Introduction à la programmation SIMD

Vous pouvez trouver les prototypes de toutes les fonction AVX ainsi que leur coût en cycles sur le site d'Intel : `http://software.intel.com/sites/landingpage/IntrinsicsGuide`.
Il suffit de filtrer les types d'instructions AVX, AVX2 et FMA à gauche de la page. Pour compiler, utilisez la commande suivante :
g++ -O2 -mavx2 -mfma fichier.cpp -o fichier

1 Copier un tableau

Le but de cet exercice est d'apprendre les bases du calcul SIMD en l'appliquant à la copie d'un tableau dans un autre.

1. Allouer deux tableaux A et B de flottants de taille N, puis initialiser A tel que $A[i] = i$. Veiller à ce que le tableau soit aligné par 32 octet pour pouvoir utiliser des instructions AVX alignées.

2. Ecrire une fonction non-vectorisé qui copie le contenue de A dans B.

3. Ecrire une deuxième fonction vectorisée qui effectue la même opération.

4. Ecrire une troisième fonction qui fait un déroulement de la boucle par un facteur de 4 (c'est à dire, qui éffectue 4 itérations de la version précédente dans une seule itération.

5. Comparér le temp d'exécution total de chaque version pour 1000 appels consécutifs. Va-t-il plus vite la version déroulée pour $N = 1024$ (Regarder la bande-passante du L1 du processeur pour load et store sur `https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake`)?

2 Produit scalaire

Le but de cet exercice est de calculer le produit scalaire de deux vecteurs avec vectorisation.

1. Allouer deux tableaux A et B de floatants taille N (divisible par 8), puis les initialiser.

2. Ecrire une fonction non-vectorisé qui calcule le produit scalaire de A et B.

3. Ecrire une deuxième fonction vectorisée qui effectue la même opération.

4. Ecrire une troisième fonction qui fait un déroulement de la boucle par un facteur de 2 et 4 (c'est à dire, qui éffectue 2 ou 4 itérations de la version précédente dans une seule itération. Combien de cycles vous attendez à passer par itération? Trouvez les "trous" dans le pipeline du processeur et essayez de réorganiser les instructions tel que le nombre de cycles attendus par itération décroit.

5. Comparér le temp d'exécution total de chaque version pour 1000 appels consécutifs.

6. Essayer de vectoriser le code automatiquement en rajoutant l'option de compilation "-ftree-vectorise" et tester les performances.

3 Calcul de filtres en SIMD

1. La fonction **vect_left1** prend deux __m128 (a, b, c, d et e, f, g, h par exemple) et renvoie un __m128 contenant les valeurs du premier registre décalées vers la gauche et en ajoutant la première valeur du deuxième registre (on obtient b, c, d, e dans notre exemple).
   Pour faire cette opération, il faudra deux **_mm_shuffle_ps**. Réalisez cette fonction.

2. La fonction **vect_right1** prend deux __m128 (a, b, c, d et e, f, g, h par exemple) et renvoie un __m128 contenant les valeurs du deuxième registre décalées vers la droite et en ajoutant la dernière valeur du premier registre (on obtient d, e, f, g dans notre exemple).
   Pour faire cette opération, il faudra deux **_mm_shuffle_ps**. Réalisez cette fonction. sont nécessaires.

3. Soit la fonction **vectoravg3_simd** permettant de réaliser un filtre moyenneur 1D tel que $m1 = \frac{1}{3}[1 \quad 1 \quad 1]$. Écrivez le code SIMD pour cette fonction en se servant des deux fonctions précédentes. Il n'est pas demandé de faire une gestion des bords.

4. Il existe des méthodes permettant d'optimiser ce calcul de filtre. Essayez de trouver cette optimisation et implémentez la dans le corps de la fonction **vectoravg3_rot_simd**. Que pensez vous des performances?

4 Inversion d'un tableau

Écrire une fonction qui effectue l'inversion d'un tableau d'entiers A avec SIMD. La taille du tableau sera toujours un multiple de 16 pour simplifier le travail. Pour ce faire, utiliser la fonction AVX **_mm256_permutevar8x32_epi32(__m256i a, __m256i idx)** afin de renverser un vecteur de 8 entièrs, puis parcourir le tableau en appliquant cette fonction des deux côtés pour l'inverser.

5 Produit matrice-vecteur

Écrire un programme qui effectue le produit d'une matrice $A \in \mathbb{R}^{N\times N}$ avec un vecteur $x \in \mathbb{R}^N$ dans un autre vecteur $y = Ax, y \in \mathbb{R}^N$. Le calcul s'effectue comme la suite:

$$y_i = \sum_{j=1}^{N} A_{ij} x_j$$

Vous pouvez aussi prendre le code OpenMP non-vectorisé du dernier TP, puis le vectoriser. Y a-t-il une amélioration des performances? Pour quelles valeurs N est-il le cas?