Performances d'une hiérarchie de mémoires et d'un pipeline processeur

Jérémie Crenne

Mars 2021

1 Introduction

Si écrire du code est une chose relativement aisée, il est parfois plus difficile de l'optimiser pour rendre son exécution performante. L'écriture d'un code n'est donc pas triviale, la manière de décrire un ensemble de tâches à effectuer pouvant ne pas correspondre à une utilisation optimale d'un processeur. Pour se faire, il est nécessaire de comprendre ce qu'il se passe réellement lors de l'exécution d'un programme au sein de l'architecture d'un microprocesseur.

Cette séance de TP est une introduction à la compréhension et au bon usage de la hiérarchie de mémoires et du pipeline d'un processeur. Il existe en effet quelques règles permettant de tirer pleinement profit de l'architecture d'un microprocesseur.

Nous allons effectuer quelques expérimentations autour de ces problématiques en utilisant le C comme langage de programmation. Il possède l'avantage d'être un langage de bas niveau et suffisamment proche du matériel pour en extraire quelques caractéristiques.

N.B : A chaque question posée (qui s'y prête) dans ce sujet, il vous ai demandé de faire le lien entre le logiciel et l'architecture matérielle sous-jacente. Soyez aussi précis que possible dans la rédaction de vos réponses. N'hésitez donc pas à utiliser des tableaux et des schémas pour bien expliquer les situations rencontrées. Une attention toute particulière sera portée au style de votre rapport sachant qu'un rapport rédigé en anglais est un plus !

2 Pré-requis

Pour commencer, démarrez votre ordinateur en choisissant un environnement UNIX. Téléchargez ensuite l'archive *mi201_lab_1_performances_tests* depuis l'adresse *http ://www.jeremiecrenne.com/enseirb-matmeca/MI201* et décompressez la dans un dossier de votre compte utilisateur. Le dossier doit main-

tenant contenir trois fichiers sources écris en langage C (*exercices.c*, *main.c* et *timer.c* et deux fichiers en-têtes *exercices.h* et *timer.h*), deux fichiers Gnuplot (*plot1* et *plot2*), ainsi qu'un fichier Makefile (*makefile*).

2.1 Utilisation du fichier Makefile

Les Makefiles sont des fichiers utilisés par l'outil Make sous UNIX pour exécuter un ensemble d'actions. Il sont très présents dans le domaine de l'informatique, car ils permettent, entre autre, d'automatiser la construction d'un fichier binaire exécutable par le système d'exploitation. Pour effectuer cette automatisation, un fichier Makefile dois répondre à certaines règles d'écriture et est par conséquent toujours structuré dans la forme donnée par le listing 1.

```
regle1: dependance1 dependance2
    commande1

dependance1:
    commande2

dependance2:
    commande3
```

Listing 1 – Mise en forme et syntaxe d'un fichier Makefile

Le fonctionnement du fichier Makefile est assez simple. Il est construit autour de trois éléments centraux :

— les règles
— les dépendances
— les commandes

Une règle possède un ensemble de dépendances et de commandes. Concernant l'exemple du listing 1, la commande *commande1* de la *regle1* n'est exécutée que si les objets *dependance1* et *dependance2* dont dépendent la règle sont créés et à jour. Si ce n'est pas le cas, l'outil make, exécute la règle des dépendances *dependance1* et *dependance2*. Celles si ne comportant pas de dépendance, la commande *commande2* puis la commande *commande3* sont exécutées dans cet ordre. Les objets étant créés et à jour, la commande *commande1* de la règle *regle1* peut désormais être exécutée. Pour exécuter le fichier Makefile, la commande serait la suivante : make *regle1*.

