UE INF2245
Devoir non surveillé
Parallélisation du produit de matrices pour un jeu d'instructions SIMD

Frédéric Raimbault

Ce travail personnel est à remettre sous la forme d'un fichier écrit en C sur l'ENT. Toute similitude détectée entre deux rendus sera sanctionnée par un zéro à la note de l'UE.

1 Algorithme séquentiel du produit de matrices

Le produit d'une matrice A de taille $d_1 \times d_3$ par une matrice B de taille $d_3 \times d_2$ est une matrice C de taille $d_1 \times d_2$ telle que $A_{i,j} = \sum_{k=1}^{d_3} A_{i,k} \times B_{k,j}$.

La figure 1 illustre quelles lignes de A et quelles colonnes de B sont impliquées dans le calcul des valeurs $c_{1,2}$ et $c_{3,3}$.

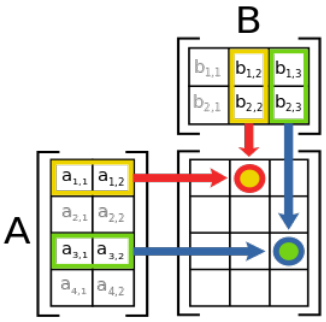

Fig. 1: Produit de deux matrices

1.1 Programme initial

Le programme suivant contient la version initiale de l'algorithme du produit de matrice écrit dans le langage C.

```
void mul_matrix_scal(float A[DIM1][DIM3],// maximal sizes
                     float B[DIM3][DIM2],
                     float C[DIM1][DIM2],
                     int dim1, // real sizes
                     int dim2,
                     int dim3)
{
    for (int x=0;x<dim1;x++) {
        for (int y=0;y<dim2;y++) {
            C[x][y]= 0;
            for (int z=0;z<dim3;z++){
                C[x][y] += A[x][z] * B[z][y];
            }
        }
    }
}
```

1.2 Linéarisation des matrices

Pour optimiser le traitement des matrices dans le langage C il est intéressant de remplacer les tableaux à 2 dimensions, exemple `A[DIM1][DIM3]`, par des tableaux à 1 seule dimension, exemple `A[]`. L'accès à l'élément `A[x][z]` est par exemple remplacé par `A[x*dim3+z]`. On obtient alors le programme suivant.

```
void mul_matrix_scal_v0(float A[], float B[], float C[],
                        int dim1, int dim2, int dim3)
{
    for (int x=0;x<dim1;x++) {
        for (int y=0;y<dim2;y++) {
            C[x*dim2+y]= 0;
            for (int z=0;z<dim3;z++){
                C[x*dim2+y] += A[x*dim3+z] * B[z*dim2+y];
            }
        }
    }
}
```

Ce programme souffre d'un défault inhérent à l'accès aux éléments de B, par colonne, ce qui provoque des défauts de cache fréquents et un remplacement de ligne de cache à chaque itération interne.

1.3 Transformation des itérations

Pour optimiser le programme séquentiel précédent et préparer sa parallélisation on va procéder à plusieurs transformations successives sur les itérations imbriquées.

Éclatement de la première itération interne en deux itérations

L'itération sur les colonnes de C (boucle `for (int y=0;y<dim2;y+)+`) est décomposée en une première itération pour initialiser la totalité de la matrice C, puis une seconde itération pour accumuler les produits entre les valeurs de A et de B.

```
void mul_matrix_scal_v1(float A[], float B[], float C[],
                        int dim1, int dim2, int dim3)
{
     for (int x=0;x<dim1;x++) {
        for (int y=0;y<dim2;y++) {
            C[x*dim2+y]= 0;
        }
        for (int y=0;y<dim2;y++) {
            for (int z=0;z<dim3;z++){
                C[x*dim2+y] += A[x*dim3+z] * B[z*dim2+y];
            }
        }
    }
}
```

Déplacement d'une boucle de la seconde itération dans la première

Au lieu d'initialiser les éléments de la matrice $C[x][y]$ à 0, on les initialise avec la première valeur du produit entre entre $A[x][0]$ et $B[0][y]$. Ce qui revient à faire la première boucle de la seconde itération $(y = 0, z = 0)$ dans la première itération et donc on commence la seconde itération à $(y = 0, z = 1)$

```
void mul_matrix_scal_v2(float A[], float B[], float C[],
                        int dim1, int dim2, int dim3)
{
     for (int x=0;x<dim1;x++) {
        for (int y=0;y<dim2;y++) {
            C[x*dim2+y]= A[x*dim3+0] * B[0*dim2+y];
        }
        for (int y=0;y<dim2;y++) {
            for (int z=1;z<dim3;z++){
                C[x*dim2+y] += A[x*dim3+z] * B[z*dim2+y];
            }
        }
    }
}
```

