



PROJET HPC

Résolution des systèmes d'équations linéaires avec la méthode de Gausse-Jordan

Elaboré par :

Borhaneddine Hamadou (SIQ2)

Mohamed Zakaria Miloudi (SIQ2)

Sous la supervision de :

Dr.HAICHOIR Amina

Selma

1 Introduction

La résolution des systèmes linéaires continue de jouer un rôle important dans le calcul scientifique. Les problèmes à résoudre sont souvent de très grande taille, de sorte que des ressources informatiques importantes, en des "supercomputers" particuliers avec une grande mémoire partagée ou des systèmes informatiques parallèles massifs à mémoire distribuée sont nécessaires pour les résoudre.

Ce travail traite deux solutions, que nous avons proposées, pour la mise en œuvre d'un algorithme parallèle d'une méthode directe pour résoudre les systèmes linéaires à savoir la méthode de Gauss-Jordan, une solution avec OpenMP et une autre avec CUDA C.

2 L'algorithme de Gauss-Jordan

On veut résoudre un système d'équations $A.X = B$, où B est un vecteur fixé, et X le vecteur inconnu. On crée un tableau à n lignes et $n + 1$ colonnes en bordant la matrice A par le vecteur B . Ainsi, on utilisera la matrice augmentée suivante :

$$(A|B) = \left(\begin{array}{ccc|c} a_{11} & \dots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{n1} & \dots & a_{nn} & b_n \end{array} \right) \quad (1)$$

La transformation de Gauss-Jordan consiste à transformer ce système en un système équivalent dont le bloc gauche est une matrice diagonale, c'est-à-dire qu'il faut modifier la matrice $(A | B)$ en utilisant les propriétés de l'algorithme. Soit :

- l_i^k : la ligne i de la matrice A à l'itération k .
- a_{ij}^k : le scalaire a_{ij} de la matrice A à l'itération k .

L'algorithme de Gauss-Jordan est le suivant :

Pour k allant de 1 à n :

Pour i allant de 1 à n :

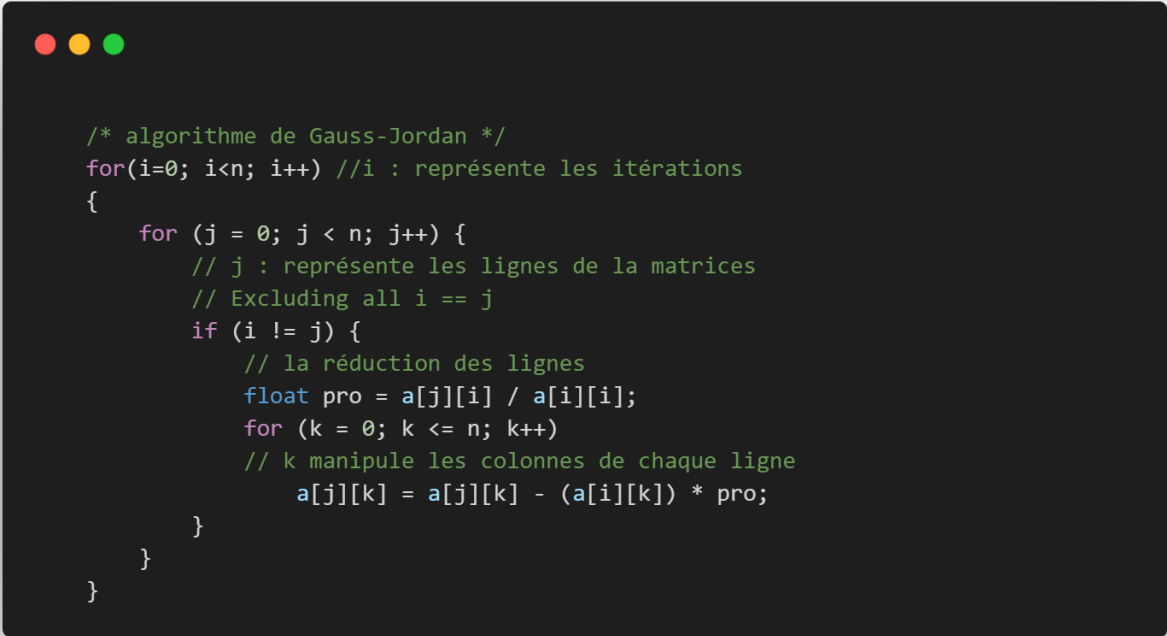
If ($i \neq k$) :

