Sorbonne Université

LIP6

ProgPar SIMD

Stencils 1D

Introduction

— Lire l'énoncé en entier, ou au moins partie par partie pour saisir l'ensemble des questions.

— Pour chaque section, il y a un fichier de calcul (à remplir) et un fichier de test (dont on pourra modifier la fonction d'appel) :
  — stencil 1D sans bord : `stencil1_without.c` et `test_stencil1_without.c`
  — stencil 1D avec bords : `stencil1_with.c` et `test_stencil1_with.c`
  — stencil 2D avec bords : `stencil2_with.c` et `test_stencil2_with.c`

— La fonctionnalité des autres fichiers est :
  — `macro_debug.h` header contenant les macros de debug.
  — `stencil.c` fonctions communes à toutes les fonctions de calculs, comme l'initialisation des paramètres, la vérification des calculs ou encore les fonctions appelées par les macros de debug.
  — `nrutil.c` et `nrutil.c` fonctions d'allocations mémoire et d'affichage. Ne pas modifier.

— Concernant le nommage des variables, il est conseillé d'utiliser un nommage court et extensible, par exemple des lettres de l'alphabet pour les colonnes et des numéros pour les lignes : `a0,a1,a2; b0,b1,b2; c0,c1,c2`. Pour les variables *réduites* par colonne, le nom de la colonne préfixée de `r` : `ra, rb, rc`.
— Enfin, pour compiler, utiliser le Makefile `Makefile` et modifier la fonction principale de test se trouvant dans `main.c`

1 Stencil 1D sans bord

Soient X et Y deux tableaux de taille n, on souhaite appliquer le stencil S_3 qui réalise la somme centrée de 3 points.

$$S_3 = \begin{bmatrix} 1 & \mathbf{1} & 1 \end{bmatrix} \tag{1}$$

Autrement dit, on aura pour $Y = S_3 * X$ au point i la somme $Y(i) = X(i-1) + X(i) + X(i+1)$
Mais comme le tableau n'a pas de bord, la définition du stencil S_3 doit être spécialisée en explicitant les cas particuliers ((eq. 2).

$$y(i) = \begin{cases} x(i) + x(i) + x(i+1) & \text{si } i = 0 \\ x(i-1) + x(i) + x(i) & \text{si } i = n-1 \\ x(i-1) + x(i) + x(i+1) & \text{sinon} \end{cases} \tag{2}$$

1. Ecrire un code C avec les deux cas particuliers à l'intérieur de la boucle dans fonction `add3_basic0_without (float *X, int n, float *Y)`.
2. Ecrire un code C avec les deux cas particuliers à l'extérieur de la boucle dans fonction `add3_basic1_without (float *X, int n, float *Y)`.
3. La fonction `compare_f32vector` est fournie mais nécessite d'avoir un résultat pour faire la comparaison et valider le second résultat. Ecrire le code C de la fonction `check_f32vector` qui valide un résultat à partir des paramètres d'initialisation ($X(i) = x_0 + i \times a$), ici `x0` et `xstep` : `check_param_f32vector(float *Y, int i0, int i1, float x0, float xstep)`.

4. Ecrire un code C avec *scalarisation* dérivant du code précédent dans fonction `add3_reg_without (float *X, int n, float *Y)`.
5. Ecrire un code C avec rotation de registres (et adapter le prologue et l'épilogue) dans fonction `add3_rot_without (float *X, int n, float *Y)`.
6. Ecrire un code C avec déroulage de boucle d'ordre 3 dans fonction `add3_lu3_without_KO (float *X, int n, float *Y)` avec une condition de sortie de boucle fausse (`i<n`) afin de volontairement provoquer une erreur *segmentation fault* et tester les macros de débug.
7. Coder les macros `load1(X, i)`, `store1(Y, i, y)` et `add3(x0, x1, x2)` pour améliorer la sémantique et debugguer la fonction fausse.
8. Ecrire un code C avec déroulage de boucle d'ordre 3 dans fonction `add3_lu3_without (float *X, int n, float *Y)`. Pour Vérifier que les codes de la boucle et de l'épilogue sont corrects, tester pour 3 valeurs consécutives de n. Par exemple $n \in 6, 7, 8$.

2 Stencil 1D avec bords

Refaire les mêmes codes, mais cette fois avoir un tableau qui a des bords. Les cases `X[-1]` et `X[n]`, les cas particuliers disparaissent et sont remplacés par une initilisation des bords : `X[-1]=X[0]` et `X[n]=X[n-1]`. Les arguments des fonctions sont tous identiques : `(float *X, int n, float *Y)`.