On permute ensuite les deux boucles **for** de la seconde itération interne et on obtient le programme suivant :

```
void mul_matrix_scal_v2_bis(float A[], float B[], float C[],
                            int dim1, int dim2, int dim3)
{
     for (int x=0;x<dim1;x++) {
        for (int y=0;y<dim2;y++) {
            C[x*dim2+y]= A[x*dim3+0] * B[0*dim2+y];
        }
        for (int z=1;z<dim3;z++){
            for (int y=0;y<dim2;y++) {
                C[x*dim2+y] += A[x*dim3+z] * B[z*dim2+y];
            }
        }
    }
}
```

Extraction des parties constantes des itérations

On extrait du corps des deux itérations internes les calculs qui ne dépendent pas des variables d'itération.

```
void mul_matrix_scal_v3(float A[], float B[], float C[],
                        int dim1, int dim2, int dim3)
{
     for (int x=0;x<dim1;x++) {
        float t= A[x*dim3];
        for (int y=0;y<dim2;y++) {
            C[x*dim2+y]= t * B[y];
        }
        for (int z=1;z<dim3;z++){
            float s= A[x*dim3+z];
            for (int y=0;y<dim2;y++) {
                C[x*dim2+y] += s * B[z*dim2+y];
            }
        }
    }
}
```

On note que désormais le parcours des tableaux se fait de manière linéaire, ce qui permettra de bénéficier de l'apport des caches.

Remplacement des opérations d'indexation par des opérations de déréférencement de pointeur

Comme un tableau en C est équivalent à un pointeur sur son premier élément, par exemple `pT=&t[0]`, on peut remplacer l'indexation d'un élément de tableau, par exemple `t[i]`, par l'accès à la valeur référencée par le pointeur plus la valeur d'index, par exemple `*(pT+i)`. En appliquant ce principe aux lignes des tableaux A, B et C on obtient la nouvelle version du programme suivante.

```
void mul_matrix_scal_v4(float A[], float B[], float C[],
                        int dim1, int dim2, int dim3)
{
     for (int x=0;x<dim1;x++) {
        float t= A[x*dim3];
        float *pC= &(C[x*dim2]);
        for (int y=0;y<dim2;y++) {
            *(pC+y)= t * B[y];
        }
        float *pA= &(A[x*dim3]);
        for (int z=1;z<dim3;z++){
            float s= *(pA+z);
            float *pB= &(B[z*dim2]);
            for (int y=0;y<dim2;y++) {
                *(pC+y) += s * *(pB+y);
            }
        }
    }
}
```

2 Parallélisation pour le jeu d'instruction Intel AVX2

Le jeu d'instruction AVX permet de manipuler des vecteurs de 8 réels simple précision (32-bit). En observant que toutes les itérations de la boucle la plus interne sont indépendantes, il est donc possible de réaliser 8 itérations de manière simultanée. Ce qui revient, dans le code séquentiel à dérouler 8 itérations de la manière suivante :

```
void mul_matrix_scal_v5(float A[], float B[], float C[],
                        int dim1, int dim2, int dim3)
{
     for (int x=0;x<dim1;x++) {
        float t= A[x*dim3];
        float *pC= &(C[x*dim2]);
        for (int y=0;y<dim2;y++) {
            *(pC+y)= t * B[y];
        }
        float *pA= &(A[x*dim3]);
        for (int z=1;z<dim3;z++){
            float s= *(pA+z);
            float *pB= &(B[z*dim2]);
            for (int y=0;y<dim2;y+=8) {
                *(pC+y+0) += s * *(pB+y+0);
                *(pC+y+1) += s * *(pB+y+1);
                *(pC+y+2) += s * *(pB+y+2);
                *(pC+y+3) += s * *(pB+y+3);
                *(pC+y+4) += s * *(pB+y+4);
                *(pC+y+5) += s * *(pB+y+5);
                *(pC+y+6) += s * *(pB+y+6);
                *(pC+y+7) += s * *(pB+y+7);
            }
        }
    }
}
```

Il vous reste à paralléliser ce code en y introduisant les instructions SIMD adéquates et à vérifier que, d'une part que vous obtenez les mêmes résultats qu'avec la version séquentielle, et d'autre part, que vous obtenez bien une accélération des calculs.

Vous trouverez sur l'ENT le fichier matmul.c qui est contient toutes les versions séquentielles décrites dans ce document, ainsi que les fonctions utilitaires pour faire vos tests de performance et comparer les résultats.