DJEBRI Maroua
BROSSARD Florian

Rapport du TP OPENMP
Parallélisme, systèmes distribués et grille
M2 ILC

Ce TP porte sur la parallélisation CPU du produit de deux matrices A et B vers une matrice résultat C avec OpenMP. La parallélisation de deux implémentations seront comparées :

- Implémentation manuelle
- Utilisation de BLAS mis à disposition dans la bibliothèque OpenBLAS

1. Implémentation manuelle

Le but de cette implémentation c'est de paralléliser les deux variantes du kernel 0, afin de résoudre le problème de parallélisation du produit matricielle. On a rencontré plusieurs cas de figures :

Parties de code parallélisable du au faite que le code soit séquentiel, c'est-à-dire que le résultat de son exécution par un thread ne dépend pas des résultats calculés par les autres threads, on peut prendre comme exemple :

- La boucle externe du kernel 0 où les itérations sont indépendantes, afin de rendre cette partie du code parallélisable il faut ajouter un « #pragma omp parallel for private(i,j,k) », afin de rendre toute la région parallèle et privatiser les variables i, j et k, afin que ces variables évoluent de façon indépendante. Quelques résultats :

Threads	Texec	Gflops	ExecBoucle
2	0.36	6.5	0.31
5	0.19	14.43	0.14

- La boucle du milieu où les itérations sont également indépendantes, il faut alors privatiser le "i" dans toute la région parallèle pour qu'il soit partagé et disponible pour tous les threads.

  La privatisation des "j, k" sont également importante pour qu'ils évoluent indépendamment. Quelques résultats :

Threads	Texec	Gflops	ExecBoucle
2	0.36	6.46	0.33
5	0.23	10.83	0.19

Parties du code où ce n'est pas séquentiel on rencontre plus de difficultés, mais on arrive à faire la parallélisation en ajoutant une zone critique afin d'empêcher la lecture et l'écriture concurrentielle. Ceci permettra à chaque thread d'accéder à la zone critique les uns après les autres, et non pas en même temps. On peut prendre comme exemple :

- La boucle interne du Kernel 0 où l'on a privatisé la variable "k" pour qu'elle soit propre à chaque thread. On ajoute ensuite la directive «#pragma omp critical » et un opérateur d'addition "+" pour que chaque thread ajoute son calcul dans l'accumulateur.

On remarque une diminution des performances car l'accès à la zone critique par les threads à tour de rôle rend l'exécution du programme plus lente.

Threads	Texec	Gflops	ExecBoucle
2	1.09	2.017	1.06
16	11.78	10.83	11.75

On remarque aux niveaux des performances, que la parallélisation de la variante "ijk" est la plus efficace car la parallélisation de boucles dont les itérations sont indépendantes est plus simple (Bonne utilisation du cache, traitement des éléments contigus en mémoire) et nous permet d'obtenir de meilleurs résultats.

Code de la version optimisée :

```
#pragma omp parallel for private(i,j,k)
    for (i = 0; i < SIZE; i++) {
```

2. Implémentation de Parallélisation d'appels BLAS

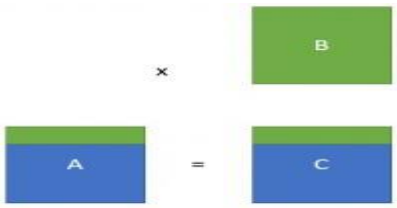
L'implémentation proposée au départ est un simple appel à la fonction cblas_dgemm de la bibliothèque BLAS. Cette fonction prend plusieurs paramètres, notamment :

- La méthode de stockage en mémoire (RowMajor, où ColumnMajor)
- Les tailles des matrices
- Un pointeur vers la case d'index (0,0) pour chacune des matrices A, B, C

Ici, la parallélisation se fait en jouant sur ces paramètres : on "coupe" les matrices en plusieurs morceaux, permettant de faire plusieurs appels à la fonction cblas_dgemm, chacun indépendant des autres :

1. On peut donner une partie de chaque matrice, à condition de bien les faires correspondre
2. Les matrices A et B sont seulement accédées en lecture
3. La matrices C est accédée en écriture, mais comme expliqué dans le point 1, chaque thread écrit dans une partie de C qui est propre à ce thread

En C et C++ La méthode de stockage utilisée naturellement est RowMajor, méthode que nous utiliserons pour chacun de ces appels, ceci impliquant que chaque thread multipliera une partie de A avec la matrice B :



En vert, la partie utilisée par un thread.

