TP Streaming SIMD Extensions (SSE)

Ce TP est une introduction aux instructions SSE https://fr.wikipedia.org/wiki/Streaming_SIMD_Extensions.
Pour connaître les différentes instructions : https://software.intel.com/sites/landingpage/IntrinsicsGuide.
Et l'utilisation de `_mm_shuffle_ps` : https://msdn.microsoft.com/fr-fr/library/4d3eabky%28v=vs.90%29.aspx.

Récupérer le code à compléter sur https://www.lri.fr/~bagneres/index.php?section=teaching&page=2015_2016_App3_archi. Le fichier `tests/simd.hpp` permet de faciliter la manipulation des registres SIMD.

0.1 Quelle est la différence entre l'utilisation de la macro `_MM_SHUFFLE` et `vfloat32_shuffle` ? Quelle est l'utilité de `vfloat32_shuffle` ?

Pour avoir une idée correcte du temps, exécuter plusieurs fois le programme (au moins 5 fois) et prendre le temps minimum.

Faire les exécutions à comparer sur la même machine et dans les mêmes conditions. Donner les caractéristiques du système d'exploitation, du compilateur et du processeur (modèle, nombre de cœurs, hyperthread, taille des caches, ... (la commande `cat /proc/cpuinfo` est un bon début)).

1 Somme Des Élements D'Un Tableau

Dans le test `sum.cpp`, on calcule la somme des éléments d'un tableau avec et sans les instructions SSE. Mesurer le temps mis avec un grand tableau.

1.1 Comparer le gain obtenu avec celui naivement espéré.

Dans le test `sum_bench.cpp`, on décide d'ajouter des calculs (uniquement dans la boucle) : `sum += std::sqrt(v[i] * v[i])`.
Faire ce calcul en utilisant les instructions SSE.

1.2 Comparer le gain obtenu avec celui de `sum.cpp`.

Dans le fichier `CMakeLists.txt` (ligne 25), enlever l'option `-fno-tree-vectorize` (uniquement pour la question suivante) qui permettait d'interdire au compilateur de vectoriser le code lui-même.

1.3 Pourquoi le gain obtenu par la version SSE a-t-il disparu ?

2 Produit Scalaire

On souhaite calculer le produit scalaire (*dot product* en anglais) de deux tableaux en utilisant la formule suivante : $\vec{x} \cdot \vec{y} = x_1 \times y_1 + x_2 \times y_2 + ... + x_n \times y_n$ https://fr.wikipedia.org/wiki/Produit_scalaire#Base_orthonormale
Implémenter ce calcul dans le fichier `tests/dot_product.cpp`.

2.1 Commenter le gain obtenu.

3 Filtre Moyenneur, Part I

On souhaite appliquer le filtre moyenneur suivant : `r[i] = (v[i-1] + v[i] + v[i+1]) / 3.f;` (sans la gestion des bords).

Pour implémenter la version SSE de ce calcul (dans le fichier `tests/avg3.cpp`), il faudra implémenter et utiliser les fonctions `vfloat32_left_1` et `vfloat32_right_1`.

La fonction `vfloat32_left_1` (dans le fichier `tests/simd.hpp`) prend deux `vfloat32_t` (`{ a, b, c, d }` et `{ e, f, g, h }` par exemple) et renvoie un `vfloat32_t` contenant les valeurs du premier registre décalées vers la gauche et en ajoutant la première valeur de deuxième registre (on obtient `{ b, c, d, e }` dans notre exemple).
Pour faire cette opération, deux `_mm_shuffle_ps` sont nécessaires.

Dans le fichier `tests/avg3_v2.cpp` implémenter l'optimisation suivante : au lieu de recharger `ve` et `vprev`, on va les copier dans les anciens registres et charger uniquement `vnext`. (Cette optimisation s'appelle la rotation de registre.)

3.1 Commenter les gains obtenus par les deux versions.

4 Filtre Moyenneur, Part II

Écrire les fonctions `vfloat32_left_2` et `vfloat32_right_2` afin d'ajouter un nouveau test `tests/avg5_v2.cpp` (en partant de `tests/avg3_v2.cpp`) (relancer `cmake ..` pour prendre en compte ce nouveau test) qui calcule le filtre moyenneur suivant : `r[i] = (v[i-2] + v[i-1] + v[i] + v[i+1] + v[i+2]) / 5.f;` (sans la gestion des bords et avec la rotation de registre).

5 Fractale de Mandelbrot

On cherche à optimiser le calcul de la fractale de MANDELBROT https://fr.wikipedia.org/wiki/Fractale

Une implémentation est fournie dans le fichier `tests/fractal_mandelbrot_v0`. Il s'agit du code séquentiel, non optimisé et avec des flottants double précision.

Écrire le test `tests/fractal_mandelbrot_v1` qui parallélise le calcul en utilisant OpenMP.

5.1 Pourquoi ce code est parallélisable ?

5.2 Tester différents `schedule` OpenMP. Quel est le plus rapide et pourquoi ? Expliquer le fonctionnement du `schedule` le plus rapide et celui par défaut.

Écrire le test `tests/fractal_mandelbrot_v2` qui transforme les calculs flottants double précision (`double`) en calculs flottants simple précision (`float`).

5.3 Commenter le gain obtenu. Dans quels conditions ce genre de transformations améliore les performances et de combien ?

Écrire le test `tests/fractal_mandelbrot_v3` qui définit les constantes dès que possible et transforme les divisions en multiplications.

5.4 Commenter le gain obtenu.

Écrire le test `tests/fractal_mandelbrot_v4` qui réécrit les calculs pour préparer la vectorisation.
Les calculs doivent être de la forme `r = a + b;`.
Extraire la condition `z_r_2 + z_i_2 < 4` de la boucle `while` en la plaçant dans un `if` qui exécute un `break`.

5.5 Comment ce code peut être vectorisé ? (Quelles sont les principales difficultés ?)

Vectoriser le code dans le test `tests/fractal_mandelbrot_v4`.
Les itérations sur `y` sont traités 4 par 4 (sans la gestion du bord).
Utiliser les instructions SSE `_mm_add_ps`, `_mm_sub_ps` et `_mm_mul_ps` pour les calculs.
Utiliser `_mm_cmple_ps` (qui retourne le `mask`) pour la comparaison et `_mm_movemask_ps` pour la condition du `if` (qui exécute le `break`).
Maintenir les 4 indices `i` (un par itération `y`) dans un `vfloat32_t` (même si cela aurait du être fait dans des entiers non signés) et incrémenter les en utilisant le `mask` après avoir obtenu 1 ou 0 grâce à `_mm_and_ps` et le vecteur SIMD qui ne contient que des 1.

5.6 Commenter le gain obtenu.

5.7 Tester sans OpenMP et commenter le gain obtenu.

6 Multiplication De Matrices (Bonus)

En partant de code de la multiplication de matrices (version `ikj`) du TP 1, optimiser le code en utilisant les instructions SSE. (Ne pas oublier de changer l'allocateur du conteneur pour aligner les données sur 16 bits.)