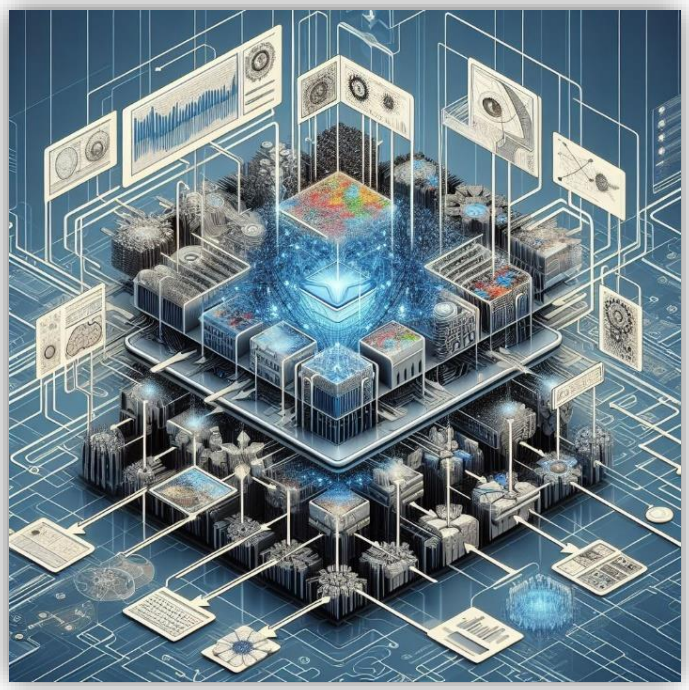

## Sommaire



| 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

| Taille des tableaux | Moyenne temps d'exécution d'un MAP pour calculer le carré des éléments d'un tableau (µs) | Moyenne temps d'exécution d'un MAP pour calculer l'addition des éléments de deux tableaux (µs) |
|---|---|---|
| **1 024** | 18 | 10 |
| **16 000** | 51 | 45 |
| **200 000** | 115 | 95 |
| **4 000 000** | 2 087 | 2 835 |
| **80 000 000** | 38 146 | 49 861 |
| **200 000 000** | 92 652 | 122 000 |
| **300 000 000** | 145 000 | 184 000 |
| **500 000 000** | 251 000 | 306 000 |

Pour des tableaux de petite taille (1 024 à 200 000 éléments), le temps d'exécution moyen pour calculer l'addition des éléments de deux tableaux est effectivement plus rapide que pour calculer le carré des éléments d'un tableau. Cependant, lorsque la taille des tableaux augmente (4 000 000 à 500 000 000 éléments), le temps d'exécution pour le calcul du carré devient plus rapide que pour l'addition.

Je pense que cela pourrait s'expliquer par le fait que le calcul du carré s'effectue sur un seul tableau, tandis que le calcul de l'addition nécessite de travailler sur deux tableaux distincts. Lorsque la taille des tableaux est petite, la différence de temps d'exécution entre les deux opérations est négligeable. Cependant, à mesure que la taille des tableaux augmente, le coût de l'accès mémoire pour deux tableaux dans le calcul de l'addition peut devenir significatif, ce qui pourrait expliquer pourquoi le calcul du carré devient plus rapide.

## Exercice 2

| Taille du tableau | Moyenne du temps d'exécution d'un REDUCE pour calculer la somme des éléments d'un tableau (µs) | Moyenne du temps d'exécution d'un REDUCE (*transform* puis *reduce*) pour calculer la somme des carrés des éléments d'un tableau (µs) | Moyenne du temps d'exécution d'un REDUCE (*transform_reduce*) pour calculer la somme des carrés des éléments d'un tableau (µs) |
|---|---|---|---|
| **1 024** | 9 | 13 | 5 |
| **16 000** | 17 | 28 | 10 |
| **200 000** | 25 | 170 | 36 |
| **4 000 000** | 1 316 | 8 207 | 969 |
| **80 000 000** | 20 448 | 614 000 | 20 097 |
| **200 000 000** | 50 943 | 1 700 000 | 49 516 |
| **300 000 000** | 76 282 | 3 530 000 | 73 942 |
| **500 000 000** | 126 000 | 4 038 000 | 126 000 |

4 000 000 : à partir de cette valeur le calcul devient faux pour être plus précis c'est à partir de 2 097 153, probablement à cause du overflow avec les floats comme dans le TP précèdent.

Les temps d'exécution du REDUCE pour calculer la somme des éléments d'un tableau et celui pour calculer la somme des carrés avec un unique transform_reduce, sont à peu près équivalents. Parfois l'un est plus rapide que l'autre et inversement.

Pour le REDUCE non optimisé pour calculer la somme des carrés des éléments d'un tableau avec l'utilisation d'un transform puis d'un reduce, j'observe que le temps d'exécution est environ 10 fois plus lent que les deux autres REDUCE. C'est normal puisque je parcours deux fois le tableau et que j'additionne le temps d'exécution d'un MAP pour calculer le carré des éléments d'un tableau et d'un REDUCE pour calculer la somme des éléments d'un tableau.

