UNIVERSIDADE FEDERAL DE SANTA CATARINA
EDUARDO BISCHOFF GRASEL - 22200355

Relatório Organização de Computadores

Florianópolis - SC
2024

Parte 1:

Como o início do programa será idêntico ao do lab anterior, irei focar nas novas funcionalidades e lógicas aplicadas ao mesmo. Inicialmente precisaremos calcular a matriz transposta, essa que será salva na memória para efetuar a soma das matrizes depois.

Para começar o cálculo da transposta, devemos percorrer as linhas em um loop externo e colunas em um loop interno, (os valores devem ser informados no .data do programa; (tamanho, linhas, colunas), esses serão recuperados antes do loop, em resumo a lógica será a mesma do lab 6, a diferença está no preenchimento da matriz, mais precisamente no loop interno, onde precisaremos calcular A[i][j], carregar o valor na memória, então armazená-lo em B[j][i]:

```
# Calcular a transposta da matriz A e armazenar em B
la      $t0, A              # carregar o endereço base da matriz A em $t0
la      $t1, B              # carregar o endereço base da matriz B em $t1
lw      $t2, m              # carregar o número de linhas em $t2 (reutilizar $t2 para linhas)
lw      $t3, n              # carregar o número de colunas em $t3 (reutilizar $t3 para colunas)

li      $t4, 0              # inicializar i = 0
```

```
transpose_inner_loop:
        bge     $t5, $t3, end_transpose_inner_loop   # se j >= n, sair do loop interno

        # Calcular endereço de A[i][j]
        mul     $t6, $t4, $t3        # t6 = i * n
        add     $t6, $t6, $t5        # t6 = i * n + j
        sll     $t6, $t6, 2          # t6 = (i * n + j) * 4
        add     $t7, $t0, $t6        # t7 = endereço de A[i][j]

        # Calcular endereço de B[j][i]
        mul     $t8, $t5, $t2        # t8 = j * m
        add     $t8, $t8, $t4        # t8 = j * m + i
        sll     $t8, $t8, 2          # t8 = (j * m + i) * 4
        add     $t9, $t1, $t8        # t9 = endereço de B[j][i]

        lw      $t6, 0($t7)          # carregar A[i][j]
        sw      $t6, 0($t9)          # armazenar A[i][j] em B[j][i]

        addi    $t5, $t5, 1          # j++
        j       transpose_inner_loop
```

Em resumo, foi colocado aqui só o loop interno, pois só a lógica dele mudou em relação às lógicas anteriores, a base permanece a mesma das matrizes percorridas nos laboratórios anteriores.

Agora que temos as duas matrizes, precisamos somá-las, iniciamos recuperando os valores:

```
# Somar as matrizes A e B e armazenar o resultado em A
la      $t0, A              # carregar o endereço base da matriz A em $t0
la      $t1, B              # carregar o endereço base da matriz B em $t1
lw      $t2, m              # carregar o número de linhas em $t2
lw      $t3, n              # carregar o número de colunas em $t3

li      $t4, 0              # inicializar i = 0
```

Do mesmo modo que fizemos anteriormente, vamos focar apenas no loop interno do laço, onde basicamente calcularemos os endereços de A[i][j] e B[i][j], mas como os dois tem os mesmos índices na matriz, podemos facilitar o cálculo, fazendo ele só para um e reutilizando no outro. uma vez tendo os dois valores recuperados, somamos ele e guardamos na mesma posição na matriz A:

```
sum_inner_loop:
        bge     $t5, $t3, end_sum_inner_loop   # se j >= n, sair do loop interno

        # Calcular endereço de A[i][j]
        mul     $t6, $t4, $t3          # t6 = i * n
        add     $t6, $t6, $t5          # t6 = i * n + j
        sll     $t6, $t6, 2            # t6 = (i * n + j) * 4
        add     $t7, $t0, $t6          # t7 = endereço de A[i][j]

        # Calcular endereço de B[i][j]
        add     $t8, $t1, $t6          # t8 = endereço de B[i][j] (mesmo offset de A[i][j])

        lw      $t9, 0($t7)            # carregar A[i][j]
        lw      $t6, 0($t8)            # carregar B[i][j]
        add     $t9, $t9, $t6          # A[i][j] = A[i][j] + B[i][j]
        sw      $t9, 0($t7)            # armazenar resultado em A[i][j]

        addi    $t5, $t5, 1            # j++
        j       sum_inner_loop
```

Parte 2:

Tendo em vista que a parte do cálculo da matriz e sua transposta são idênticos, partiremos para a mudança no cálculo de soma.