Nous allons nous servir d'un fichier Makefile et de l'outil make pour compiler l'ensemble des fichiers sources du projet et lier les fichiers objets pour générer le binaire exécutable. Travaillant avec le langage C, nous utiliserons le compilateur GCC. Pour rappel, ce compilateur peut être invoqué avec un grand nombre d'options par l'intermédiaire de flags (ou commutateurs). En voici quelques uns pour mieux comprendre le fichier Makefile *makefile* :

— le flag -o spécifie le nom de l'objet
— le flag -c permet la compilation d'un fichier source
— le flag -Wall active l'affichage de tous les messages d'avertissement du compilateur
— le flag -std=c99 permet l'utilisation de la norme standard C99 du langage C

Travail :

— Q1 : étudiez le fichier *makefile* et décrivez précisement son fonctionnement

2.2 Utilisation de Gnuplot

Gnuplot est un outil sous UNIX permettant de tracer des graphiques en deux ou trois dimensions. Pour exécuter Gnuplot, la commande est la suivante : gnuplot *paramètres*. L'argument de Gnuplot est ici un fichier de script de paramètres (voir l'exemple du listing 2) au format ASCII qui permet d'indiquer, par exemple, le type de graphique, sa dimension ou encore son format de sortie.

```
set title 'title'
set xlabel 'labelx'
set ylabel 'labely'
set xtics 1,1,27
set grid
set terminal png size 1024, 768 enhanced font "Helvetica, 11"
set output "plot1.png"
plot "perfgo1" with linespoints
```

Listing 2 – Un exemple de fichier de paramètres Gnuplot

3 Étude de la performance de la hiérarchie de mémoires

L'étude qui suit permet de mettre en évidence les problématiques liées aux mémoires caches et plus généralement à la hiérarchie de mémoires. Pour finement étudier et comparer vos résultats, les commandes *lscpu* et *cat /proc/cpuinfo* peuvent vous aider, notamment pour connaître les tailles des caches. Veuillez noter que selon votre machine, les résultats obtenus peuvent être différents.

3.1 Influence de la taille d'une ligne de cache sur le débit

Travail :

— Q1 : lancez un terminal et utilisez la commande *make all*

— Q2 : exécutez le binaire avec l'option Go1 (./test Go1)et effectuez le tracé de la courbe (le fichier de sortie est nommé *plot1.png*) à l'aide de Gnuplot et du fichier de paramètres *plot1* Le fichier *plot1* utilise les données du fichier de sortie *perfgo1*
— Q3 : expliquez l'allure de la courbe obtenue à l'aide du listing 3 et de son appel depuis le fichier source *main.c* et concluez

```
void Go1( unsigned int count, unsigned int stride ) {
        const unsigned int arrSize = 64 * 1024 * 1024;
        int * ary = (int *)malloc( sizeof( int ) * arrSize );
        StartTimer();
        for ( unsigned int i = 0; i < count; ++i ) {
                for ( unsigned int j = 0; j < arrSize; j +=
                    stride ) {
                        ary[j] *= 17;
                }
        }
        double t = GetTimer();
        PrintFloatToFile( pPerf1, t );
        printf( "(II) Executed in %f ms\n", t );
        free( ary );
}
```

Listing 3 – l'exemple Go1

3.2 Influence de la hiérarchie de mémoires sur le débit

Travail :

— Q1 : exécutez le binaire avec l'option Go2 et effectuez le tracé de la courbe (le fichier de sortie est nommé *plot2.png*) à l'aide de Gnuplot et du fichier de paramètres *plot2*. Le fichier *plot2* utilise les données du fichier de sortie *perfgo2*
— Q2 : expliquez l'allure de la courbe obtenue à l'aide du listing 3 et de son appel depuis le fichier source *main.c*. En quoi elle est caractéristique de la hiérarchie de mémoires du système ? (pensez à faire le lien avec les caractéristiques des mémoires du microprocesseur, en recherchant sa référence sur internet)

```
void Go2( unsigned int steps, unsigned int arrSize, unsigned int
     clSize ) {
        int * ary = (int *)malloc( sizeof( int ) * arrSize );
        unsigned int asm1 = arrSize - 1;
        StartTimer();
        for ( unsigned int i = 0; i < steps; ++i) {
                ary[( i * clSize ) & asm1]++;
        }
        double t = GetTimer();
        PrintFloatToFile( pPerf2, ( ( steps * clSize * 4 ) ) / t
             / 1024 / 1000 );
```

```
        printf( "(II) Executed in %f ms\n", t );
        free( ary );
}
```

Listing 4 – l'exemple Go2

4 Étude de la performance d'un pipeline processeur

Dans cette partie, nous nous intéressons au fonctionnement du pipeline processeur et à son étude avec l'outil Compiler Explorer [1] qui est disponible en ligne. Deux stratégies d'optimisation sont également proposées et décrites.

