

Optimisations de code pour le calcul régulier Lionel Lacassagne

Introduction

Le but de ce TP est de prendre en main les outils de compilation et de chronométrie et d'appliquer des techniques d'optimisation de code à des algorithmes d'algèbre linéaire et de traitement du signal. Les fichiers C nécessaires ainsi qu'une documentation du compilateur Intel se trouvent à l'adresse www.ief.u-psud.fr/~lacas/Teaching.

Pour cela, nous allons analyser l'impact sur le temps d'exécution :

- des optimisations logicielles,
- des options de compilation,
- de l'architecture (pipeline et cache)

Le déroulement de ce TP sera le suivant :

1. partie *Debug*: codage, mise au point des algorithmes de base, optimisation logicielle manuelle et validation via des tests unitaires
2. partie *configuration*: création de plusieurs Makefile pour évaluer l'impact des options d'optimisation du compilateur
3. partie *benchmark*: évaluation de performance d'une fonction, pour différentes tailles de données et différents Makefile.

Afin d'avoir des types normalisés (dont la taille et la précision restent identiques d'une architecture à une autre), les types classiques du C: `int`, `float` et `double` sont remplacés par des `sint32`, `float32` et `float64`. Les fonctions les manipulant seront *typées* avec `si32`, `f32` et `f64`.

Les tableaux 1D et 2D sont décrits de différentes façons:

- méthode classique: n est la taille des données. En 1D, l'espace d'itération de la boucle j est sur $[0..n-1]$. En 2D l'espace d'itération du nid de boucles (i, j) est sur $[0..n-1] \times [0..n-1]$.
- méthode NRC: l'intervalle utile (*range*) est passé en paramètre. En 1D ce sera $j \in [j_0..j_1]$ et en 2D $(i, j) \in [i_0..i_1] \times [j_0..j_1]$.

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 parenthèses),
- `BENCH(X)` ne fait rien,
- `DEBUG(X)` est la fonction identité: `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.

Le TP et le compte rendu de TP sont à rendre de la manière suivante:

- par mail, dans une archive `zip` qui contiendra un répertoire à votre **nom**,
- le sujet du mail sera `CCI TP` et rien d'autre,
- le compte rendu sera en `pdf` (les formats Microsoft et Open-Office seront refusés) à votre **nom** dans le répertoire à votre **nom** ainsi que les fichiers fournis.

Une façon simple de respecter ces contraintes est de renommer avec votre **nom** le répertoire contenant le TP et d'y copier votre compte-rendu. Ne pas oublier de commenter vos codes et de les indenter correctement.

1 Opérateurs 1D

La fonction de test et d'allocation mémoire est `test_dup_add_dot_sum3_vector` dans le fichier `vector.c`.

1.1 Copies de vecteurs (tableaux 1D)

Soient les vecteurs X et Y de taille n , alors:

$$Y(j) \leftarrow X(j), \quad j \in [0..n-1] \quad (1)$$

1. Codez cette copie dans les fonctions `dup_si32vector`, `dup_f32vector` et `dup_f64vector` pour les types `sint32`, `float32` et `float64`.

1.2 Addition de deux vecteurs (tableaux 1D)

Soient les vecteurs X_1 et X_2 de taille n et Y leur somme:

$$Y \leftarrow X_1 + X_2 = \sum_{j=0}^{n-1} X_1(j) + X_2(j) \quad (2)$$

1. Codez cette addition dans les fonctions `add_si32vector`, `add_f32vector` et `add_f64vector`.

1.3 Produit scalaire de deux vecteurs (tableaux 1D): réduction totale

Soient les vecteurs X_1 et X_2 de taille n et d leur produit scalaire:

$$d = X_1 \cdot X_2 = \sum_{j=0}^{n-1} X_1(j) \times X_2(j) \quad (3)$$

1. Codez ce produit scalaire dans les fonctions `dot_si32vector`, `dot_f32vector` et `dot_f64vector`.

1.4 Somme sur un voisinage de 3 points (tableaux 1D): réduction partielle

Les opérateurs précédents réalisent des calculs point à point en scalaire. Cet opérateur se différencie des précédents car il réalise des calculs sur un voisinage. Soit le tableau X . On souhaite que le tableau Y contienne en tout point la somme de trois points du tableau X :

$$Y(j) = X(j - 1) + X(j) + X(j + 1), \quad j \in [0..n - 1] \quad (4)$$

1. Codez cet opérateur dans les fonctions `sum3_f32vector` et `sum3_f64vector`.
2. Afin d'éviter de recharger les points déjà chargés lors des précédentes itérations de la boucle, appliquez la transformation dite de *Rotation de Registres*. Codez cet opérateur dans la fonction `sum3_f32vector_RR`.
3. Afin d'éviter les copies registres à registres, appliquez la transformation dite de *Déroulage de Boucle (Loop Unrolling)*. Codez cet opérateur dans les fonctions `sum3_f32vector_LU` et `sum3_f64vector_LU`.

