Travaux Pratique 2

Ce sujet est en lien avec les deux premiers chapitres du cours, et concerne les *machines vectorielles*. L'idée est de manipuler des ***instructions AVX***, qui travaillent sur des registres 256 bits, soit 8×4 octets. Elles permettent de manipuler 8 réels en simple précision via une unique instruction, et cela de façon native dans les processeurs modernes. Les instructions AVX disponibles sur les différents processeurs sont documentées en ligne :

https://software.intel.com/sites/landingpage/IntrinsicsGuide/#

Nous en profiterons pour faire quelques calculs simples, illustrant l'intérêt de ces instructions, notamment en lien avec du multithread. Attention, dans ce dernier cas le modèle n'est plus simplement vectoriel, et donc il faut ajouter des synchronisations (enfin, lorsque nécessaire) pour gérer l'aspect multiprocesseurs vectoriels. Pour chaque exercice, il est demandé deux versions : l'une séquentielle (manipulant des `float`), et l'autre vectorielle (manipulant des `__m256`, des paquets de 8 réels simples précisions).

Votre travail est à rendre (comme à chaque fois) sous la forme du code (et uniquement la partie « `student` ») accompagnée d'un rapport au format PDF, le tout dans une archive compressée au format ZIP. Ne respecter pas ces contraintes, et votre note en sera diminuée de quelques points.

Vous ne devez modifier que ce qui se trouve dans le répertoire « `./student/` » !

Notez que chaque exercice vient avec un squelette, qui s'occupe de la ligne de commande, du lancement de votre code (défini dans une classe particulière), et d'une vérification sommaire du résultat (lorsque c'est possible). Utilisez l'option « `-h` » ou « `--help` » pour connaître le fonctionnement de la ligne de commande …

Deux scripts sont fournis pour compiler : sous Windows, il s'agit de « `./build.bat` » qui génère les exécutables dans le répertoire « `./win32/Release` » ; pour les autres systèmes, utilisez « `bash ./build.sh` » qui génère les exécutables dans « `./linux/` ». Ces scripts utilisent « `cmake` », et soit `GCC` sous linux/mac, soit `Visual C++ 19` sous Windows. Leur comportement a été testé sous Windows, WSL/Debian, et Linux Mint.

Exercice 1

Etudiez le code fourni dans la partie `student`. Notez dans le rapport le type des données utilisé, et le résultat obtenu. N'hésitez pas à consulter la documentation des fonctions utilisées (cf. le lien plus haut). Aucun code n'est à produire ici !

Exercice 2

Vos deux fonctions reçoivent un tableau de valeurs, et doivent retourner le tableau des racines carrées de ces valeurs (patron MAP). La première utilise des `float`, la seconde des `__m256`. L'accélération obtenue (entre les deux versions) dépend de la machine utilisée. Testez différentes tailles de tableau (ligne de commande) et mesurez *l'efficacité*. Notez le tout dans votre rapport !

Exercice 3

Même principe que l'exercice précédent, mais ici en utilisant plusieurs threads pour optimiser encore plus. Qu'observez-vous ? Eh oui, le syndrome du retour de week-end !

Exercice 4

Vous recevez un tableau de valeurs, et devez calculer leur somme le plus rapidement possible … Par exemple si le tableau contient $\{1,2,3,\ldots,10\}$, le résultat est

$$\sum_{i=1}^{10} i = \frac{10(10+1)}{2} = 55$$

Plusieurs fonctions AVX sont à utiliser, et il y a différentes solutions possibles. Donc, lisez et fouillez la documentation !

NB : validez votre travail avec peu de valeurs au début (e.g. 640.000), puis observez ce qu'il se passe avec plus de valeurs (e.g. 6.400.000). Pourquoi ce résultat ?

Exercice 5

L'idée ici est de calculer rapidement un produit matrice par vecteur (en ligne de commande vous spécifiez la taille du vecteur divisé par 8). Commencez par faire ce calcul sur papier pour une matrice 8×8. Puis passez à son implantation (d'abord en séquentiel, puis en vectoriel) !

Quelle est l'accélération ? Quelle est l'efficacité ?

NB :

- Ici aussi le calcul devient faux en augmentant la taille de la matrice. Donc, validez votre code avec de petites matrices, puis augmentez pour mesurer les temps d'exécution.
- Les fonctions utiles sont `_mm256_add_ps`, `_mm256_set1_ps` et `_mm256_setzero_ps`, `_mm256_mul_ps`, et `_mm256_dp_ps`.