**RAPPORT DE PROJET HPC**

| **Parallélisation de la résolution de système d’équations linéaire par la méthode d’élimination Gauss-Jordan** |
| --- |

**Réalisé par :**

* LAMDANI Wilem - Groupe : 2CS SIQ3
* BELKESSA Linda - Groupe : 2CS SIQ3

**Encadré par :**

* Mme. Haichour Salima

**Sommaire :**

[**1°/ Introduction :**](#_q2zdccirsub) **.................................................................................... 3**

[**2°/ Présentation du problème choisi :**](#_1kzd7gw4ro7k) **................................................. 4**

[**3°/ Implémentation séquentielle :**](#_ct3dihydspwb) **......................................................... 6**

[**4°/ Méthodologie de parallélisation avec openMP :**](#_5brtz7d30dff) **........................... 7**

[**5°/ Implémentation du code avec openMP :**](#_ob25le8a33kj) **........................................ 8**

[**6°/ Méthodologie de parallélisation avec CUDA:**](#_wnh76ejpjjxo) **................................ 9**

[**7°/ Temps d'exécutions et discussion des résultats :**](#_s4v7w0cdiwgw) **...................... 11**

[**8°/ Conclusion :**](#_ykdespyie061) **................................................................................... 12**

[**9°/ Guide d’utilisation :**](#_pc43l73rpmoj) **........................................................................ 12**

# **1°/ Introduction :**

La résolution des systèmes linéaires est une opération de grande importance qui se manifeste dans tous les sujets scientifiques nécessitant du calcul numérique, plusieurs techniques numériques et analytiques ont été développées dans le but d’accélérer les calculs et réduire la complexité des algorithmes connus.

La première méthode mise en place fut celle de Cramer, une méthode exacte mais avec un espace de complexité très élevé, une amélioration a conduit à la résolution des systèmes linéaires en se basant sur l’inversion des matrices par l’algorithme d’élimination de Gauss-Jordan.

La méthode de Gauss Jordan pour le calcul de l'inverse d'une matrice est une des méthodes les plus anciennes. Elle est robuste, précise sur une gamme de matrices et ne nécessite pas de contrôles multiples selon le type de la matrice impliquée. Les travaux existants sur la parallélisation de cet algorithme ont été limités principalement due aux limitations matérielles de l'époque.

Nous élaborons, dans ce rapport notre solution pour paralléliser l’algorithme dans un contexte d’exécution CPU mis en oeuvre grâce à l’API openMP, ensuite, nous exploitons le gain en accélération grâce à la parallélisation extrême sur une architecture GPU implémentée à l’aide de CUDA.

La nature massivement parallèle des GPU les rend capables de produire des taux de GFlops théoriquement beaucoup plus élevés que les processeurs de pointe actuels.

Les performances augmentent également beaucoup plus rapidement que les performances du processeur en raison de parallélisme explicite. La quantité de puissance de calcul à récolter a également attiré la communauté du calcul haute performance (HPC), et nous avons vu de nombreuses applications scientifiques mises en œuvre avec succès avec des gains de performances significatifs sur le GPU.

# **2°/ Présentation du problème choisi :**

L'élimination de Gauss-Jordan peut résoudre un système d'équations AX = B, où A est une matrice n × m de rang r, B est un vecteur fixé, et X le vecteur inconnu. On crée un tableau à n lignes et m + 1 colonnes en bordant la matrice A par le vecteur B. On réduit la matrice sous forme échelonnée réduite.

Si les pivots de la matrice échelonnée réduite associée à (A|B) sont situés uniquement dans les m premières colonnes (ce qui est toujours le cas si r = n ) et ont pour indice de colonnes k1, …, kr , alors la dernière colonne fournit une solution particulière, obtenue en prenant tous ses termes nuls sauf ceux situés à la ligne d'indice ki et à qui on donne la valeur du terme situé à la ligne i de la dernière colonne, i variant de 1 à r.

On obtient la solution générale du système en ajoutant à cette solution particulière un élément quelconque du noyau de A. Celle-ci s'obtient en donnant des valeurs quelconques aux coefficients de X situés à un indice de ligne autre que les ki, et en déterminant les coefficients situés aux lignes d'indice ki de façon à satisfaire le système (ce qui est facile compte tenu de la forme échelonnée de la matrice).

Si le dernier pivot de la matrice échelonnée réduite associée à (A|B) se situe dans la dernière colonne, alors il n'y a pas de solution.

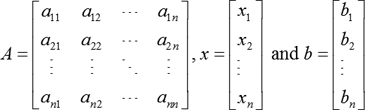
Si la matrice A est carrée inversible (autrement dit, le système est de Cramer), alors on obtient dans la dernière colonne l'unique solution X du système.

