Université Pierre et Marie Curie
———
LIP6

Calcul Haute Performance

Optimisation de codes scalaires pour le calcul matriciel et le traitement du signal

Introduction

Le TP est composé de deux parties. La première partie consiste à implémenter des algorithmes et à les valider par des tests unitaires. La seconde consiste à évaluer ls codes, à analyser les réultats de benchmarks et enfin, à comprendre et expliquer les résultats.

Typage fort, allocation mémoire, *debug* et *benchmarking*

Le mode de fonctionnement du programme est lié à la macro `#define ENABLE_BENCHMARKING` et aux macros `CHRONO`, `DEBUG` et `BENCH` (fichier `mymacro.h`).

En mode de mise au point (`ENABLE_BENCHMARKING` en commentaire) :
- `CHRONO(X,t)` est la fonction identité (exécute une fois le contenu `X` entre parenthèes),
- `BENCH(X)` ne fait rien (élément absorbant),
- `DEBUG(X)` est la fonction identitée : `X` est exécuté une fois.

En mode Benchmark (`ENABLE_BENCHMARKING` défini) :
- `CHRONO(X,t)` réalise un chronométrage de `X` : le temps d'exécution en cycles est dans `t`,
- `BENCH(X)` est la fonction identité : `X` est exécuté une fois.
- `DEBUG(X)` ne fait rien.

On peut donc passer du mode mise-au-point au mode *benchmarking* sans toucher au code et donc sans risquer d'introduire des erreurs de frappe...

Les codes utilisés et ceux à écrire respecte un *typage fort* : le types utilisé pour les calculs a en suffixe le nombre de bits sur lequel il est codé : ainsi `float` et remplacé par `float32`. De même les fonctions d'allocation - au format *Numerical Recipes in C* - NRC (`www.nr.com`) sont préfixées par le type manipulé : `f32vector` et `f32matrix`. Les fonctions d'affichage suivent la même logique : `display_f32vector` `display_f32matrix`.

Lors de la phase de validation (tests unitaire), il faut vérifier la validité des calculs, en comparant le résultat du code à celui obtenu via un tableur (Excel, Numbers, Calc, ...).

1 Implémentation et validation

1.1 Copie de matrices

Soient A, B deux matrices carrées de taille n. On souhaite copier A dans B : $B \leftarrow A$.

1. Implémenter la copie par balayage horizontal (de la mémoire) dans la fonction `dup_f32matrix_ij`.
2. Implémenter la copie par balayage vertical (de la mémoire) dans la fonction `dup_f32matrix_ij`.

1.2 Addition de matrices

Soient A, B et C trois matrices carrées de taille n. Nous avons :

$$C_H(i,j) = \sum_{i=0}^{i=n-1} \sum_{j=0}^{j=n-1} A(i,j) + B(i,j) \quad (1)$$

$$C_V(i,j) = \sum_{j=0}^{j=n-1} \sum_{i=0}^{i=n-1} A(i,j) + B(i,j) \quad (2)$$

$$(3)$$

1. Implémenter l'addition par balayage horizontal (de la mémoire) dans la fonction `add_f32matrix_ij`.
2. Implémenter l'addition par balayage vertical (de la mémoire) dans la fonction `add_f32matrix_ji`.

1.3 Réduction de matrice

Les projections horizontales et verticales sont des réductions partielles qui produisent, à partir d'une matrice, un vecteur horizontal ou vertical. Soient M une matrice carrée de taille n et P_H et P_V, les projections horizontale et verticale. Nous avons :

$$P_H(i) = \sum_{j=0}^{j=n-1} M(i,j) \quad (4)$$

$$P_V(j) = \sum_{i=0}^{i=n-1} M(i,j) \quad (5)$$

$$(6)$$

1. Implémenter la projection P_H dans la fonction `projection_H`.
2. Implémenter la projection P_V dans la fonction `projection_V`.
3. L'implémentation naïve de la projection P_V pose des problèmes de cache. Faire une implémentation optimisée de la projection P_V qui tienne compte des caches dans la fonction `projection_VH`.

1.4 Multiplication de matrices

Soient A, B et C trois matrices carrées de taille n. On souhaite réaliser la multiplication de A par B dans C : $C = A \times B$.
Le nommage des versions est basé sur l'ordre des boucles, de la plus externe à la plus interne. Ainsi pour la version de référence : `"ijk"`, les boucles i et j balaient la mémoire et la boucle k est utilisée pour réaliser le produit scalaire de la ligne i par la colonne j. Ce produit scalaire est rangé dans $C(i,j)$. Nous avons :

$$C(i,j) = \sum_{k=0}^{i=n-1} A(i,k) \times B(k,j) \tag{7}$$

Il existe 6 versions obtenues par permutation du nid de boucles

1. Implémenter la version `ijk` dans la fonction `multiplication_ijk`.
2. Implémenter la version `ikj` dans la fonction `multiplication_ikj`.
3. Implémenter la version `jik` dans la fonction `multiplication_jik`.
4. Implémenter la version `jki` dans la fonction `multiplication_jki`.
5. Implémenter la version `kij` dans la fonction `multiplication_kij`.
6. Implémenter la version `kji` dans la fonction `multiplication_kji`.

