**Relatório 8 – Organização de Computadores I – INE5411**

**Alunas: Ana Luiza Sales Gobbi (24105453), Fernanda Bertotti Guedes (24100601) e Laísa Ágathe Bridi Dacroce (24100598)**

**Data: 11/06/25**

# 1 INTRODUÇÃO

O presente relatório visa descrever as soluções desenvolvidas para os exercícios propostos no Laboratório 8 da disciplina de Organização de Computadores I. Nesta atividade, foi possível explorar a influência do padrão de acesso à memória sobre o desempenho do cache, por meio de duas implementações de soma entre uma matriz e a transposta de outra. Além disso, foram observados os efeitos da técnica de cache blocking, destacando a importância da localidade espacial para a eficiência da memória cache.

# 2 EXERCÍCIOS PROPOSTOS

## 2.1 EXERCÍCIO 1

O primeiro exercício propôs a implementação de um algoritmo, em linguagem Assembly MIPS, que realiza a soma de duas matrizes A e $B^t$, onde $B^t$ é a transposta da matriz B. Ambas as matrizes são quadradas, com dimensão MAX x MAX, e os elementos são representados em ponto flutuante de precisão simples (.float).

Para resolver esse problema, foi utilizado um loop duplo para percorrer todos os elementos A[i][j] e B[j][i]. O acesso à transposta foi feito invertendo os índices ao acessar B. O valor somado A[i][j] + B[j][i] foi armazenado de volta na posição correspondente de A[i][j], substituindo seu valor original. Para isso, os endereços de memória foram calculados com base na fórmula (índice * MAX + índice) * 4, visto que cada elemento .float ocupa 4 bytes.

A lógica foi implementada utilizando dois registradores de controle de loop ($t1 para as linhas e $t2 para as colunas) e instruções como l.s, add.s e s.s para manipulação dos valores em ponto flutuante simples. Após a realização da soma, foi incluído um procedimento para impressão da matriz A atualizada, linha a linha, utilizando chamadas syscall para exibição de floats no console e uma quebra de linha ao final de cada linha da matriz. Essa

etapa final possibilita a visualização do resultado da operação e a verificação da correção do algoritmo.

Além disso, para verificar o correto funcionamento do algoritmo, realizamos o cálculo manual da matriz 3x3 original, considerando a operação A[i][j] + B[j][i], e utilizamos esse resultado como base de comparação. Ainda, inserimos um trecho de código no final do programa para imprimir a matriz A atualizada no console, o que permitiu validar os valores obtidos diretamente no MARS. A inclusão desse print facilitou a depuração e serviu como apoio visual no teste dos resultados. As figuras a seguir mostram o trecho responsável pela impressão e a saída gerada no console.

```
60  li $t1, 0        # i = 0
61
62  print_linhas:
63      li $t2, 0        # j = 0
64
65  print_colunas:
66      # endereço de A[i][j]
67      mul $t3, $t1, $s0
68      add $t3, $t3, $t2
69      sll $t3, $t3, 2
70      add $t4, $s1, $t3
71
72      l.s $f12, 0($t4)      # carrega A[i][j] em $f12
73      li $v0, 2            # syscall 3 = imprimir double
74      syscall
75
76      # Imprimir espaço (opcional)
77      li $v0, 4
78      la $a0, newline       # usar newline como espaço temporário (poderia ser " ")
79      syscall
80
81      addi $t2, $t2, 1
82      blt $t2, $s0, print_colunas
83
84      # Nova linha
85      li $v0, 4
86      la $a0, newline
87      syscall
88
89      addi $t1, $t1, 1
90      blt $t1, $s0, print_linhas
91
```

**Figura 1 -** Trecho do código responsável pela impressão da matriz A no console.

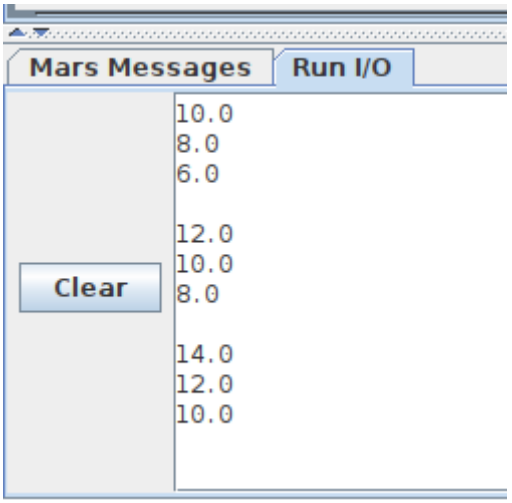


**Figura 2 -** Resultado da matriz A após a operação A + $B^t$, exibido no console do MARS.

## 2.2 EXERCÍCIO 2

O segundo exercício teve o mesmo objetivo do primeiro (somar a matriz A com a transposta da matriz B), porém com a aplicação da técnica de cache blocking. A ideia principal é dividir a matriz em blocos menores de tamanho block_size x block_size e fazer a soma bloco a bloco, com o intuito de melhorar a organização do acesso à memória.

