TP 8 : 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=2016_Et4_archi_parallele. 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` ?

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 Pourquoi le gain naivement espéré par la version SSE n'est pas atteint ?

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 Pourquoi le gain obtenu par la version SSE est-il meilleur ?

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

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 Multiplication De Matrices (Bonus)

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