Variante : dans l'algorithme précédent, si on se borne à obtenir une matrice échelonnée (non réduite), on obtient une matrice triangulaire supérieure. Il ne reste plus qu'à « remonter » pour retrouver les valeurs des coefficients de X.

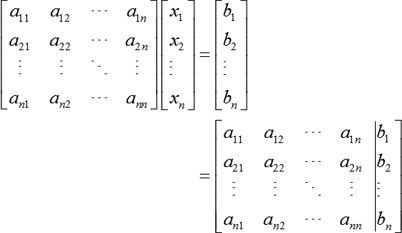
Complexité algorithmique = O(n^3)

**Algorithme général (séquentiel) :**

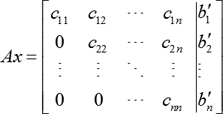
(1) Pour effectuer une élimination gaussienne en commençant par le système d'équations :

****

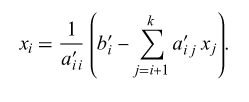
(2) Composer la matrice augmentée



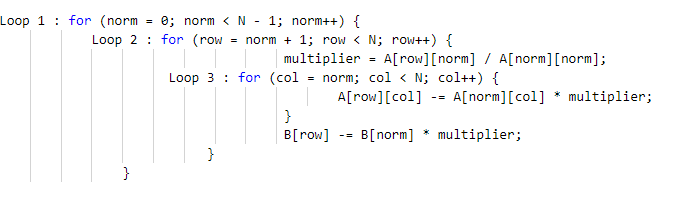
(3) Mettre la matrice augmentée sous forme de matrice triangulaire supérieure



(4) Résoudre l'équation de la ième ligne pour , puis remettez-la dans l'équation de la (k-1)ère ligne pour obtenir une solution pour Xk-1, etc., selon la formule

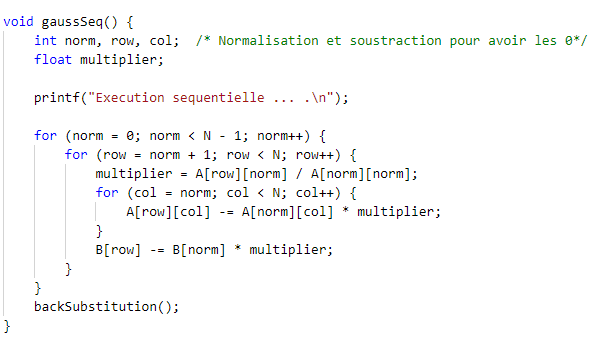


**Le pseudo code pour l’étape d'élimination est le suivant :**

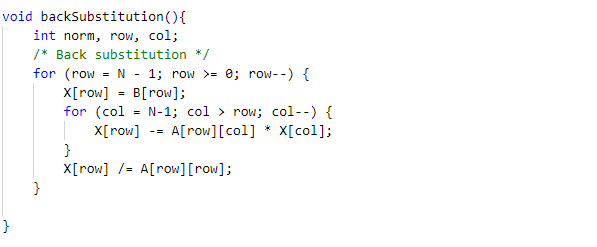


# **3°/ Implémentation séquentielle :**

**Code pour l’élimination de Gauss-Jordan:**

****

**Code pour la remontée (Backward substitution) :**

****

# **4°/ Méthodologie de parallélisation avec openMP :**

Pour pouvoir bien utiliser openMP, il faut dans un premier temps répondre aux questions suivantes :

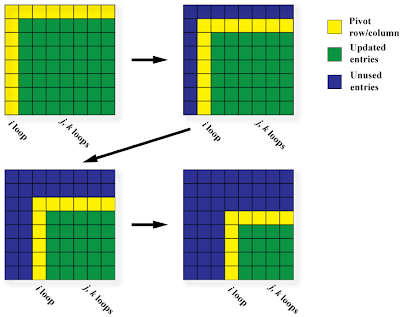
1- Quelles sont les boucles parallélisables ?

Les boucles dont le nombre d'itérations est connu dès le départ et qui ne change pas

Les boucles où chaque itération est indépendante des autres

Les boucles qui ne contiennent pas une dépendance de données (avec les boucles externes en général)

Observons d'abord comment s'effectue l'élimination de Gauss-Jordan visuellement, 4 scénarios sont possibles:



