TDP1 - PRCD

Mise en oeuvre des BLAS

Benjamin Angelaud Adrien Guilbaud

22 octobre 2015

1 Produit scalaire séquentiel

Nous avons commencé par coder la fonction affiche(m,n,a,lda,flux), permettant d'afficher une matrice a de taille m*n. Le lda Leading Dimension de la matrice a) correspond à la taille permettant de passe d'une valeur de la première dimension de la matrice, à la valeur suivante. Cette valeur dépend de la manière dont est stocké la matrice (Raw major ou Column Major). Nous avons ensuite créé des modules d'allocation et d'initialisation de matrice et de vecteur : alloc_vector(int n), init_vector(int n, double * v), alloc_matrix(int m, int n), init_matrix(int m, int n, double * a, int F). Toutes ces fonctions sont définies dans util.c.

Dans ce Tp nous allons nous intéresser à la bibliothèque BLAS (Basic Linear Algebra Subprograms). Les opérations de type BLAS1 sont des opérations de vecteur sur vecteur, les BLAS2 de matrice sur vecteur et les BLAS3 de matrice sur matrice. zdot est une routine BLAS1, elle permet de faire une multiplication de vecteur composés de nombres complexes à double précision. saxpy, permet de faire une multiplication d'un scalaire et d'un vecteur, qu'elle additionne ensuite avec un autre vecteur. Les nombres sont ici des réels à simple précision.

Nous avons donc réaliser une fonction cblas_ddot(const int N, const double *X, const int incX, const double *Y, const int incY) qui implémente la routine BLAS1 ddot. La complexité de calcul est de O(n), car les vecteurs sont de tailles n et nous calculons le produit des deux. Nous avons donc n calcul concernant les multiplications et n calculs concernant les additions, soit 2n calculs. La complexité en nombre de références mémoire est O(n), car nous avons 2*n lectures et n écritures pour une multiplication de vecteurs, soit 3n accès mémoires. Le fait que la complexité en nombre de calculs et en nombre de référence mémoire soit la même ne nous permet pas d'améliorer de manière significative les performances de calcul.

Test de validité Pour vérifier que les résultats fournis par notre fonction soient correct, nous avons dû mettre en place un test de validité. N'ayant pas vraiment d'idée sur la manière de procéder, nous avons regardé comment les librairies comme cblas exécutaient leur test. Nous sommes donc parvenu à une première formule qui était : $\frac{|trueValue - computedValue|}{errorBound}$. Ce calcul nous permettait d'obtenir un ratio qui devait être inférieur à 1 pour que le test soit validé. Après avoir était aiguillé par notre professeur, nous sommes arrivé au test final qui est le suivant : si $\frac{|trueValue - computedValue|}{|trueValue|} < errorBound$ alors le test est bon. Le défi était

ensuite de trouver une valeur de errorBound cohérente pour nos tests. En premier lieu nous avons choisi, suite aux informations trouvées, le epsilon machine disponible dans float.h (DBL_EPSILON). Mais le fait est que, sur un produit scalaire, les erreurs s'accumulent et il est très difficile d'atteindre la précision machine. Nous avons donc chercher des informations complémentaires pour enfin trouver comme valeur 1E-6, mais il nous a été conseillé d'utiliser pour ce TP 1E-5. Ce test nous permet donc de voir que notre fonction réalise les opérations désirés et que nous obtenons des résultats corrects.

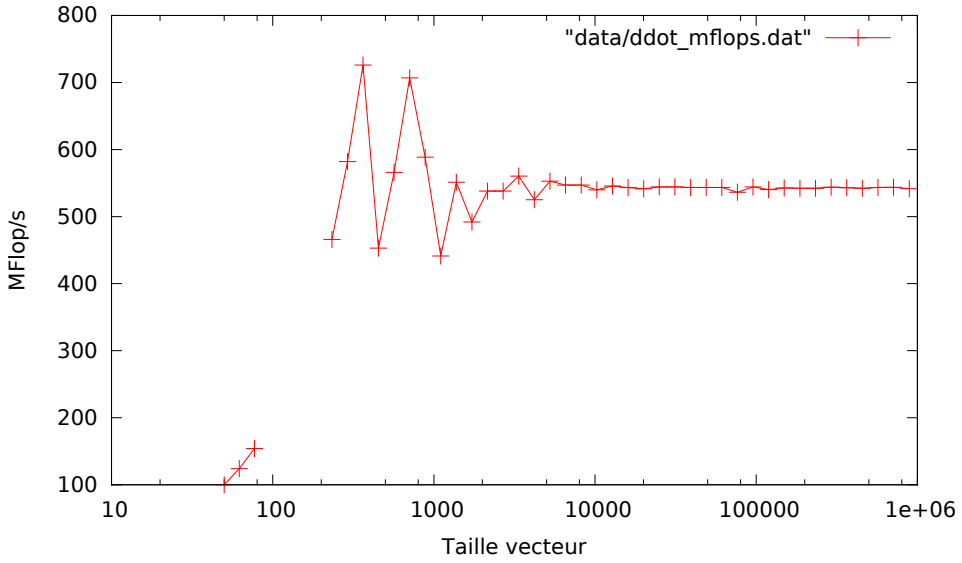

FIGURE 1 – Résolution du produit de vecteurs

Les tests de performances sont effectués sans flush dans le cache après l'initialisation des données.

