ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

Rapport Projet HPC

Réalisé par :

- AIN GUERAD Manel
- MAMMA Salima

1- Introduction :

L'avènement des nouvelles technologies de vision par ordinateur et de graphisme ont nourri le besoin incessant d'effectuer des opérations de plus en plus complexes sur les matrices et ce en un minimum de temps.
En effet, les matrices jouent un rôle crucial dans les graphiques. En partant du principe que toute image est représentée par une matrice où chaque chiffre représente l'intensité d'une certaine couleur à un certain point de grille, la reconnaissance d'images est ainsi réduite en grande partie à des opérations matricielles tels que les inversions, les décompositions,

Néanmoins, le graphisme est loin d'être le seul domaine d'utilité des matrices, surtout avec l'ampleur qu'ont pris les Big Data qui sont une collection de vecteurs reliant plusieurs points de données. De nombreux algorithmes de classification et d'analyse de données reposent sur des systèmes linéaires et des opérations matricielles.

Ces raisons ont motivé notre choix de problème qui s'est naturellement orienté vers une opération matricielle.

2- Problème choisi :

Nous avons donc opté pour le problème de la décomposition gaussienne d'une matrice, dite échelonnement par ligne. Une méthode très utile pour la résolution des systèmes linéaires avec la méthode de Gauss, elle est également utilisée lors de la factorisation LU d'une matrice qui est à son tour un outil puissant pour les algorithmes d'inversion de Matrice et de résolutions de systèmes complexes.

- ❖ **Commençons d'abord par définir, ce qu'est une matrice échelonnée par ligne :**

En algèbre linéaire, on dit qu'une matrice carrée est échelonnée en lignes si elle est triangulaire supérieure, c'est à dire le nombre de zéros précédant la première valeur non nulle d'une ligne augmente ligne par ligne jusqu'à ce qu'il ne reste éventuellement plus que des zéros.

Toute matrice peut être transformée en sa matrice échelonnée réduite au moyen d'opérations élémentaires sur les lignes qui sont :

- permuter deux lignes .
- multiplier une ligne par une constante non nulle .
- ajouter à une ligne le multiple d'une autre ligne.

❖ **Algorithme d'échelonnement de matrice :**

Parmi les algorithmes existants pour échelonner une matrice, nous avons opté pour l'élimination de Gauss sans permutations, dont le pseudo algorithme se présente comme suit :

```
 Gauss
   Pour pour i allant de 1 à n − 1, on effectue les calculs suivants :
     |   On ne change pas la i-ème ligne (qui est la ligne du pivot)
     |   Pour k allant de i + 1 à n :
       |    L_{k,i} = U_{k,i}/ U_{i,i}
       | Pour j allant de i + 1 à n,
     |     |     | U_{k,j} = U_{k,j} − L_{k,i}U_{i,j}
     |     | Fin pour
     |
     |   Fin pour
     Fin Pour
  Fin Gauss
```

❖ **Complexité de l'algorithme et nécessité de paralléliser :**

La complexité algorithmique asymptotique de l'élimination de Gauss est O(n3), telle que n×n est la taille de la matrice et le nombre d'instructions à réaliser est proportionnel à n3.

Cet algorithme est généralement utilisé sur un ordinateur pour des systèmes avec des milliers d'inconnues et d'équations d'où la nécessité de paralléliser pour améliorer les performances.

3- Code séquentiel en langage C :

Le fichier comportant le code écrit en langage C est joint à ce rapport sur classroom .

