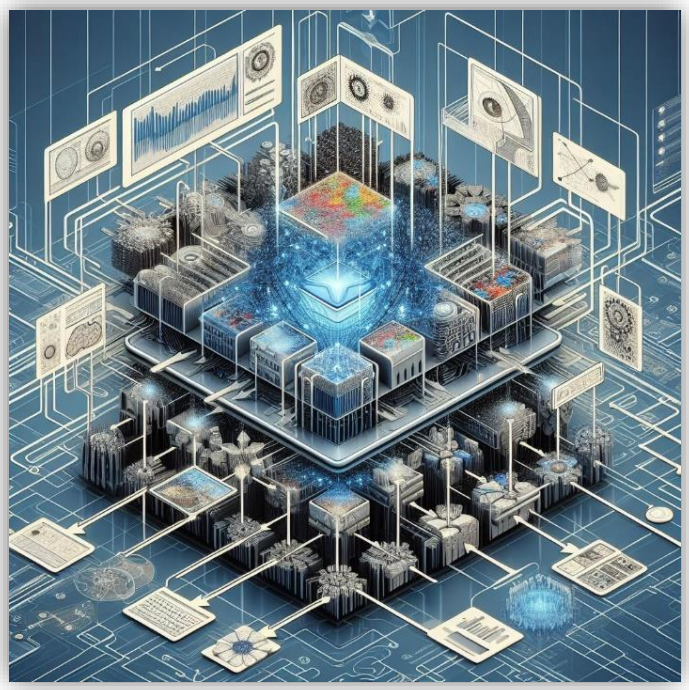

Sommaire

Spécifications de ma machine	
Processor	AMD Ryzen 7 3750H with Radeon Vega Mobile Gfx 2.30 GHz
Installed RAM	16.0 GB (13.9 GB usable)
System type	64-bit operating system, x64-based processor
Windows	Windows 11 Home

Exercice 1

1- Type de données utilisé :

Le code utilise principalement le type de données __m256 qui est un type de données stocké sur 256 bits, spécifique à AVX (Advanced Vector Extensions). Il représente un vecteur de 8 nombres à virgule flottante de 32 bits. De plus, un tableau de float est utilisé pour stocker les valeurs initiales.

2- Fonctions utilisées, il y a trois fonctions AVX :

- _mm256_load_ps(values) :

  Cette fonction est utilisée pour charger huit valeurs de type float consécutives à partir d'une adresse mémoire alignée dans un registre __m256. L'argument values est un pointeur vers le premier élément d'un tableau de float. Les valeurs sont chargées dans le registre dans l'ordre de la mémoire, c'est-à-dire que la première valeur chargée est à l'index 0 du registre __m256.

- _mm256_add_ps(a, b) :

  Cette fonction est utilisée pour effectuer une addition élément par élément de deux vecteurs __m256. Les arguments a et b sont des vecteurs __m256. Pour chaque index de 0 à 7, l'élément à cet index dans le vecteur résultant est la somme des éléments correspondants dans a et b.

- _mm256_sqrt_ps(a) :

  Cette fonction est utilisée pour calculer la racine carrée de chaque élément d'un vecteur __m256. L'argument a est un vecteur __m256. Pour chaque index de 0 à 7, l'élément à cet index dans le vecteur résultant est la racine carrée de l'élément correspondant dans a.

3- Résultat obtenu :

➔ Ici on charge un __m256 avec le paquet {1, 2, 3, 4, 5, 6, 7, 8}
➔ Puis l'addition de ce __m256 avec lui-même donne {2, 4, 6, 8, 10, 12, 14, 16}
➔ Enfin la racine carrée du résultat de l'addition donne {1.41421, 2, 2.44949, 2.82843, 3.16228, 3.4641, 3.74166, 4}

Exercice 2

Nous allons calculer le travail, l'accélération et l'efficacité. Pour cela, rappelons les définitions :

1- Travail :

Le travail est le temps total que tous les processeurs passent à exécuter le programme. Il est calculé comme le produit du temps d'exécution de la version séquentielle et du nombre de registres AVX.

2- Accélération :

L'accélération est une mesure de l'amélioration des performances obtenue en passant d'une version séquentielle à une version parallèle d'un algorithme. Elle est calculée en divisant le temps d'exécution de la version séquentielle par le temps d'exécution de la version vectorielle.

3- Efficacité :

L'efficacité est une mesure de l'utilisation des ressources de calcul. Elle est calculée en divisant le temps d'exécution de la version séquentielle par le produit du temps d'exécution de la version vectorielle et du nombre de registres AVX.

■ Testons maintenant avec plusieurs nombres de registres AVX :

Nombre de registres AVX	Temps d'exécution de la version séquentielle (µs)	Temps d'exécution de la version vectorielle (µs)	Travail	Accélération	Efficacité
1 024	11	3	8*3 = 24	11/3 = 3.67	11/24 = 0.45833
32 768	332	84	672	3.95	0.49404
65 536	787	394	3 152	2	0.24968
131 072	1 552	570	4 560	2.72	0.34035
8 388 608	86 594	36 320	290 560	2.38	0.29802
15 352 120	160 000	65 210	521 680	2.45	0.30670
30 000 000	327 000	132 000	1 056 000	2.47	0.30965
100 000 000	1 088 000	465 000	3 720 000	2.34	0.29247

8 : le nombre de «« threads »» utilise avec AVX.