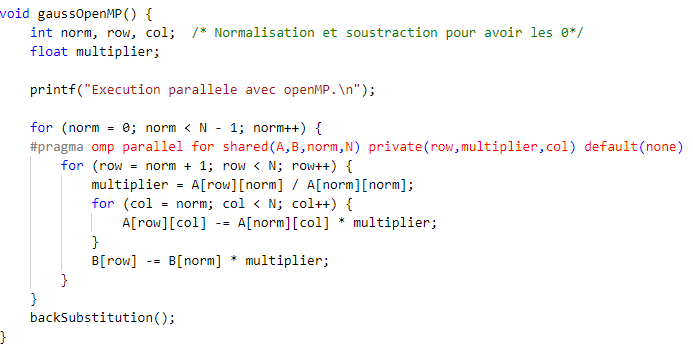
* La boucle i est représentée par la ligne et la colonne jaunes. Les entrées dans la ligne et la colonne jaunes sont utilisées pour mettre à jour la sous-matrice verte avant de passer à la ligne/colonne i+1, ce qui signifie que les valeurs des entrées dans la (i+1)ère zone jaune dépendent des opérations effectuées sur eux aux valeurs précédentes de i. Par conséquent, nous ne pouvons pas utiliser OpenMP pour paralléliser cette boucle en raison de la dépendance des données.
* La boucle j a un nombre d'itérations qui varie avec i, mais nous connaissons le nombre d'itérations à chaque fois que nous sommes sur le point d'entrer dans la boucle. Aucune des itérations ultérieures ne dépend des précédentes et les itérations peuvent être calculées dans n'importe quel ordre ! La boucle j est donc parallélisable.
* La boucle k, comme la boucle j, a un nombre d'itérations qui varie mais qui est calculable pour chaque i. Aucune des itérations ultérieures ne dépend des précédentes, et elles peuvent toutes être calculées dans n'importe quel ordre. Par conséquent, la boucle k est également parallélisable.

**Conclusion :** Il est préférable de sélectionner la boucle externe (j), car nous aurons alors plus de parallélisme ininterrompu et moins de fork et de join.

# **5°/ Implémentation du code avec openMP :**

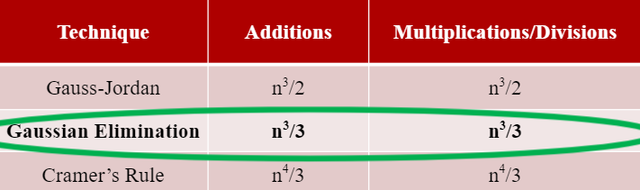
* La boucle externe (Loop 1) ne peut pas être parallélisée car elle contient une variable liée au contrôle de la boucle interne (Loop 2)
* La boucle interne (Loop 2) ne contient aucune dépendance de donnée d'où elle est parallélisable
* La matrice A, le vecteur B et l'index de la boucle externe 'norm' et N seront partagées (shared).

**Idée globale :** Durant la première itération de la boucle externe (Loop 1), on initialise toutes les entrées dans column[0] en commençant de row[1]. Dans la boucle interne (Loop 2) chaque thread s'occupe d'additionner une ligne un multiple (par un scalaire) d'une autre ligne à fin d'avoir 0. à la sortie de la boucle interne, la boucle externe passe à la prochaine colonne, il s'agit ici de l'échelonnement par colonnes. Puis, on effectue une opération de back substitution pour résoudre l'équation.

****

# **6°/ Méthodologie de parallélisation avec CUDA:**

Nous commençons par remarquer la complexité qui s'incrémente drastiquement avec l'augmentation de n.



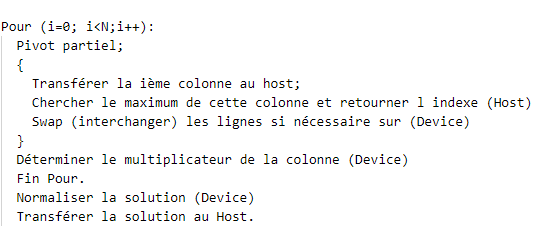
De même, A[j][]=A[j][] –m[j][i]\*matrix pivot row où m est déterminée à l'entrée de la boucle ce qui implémente parfaitement une architecture SIMD (Single Instruction Multiple Data).

**Organisation des blocs et grilles :**

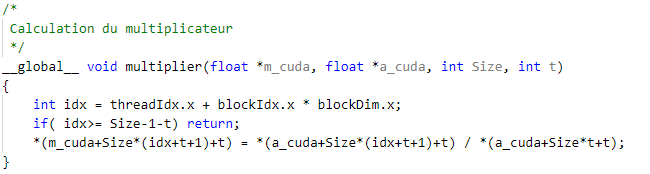
Pour une distribution de calcul optimale et efficace nous considérons l’organisation suivante:

* Pour la fonction du multiplicateur : (on se contente d’une organisation 1D)
  + Taille de bloc = MAXBLOCKSIZE = 512 (peut etre modifié)
  + Taille de la grille = Taille de la matrice / Taille de bloc + 1 (une condition est vérifiée d’abord avant de mobiliser ce bloc de plus : taille de la matrice MOD taille du bloc <> 0)
* Pour la fonction de triangularisation : (organisation 2D)
  + Taille de bloc = 4\*4 (16 threads au total)
  + Taille de la grille = Taille de la matrice / taille de bloc + 1 (vérification de la condition d’abord)