Définition des fonctions :

print_matrix : Pour afficher une matrice carrée d'ordre size passé en paramètre

gaussian : Faire la transformation en matrice échelonnée réduite gaussienne d'une matrice donnée en entrée d'ordre size suivant l'algorithme présenté plus haut

randomfill : Remplit aléatoirement une matrice d'ordre size passé en paramètre

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define N 250


void print_matrix( float  matrix[N][N] , int size)
{

    for (int i = 0; i < size; i++)
    {

        for (int j = 0; j < size; j++)
        {
     printf("%f " , matrix[i][j]);

        }

     printf("\n");
    }
}
```

```c
void gaussian (float a[N][N] , int size)
{
    float l[N];
    for (int i = 0; i < size; i++)
    {

        for (int j = i ; j < size; j++)
        {
     //* printf(" a [j][0] %f a[i][0] %f \n", a[j][0] ,

     if (i!=j) l[j] = a[j][i] / a[i][i];

      //*printf("%f \n %d" , l[j] , j);

      for (int k =0 ; k < size ; k++)
     {
        if (j == i) a[j][k] = a[i][k];
      else {
          if ( k < j) a[j][k] = 0 ;
          else a[j][k] = a[j][k] - l[j] * a[i][k];

      }
      //*printf("%f  " , a[j][k]);
        //*printf("\n");
     }
```

```c
void random_fill(float matrix[N][N], int size)
{

    srand(time(0));
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            matrix[i][j] = ((rand()%20)+1) ;
        }
    }

}
```

```c
int main(void )
{
    int size = N ;
    float a [N][N];
    srand(time(0));
    random_fill(a, size);
    //*printf(" ********************************A********************** \n ");
    //*print_matrix(a, size);
    clock_t start_t=clock(); //Debut temps
    gaussian(a, size);
    clock_t end_t=clock(); //fin temps
    double runing_t =(double)(end_t-start_t)/CLOCKS_PER_SEC;
    //*printf("\n ********************************U********************** \n");
    //*print_matrix(a, size);
    printf("\nOrdre de la matrice = %d \n " , N);
    printf("\nLe temps séquentiel pour le calcul de la décomposition gaussiene de l
    return 0;
}
```

- **Exemple de Test avec affichage de la matrice :**

  Test avec une matrice d'ordre N =6 générée aléatoirement

```
 ********************************A************************
 7.000000 17.000000 10.000000 11.000000 1.000000 5.000000
13.000000 17.000000 16.000000 9.000000 15.000000 18.000000
2.000000 15.000000 15.000000 9.000000 14.000000 15.000000
19.000000 18.000000 17.000000 7.000000 14.000000 19.000000
3.000000 12.000000 13.000000 5.000000 8.000000 3.000000
12.000000 7.000000 11.000000 13.000000 17.000000 3.000000

 ********************************U************************
7.000000 17.000000 10.000000 11.000000 1.000000 5.000000
0.000000 -14.571428 -2.571428 -11.428572 13.142858 8.714286
0.000000 0.000000 12.142857 5.857142 13.714286 13.571428
0.000000 0.000000 0.000000 -22.857141 11.285714 5.428572
0.000000 0.000000 0.000000 0.000000 7.571429 0.857143
0.000000 0.000000 0.000000 0.000000 0.000000 -5.571428

Ordre de la matrice = 6

Le temps séquentiel pour le calcul de la décomposition gaussiene de la matrice 0.000003 s.
```

4- Code parallélisé avec openmp :

Nous avons parallélisé la boucle for la plus extérieure avec la directive de partage de travail #pragma omp parallel for schedule(static).

```
%%writefile first.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.h>
#define N 1000
#define n_threads 20


void print_matrix(float  matrix[N][N] , int size)
{

for (int i = 0; i < size; i++)
{

for (int j = 0; j < size; j++)
{
      printf("%f " , matrix[i][j]);
}

printf("\n");
}
}
```

```
void gaussian(float a[N][N], int size)
{
float l[N];

#pragma omp parallel for schedule (static)
for (int i = 0; i < size; i++)
{
for (int j = i ; j < size; j++)
{

     //* printf(" a [j][0] %f a[i][0] %f \n", a[j][0] , a[i][0]);
     if (i!=j) l[j] = a[j][i] / a[i][i];
      //*printf("%f \n %d" , l[j] , j);
      for (int k =0 ; k < size ; k++)
     {
        if (j == i) a[j][k] = a[i][k];
       else {
           if ( k < j) a[j][k] = 0 ;
           else a[j][k] = a[j][k] - l[j] * a[i][k];

