L3 MI / PROG
TP#1: Produit scalaire

2023-09-12

Exercice 1 : produit scalaire, compilation

Cet exercice a pour but de constater l'impact des optimisations de compilation sur un programme très simple.

En fonction de votre maîtrise de la compilation séparée, vous pouvez n'écrire vos fonctions que dans un seul fichier ou bien adopter la structure indiquée dans les questions. Dans ce dernier cas, vous pouvez également écrire un `Makefile` pour piloter votre processus de compilation.

Rappels :

- Pour utiliser le type `size_t`, il faut inclure le fichier `stdlib.h` via la directive préprocesseur `#include <stdlib.h>`.
- Pour utiliser les types `uint32_t`, `uint64_t`, il faut inclure le fichier `stdint.h` via `#include <stdint.h>`.
- Pour afficher les mêmes types via `printf` sur une machine 64 bits, utilisez les formats d'affichage `%u` et `%lu` ou `%llu` respectivement.
- Pour mesure le temps d'exécution d'un programme `prog`, dans un shell UNIX, faire `time prog`.

Q.1 : Implémentez les fonctions de calcul de produit scalaire suivantes pour des vecteurs de dimension `dim` et à coefficients de types respectivement `double`, `float`, `uint64_t`, `uint32_t` :

- `double psd (size_t dim, double x[dim], double y[dim]);`
- `float psf (size_t dim, float x[dim], float y[dim]);`
- `uint64_t psu64(size_t dim, uint64_t x[dim], uint64_t y[dim]);`
- `uint32_t psu32(size_t dim, uint32_t x[dim], uint32_t y[dim]);`

dans un fichier `ps.c`, et déclarez les dans un fichier `ps.h` correspondant.

Q.2 : Implémentez les fonctions de test suivantes :

- `void test_double(size_t dim, int repet)`
- `void test_float (size_t dim, int repet)`
- `void test_u64 (size_t dim, int repet)`
- `void test_u32 (size_t dim, int repet)`

qui déclarent deux vecteurs `x` et `y` de type approprié et de taille `dim` (par ex. via la construction `type x[dim], y[dim];`), les initialisent avec tous leurs coefficients à 1, calculent `repet` fois leur produit scalaire en dimension `dim`, accumulent (c-à-d somment) le résultat (dans une variable du type approprié, initialisée à zéro), et l'affichent sur la sortie standard.

Ces fonctions devront être écrites dans un fichier `psm.c`, qui comportera aussi votre fonction `main`.

Q.3 : Si vous êtes familier avec `make`, écrivez un `Makefile` vous permettant de compiler votre programme et de spécifier aisément le compilateur et les options de compilation à utiliser.

Q.4 : Lancez chacune de vos fonctions de tests avec les arguments 1000 et 10000000 avec au moins deux compilateurs (par ex. clang et gcc) et les options suivantes :

- `-O0`
- `-O2`
- `-O2 -march=native`

Comment pouvez-vous expliquer : 1) les temps de calcul obtenus ; 2) les différentes valeurs des résultats ?

Q.5 ***(Pour aller (beaucoup) plus loin)*** **:**

1. Modifiez vos fonctions de la question 1 en déroulant la boucle, par exemple sur 4 ou 8 instructions. Pour simplifier les choses, vous pouvez supposer que la dimension de l'entrée est divisible par 8. Faites cependant attention à ne pas avoir de dépendances du flot des données entre chaque instruction de votre boucle déroulée [1].
2. Faites à nouveau les tests de la question précédente. Que constatez-vous ?
3. Comment pouvez-vous expliquer ces résultats par la présence d'instructions *vectorielles* sur votre processeur [2] ?
4. Confirmez (ou non) vos hypothèses en inspectant le code assembleur généré par votre compilateur (en utilisant l'option `-S`).

1. Dans la suite d'instruction suivante : `acc += a; acc += b`, il y a une dépendance du flot des données car les deux valeurs `a` et `b` sont ajoutées à la même variable `b`. Il n'y a par contre pas de telle dépendance dans la suite d'instruction `acc0 += a; acc1 += b`.
2. Cf. par exemple : https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=vmulpd&ig_expand=4701