$$l_i^k \leftarrow l_i^{k-1} - a_{ik}^{k-1} \times l_k^k$$

L'algorithme se compose de n itérations, dans chaque itération k on va annuler tous les coefficients de la colonne k sauf l'élément diagonale, et donc dans chaque itération on va transformer $n-1$ lignes de la matrice augmentée $(A|B)$.

3 Implémentation séquentielle de l'algorithme

Voici un algorithme séquentiel en C, qui permet de résoudre un système d'équations linéaires par la méthode de Gauss-Jordan :



```
/* algorithme de Gauss-Jordan */
for(i=0; i<n; i++) //i : représente les itérations
{
    for (j = 0; j < n; j++) {
        // j : représente les lignes de la matrices
        // Excluding all i == j
        if (i != j) {
            // la réduction des lignes
            float pro = a[j][i] / a[i][i];
            for (k = 0; k <= n; k++)
                // k manipule les colonnes de chaque ligne
                a[j][k] = a[j][k] - (a[i][k]) * pro;
        }
    }
}
```

Figure 1: Programme séquentielle : Algorithme de Gauss-Jordan

Cet algorithme fonctionne parfaitement avec des valeurs petites de n , mais

plus n est grand, plus il est considérablement moins efficace.



Remarque

La parallélisation de l'algorithme se fait au niveau des opérations de transformation des lignes dans chaque itération. On fait, on peut pas paralléliser les itérations car chaque itération k besoin de la matrice résultat de l'itération $(k-1)$, donc les itérations doivent s'exécuter dans l'ordre.

4 Implémentation parallèle de l'algorithme

4.1 Implémentation avec OpenMP

```
1      start= omp_get_wtime();
2      for(i=0; i<n; i++)
3      {
4          #pragma omp parallel for shared(i) private(j, k)
5          for (j = 0; j < n; j++) {
6              // Excluding all i == j
7              if (i != j) {
8                  // Converting Matrix to reduced row
9                  // echelon form(diagonal matrix)
10                 double pro = a[j][i] / a[i][i];
11                 for (k = 0; k <= n; k++)
12                     a[j][k] = a[j][k] - (a[i][k]) * pro;
13             }
14         }
15     }
16     end = omp_get_wtime();
```

Figure 2: Programme Parallèle avec OpenMP : Algorithme de Gauss-Jordan

Dans ce cas, nous avons distribué les itérations de cette boucle (qui représente les lignes) sur les threads disponibles, donc chaque thread est chargée de faire les transformations d'une ligne, du coup, plusieurs lignes vont être transformées à la fois, ce qui permet d'accélérer l'exécution du programme.

4.2 Implémentation avec CUDA

```
1 //Le code du kernel
2 __global__ void gauss(double *a, int n, int currentItem)
3 {
4     int j = blockIdx.x, k = threadIdx.x;
5     if(j != currentItem){
6         float pro = a[j * (n+1) + currentItem] / a[currentItem*(n+1) + currentItem];
7         a[j*(n+1)+k] = a[j*(n+1)+k] - (a[currentItem*(n+1)+k]) * pro;
8     }
9 }
```

Figure 3: Programme Parallèle avec CUDA : Le Kernel

Avec le GPU, toutes les opérations d'une itération auront lieu en un seul cycle, chaque block est chargé de faire la transformation d'une ligne, et chaque thread dans le block fait une opération sur une case. Il suffit de prendre un nombre de blocks égale au nombre de ligne n , et un nombre de threads égale au nombre de colonnes $(n+1)$, voici l'appel du kernel au niveau du programme principal :

```
1     cudaEvent_t start, stop;
2     cudaEventCreate(&start);
3     cudaEventRecord(start,0);
4
5     for(i=0; i<n; i++){
6         gauss<<<n, n+1>>>(cuda_a, n, i);
7     }
8
9     cudaEventCreate(&stop);
10    cudaEventRecord(stop,0);
11    cudaEventSynchronize(stop);
12    cudaEventElapsedTime(&elapsedTime, start,stop);
```

Figure 4: Programme Parallèle avec CUDA : Appel du kernel

A chaque itération de l'algorithme, On fait un appel au kernel.