       }
       //*printf("%f  " , a[j][k]);
          //*printf("\n");
     }
}
```

```
int main(void )
{

double runtime;
int size = N ;
srand(time(0));
// set le nombre de threads a utiliser
omp_set_num_threads(n_threads);
float a [N][N];
random_fill(a, size);
//*printf("*********************************A************************\n");
//*print_matrix(a, size);
runtime = omp_get_wtime();
gaussian(a, size);
runtime = omp_get_wtime() - runtime;

//*printf("*********************************U************************\n");
/*print_matrix(a, size);
 printf("\nOrdre de la matrice : %d \n" , N);
 printf("\nNombre de Threads : %d \n" , n_threads);
 printf("\nLe temps parallele avec openmp pour le calcul de la décomposition gaussiene de la matrice %f s.\n", runt
  return 0 ;
}
```

- **Exemple de Test avec affichage de la matrice :**

  Test sur une matrice d'ordre N = 6 générée aléatoirement :

```
**********************************A*************************
19.000000 3.000000 9.000000 3.000000 4.000000 13.000000
18.000000 4.000000 3.000000 10.000000 20.000000 14.000000
5.000000 8.000000 13.000000 9.000000 4.000000 14.000000
13.000000 12.000000 10.000000 4.000000 15.000000 10.000000
9.000000 17.000000 20.000000 8.000000 2.000000 18.000000
2.000000 20.000000 12.000000 2.000000 14.000000 15.000000

**********************************U*************************
19.000000 3.000000 9.000000 3.000000 4.000000 13.000000
0.000000 1.157895 -5.526316 7.157895 16.210526 1.684210
0.000000 0.000000 7.000000 -11.000000 -36.000000 -14.000000
0.000000 0.000000 0.000000 -2.923077 11.923077 -0.769231
0.000000 0.000000 0.000000 0.000000 -4.153846 -3.538462
0.000000 0.000000 0.000000 0.000000 0.000000 2.076923

Ordre de la matrice 6

Nombre de Threads 6

le temps parallel avec openmp pour le calcul de la décomposition gaussiene de l
a matrice 0.001028 s.
```

5- Code parallélisé avec CUDA :

Pour paralléliser sur Cuda, nous avons opté pour la logique suivant :

- On pose le nombre de thread égal à l'ordre de la matrice
- Chaque Thread va réduire une ligne de la matrice à la fois.
- Étant donné que la réduction se fait sur plusieurs itérations et le nombre de lignes à réduire diminue au fur et à mesure. Par exemple, pour la première itération nous gardons la première ligne intacte et réduisons toutes les autres tandis que pour la seconde itération, nous fixons les deux premières lignes pour effectuer les opérations sur les (N-2) restantes et ainsi de suite. Donc le nombre de thread opérant dans chaque itération de l'opération d'échelonnement diminue au fur et à mesure, c'est pour cette raison que nous avons introduit l'appel du kernel à l'intérieur d'une boucle et le nombre de thread par bloc se décrémente.

Aperçu du code parallèle (Envoyé en fichier également) :

```
__global__ void gaussian_reduction(float *a, int size ,int index, float * U ){
  int i;
  int tid;
  tid = threadIdx.x ;
  float b ;
  float k ;
  float f ;


  int start =  ((index+tid+1)*size+index);
  int end= ((index+tid+1)*size+size);

  //*printf("start %d %d \n" , start , end);
  //*  printf("%f %f %f id %d\n " , f , k , b, tid);

          f =  a [start] ;
          k = a[index*size + index];
          b = f/k;
          //*printf("valeur de b pour id : %f %d  f= %f k = %f %d \n" , b , tid , f, k , index*size);
          for(i= start;i< end;i++){
               a[i]=a[i]- b*a[(index*size)+ (index + (i-start))];
               //*printf("%f \n" ,a[i + tid*size] );
        }
}
```