a ideia é usar a técnica de blocking cache, portanto precisamos determinar o tamanho do bloco que queremos, isso será definido no .data, agora para percorrer a matriz precisaremos de 4 loops: outer_block_loop, inner_block_loop, ii_loop, jj_loop, a soma será realizada dentro do jj_loop, os loops são destinados a fazer o controle do bloco, formando esse padrão:

```
1  float A[MAX, MAX], B[MAX, MAX];
2  for (i=0; i< MAX; i+=block_size) {
3    for (j=0; j< MAX; j+=block_size) {
4      for (ii=i; ii<i+block_size; ii++) {
5        for (jj=j; jj<j+block_size; jj++) {
6          A[ii,jj] = A[ii,jj] + B[jj, ii];
7        }
8      }
9    }
10 }
```

```
        # Calcular a soma das matrizes A e B utilizando o padrão de bloco
        li      $t5, 0                  # inicializar i = 0

outer_block_loop:
        bge     $t5, $t2, end_outer_block_loop   # se i >= m (linhas), sair do loop externo

        li      $t7, 0                  # inicializar j = 0

inner_block_loop:
        bge     $t7, $t3, end_inner_block_loop   # se j >= n, sair do loop interno

        li      $t8, 0                  # inicializar ii = i
        add     $t8, $t8, $t5           # t8 = i

ii_loop:
        bge     $t8, $t2, end_ii_loop            # se ii >= m, sair do loop

        li      $t9, 0                  # inicializar jj = j
        add     $t9, $t9, $t7           # t9 = j
```

```
jj_loop:
        bge     $t9, $t3, end_jj_loop            # se jj >= n, sair do loop

        mul     $s0, $t8, $t3           # s0 = ii * n
        add     $s0, $s0, $t9           # s0 = ii * n + jj
        sll     $s0, $s0, 2             # s0 = (ii * n + jj) * 4
        add     $s1, $t0, $s0           # s1 = endereço de A[ii][jj]

        mul     $s2, $t9, $t2           # s2 = jj * m
        add     $s2, $s2, $t8           # s2 = jj * m + ii
        sll     $s2, $s2, 2             # s2 = (jj * m + ii) * 4
        add     $s3, $t1, $s2           # s3 = endereço de B[jj][ii]

        lw      $s4, 0($s1)             # carregar A[ii][jj]
        lw      $s5, 0($s3)             # carregar B[jj][ii]
        add     $s4, $s4, $s5           # A[ii][jj] = A[ii][jj] + B[jj][ii]
        sw      $s4, 0($s1)             # armazenar resultado em A[ii][jj]

        addi    $t9, $t9, 1             # jj++
        j       jj_loop

end_jj_loop:
        add    $t8, $t8, 1              # ii++
        j       ii_loop

end_ii_loop:
        add    $t7, $t7, $t4            # j += block_size
        j       inner_block_loop
```

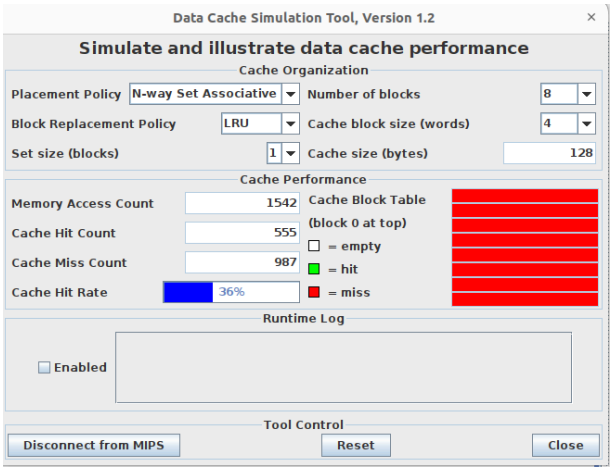
```
end_inner_block_loop:
        add    $t5, $t5, $t4            # i += block_size
        j       outer_block_loop

end_outer_block_loop:

        # Fim do programa
        li      $v0, 10                 # código para sair do programa
        syscall
```

Como podemos ver, apenas copiamos os loops dados no enunciado, transferindo para assembly e inicializamos as variáveis i, j, ii, jj, para realizar os cálculos, dentro do jj_loop, faremos a adição das duas matrizes, seguindo o padrão de bloco.

Parte 3:

Rápida comparação entre os 2 códigos e como eles são apresentados com os mesmos dados de simulação:

Parte 1:



Parte 2:

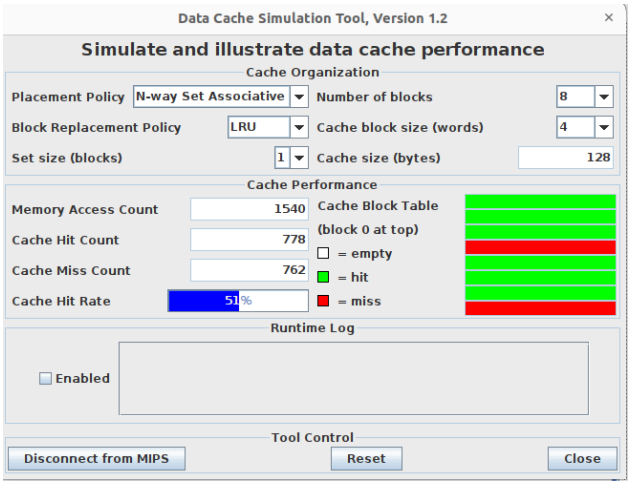


Tabela de comparação dos cache misses OBS: no código na parte dois, em cada análise eu alterei o valor do tamanho do bloco para se encaixar com aquele dado no simulador.

	Hit rate			
	8 blocos 4 block size	8 blocos 8 block size	16 blocos 8 block size	16 blocos 16 block size
Parte 1:	36%	42%	43%	60%
Parte 2:	51%	56%	58%	90%

Em ambos os casos, conforme a cache aumenta o hit rate aumenta, entretanto pela técnica 2 envolver diretamente a manipulação da matriz para que ela seja otimizada em relação ao cache block size definido previamente, acaba por aumentar mais, tanto que a partir do momento que o block size foi igual a dimensão da matriz testada (16x16) o aumento foi absurdo, diferente da técnica 1, a qual não pode fazer otimizações relacionadas ao block cache size, apenas sendo executada de forma trivial.

Conclusão:

Ao fim do trabalho foi visível o quão não trivial é trabalhar com matrizes em assembly mips, ainda mais quando essas matrizes são usadas dentro de muitos laços de repetição, é extremamente confuso, além de que precisamos tratar um “array” como uma matriz e fazer funcionar como uma matriz. Fora isso foi interessante ver que o código mais complexo e confuso, cujo ao bater o olho é quase impossível imaginar ser mais eficiente, é de fato muito superior ao seu competidor por explorar diretamente a cache e seu funcionamento.