1.5 Convolution / *Stencil*

Soient A, B deux matrices matrices carrées de taille n, On souhaite maintenant évaluer l'impact des transformations de boucles sur le calcul d'une convolution (dans le vocabulaire du traitement du signal et *stencil* dans le vocabulaire du HPC). L'opération à réaliser est la somme sur un voisinage 3×3.

$$B(i,j) = \sum_{\delta i=-1}^{\delta i=+1} \sum_{\delta j=-1}^{\delta j=+1} A(i+\delta i, j+\delta j) \tag{8}$$

Il faut donc deux boucles pour parcourir la matrice d'arrivée (i,j) et deux autres boucles pour parcourir le voisinage $(\delta i, \delta j)$ autour du *point coïncidant* dans la matrice de départ.

1. Implémenter la version avec 4 boucles dans la fonction `sum3_f32matrix_loop`. Par la suite, on réalisera systématiquement un *loop unwinding* des deux boucles les plus internes.
2. En partant de la version `loop`, réalisez un *loop unwinding* des deux boucles les plus internes. N'utilisez que des notations tableaux sans *scalarisation*. Codez cette nouvelle version dans la fonction `sum3_f32matrix_array`.
3. En partant de la version `array`, réalisez une *scalarisation*. Codez cette nouvelle version dans la fonction `sum3_f32matrix_reg`.
4. En partant de la version `reg`, réalisez une *rotation de registres*. Codez cette nouvelle version dans la fonction `sum3_f32matrix_rot`.
5. En partant de la version `reg`,réalisez un *loop unrolling* de la boucle interne permettant une *scalarisation complète*. Codez cette nouvelle version dans la fonction `sum3_f32matrix_lu`.
6. En partant de la version `rot`,réalisez une *réduction par colonne* en utilisant le fait que la somme 2D est décomposable en deux sommes 1D. Codez cette nouvelle version dans la fonction `sum3_f32matrix_red`. Pour cela, aidez-vous de la publication "High Level Transforms for SIMD and Low-Level Computer Vision Algorithms".

On souhaite maintenant enchainer plusieurs sommations $S_{3\times3} : B = S_{3\times3}(A)$ puis $C = S_{3\times3}(B)$ avec C une troisième matrice carrée de taille n.

7. Codez dans la fonction `sum3x2_f32matrix_array` un double appel à la fonction simple `sum3_f32matrix_array`.
8. Faire de même avec la réduction par colonne `sum3x2_f32matrix_red` appelle deux fois `sum3_f32matrix_red`.

Cette façon de parcourir la mémoire est inefficace dès que les matrices ne tiennent plus dans les caches. Ils faut *pipeliner* les traitement pour *maximiser* la persistance des données dans les caches.

9. En partant de la fonction `sum3_f32vector_red`, codez la fonction *ligne* `sum3_f32vector_reg` qui traite une ligne (d'indice i - voir le prototype de la fonction). Puis codez la fonction `sum3x2_f32matrix_reg_pipe` qui traite en pipeline les matrices A, B et C. Après un prologue qui permet de produire suffisamment de ligne de B, il est possible de produire en séquence une ligne de B puis une ligne de C. Faire un schéma.
10. Faire de même avec les fonction `sum3_f32vector_red` et `sum3x2_f32matrix_red_pipe`.
11. En fonction du compilateur et des options d'optimisations, il peut être utile d'*inliner* le code de la fonction ligne. Codez pour cela la fonction `sum3x2_f32matrix_red_pipe_inline`.
12. Conclure

2 Benchmark et analyse

Il y a deux paramètres pour les benchmarks :

— La taille des matrices, typiquement trois cas : matrices de petite taille qui tiennent dans les caches, matrices de grande taille qui ne tiennent pas dans les caches et matrices de taille intermédiaire qui est de l'ordre de la taille des caches. Typiquement $\{512, 1024, 2048\}$ (pour la multiplication matricielle, prendre des valeurs deux fois plus petites). Pour voir s'il y a un problème avec les puissances de 2, on peut aussi prendre des multiples de 10 proches : $\{500, 1000, 2000\}$.

— Les options d'optimisations pour la compilation : `-O1`, `-O3`

La taille des matrices devra être adaptée à la taille des caches des processeurs des machines utilisées en TP (vérifier les tailles sur `ark.intel.com`). Faire un tableau récapitulatif et analyser les résultats.