Ici les performances sont limitées car la complexité des accès mémoire et des opérations est la même. Donc même si les opérations sont effectués plus rapidement, l'exécution sera quand même limitée par les accès mémoire.

2 Produit de matrices séquentiel

Dans cette partie nous avons commencé par réaliser une fonction implémentant la routine cblas dgemm, cblas_dgemm_scalaire(const int M, const int N, const double *A, const int lda, const double *B, const int ldb, double *C, const int ldc), qui fait le produit de la transposé de la matrice A et de la matrice B pour ensuite stocker le résultat dans la matrice C (matrices de taille M*N).

Cette fonction a ensuite été décliné en 3 autres fonctions, chacune explorant les matrices de façons différentes. Ces 3 fonctions sont les suivantes :

— cblas_dgemm_scalaire_ijk(const int M, ...)

— cblas_dgemm_scalaire_kij(const int M, ...)

— cblas_dgemm_scalaire_jik(const int M, ...)

On peut évaluer la complexité en nombre de références mémoire à O(n^2), car nous faisons 2*(n^2) opérations de lecture sur les matrices A et B et n^2 opérations d'écriture sur la matrice C, soit 3*n^2 références mémoire. Pour la complexité en nombre de calculs, nous pouvons l'évaluer à O(n^3), car il faut n calculs pour calculer un terme de la matrice, donc n^2 pour calculer un vecteur de la matrice et n^3 pour calculer la matrice complète. On note ici que la complexité en nombre de référence mémoire est inférieur à la complexité en nombre de calcul, on va donc pouvoir espérer tirer de cela une bonne amélioration de performances en utilisant correctement les caches. Nous observons donc logiquement des performances différentes suivant le parcours de matrices choisit.

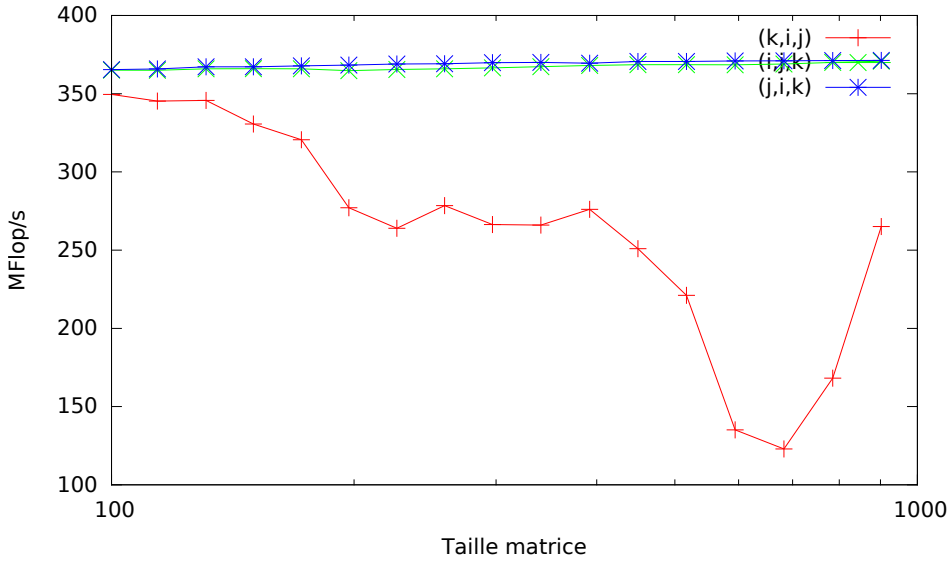

FIGURE 2 – Résolution du produit de matrices suivant différents ordres de parcours

Ces différences s'expliquent par le fait que selon la méthode de parcours des matrices, nous faisons des accès à des valeurs qui ne sont pas contiguës en mémoire et nous faisons donc face à un nouveau remplissage de cache pour chaque valeur (ou presque). En revanche lorsque nous accédons à des valeurs contiguës en mémoire, les valeurs sont directement piochés dans le cache et cela permet un gain de temps qui se ressent sur les courbes précédentes.

dgemm Nous avons par la suite attaqué la partie dgemm par block. Dans un premier temps nous avons codé la fonction cblas_dgemm_block() qui nous calculait le produit de la transposé de la matrice A et la matrice B. Nous avons ensuite lancé les tests de validité sur cette fonction, comme montré précédemment pour ddot. La problème à ensuite était de coller à l'interface cblas : cblas_dgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const double alpha, const double *A, const int lda, const double *B, const int ldb, const double beta, double *C, const int ldc).