1. `add3_basic0_with`
2. `add3_basic1_with`
3. `add3_reg_with`
4. `add3_rot_with`
5. `add3_lu3`

Toujours vérifier que le code `lu3` est correct pour 3 valeurs de n ...

3 Stencil 1D et pipeline logiciel

Implémenter les algorithmes avec pipeline logiciel dans l'ordre indiqué en TD :

1. `add3_sp3`
2. `add3_sp2`
3. `add3_sp_rot2`
4. `add3_sp_lu2`
5. `add3_sp3_without`
6. `add3_sp2_without`
7. `add3_sp_rot2_without`
8. `add3_sp_lu2_without`

4 Stencil 2D avec bord

Dans cette partie, les données en entrée ont toutes des bords et les fonctions passent de 1 paramètre (`n`) à 4 pour décrire une *Region of Interest*. De plus, les notations NRC sont pour des intervalles fermés $[i_0, i_1] \times [j_0, j_1]$ et non semi-ouverts $[0, n[\times [0, n[$. Enfin les fonctions 2D appelleront des fonctions 1D. Pour la fonction `add3_f32matrix_basic` on aura :

```
void line_add3_f32matrix_basic(float32 **X, int i, int j0, int j1, float32 **Y)
{
    for(int j=j0; j<=j1; j++)
        Y[i][j] = add9(X[i-1][j-1], X[i-1][j], X[i-1][j+1],
                       X[i  ][j-1], X[i  ][j], X[i  ][j+1],
                       X[i+1][j-1], X[i+1][j], X[i+1][j+1]);
}
void add3_f32matrix_basic(float32 **X, int i0, int i1, int j0, int j1, float32 **Y)
{
    for(int i=i0; i<=i1; i++)
        line_add3_f32matrix_basic(X, i, j0, j1, Y);
}
```

Afin de simplifier le codage des fonctions et de supprimer les cas particuliers (et la taille du code) liés aux 4 coins et aux 4 bords (soit 8 spécialisations du code générale de la boucle),

— l'image d'entrée a un bord de rayon $r = 1$ adapté aux stencils de diamètre 3×3 ($d = 2r + 1$).

— les bords sont initialisés à zéro, il n'est pas nécessaire d'y dupliquer les valeurs.

En vous inspirant des fonctions pour stencils 1D,

1. Ecrire une fonction avec *scalarisation* : `add3_f32matrix_reg` et `line_add3_f32matrix_reg`.
2. Ecrire une fonction avec *rotation de registres* : `add3_f32matrix_rot` et `line_add3_f32matrix_rot`.
3. La fonction `compare_f32matrix` est fournie mais nécessite d'avoir un résultat pour faire la comparaison et valider le second résultat. Ecrire le code C de la fonction `check_f32matrix` qui valide un résultat à partir des paramètres d'initialisation ($X(i, j) = x_0 + i \times a + j \times b$), ici `x0`, `xstep` et `ystep` : `check_param_f32matrix(float *Y, int i0, int i1, int j0, int j1, float x0, float xstep, float ystep)`.
4. Ecrire une fonction utilisant le fait que le stencil 2D est séparable en 2 stencils 1D : un stencil horizontal S_H avec parcours horizontal de la mémoire et un stencil vertical S_V avec parcours vertical de la mémoire : `add3_f32matrix_sep0`.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

5. Faire de même mais avec un parcours horizontal de la mémoire pour le stencil vertical S_V : `add3_f32matrix_sep1`.
6. Ecrire une fonction avec *reduction* par colonne (en s'inspirant des versions séparables) `add3_f32matrix_red` et `line_add3_f32matrix_red`.
7. Ecrire une fonction avec *déroulage* de la boucle interne `add3_f32matrix_ilu3` et `line_add3_f32matrix_ilu3`.
8. Ecrire une fonction avec *déroulage* de boucle interne et *reduction* par colonne : `add3_f32matrix_ilu3_red` et `line_add3_f32matrix_ilu3_red`.
9. Ecrire une fonction avec *déroulage* de boucle externe et *reduction* par colonne : `add3_f32matrix_elu2_red` et `line_add3_f32matrix_elu2_red`.
10. Ecrire une fonction avec *déroulage* de boucle externe, *reduction* par colonne et *factorisation* des calculs (optimisation *Common Sub Expression Elimination*) : `add3_f32matrix_elu2_red_factor` et `line_add3_f32matrix_elu2_red_factor`.
11. Ecrire une fonction avec *déroulage* de boucle externe, *reduction* par colonne et *factorisation* des calculs (optimisation *Common Sub Expression Elimination*) : `add3_f32matrix_ilu3_elu2_red_factor` et `line_add3_f32matrix_ilu3_elu2_red_factor`.