Assim como no exercício anterior, as matrizes A e B são quadradas, com dimensão MAX x MAX, e os elementos são representados em ponto flutuante de precisão simples (.float). Os valores de MAX e block_size foram definidos no segmento .data, permitindo que o tamanho das matrizes e dos blocos seja facilmente alterado para testes e simulações.

O código foi estruturado com quatro loops aninhados, representando a varredura por blocos e, dentro de cada bloco, a varredura elemento por elemento:

- Laço externo: varre os blocos na vertical (i);
- Segundo laço: varre os blocos na horizontal (j);
- Terceiro laço: percorre as linhas dentro do bloco (ii);
- Quarto laço: percorre as colunas dentro do bloco (jj).

A operação A[ii][jj] += B[jj][ii] foi realizada nos loops internos, acessando corretamente os índices invertidos da transposta e armazenado o novo valor em A. Além disso, ao final da operação, foi implementado um procedimento de impressão da matriz A no console, utilizando chamadas syscall:

- syscall 2 para imprimir valores de ponto flutuante (float);
- syscall 4 para pular linha, utilizando a string \n definida no segmento .data.

## 2.3 EXERCÍCIO 3

Para a análise da cache dos exercícios 1 e 2, optamos pelo uso do mapeamento direto, pois, das três políticas de mapeamento possíveis da cache, essa apresenta o pior cenário para o desempenho do código, devido ao aumento da ocorrência de conflitos em razão de sua menor flexibilidade. Portanto, foi possível concluir as melhores configurações da cache para um cenário não favorável.

Nas análises do exercício 1, primeiramente fixamos o tamanho da cache em 128 bytes (8 blocos de 4 words) e variamos apenas o tamanho das matrizes A e B. Foi possível notar que, se a soma da quantidade de bytes das matrizes A e B for igual ou inferior ao tamanho da cache, o código do exercício 1 apresenta boas taxas de acerto. Por exemplo, para matrizes 3x3 e 4x4, de tamanhos 36 e 64 bytes, respectivamente, a taxa de acertos de ambas é 82%.

Por outro lado, ao manipular matrizes que, juntas, excedem o tamanho da cache, as taxas de falha aumentam. Isso pode ser verificado ao usar matrizes 5x5 e 6x6, de 100 e 144 bytes, respectivamente, as quais apresentaram taxas de acertos de 75% e 65%.

Em uma segunda análise, aumentando o tamanho da cache para 256 bytes (mudando o tamanho do bloco na cache para 8 words), a taxa de acertos para matrizes 5x5 subiu para 91%. Isso pode ser explicado, uma vez que agora ambas matrizes A e B cabem na cache. Essa é uma tendência que se segue para quando o tamanho das matrizes é limitado ao tamanho da cache.

Entretanto, mesmo quando as matrizes cabem inteiramente na cache, o desempenho não atinge 100% de acertos devido a conflitos no mapeamento direto, que fazem blocos diferentes da memória competirem pelo mesmo índice da cache, causando substituições desnecessárias. Além disso, o acesso transposto à matriz B (na forma B[j][i]) resulta em baixa localidade espacial, pois percorre colunas em vez de linhas, gerando mais falhas.

Para o exercício 2, testamos diferentes configurações de block size e número de blocos da cache para verificar como esses parâmetros afetam o desempenho. Observamos que, mantendo o cache em 128 bytes, aumentar o tamanho dos blocos (block_size) melhora a taxa de acerto, principalmente quando o tamanho do bloco é suficiente para armazenar linhas completas ou grandes porções da matriz. Por exemplo, ao configurar o block_size como 16 ou 32, a taxa de acerto aumentou significativamente, chegando a 99% e 100%, respectivamente, mesmo com matrizes maiores como 5x5.

Por outro lado, diminuir o block_size compromete o desempenho, com block_size = 2, por exemplo, o desempenho caiu para 72%, e com block_size = 1, reduziu-se ainda mais

para 44%. Isso ocorre porque blocos menores reduzem a localidade espacial e exigem mais acessos distintos à memória principal, aumentando os misses.

Nossa suposição inicial seria que o melhor desempenho para as simulações do exercício 2 ocorreria com o block_size igual ao cache block size. Porém, esse não foi necessariamente o caso nos testes: em geral, um cache block size igual ou maior que block_size permitiu um aumento na taxa de acerto da cache. Através desse exercício, foi possível perceber que a otimização proporcionada pelo algoritmo cache blocking não depende apenas do tamanho da linha de cache.

Constatamos o benefício do uso da técnica de cache blocking no exercício 2, uma vez que, com a alteração dos parâmetros block_size (no código) e cache block size (no simulador), foi possível alcançar uma taxa de acertos da cache de 100%, mesmo com a política de mapeamento direto. Nesse sentido, a ocorrência de conflitos em cache pôde ser mitigada pela reutilização de dados.