Examen CEAMC/SEOC3A – Partie 2 – 2021/2022

January 3, 2022

Factorielle

On considère un programme C calculant la factorielle d'un entier naturel positif. On rappelle que factorielle de n, $n!$, est définie comme :

$$n! = \prod_{i=1}^{n} i$$

avec $0! = 1$ par définition de la factorielle. On a donc :

$$1! = 1 \times 0! = 1 \times 1 = 1$$
$$2! = 2 \times 1! = 2 \times 1 \times 1 = 2$$
$$3! = 3 \times 2! = 3 \times 2 \times 1 \times 1 = 6$$

Cette fonction peut être traduite en langage C par le code suivant :

```
unsigned long long fact(unsigned long long n)
{
  unsigned long long result = 1;

  while (n != 0) {
    result = result * n;
    n--;
  }

  return result;
}
```

Ce code C peut-être traduit dans l'assembleur x86_64 ci-après. Ce jeu d'instructions définit 16 registres généralistes 64-bit, nous n'en utilisons que deux ici, *rax* et *rdi*. De plus, ce jeu d'instructions utilise le premier opérande source comme destination, il faut donc par exemple lire *imul rax, rdi* comme $rax = rax * rdi$.

```
                    % n est dans le registre rdi
.fact:
  mov     rax, 1    % result <= 1 (result est dans le registre rax)
  test    rdi, rdi % test fait le ET bit-a-bit et jette le resultat
  je      .end      % Pris si (test rdi, rdi) == 0, sinon non pris
.loop:
  imul    rax, rdi % rax (result) = rax (result) * rdi (n)
  sub     rdi, 1    % rdi (n) = rdi (n) - 1
  jne     .loop     % Pris si (sub rdi, 1) != 0 sinon non pris
.end:
  ret               % Retour de fonction (result est dans registre rax)
```

On note que le code a déjà été optimisé. Premièrement, les variables sont conservées dans des registres, ce qui évite le coût d'accéder à la mémoire à chaque tour de boucle. Deuxièmement, le test de la condition du *while* a été fusionné avec la mise à jour de n : $sub, rdi, 1$[1] met à jour les flags définis dans l'architecture x86_64 (notamment le flag *Zero*[2]), qui sont lus par les branchements conditionnels (il y a donc une dépendance de donnée sur les flags entre *sub rdi, 1* et *jne .loop*). Cela permet de déterminer directement si la boucle doit continuer, plutôt que de sauter vers le début du corps de boucle puis tester la condition et refaire un branchement en conséquence.

1 Ordonnancement

Question 1.1 (0,5pt): Rappelez les différents types de dépendances de données via les registres.

Question 1.2 (0,5pt): Existe-t-il d'autres types de dépendances de données qui pourraient limiter la performance dans ce programme ? En général ?

Question 1.3 (1pt): Mettre en lumière les dépendances de données via les registres présentes dans l'assembleur implémentant la fonction factorielle. Attention à ne pas oublier les dépendances entre plusieurs itérations de la boucle.

Question 1.4 (2pt): En ne considérant que les dépendances de données via les registres mises en lumière à la question précédente, et en considérant que chaque instruction requiert un seul cycle pour s'exécuter, quel est le parallélisme d'instructions présent dans la boucle de la fonction *fact(n)*, en instructions par cycle (IPC), en admettant que n soit grand. Attention, on considère ici que deux instructions dépendantes ne peuvent **jamais** être ordonnancer dans le même cycle, même si certains designs pourraient se le permettre.

Question 1.5 (1,5pt): Rappelez brièvement le principe du renommage de registres et ses avantages, les structures matérielles requises pour l'implémenter et leur(s) rôle(s).

Question 1.6 (1.5pt): Même question que 1.3, mais en ignorant les dépendances de données via les registres que le renommage de registres permet d'ignorer (on considère qu'on a un nombre infini de registres physiques à notre disposition).

[1] C'est à dire $rdi = rdi - 1$

[2] Qui vaut 1 si le résultat de l'opération vaut 0, sinon il vaut 0.

2 Single Instruction Multiple Data

La fonction *fact()* possède du parallélisme au niveau données. On note par exemple que 8! requiert d'effectuer les "blocs" d'opérations suivants:

$$tmp_0 = 8 \times 7$$
$$tmp_1 = 6 \times 5$$
$$tmp_2 = 4 \times 3$$
$$tmp_3 = 2 \times 1$$

Puis:

$$tmp_4 = tmp_0 \times tmp_1$$
$$tmp_5 = tmp_2 \times tmp_3$$

Et enfin:

$$result = tmp_4 \times tmp_5$$

Bien qu'il y ait des dépendances de données entre les différentes étapes, les opérations à l'intérieur d'un bloc sont indépendantes. On peut donc accélérer le calcul de *fact()* en parallélisant les traitements via le paradigme **Single Instruction Multiple Data** (SIMD).

Question 2.1 (0,5pt): Rappelez le principe du paradigme SIMD.

On se place dans le contexte de l'extension vectorielle AVX (*Advanced Vector Extensions*) du jeu d'instructions x86_64. Cette extension permet de manipuler des vecteurs de 4 éléments de 64-bit via des registres et instructions vectorielles dédiés.

Question 2.2 (1pt): Quelle(s) difficulté(s) va-t-on rencontrer lors de la conversion du code scalaire de fact() en code vectoriel avec vecteur de taille 4 ? On pourra notamment se poser la question de l'exécution de *fact(10)*.

Question 2.3 (2.5pt): Fournir le pseudo-code de la fonction fact() en exprimant un parallélisme de données de degré 4 (par exemple en utilisation la notation *var[0,1,2,3] = ...*)

Question 2.4 (1pt): La vectorisation va nous permettre de traiter plus d'éléments à chaque tour de boucle et donc augmenter la performance en augmentant le débit d'instructions. Cependant, étant donné les réponses aux questions 2.2 et 2.3, est-t-il forcément plus efficace d'utiliser la version vectorisée de *fact(n)* (par rapport à la version scalaire) ? On pourra notamment se poser la question de l'exécution de *fact(2)*. On attend naturellement une réponse argumentée.