Lorsque le nombre de registres AVX est inférieur à 65 536 (2^16), l'efficacité est en moyenne supérieure à 0.4, ce qui indique une utilisation efficace des ressources de calcul disponibles. Cela se traduit par une réduction significative du temps d'exécution par rapport à la version séquentielle.

Cependant, lorsque le nombre de registres AVX dépasse 65 536, l'efficacité diminue, passant en dessous de 0.4. Cela suggère que l'algorithme devient moins efficace à mesure que le nombre de registres AVX augmente. En d'autres termes, l'augmentation du nombre de registres AVX n'entraîne pas une réduction proportionnelle du temps d'exécution.

Il est également intéressant de noter que malgré la diminution de l'efficacité, l'accélération reste relativement constante, oscillant autour de 2.4 pour les nombres de registres AVX supérieurs à 65 536. Cela pourrait indiquer que bien que l'efficacité diminue, l'algorithme parallèle est toujours capable de fournir une accélération significative par rapport à la version séquentielle.

Exercice 3

Voici les résultats de l'efficacité obtenus en fonction du nombre de threads et de registres AVX :

Nombre de threads / Nombre de registres AVX	1	2	4	8
1 024	0.45	0.45	0.40	0.41
4 096	0.49	0.40	0.38	0.38
32 768	0.24	0.17	0.17	0.15
65 536	0.34	0.30	0.28	0.28
131 072	0.29	0.22	0.22	0.22
8 388 608	0.30	0.20	0.20	0.20
15 352 120	0.30	0.21	0.22	0.21
30 000 000	0.29	0.25	0.22	0.22

L'augmentation du nombre de threads n'optimise pas le calcul. En réalité, plus le nombre de threads augmente, plus l'efficacité diminue. On attribue cela au fait que le calcul se fait rapidement (dans l'ordre des microsecondes), ce qui rend difficile l'optimisation avec des threads. De plus, le lancement des threads et l'attente de leur synchronisation prennent du temps, ce qui réduit l'efficacité.

Cependant, il est important de noter que l'efficacité ne diminue pas de manière linéaire avec l'augmentation du nombre de threads. Par exemple, l'efficacité reste relativement stable lorsque le nombre de threads passe de 2 à 4, puis à 8, pour un nombre de registres AVX de 131 072 à 30 000 000. Cela suggère que, bien que l'efficacité globale diminue avec l'augmentation du nombre de threads, l'impact est moins prononcé pour un nombre plus élevé de registres AVX.

Exercice 4

Lorsqu'on exécute le programme de cet exercice avec un grand nombre de valeurs, on remarque que l'approche séquentielle donne de mauvais résultats. Pour être plus précis, le problème vient à partir de 2 097 153 valeurs. Le problème vient du fait que l'on calcule plus de 16 777 216 floats, qui est la limite d'incrémentation d'un float en C++ et dans de nombreux autres langages. Cette valeur est égale à 2^{24}, or ici notre float a sa mantisse codée sur 23 bits, ce qui fait que la mantisse se remplit de 0 et que le float reste bloqué à la valeur 16 777 216 sans pouvoir s'incrémenter.

On remarque que si on donne plus de 16 777 216 valeurs à notre programme, alors on a le même problème avec l'approche vectorielle pour la même raison car un __m256 contient en interne des floats.

Exercice 5

Un tableau comparatif des performances de la version vectorielle du produit matrice-vecteur en utilisant les instructions AVX avec une version séquentielle de base. Voici les résultats que j'ai obtenus :

Nombre de registres AVX	Temps d'exécution de la version séquentielle (µs)	Temps d'exécution de la version vectorielle (µs)	Accélération	Efficacité
10	7	1	7/1 = 7	7/8*1= 0.87
16	14	2	7	0.87
22	33	4	8.25	1.03
28	41	8	5.12	0.64
34	63	11	5.72	0.71
40	84	19	4.42	0.55
55	195	36	5.41	0.67
500	12 950	3 476	3.72	0.46
2 000	208 000	58 549	3.55	0.44
4 000	874 000	244 000	3.58	0.44
5 000	1 327 000	404 000	3.28	0.41

55 : à partir de cette donnée dans le tableau que le calcul devient faux, si on veut être plus précis c'est à partir de 41.

Les résultats montrent clairement que la version vectorielle est nettement plus rapide que la version séquentielle, avec une accélération allant de 3.28 à 8.25. Cela confirme l'efficacité des instructions AVX pour paralléliser les calculs et accélérer les opérations sur les données.

L'efficacité, varie de 0.41 à 1.03. Ces valeurs indiquent que la version vectorielle utilise de manière relativement efficace les ressources de calcul disponibles. Ainsi, on peut dire que pour implémenter le calcul d'un produit entre une matrice et un vecteur, il est intéressant d'utiliser un algorithme vectoriel pour calculer plus rapidement.

Il est important de noter que l'efficacité des instructions AVX peut dépendre de nombreux facteurs, y compris la nature spécifique des calculs, la manière dont le code est écrit et optimisé, et les caractéristiques spécifiques du matériel sur lequel le code est exécuté.