Laboratoire 3: Optimisations de compilation

Département: **TIC**

Unité d'enseignement: **HPC**

Auteur(s):

- **CECCHET Costantino**

Professeur:

- **DASSATTI Alberto**

Assistant:

- **DA ROCHA CARVALHO Bruno**

Date:

- **28/03/2024**

[Page de mise en page, laissée vide par intention]

Introduction

Pour ce laboratoire, nous devons utiliser **godbolt** pour analyser du code C et montrer les différences entre différentes optimisations de compilation.

Je vais utiliser X86-64 GCC 13.2 avec l'option `-O3` et sans optimisation.

Code simple

Voici une fonction tres simple qui calcule le carré d'un nombre:

```
int square(int num) {
    int i = num;
    return i * i;
}
```

Cette fonction est clairement mal écrite car elle utilise une variable inutile. En effet, on pourrait simplement retourner `num * num` et cela fonctionnerait de la même manière.

Mais en voici néanmoins **le code assembleur**.

Si l'on corrige cette erreur, le code sera plus simple et plus rapide:

```
int square(int num) {
    return num * num;
}
```

en voici **le code assembleur**.

nous remarquons tout de suite une amélioration dans le code assembleur généré.

Mais nous pouvons faire mieux grâce à **l'optimisation de compilation `-O3`**.

Nous remarquons que depuis notre code initial, nous avons pu réduire le nombre d'instructions de 9 à 3, ceci va accélérer l'exécution de notre programme.

Grâce a l'utilisation du registre EDI qui contient déjà la valeur de `num`, nous avons pu éviter de faire une opération inutile de déplacement de `num` dans un autre registre.

Ceci est possible grâce aux conventions d'appel de la fonction qui permettent de stocker les paramètres de la fonction dans des registres spécifiques.

Code simple exemple 2

Voici une fonction qui calcule la somme des éléments d'un tableau:

```
int sum(int *arr, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Voici **le code assembleur**.

Nous pouvons optimiser ce code en utilisant un pointeur `p` pour éviter de faire des calculs inutiles à chaque itération de la boucle:

```c
int sum(int *arr, int size) {
    int sum = 0;
    int *p = arr;
    for (int i = 0; i < size; i++) {
        sum += *p++;
    }
    return sum;
}
```

Voici **le code assembleur**.

Puis en utilisant **l'optimisation de compilation** `-O3`:

Ici nous remarquons que le code généré fait le double de la taille du code généré sans optimisation, mais il est plus rapide dans certains cas.

Ceci est du a l'utilisation d'operation SIMD si possible, ce qui permet de faire des calculs sur plusieurs éléments en même temps, ce qui accélère l'exécution de notre programme.

Code simple exemple 3

Voici une fonction qui calcule une valeur absolue:

```c
int abs(int num) {
    if (num < 0) {
        return -num;
    }
    return num;
}
```

Voici **le code assembleur**.

Nous pouvons optimiser ce code en utilisant un masque pour éviter de faire une comparaison inutile ce qui pourrait créer un branchement:

```c
int abs(int num) {
    int mask = num >> 31;
    return (num + mask) ^ mask;
}
```

Cette manipulation de bits permet de faire la même chose que la condition `if (num < 0)`, mais de manière plus rapide sans utiliser de branchement.

Voici **le code assembleur**.

Et pour finir, en utilisant **l'optimisation de compilation** `-O3`:

Nous voyons que le compilateur utili se une instruction `cmovs` pour faire la même chose que notre manipulation de bits mais evite de faire un branchement, ce qui accélère l'exécution de notre programme.

Code plus complexe du Laboratoire 1

Voici une fonction du code du laboratoire 1:

```c
void rgb_to_grayscale_1D(const struct img_1D_t *img, struct img_1D_t *result)
{
```

```
    int components = img->components;

    if (components != COMPONENT_RGB && components != COMPONENT_RGBA)
    {
        fprintf(stderr, "Error: image is not in RGB format\n");
        return;
    }

    int width = img->width;
    int height = img->height;
    int size = width * height * components;
    int j = 0;

    for (int i = 0; i < size; i += components)
    {

        uint8_t r = img->data[i + R_OFFSET];
        uint8_t g = img->data[i + G_OFFSET];
        uint8_t b = img->data[i + B_OFFSET];

        result->data[j++] = (uint8_t)(FACTOR_R * r + FACTOR_G * g + FACTOR_B * b);
    }

    result->width = width;
    result->height = height;
    result->components = COMPONENT_GRAYSCALE;
}
```

Cette fonction génère **un code assembleur assez long**.

Ce code est pas optimisé, on peux le simplifier en utilisant les pointeurs `src` et `dst` pour éviter de faire des calculs inutiles à chaque itération de la boucle, et on peux aussi simplifié la condition de la boucle. On peux optimiser ce code nous même comme suit

```
void rgb_to_grayscale_1D(const struct img_1D_t *img, struct img_1D_t *result)
{
    int components = img->components;

    if (components != COMPONENT_RGB && components != COMPONENT_RGBA)
    {
        fprintf(stderr, "Error: image is not in RGB format\n");
        return;
    }

    int width = img->width;
    int height = img->height;
    int size = width * height;

    uint8_t *src = img->data;
    uint8_t *dst = result->data;
```

```
    for (int i = 0; i < size; i++)
    {
        int index = i * components;
        uint8_t r = src[index + R_OFFSET];
        uint8_t g = src[index + G_OFFSET];
        uint8_t b = src[index + B_OFFSET];

        dst[i] = (uint8_t)(FACTOR_R * r + FACTOR_G * g + FACTOR_B * b);
    }

    result->width = width;
    result->height = height;
    result->components = COMPONENT_GRAYSCALE;
}
```

Ce code génère **le code assembleur suivant**

On peux deja voir que le code est plus court et plus lisible, mais on peux encore l'optimiser en utilisant **l'optimisation de compilation** `-O3`:

Grâce a l'optimisation de compilation `-O3`, on a pu réduire le code assembleur de la moitié environs ce qui va accélérer l'exécution de notre programme.

Cette optimisation se base sur une incrementation basée sur le nombre de composants de l'image, ce qui permet de réduire le nombre d'instructions de la boucle, puis des multiplication directes pour calculer la valeur de la composante de l'image en niveaux de gris sans passer par des variables intermédiaires et ceci aussi dans les calculs RGB.

Tout ceci reduit considérablement le nombre d'instructions et donc accélère l'exécution de notre programme.

Conclusion

Nous avons pu voir que l'optimisation de compilation est très importante pour accélérer l'exécution de notre programme, et que l'optimisation de compilation `-O3` est très efficace pour réduire le code assembleur généré.

Mais il est aussi important de noter, comme vu en cours, que parfois les optimisation ne se font pas car le compilateur n'a pas toutes les informations nécessaires et qu'il faut dont optimiser à la main du code pour que le compilateur puisse faire des optimisations plus poussées.