5 Tests et résultats

Afin de comparer entre les temps d'exécutions des trois algorithmes, nous avons choisi un système de 1000 équations à 1000 inconnus (matrice remplie aléatoirement) pour bien remarquer les différences, et voici les résultats obtenus :

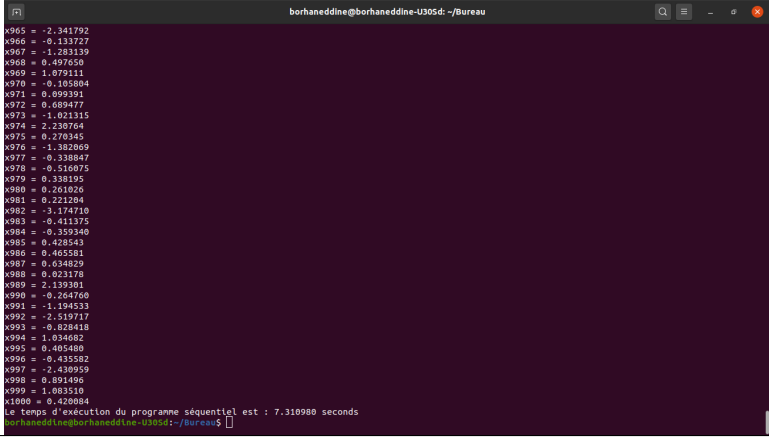
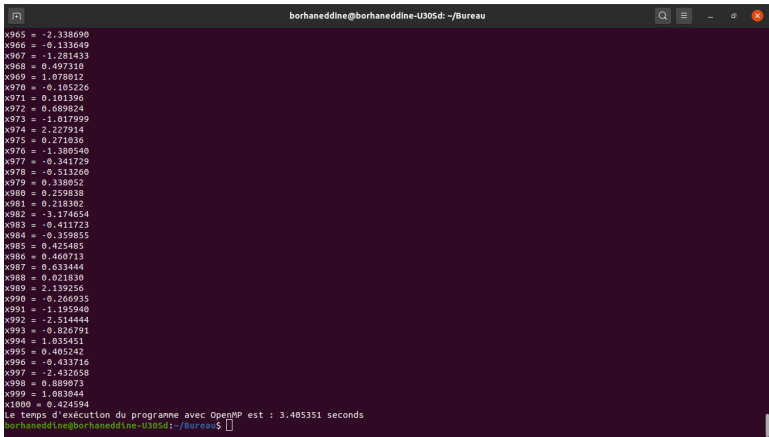