## Exercice 3

| Taille des tableaux | Moyenne du temps d'exécution d'un MAP avec découpage fixe par bloc pour calculer le carré des éléments d'un tableau (µs) | Moyenne du temps d'exécution d'un MAP avec découpage fixe par bloc pour calculer l'addition des éléments de deux tableaux (µs) |
|---|---|---|
| **1 024** | 29 | 26 |
| **16 000** | 36 | 37 |
| **200 000** | 76 | 84 |
| **4 000 000** | 2 026 | 2 901 |
| **80 000 000** | 37 673 | 53 101 |
| **200 000 000** | 94 862 | 127 000 |
| **300 000 000** | 144 000 | 187 000 |
| **500 000 000** | 257 000 | 334 000 |

Pour des tableaux de petite taille, le temps d'exécution est significativement plus long avec ma version de la fonction transform par rapport à la version standard de C++. Cependant, lorsque la taille des valeurs augmente, on observe que le temps d'exécution pour le calcul du carré et pour l'addition devient comparable à celui de la fonction de la bibliothèque standard de C++.

Cela suggère que ma version de la fonction transform avec un découpage fixe par bloc est bien adaptée pour les grands tableaux. Pour les petits tableaux, la fonction transform de la librairie standard reste la meilleure option car elle est optimisée pour toute taille de tableau. Je suppose que cette fonction choisit automatiquement le nombre de threads à lancer en fonction de la taille du tableau. Ainsi, si le tableau est petit, la fonction lance un petit nombre de threads et si le tableau est grand, alors la fonction lance le maximum de threads disponible.

Il serait intéressant d'explorer le découpage fixe par modulo. Bien que je n'aie pas testé cette méthode, elle pourrait offrir des avantages en termes de distribution de charge, surtout dans les cas où la charge de travail n'est pas uniformément répartie. Cela pourrait potentiellement améliorer les performances, en particulier pour les grands tableaux, le code pourrait ressembler à ça :

```
        ================= first version =================
for (int i = 0; i < nbTasks; ++i) {
    futures.emplace_back([i, nbTasks, fullSize, aBegin, oBegin, functor] () {
        for (int iter = i ; iter < fullSize ; iter += nbTasks)
            oBegin[iter] = functor(aBegin[iter]);
        }
    )
};
        ================= second version =================
for (int i = 0; i < nbTasks; ++i) {
    futures.emplace_back(
            [i, nbTasks, fullSize, aBegin, bBegin, oBegin, functor] () {
                for (int iter = i; iter < fullSize; iter += nbTasks)
                    oBegin[iter] = functor(aBegin[iter], bBegin[iter]);
            }
    )
};
```

## Exercice 4

| Taille des tableaux | Moyenne du temps d'exécution d'un GATHER avec découpage fixe par bloc (µs) | Moyenne du temps d'exécution d'un SCATTER avec découpage fixe par bloc (µs) |
|---|---|---|
| **1 024** | 16 | 32 |
| **16 000** | 51 | 54 |
| **200 000** | 69 | 121 |
| **4 000 000** | 3 300 | 4 299 |
| **80 000 000** | 61 802 | 87 710 |
| **200 000 000** | 153 000 | 218 000 |
| **300 000 000** | 228 000 | 316 000 |
| **500 000 000** | 528 000 | 619 000 |

Pour des tableaux de petite taille, on peut observer que le temps d'exécution moyen pour l'opération SCATTER est systématiquement plus long que pour l'opération GATHER, quel que soit la taille du tableau. Cela pourrait s'expliquer par le fait que l'opération SCATTER nécessite plus de coordination entre les threads pour éviter les conflits d'accès en écriture.

De plus, on peut constater que le temps d'exécution augmente avec la taille du tableau pour les deux opérations. Cela est attendu car plus le tableau est grand, plus il y a d'éléments à traiter.

Enfin, il est intéressant de dire que l'écart de performance entre GATHER et SCATTER semble augmenter avec la taille du tableau. Cela pourrait montrer que l'opération SCATTER est plus sensible à la taille du tableau que l'opération GATHER.

## Exercice 5

| Taille des tableaux | Moyenne du temps d'exécution d'un REDUCE (avec le ThreadPool) pour calculer la somme des éléments d'un tableau (µs) | Moyenne du temps d'exécution d'un REDUCE (avec la méthode fourni par C++) pour calculer la somme des éléments d'un tableau (µs) [données du Ex2] |
|---|---|---|
| **1 024** | 25 | 9 |
| **16 000** | 34 | 17 |
| **200 000** | 63 | 25 |
| **4 000 000** | 1 230 | 1 316 |
| **80 000 000** | 21 498 | 20 448 |
| **200 000 000** | 45 922 | 50 943 |
| **300 000 000** | 70 414 | 76 282 |
| **500 000 000** | 121 000 | 126 000 |

Dans les résultats que j'ai obtenus, deux tendances distinctes se dégagent en fonction de la taille du tableau.

Pour les petits tableaux (de 1 024 à 80 000 000 éléments), le temps d'exécution de ma version de reduce est nettement plus long que celui de la version standard de C++. Donc ma version de reduce n'est pas optimisée pour les petits tableaux. Cela pourrait être dû à l'overhead de la création et de la gestion des threads, qui est plus perceptible sur les petits tableaux où le temps de calcul est déjà très court.

Pour les grands tableaux (de 200 000 000 à 500 000 000 éléments), le temps d'exécution de ma version de reduce devient comparable à celui de la version standard de C++ (voir plus performant). Donc ma version de reduce est bien adaptée pour les grands tableaux. En effet, pour ces tailles de tableau, l'overhead de la gestion des threads est amorti par le temps de calcul plus long, ce qui rend ma version de reduce plus intéressante par rapport à la version standard.