TPC9 e Guião Laboratorial

Resolução/explicação dos exercícios

1.b) Identifique as instruções responsáveis pelos dois ciclos `for` na função `media` e apresente uma estimativa do número total de instruções executadas nessa função.
Tal como já foi visto em TPCs anteriores, as estruturas de controlo (ciclos) têm uma instrução de salto para um endereço menor, por forma a repetir a execução de um bloco de instruções (nota: um exceção será o caso do ciclo conter poucas iterações, porque o compilador pode remover completamente o ciclo, aplicando a otimização "loop unrolling"). Neste exercício apenas existem duas instruções de salto, cada uma correspondente a um dos ciclos `for` da função.

1º ciclo -> 4*100 = 400 instruções

```
.L14:
        incl    4(%edx,%eax,8)
        incl    %eax
        cmpl    $99, %eax
        jle     .L14
```

2º ciclo -> 4*100 = 400 instruções

```
.L19:
        addl    4(%edx,%eax,8), %ecx
        incl    %eax
        cmpl    $99, %eax
        jle     .L19
```

Outras instruções em media -> 11 instruções

Uma boa estimativa é contabilizar só as instruções dos ciclos, neste caso, seriam 800 instruções.

1.c) Repita a alínea anterior, agora compilando o código sem otimização (`gcc -O0 ...`).

Repetindo o processo anterior obtemos 1000 + 1200 = 2200 instruções.

1.d) Qual será o ganho esperado ao compilar com otimização ($Texe_{O0}/Texe_{O2}$)?

Assumindo que o Tcc será o mesmo nos dois casos, pois o código é executado na mesma máquina, e que o CPI também é igual (o que por norma não acontece), temos IexecO0 /IexecO2 = 2200/800 = 2.75. Logo o ganho esperado seria de 2.75 no tempo de execução.

1.e) Quantas instruções são usadas para calcular `soma/N` nas duas versões? Porquê?

```
        movl    $1374389535, %eax
        mull    %ecx
        movl    %edx, %eax
        shrl    $5, %eax
```

Na versão compilada com O0 é usada mais uma instrução para carregar o valor de soma para registo. Este exercício mostra que o compilador evita, a todo custo, a utilização da divisão (mesmo quando não pretendemos código otimizado!), pois é uma operação com um custo elevado no tempo de execução (ou seja, tem um CPI alto). Neste caso, a divisão é realizada multiplicando por 1/N (nota: ver explicação mais detalhada em anexo). Globalmente, estas instruções reduzem o **Texe.**

Análise da localidade no acesso aos dados

2.a) Compare a forma como as variáveis `i` e `soma` são iniciadas nas versões `O0` e `O2`.

O0 -> variáveis são alocadas em memória (na pilha)

```
        movl    $0, -8(%ebp)
        movl    $0, -4(%ebp)
```

O2 -> variáveis são alocadas em registos

```
        xorl    %ecx, %ecx
        xorl    %eax, %eax
```

Em O0 é utilizado um `movl` de uma constante (a instrução que ocupa mais espaço). No outro caso é utilizado uma operação logica para colocar o registo a 0.

2.b) Indique todas as instruções que acedem à memória na função `media`, compilada com `O2`.

```
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx
...
.L14:
        incl    4(%edx,%eax,8)
        incl    %eax
        cmpl    $99, %eax
        jle     .L14
        xorl    %eax, %eax
.L19:
        addl    4(%edx,%eax,8), %ecx
        incl    %eax
        cmpl    $99, %eax
        jle     .L19
...
        shrl    $5, %eax
        leave
        ret
```

Não esquecer as instruções de `push`, `ret` e `pop` (leem/escrevem valores na pilha).

2.c) Nos acessos aos elementos do vector dentro da função `media` indique a localidade temporal e espacial existente.

Localidade temporal: cada posição do `vec` é acedida duas vezes (uma em cada ciclo, ver 2b)

Localidade espacial: os elementos são acedidos tal como estão armazenados em memória, depois de aceder ao elemento i acedemos ao elemento i+1.

2.d) Altere o código C do programa para tirar melhor proveito da localidade temporal nos acessos ao vector. Estime o ganho em número de acessos à memória desta melhoria.

```
for (i=0 ; i<N ; i++){
  vec[i].a++;
  soma += vec[i].a;
}
```

Neste caso, cada elemento do vetor é acedido uma única vez, logo o ganho é 2 (a 2ª instrução a vermelho em 2b já não é necessária).

2.e) Altere o código C do programa para tirar melhor proveito da localidade espacial nos acessos ao vector. Estime o ganho em termos de *cache misses* desta melhoria.

Uma forma possível será declarar um vetor para os valores de `a` e outro para os valores de `s` (ou seja alterar a estrutura de dados de **A**rray **o**f **S**tructures (AoS) para **S**tructure **o**f **A**rray (SoA)).

Assim, reduzimos para metade o tamanho de cada elemento logo o ganho será cerca de 2.

Anexo - explicação sobre implementar X/N versus X * 1/N

Nesta explicação vamos ver o caso de N=3 com a compilação do código para 64 bits (x86-64), por ser mais simples de entender (o provavelmente o código que irão obter nos seus PCs).

A primeira questão a perceber é que, na multiplicação, a vírgula pode ser colocada apenas no final da operação. Por exemplo, em binário, 110 * 0,101 <=> 110 * 101 * 2^{-3} . Ou seja, em vez de multiplicar 6 por 0,625, multiplica-se 6 por 5 e divide-se o resultado por 8. Recorde-se, que, em binário, a divisão por potências de base 2 é trivial, bastando, neste caso, deslocar os bits do resultado três posições para a direita (*shift right*) colocando 0s no lado esquerdo. Isto permite fazer contas com valores fracionários utilizando apenas aritmética de inteiros.

Vejamos o código gerado pelo `gcc` para o exemplo da `soma/N`, quando compilado para 64 bits (`%rdx` e `%rax` são registos com 64 bits; `%rdx` inclui o valor de `%edx` nos 32 bits menos significativos, e `rax` inclui o valor da `soma` nos 32 bits menos significativos):

```
movl    $2863311531, %edx# 0xAAAA AAAB = 2^32 * 0,6666666667_10
imulq   %rdx, %rax
shrq    $33, %rax         # divide %rax por 2^33
```

Neste exemplo o compilador optou por utilizar $0,6666666667_{10}$ em vez $0,3333333334_{10}$ ou seja, multiplicou por 2*1/3 (=0,66) e dividiu 2*2^{32} (= 2^{33}), por causa dos arredondamentos.

No exemplo da aula é utilizada a mesma estratégia, mas o código é compilado para 32 bits. Nesse caso, o resultado da multiplicação também tem 64 bit, mas tem que ser armazenado num par de registos, cada um com 32 bits (`%edx:%eax`). O valor da `soma` está inicialmente em %`ecx`. Assim:

```
movl    $1374389535, %eax          # = 2^32 * 2^5 * 1/100
mull    %ecx                       # %edx:%eax = %eax*%ecx
movl    %edx, %eax                 # equivalente a dividir por 2^32
shrl    $5, %eax                   # divide por 2^5
```