Il nous a donc fallu introduire l'éventualité que la matrice A ne soit pas transposée. Pour cela nous avons simplement modifié la façon de calculer un terme de la matrice résultat en inversant les variables de parcours i et k sur la matrice A, et avons aussi modifier la façon de calculer le pointeur sur le début de bloc. Nous nous retrouvons donc avec une fonction

dgemm permettant de transposé A ou non. Comme indiqué dans le TP nous stockons nos matrices seulement en ColumnMajor et ne prenons pas en compte le cas du RawMajor. Pour ces même raisons, B n'est pas non plus transposable.

En utilisant le produit matriciel par blocs, cela nous permet de mieux gérer le cache. En effet, en utilisant les blocs, nous pouvons décider de la façon dont nos données occupent le cache. Nos blocs, en l'état actuel du code, ne sont pas optimisés pour bien remplir le cache.

Si A n'est pas transposé il faut changer notre façon de gérer les blocs, car les valeurs d'une même colonne ne sont plus contiguë en mémoire, ce sont désormais les lignes. Si cette composante n'est pas pris en compte, les performances en seront grandement réduites car nous feront face à des défauts de cache à chaque chargement d'une donnée (ou presque).

Pour déterminer la taille de nos blocs, en considérant que les blocs contiennent des données contiguë en mémoire, il faut simplement se contenter de remplir le cache avec les blocs nécessaires à la mise à jour du bloc solution. Nous pensons que le cache peut aussi contenir le bloc résultat et il faudrait donc ajuster les blocs en conséquence, mais nous ne savons pas vraiment si les valeurs résultats sont écrites ou non dans le cache, ni dans quels cas, c'est donc une amélioration que nous pourrions apporter à notre code.

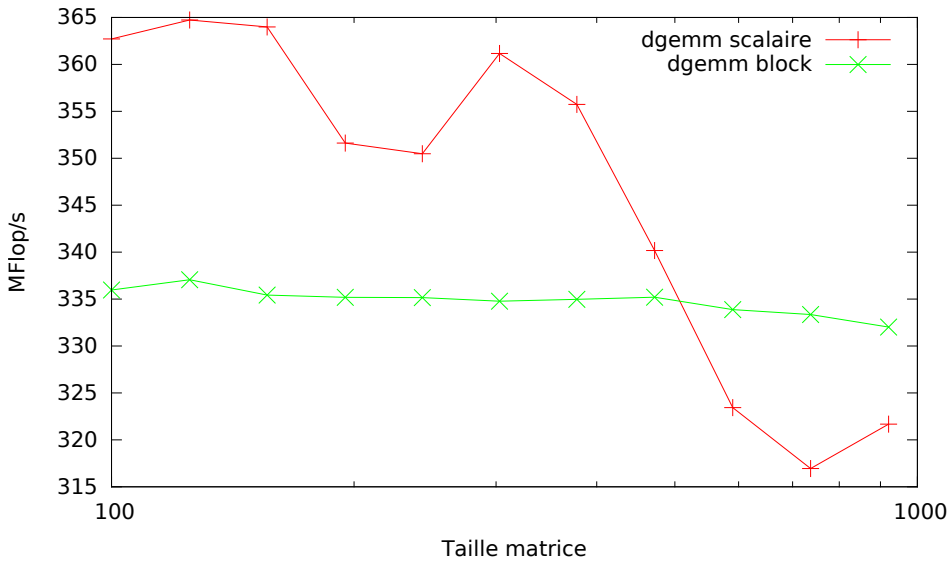

FIGURE 3 – Comparaison entre dgemm scalaire et dgemm block

A partir d'un certain seuil, les performances de dgemm scalaire s'effondrent à cause des défauts de cache, alors que le dgem bloc, qui utilise toujours une taille constante et donc occupe le cache toujours de la même manière, ne souffre pas de perte de performances avec l'augmentation des tailles de matrices.

3 Produit de matrices pour multi-coeur

Cette partie n'est pas implémenté dans notre code. Nous avons néanmoins recherché des informations sur le sujet et avions mis en place un algorithme.

Notre but ici était de créer un système de tâche, grâce à une file. Les tâches contiennent des paires de blocs, un sur A et un sur B. Lorsque les threads ne sont pas occupés, ils "piochent" une tâche dans la file pour l'exécuter et ce jusqu'à ce que la file soit vide. Une fois la tâche allouée, le thread calcul le résultat partiel de la matrice résultat. La mise à jour est ensuite faite dans une section critique, pour éviter les accès concurrents.