2 Benchmarking 1D

1. En cherchant dans la documentation du compilateur Intel et en utilisant le Makefile fourni initialement, en faire 3 versions (indiquez qu'on fait l'hypothèse qu'il n'y a pas d'aliasing de pointeur):
 - `Makefile1`: niveau d'optimisation 1, vectorisation désactivée
 - `Makefile3`: niveau d'optimisation 3, vectorisation désactivée
 - `Makefile4`: niveau d'optimisation 3, vectorisation activée
2. Compilez le projet avec les différents Makefile et pour les tailles de données suivantes:
 - `n=10*10` pour des données très petites tenant dans le cache
 - `n=100*100` pour des données petites pouvant tenir (ou pas dans) dans le cache
 - `n=1000*1000` pour des données de grande taille
 - `n=1000*1000` pour des données de très grande taille ne tenant pas dans le cache
3. Pour chaque fonction, collectez les différents temps d'exécution en *cpp* (cycles par points) dans un tableur, indiquez :
 - le gain dû aux optimisations des Makefile,
 - le gain dû aux optimisations logicielles (s'il y a lieu),
 - la perte de gain due à l'augmentation de la taille des données.
4. Notez le nom du processeur Intel utilisé ainsi que sa fréquence.

3 Opérateur 2D

La fonction de test est `main_test_matrix` dans le fichier `matrix.c`.

3.1 Copie de matrice

1. Codez les versions `dup_f32matrix_ij`, `dup_f32matrix_ji`, `dup_f64matrix_ij`, `dup_f64matrix_ji`, pour les types `float32` et `float64`, avec les boucles dans le *bon* ordre (ie `ij`) et dans le *mauvais* ordre (`ji`), puis faire de même en `float64`.

3.2 Addition matricielle

1. Codez les versions suivantes `add_f32matrix_ij`, `add_f32matrix_ji`, `add_f64matrix_ij`, `add_f64matrix_ji`, pour les types `float32` et `float64`, avec les boucles dans le *bon* ordre (ie `ij`) et dans le *mauvais* ordre (`ji`), puis faire de même en `float64`.

3.3 Multiplication matricielle

Soient A , B et C trois matrices carrées de taille $n \times n$.

1. Coder les 6 versions `ijk`, `ikj`, `jik`, `jki`, `kij`, `kji` en `float32` dans les fonctions `mul_ijk_f32matrix`, `mul_ikj_f32matrix`, `mul_jik_f32matrix` et `mul_jki_f32matrix`, `mul_kij_f32matrix` et `mul_kji_f32matrix`.
2. Faire de même en `float64`.

3.4 Somme sur un voisinage de 3×3 : réduction partielle en 2D

Soient les matrices X et Y de taille $n \times n$

$$Y(i, j) = \sum_{\delta_i=-1}^{+1} \sum_{\delta_j=-1}^{+1} X(i + \delta_i, j + \delta_j), \quad (i, j) \in [0..n - 1] \times [0..n - 1] \quad (5)$$

1. Codez la version basique (avec une double boucle pour parcourir le voisinage) dans la fonction `sum3_f32vector_Loop`.
2. Codez la version standard où la double boucle interne est complètement déroulée (car on connaît à l'avance les indices de début et fin de cette double boucle) dans la fonction `sum3_f32vector`.
3. Codez la version avec *Rotation de Registres* dans la fonction `sum3_f32vector_Rot` afin d'éviter de recharger les points déjà chargés lors des précédentes itérations de la boucle (ie les 6 points des deux premières colonnes du voisinage). Les variables utilisées seront `a0`, `a1`, `a2` pour la première colonne, `b0`, `b1`, `b2` pour la seconde et `c0`, `c1`, `c2` pour la troisième.
4. Afin de diminuer le nombre de calcul et d'éviter de refaire plusieurs fois les mêmes calculs par colonnes, calculez des sommes réduites par colonnes (`ra=a0+a1+a2`, `rb=b0+b1+b2` et `rc=c0+c1+c2`) et faire uniquement la rotation de ces variables réduites. Nom de la fonction: `sum3_f32vector_Red`.
5. Enfin faites un déroulage de boucle d'ordre 3 (avec toujours le mécanisme de réduction par colonne) dans la fonction `sum3_f32vector_LU3`. Validez la gestion de l'épilogue sur 3 valeurs contiguës de n .

4 Benchmarking 2D

1. Faire de même que précédemment avec les tailles de données suivantes: 100, 128, 200, 256, 512, 1024. Attention: les temps de calcul augmentant avec le cube de n pour la multiplication matricielle ... donc prévoir le temps de calcul.
2. Collectez les résultats et faire les même analyses que précédemment.