4.1 Rupture dans un pipeline

Travail :

— Q1 : prenez en main l'outil Compiler Explorer
— Q2 : étudiez le listing 5 avec Compiler Explorer et expliquez pourquoi son exécution est inefficace d'un point de vue de l'utilisation du pipeline processeur
— Q3 : Y-a-t-il une solution pour éviter ce problème ? Si oui, évaluez et comparez les performances sans et avec votre solution

```
void Go3() {
        for ( int i = 0; i < 1000; i++ ) {
                for ( int j = 0; j < 1000; j++ ) {
                        if ( j % 2 == 0 ) {
                                DoSomething( i, j );
                        }
                }
        }
}
```

Listing 5 – l'exemple Go3

4.2 Dépendances dans un pipeline

Travail :

— Q1 : exécutez le binaire avec l'option Go4.
— Q2 : étudiez le listing 6 avec Compiler Explorer et expliquez pourquoi le corps de la seconde boucle for s'exécute plus rapidement que celui de la

1. www.godbolt.org

première.

```
void Go4() {
        int steps = 256 * 1024 * 1024;
        int * a = (int *) malloc( sizeof( int ) * 2 );
        StartTimer();
        for ( int i = 0; i < steps; i++ ) { a[0]++; a[0]++; }
        double t = GetTimer();
        printf( "(II) Executed in %f ms\n", t );
        StartTimer();
        for ( int i = 0; i < steps; i++ ) { a[0]++; a[1]++; }
        t = GetTimer();
        printf( "(II) Executed in %f ms\n", t );
}
```

Listing 6 – l'exemple Go4

4.3 Le "loop unrolling"

Travail :

— Q1 : exécutez le binaire avec l'option Go5.
— Q2 : étudiez le listing 7 avec Compiler Explorer et expliquez pourquoi le corps de la seconde boucle for s'exécute plus rapidement que celui de la première.
— Q3 : proposez une réécriture du listing permettant d'augmenter la vitesse d'exécution. Testez votre proposition. Y-a-t-il un ou des désavantages à votre solution ?

```
void Go5() {
        StartTimer();
        for ( int i = 0; i < 100000000; ++i ) {
                DoSomethingA( i );
        }
        printf( "(II) Executed in %f ms\n", GetTimer() );
        for ( int i = 0; i < 100000000; i += 2 ) {
                DoSomethingA( i );
                DoSomethingA( i + 1 );
        }
        printf( "(II) Executed in %f ms\n", GetTimer() );
}
```

Listing 7 – l'exemple Go5

4.4 Le "software pipelining"

Travail :

— Q1 : exécutez le binaire avec l'option Go6.

— Q2 : étudiez le listing 8 avec Compiler Explorer et expliquez précisément les temps d'exécution obtenus.

```
void Go6() {

        int a1, a2;

        StartTimer();

        for ( int j = 0; j < 10; j++ ) {
                for ( int i = 0; i < 100000000; i += 2 ) {
                        DoSomethingA( i );
                        DoSomethingB( i );
                        DoSomethingA( i + 1 );
                        DoSomethingB( i + 1 );
                }
        }

        printf( "(II) Executed in %f ms\n", GetTimer() );

        StartTimer();

        for ( int j = 0; j < 10; j++ ) {
                for ( int i = 0; i < 100000000; i += 2 ) {
                        a1 = DoSomethingA( i );
                        DoSomethingB( a1 );
                        a2 = DoSomethingA( i + 1 );
                        DoSomethingB( a2 );
                }
        }

        printf( "(II) Executed in %f ms\n", GetTimer() );

        StartTimer();

        for ( int j = 0; j<10; j++ ) {
                for ( int i = 0; i < 100000000; i += 2 ) {
                        a1 = DoSomethingA( i );
                        a2 = DoSomethingA( i + 1 );
                        DoSomethingB( a1 );
                        DoSomethingB( a2 );
                }
        }

        printf( "(II) Executed in %f ms\n", GetTimer() );
}
```

Listing 8 – l'exemple Go6