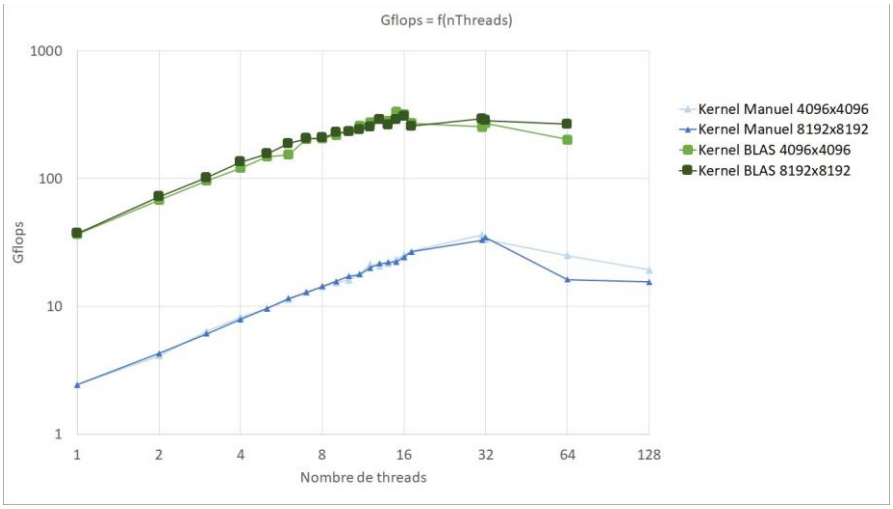
Le nombre de colonnes des matrices A et C est calculée fonction du nombre de threads totaux, du numéro du thread actuel et de la taille des matrices.

Code de la version optimisée :

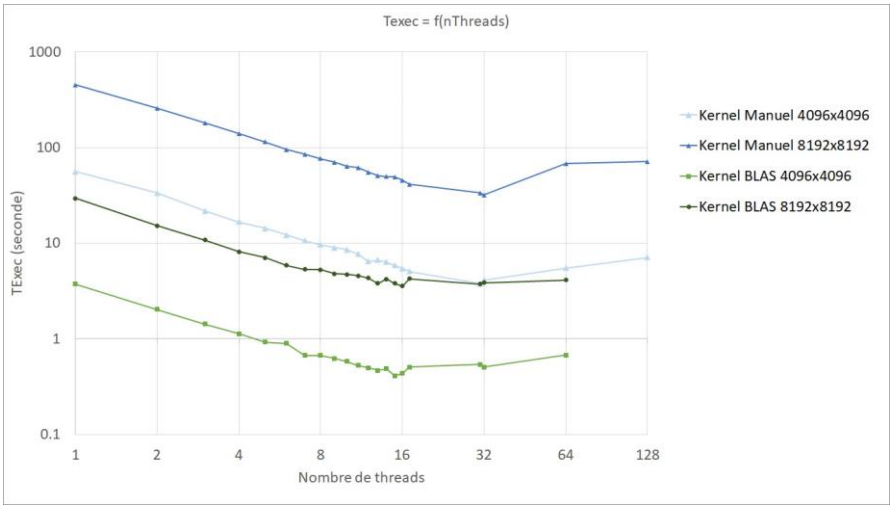
```
#pragma omp parallel
    {
        int slice_size = SIZE / omp_get_num_threads();
        int colonne = slice_size * omp_get_thread_num();
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            slice_size, SIZE, SIZE,
            1.0, &A[colonne][0], SIZE,
            &B[0][0], SIZE,
            0.0, &C[colonne][0], SIZE);
            #pragma omp single
            {
                int reste = (SIZE % omp_get_num_threads());
                if(reste != 0)
                {
                    colonne = SIZE - reste;
                    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                        reste, SIZE, SIZE,
                        1.0, &A[colonne][0], SIZE,
                        &B[0][0], SIZE,
                        0.0, &C[colonne][0], SIZE);
                }
            }
    }
```

3. Comparaison et analyse des performances

Puissance de calcul par nombre de threads :



Temps d'exécution par nombre de threads :



On remarque que L'utilisation des BLAS (en vert), rend l'exécution du produit matricielle au moins 10 fois plus rapide que l'utilisation de l'implémentation manuelle. On peut donc en déduire que l'utilisation des BLAS est toujours au moins aussi performante que l'utilisation du kernel implémenté manuellement, même si la différence au niveau du temps d'exécution peut être bien moins visible pour des matrices de plus petites tailles.

Les courbes ne sont pas tout à fait droites, on peut observer certains creux. Ceux-ci peuvent s'expliquer par la présence de plusieurs utilisateurs sur le serveur où les tests ont été lancés, et également par l'effet de warmup des processeurs. De plus, l'implémentation de la parallélisation proposée implique qu'un des threads auras un travail supplémentaire par rapport aux autres s'il y a un reste dans la division de la taille des matrices par le nombre de threads. Ceci explique entre autres le creux observable dans les deux courbe CBLAS du premier graphique lorsque le nombre de threads est à 17. On observe à ce même endroit un pique dans le temps d'exécution (deuxième graphique).