b s < test_vect.txt
real0m11.249s
user0m11.047s
sys  0m0.074s
```

La encore la différence entre les options de compilations est flagrante. On atteint quasiment un gain de 335%.

## V – Calcul du nombre d'opérations flottantes secondes

Par commodité ce calcul sera réalisé dans le cas le plus performant, à savoir :

```
time test_vect f s s < test_vect.txt
real0m0.120s
user0m0.112s
sys  0m0.005s
```

Nous avons donc deux vecteurs de 100000 composantes chacuns, et nous allons multiplier ces composantes ensembles. Nous avons donc 1000000 x 100000 multiplications de ainsi que 1000 x 100000 additions flottants en 11,2 secondes.
Nous arrivons donc à : 1 785 000 000 multiplications flottantes à la seconde.
Nous avons donc un processeur a 1,785 GFLOPS pour ce programme. On est à 1/8 de la crête pour notre processeur. On pourrait encore optimiser ce résultat en vectorisant.

## VI – Conclusion

Inutile de tester toutes les opérations et l'option O2 pour conclure. En utilisant le cache correctement et en optimisant la compilation le temps d'éxécution d'un programme peut

changer du tout au tout. En cumulant les deux on peut constater une diminution de ce dernier de près de 100 fois.