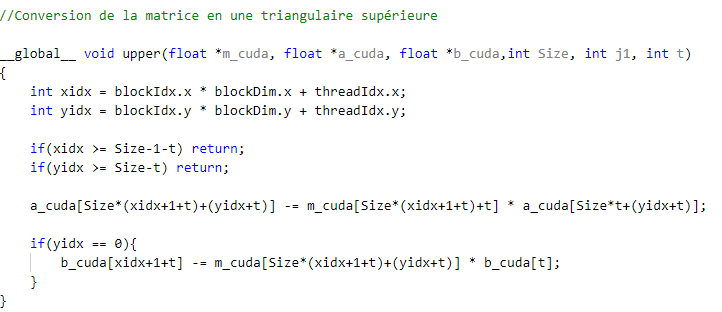
Le pseudo-code pour les instructions avec CUDA :



**Fonction kernel pour déterminer le multiplicateur (multiplier) :**

****

**Fonction kernel pour effectuer les swap et transformer en triangulaire supérieure :**

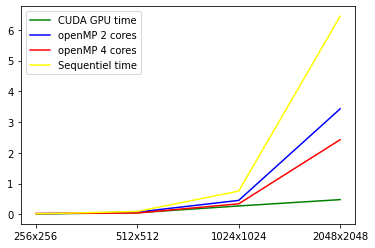
****

# **7°/ Temps d'exécutions et discussion des résultats :**

Nous avons prélevé les temps d’exécutions, et les résultats sont comme suit :

le temps est en secondes.

| **Taille de la matrice** | **code séquentiel** | **openMP (2 cores)** | **openMP (4 cores)** | **CUDA** |
| --- | --- | --- | --- | --- |
| **256x256** | 0.012965 | 0.015634 | 0.026537 | **0.011801** |
| **512x512** | 0.092065 | 0.071215 | **0.041481** | 0.051452 |
| **1024x1024** | 0.754473 | 0.451438 | 0.340611 | **0.270875** |
| **2048x2048** | 6.44276 | 3.43068 | 2.42651 | **0.476649** |

****

Les résultats obtenus s’alignent avec nos attentes théoriques, le temps d’exécution a drastiquement diminué avec CUDA pour les matrices larges ce qui rend ce code stable et efficace pour la résolution des systèmes linéaires.

OpenMP a amélioré le temps d’exécution pour les grandes matrices relativement à l’exécution séquentielle, mais pour les petites matrices la différence reste négligeable, voir même un temps séquentiel meilleur et ceci car le temps de communication et synchronisation entre les threads (cores) s’ajoute et ralenti le code openMP.

# **8°/ Conclusion :**

La résolution des systèmes d’équations linéaires par la méthode d’élimination de Gauss-Jordan peut être adoptée pour les calculs numériques à grande échelle, en se dotant des ressources GPU nécessaires, même si d’autres méthodes peuvent être considérées comme la décomposition LU, QR ou la méthode de Cholesky, l’élimination de Gauss-Jordan reste la plus simple à mettre en oeuvre et ses lacunes par rapport au temps d'exécutions peuvent être surmontées grâce à l’implémentation parallèle avec CUDA.

# **9°/ Guide d’utilisation :**

**Le dossier** [**drive**](https://drive.google.com/drive/folders/1pZKfEIO-IZTV1Qs1S6V-V5f_y1Ca5PAR?usp=sharing) **contient tous les fichiers de codes.**

Pour exécuter le **programme séquentiel et openMP**, utilisez le notebook sur Google Colab :

“projet\_HPC\_Lamdani\_Belkessa\_2cssiq3.ipynb”, ou pour utiliser un **environnement d’exécution local** :

ouvrez jupyter notebook dans un terminal et uploader le fichier “notebook\_openMP.ipynb” ainsi que “sequentiel.c” et “openMP.c”

Ou bien, compilez les fichiers “openmp.c” et “sequentiel.c” directement avec gcc, la syntaxe est la suivante:

“gcc sequentiel.c -o sequentiel”

“gcc -fopenmp openMP.c -o openmp”

**Pour l’exécution :**

“./sequentiel <taille de la matrice> <seed alétoire pour générer les valeurs de la matrice>”

“./openmp <taille de la matrice> <seed alétoire pour générer les valeurs de la matrice>”

**Pour exécuter le code CUDA :**

Ouvrez le fichier notebook ''projet\_HPC\_Lamdani\_Belkessa\_2cssiq3.ipynb'' avec colab et rajoutez les fichiers ''openMP.c'' ,''sequentiel.c'' ,''randcuda.c'' à la session colab.