Algorithme	Résultat	Temps d'exécution
Séquentiel	 <pre> x965 = -2.341792 x966 = -0.133727 x967 = -1.283139 x968 = 0.497650 x969 = 1.079111 x970 = -0.105804 x971 = 0.099391 x972 = 0.689477 x973 = -1.021315 x974 = 2.230764 x975 = 0.210345 x976 = -1.302869 x977 = -0.338847 x978 = -0.516075 x979 = 0.330195 x980 = 0.261026 x981 = 0.222184 x982 = -3.174710 x983 = -0.411375 x984 = -0.359340 x985 = 0.428543 x986 = 0.465581 x987 = 0.614829 x988 = 0.023178 x989 = 2.119301 x990 = -0.264768 x991 = -1.194533 x992 = -2.519717 x993 = 0.028418 x994 = 1.034682 x995 = 0.405400 x996 = -0.435582 x997 = -2.430959 x998 = 0.891496 x999 = 1.083510 x1000 = 0.420084 Le temps d'exécution du programme séquentiel est : 7.31998 seconds borhaneddine@borhaneddine-U305d: ~/Bureau\$ </pre>	7.31(s)
OpenMP	 <pre> x965 = -2.338690 x966 = -0.133649 x967 = -1.281433 x968 = 0.497310 x969 = 1.078012 x970 = -0.105226 x971 = 0.101396 x972 = 0.689824 x973 = -1.017999 x974 = 2.227914 x975 = 0.217036 x976 = -1.380540 x977 = -0.541729 x978 = -0.513509 x979 = 0.338052 x980 = 0.259838 x981 = 0.218302 x982 = -3.174654 x983 = -0.411723 x984 = -0.359855 x985 = 0.425485 x986 = 0.460713 x987 = 0.633444 x988 = 0.021830 x989 = 2.119256 x990 = -0.266935 x991 = -1.195940 x992 = -2.514444 x993 = -0.826791 x994 = 1.035451 x995 = 0.405242 x996 = -0.433716 x997 = -2.432058 x998 = 0.889073 x999 = 1.083044 x1000 = 0.424594 Le temps d'exécution du programme avec OpenMP est : 3.405351 seconds borhaneddine@borhaneddine-U305d: ~/Bureau\$ </pre>	3.40(s)
CUDA	 <pre> x967 = 1.897143 x968 = 0.586207 x969 = 0.862069 x970 = 0.230769 x971 = 2.250000 x972 = 1.400000 x973 = 2.800000 x974 = 0.850000 x975 = 0.266667 x976 = 0.200000 x977 = 1.714286 x978 = 0.548387 x979 = 1.466667 x980 = 1.086957 x981 = 0.744096 x982 = 1.785714 x983 = 1.784706 x984 = 1.333333 x985 = 0.827586 x986 = 4.580000 x987 = 0.280321 x988 = 0.172414 x989 = 0.010182 x990 = 0.555556 x991 = 7.333333 x992 = 0.613846 x993 = 3.285714 x994 = 1.357143 x995 = 0.560000 x996 = 5.166667 x997 = 0.292900 x998 = 0.840000 x999 = 2.580000 x1000 = 0.500000 Le temps d'exécution du programme avec le GPU est : 0.001394 seconds </pre>	0.0013(s)

Tableau 1: Temps d'exécutions des algorithmes

5.1 Discussion des résultats

La complexité de l'algorithme de Gauss-Jordan est de $O(n^3)$ ce qui explique le temps d'exécution considérable pris par le programme séquentielle (qui est très inefficace dans ce cas). Si tous les n^3 calculs peuvent être faits en parallèle, la complexité de l'algorithme de Gauss Jordan sur une architecture parallèle peut se réduire jusqu'à $O(n)$, qui est le minimum possible (car on doit passer par tous les itérations). L'algorithme d'OpenMP prend environ la moitié du temps pris par l'algorithme séquentiel, ça c'est très logique car le test a été fait sur un PC avec 2 cores, et cela signifie que uniquement 2 threads à la fois sont en cours d'exécution. Pour que tout le calcul soit en parallèle, on doit utiliser un GPU, qui permet que $n*(n+1)$ threads contribuera à l'exécution du programme et donc de transformer toutes les lignes dans une itération en une seule opération, donc une complexité proche de $O(n)$, et cela explique le temps d'exécution négligeable avec CUDA par rapport aux autres programmes.



Remarque

Les codes des 3 programmes sont disponibles sur github via le lien :
[Gausse-Jordan-Algorithm](#).

6 Conclusion

La capacité de la résolution rapide des systèmes linéaires de grandes dimensions détermine l'efficacité des diverses manières de parallélisation. Les architectures GPUs sont parfaitement adaptés aux tâches massivement parallèles, car ils sont capables de créer un très grand nombre de threads qui travaillent simultanément. Selon les résultats obtenus, on peut déduire facilement que la parallélisation basée sur le GPU pour l'algorithme de Gauss-Jordan est beaucoup plus rapide et efficace que la parallélisation basée sur le CPU.