```
int main(){

  float result[N][N]; float *a ; float c[N*N]; float d[N*N];
  float *dev_a, *dev_b, *dev_c;
  int i; int j; int k;
  float l1; float u1; float b[N][N];  float tab[N][N];
  cudaEvent_t start, stop;
  float elapsedTime;
  cudaEventCreate(&start);
  cudaEventRecord(start,0);
  a=(float *)malloc(sizeof(float)*N*N);
  cudaMalloc ( (void**)&dev_a, N*N* sizeof (float) );
  cudaMalloc ( (void**)&dev_b, N*N* sizeof (float) );
  cudaMalloc ( (void**)&dev_c, N*N* sizeof (float) );
  srand(time(0));
  int indice = 0 ;
  printf("***************************** A ***************************** \n");
  for ( i = 0; i < N; i++)
{
    for ( j=0 ; j< N ; j++) {
    tab[i][j] =rand()%1000;
    a[indice] = tab[i][j] ;
    printf("%f  " , tab[i][j]);
    indice ++ ;
 }
```

```
cudaMemcpy( dev_a, a, N*N*sizeof(float), cudaMemcpyHostToDevice);//copy array to device memory
  for(i=0;i<N;i++){
     gaussian_reduction<<<1,N-1-i>>>(dev_a,N,i ,dev_c );
}
cudaDeviceSynchronize ();
cudaEventCreate(&stop);
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start,stop);


cudaMemcpy( &c, dev_a, N*N*sizeof(float),cudaMemcpyDeviceToHost );//copy array back to host

printf("************************** U **************************** \n");
       for(i=0;i<N;i++){
           for(j=0;j<N;j++){
               if (abs (c[i*N + j]) < 0.005) c[i*N + j] = 0 ;
  printf("%f " , c[i*N + j]);
  result[i][j]=c[i*N+j];

  }
  printf("\n");
      }
printf("\nOrdre de la matrice : %d \n" , N);
printf("Le temps parallèle avec CUDA pour le calcul de la décomposition gaussiene de la matrice : %f s \n" ,elapsedTime/1000);
```

- **Exemple de Test avec affichage de la matrice :**

  Test sur une matrice d'ordre N = 6 générée aléatoirement :

```
*************************** A ******************************
340.000000  874.000000  528.000000  538.000000  451.000000  256.000000
379.000000  658.000000  662.000000  759.000000  911.000000  816.000000
935.000000  578.000000  858.000000  62.000000  448.000000  861.000000
265.000000  931.000000  362.000000  376.000000  767.000000  757.000000
469.000000  861.000000  828.000000  460.000000  440.000000  923.000000
971.000000  780.000000  149.000000  499.000000  671.000000  953.000000
*************************** U ******************************
340.000000 874.000000 528.000000 538.000000 451.000000 256.000000
379.000000 658.000000 662.000000 759.000000 911.000000 816.000000
935.000000 578.000000 858.000000 62.000000 448.000000 861.000000
265.000000 931.000000 362.000000 376.000000 767.000000 757.000000
469.000000 861.000000 828.000000 460.000000 440.000000 923.000000
971.000000 780.000000 149.000000 499.000000 671.000000 953.000000

