Rapport – Travaux Pratique 2

Sommaire

[Exercice 1 1](#_Toc93249898)

[Exercice 2 2](#_Toc93249899)

[Exercice 3 2](#_Toc93249900)

[Exercice 4 3](#_Toc93249901)

[Exercice 5 3](#_Toc93249902)

# 

# Exercice 1

Dans le code on utilise le type \_\_m256. Ce type est stocké sur 256 bits et permet de manipuler des instructions AVX. Pour charger une variable de ce type, il faut un paquet de 8 réels simples précisions (float en C++).

Plusieurs opérations sont disponibles sur les \_\_m256, dans cet exercice il y a l’addition entre deux \_\_m256 et la racine carrée d’un \_\_m256.

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Nombre de registres AVX | Temps d’exécution de la version séquentielle (µs) | Temps d’exécution de la version vectorielle (µs) | Accélération | Travail | Efficacité |
| 1 024 | 17 | 4 | 17 / 4 = 4.25 | 8 x 4 = 32 | 1 / 8 \* 4.25 = 17 / 32  = 0.53 |
| 4 096 | 68 | 17 | 4.85 | 136 | 0.5 |
| 32 768 | 547 | 137 | 3.99 | 1 096 | 0.49 |
| 65 536 | 1 098 | 318 | 3.45 | 2 544 | 0.43 |
| 131 072 | 2 220 | 1 181 | 1.87 | 9 448 | 0.23 |
| 8 388 608 | 145 000 | 77 895 | 1.86 | 623 160 | 0.23 |

Lorsque le nombre de registres AVX est inférieur à 216 l’efficacité est en moyenne égale à 50%, par contre quand le nombre de registre est plus grand que 216 l’efficacité est deux fois moins grande.

# Exercice 3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nombre de threads / Nombre de registres AVX | 1 | 2 | 4 | 8 |
| 1 024 | 0.13 | 0.13 | 0.11 | 0.12 |
| 4 096 | 0.19 | 0.15 | 0.14 | 0.14 |
| 32 768 | 0.21 | 0.17 | 0.17 | 0.15 |
| 65 536 | 0.19 | 0.18 | 0.16 | 0.16 |
| 131 072 | 0.20 | 0.13 | 0.13 | 0.13 |
| 8 388 608 | 0.23 | 0.13 | 0.13 | 0.13 |

Le nombre de threads n’optimise pas le calcul, bien au contraire plus on lance de threads, plus l’efficacité diminue. Cela s’explique par le fait que le calcul se fait rapidement (dans l’ordre des microsecondes) et qu’on ne peut pas l’optimiser avec des threads. De plus lancer les threads et attendre leur synchronisation prend du temps et fait baisser l’efficacité.

# Exercice 4

Lorsqu’on exécute le programme de cet exercice avec un grand nombre de valeurs, on remarque que l’approche séquentiel donne de mauvais résultats. Pour être plus précis c’est à partir de 2 097 153 valeurs que l’on a un problème. 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 beaucoup d’autres langages. Cette valeur est égal 2^24, or ici notre float à sa mantisse codé sur 23 bits, ce qui fait que la mantisse se remplie 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 vectoriel pour la même raison car un \_\_m256 contient en interne des floats.

# Exercice 5

|  |  |  |
| --- | --- | --- |
| Nombre de registres AVX | Accélération | Efficacité |
| 10 | 7 | 0.88 |
| 16 | 5.75 | 0.72 |
| 22 | 7.4 | 0.93 |
| 28 | 6.78 | 0.85 |
| 34 | 6.5 | 0.82 |
| 40 | 6.4 | 0.8 |

On remarque que l’accélération et l’efficacité de l’algorithme vectoriel sont important, 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.