Ordre de la matrice : 6
Le temps parallèle avec CUDA pour le calcul de la décomposition gaussiene de la matrice : 0.000207 s
```

6- Comparaison et interprétation des résultats :

❖ Comparaison :

N =	Séquentiel
250	Ordre de la matrice = 250 Le temps séquentiel pour le calcul de la décomposition gaussiene de la matrice 0.023009 s.
500	Ordre de la matrice = 500 Le temps séquentiel pour le calcul de la décomposition gaussiene de la matrice 0.195346 s.
750	Ordre de la matrice = 750 Le temps séquentiel pour le calcul de la décomposition gaussiene de la matrice 0.687381 s.
1000	Ordre de la matrice = 1000 Le temps séquentiel pour le calcul de la décomposition gaussiene de la matrice 1.642016 s.

N =	OpenMp
250	Ordre de la matrice 250 Nombre de Threads 20 le temps parallel avec openmp pour le calcul de la décomposition gaussiene de l a matrice 0.025357 s.
500	Ordre de la matrice 500 Nombre de Threads 20 le temps parallel avec openmp pour le calcul de la décomposition gaussiene de l a matrice 0.092320 s.
750	Ordre de la matrice 750 Nombre de Threads 20 le temps parallel avec openmp pour le calcul de la décomposition gaussiene de l a matrice 0.228965 s.

1000	Ordre de la matrice 1000 Nombre de Threads 20 le temps parallel avec openmp pour le calcul de la décomposition gaussiene de l a matrice 0.460565 s.

N =	**Cuda**
250	Ordre de la matrice : 250 Le temps parallèle avec CUDA pour le calcul de la décomposition gaussiene de la matrice : 0.001962 s
500	Ordre de la matrice : 500 Le temps parallèle avec CUDA pour le calcul de la décomposition gaussiene de la matrice : 0.006928 s
750	Ordre de la matrice : 750 Le temps parallèle avec CUDA pour le calcul de la décomposition gaussiene de la matrice : 0.015568 s
1000	Ordre de la matrice : 1000 Le temps parallèle avec CUDA pour le calcul de la décomposition gaussiene de la matrice : 0.025719 s

❖ **Rapport entre temps séquentiel et temps parallèle :**

	t_seq / t_omp	**t_seq / t_cuda**	**t_omp / t_cuda**
250	**0.9**	**11.73**	**12.92**
500	**2.12**	**28.2**	**13.32**
750	**3.002**	**44.15**	**14.7**
1000	**3.56**	**63.84**	**17.9**

❖ **Résumé des résultats :**

- Le temps d'exécution en séquentiel est plus lent que celui du code parallélisé avec openMp, qui est à son tour plus lent que le code avec CUDA.
- La différence entre temps séquentiel et temps du programme parallèle avec openMp ou cuda devient d'autant plus grande quand l'ordre de la matrice augmente.
- Pour une matrice d'ordre 1000, le programme openmp est presque 4 plus rapide que le séquentiel et le programme CUDA est quasiment 64 fois plus rapide que le séquentiel et 18 fois plus rapide que le programme openmp

❖ **Interprétation :**

- Dans les deux cas, la parallélisation améliore les performances du code étant donné que les threads se répartissent la charge de travail qui est effectué par un seul dans le cas du code séquentiel. La portion de code parallélisée est importante dans notre cas

- Plus l'ordre est grand, et plus la portion du code parallélisable prend de l'ampleur et la différence entre le temps séquentiel et parallèle devient plus importante.
- La parallélisation CUDA est plus efficace que la parallélisation avec OpenMP et donne de meilleurs résultats dans notre cas car notre problème est très adapté à une parallélisation CUDA, nous effectuons le même ensemble d'instructions sur des indices différents de l'ensemble de vecteurs (Single Instruction Multiple Data) c'est pour cette raison que nous avons pu tirer un maximum de bénéfice de la puissance de calcul qu'offre le GPU qui est nettement plus grande que le CPU pour les calculs vectoriels
- Pour une matrice d'ordre petit le programme openmp ne donne pas d'amélioration par rapport au programme séquentiel et ceci est dû au temps additionnels liés à la gestion des threads et leur synchronisation qui est égale au temps gagné par la parallélisation étant donné que la portion de code parallélisable n'est pas très importante.

7- Conclusion :

Les calculs parallèles n'ont plus besoin de prouver leur importance dans l'industrie et le monde de la recherche scientifique. La nécessité d'une précision accrue et le besoin croissant de résoudre des problèmes de plus en plus complexes ont conduit à l'évolution et à la maturité des techniques de parallélisation.

Il existe plusieurs architectures parallèles et technologies, nous avons exploré deux techniques qui sont la parallélisation OpenMp et CUDA qui sont parmi les plus utilisées et offrant d'excellents résultats.

Étant donné son exploration du GPU , CUDA est plus efficace pour les calculs matriciels, néanmoins lorsque l'on a affaire à des codes complexes avec beaucoup de contrôles la parallélisation CUDA est plus difficile à mettre en œuvre et exploite moins les capacités du GPU. Le choix de l'une ou l'autre des méthodes dépend ainsi en partie du type de problème auquel on a affaire.