

Universidade Federal de Santa Catarina  
Centro Tecnológico  
Departamento de Informática e Estatística  
Ciência da Computação  
INE5411 - Organização de Computadores I

## **Relatório Laboratório 8**

Joshua Cruz do Amaral (24205457)  
Julia Macedo de Castro (23250860)

Florianópolis  
2025

## Questão 1: Soma matriz transposta (sem cache blocking)

Exemplo do funcionamento da soma das matrizes (A+B):

```
A: .float 1.0, 1.0, 1.0, 1.0
    .float 1.0, 1.0, 1.0, 1.0
    .float 1.0, 1.0, 1.0, 1.0
    .float 1.0, 1.0, 1.0, 1.0

B: .float 2.0, 0.0, 0.0, 0.0
    .float 0.0, 2.0, 0.0, 0.0
    .float 0.0, 0.0, 2.0, 0.0
    .float 0.0, 0.0, 0.0, 2.0
```

ssages	Run I/O
3.0 1.0 1.0 1.0	
1.0 3.0 1.0 1.0	
1.0 1.0 3.0 1.0	
1.0 1.0 1.0 3.0	
-- program is finished running --	

```

30  LOOP_j:
31      bge $t1, $s2, END_j # if (j >= size)
32
33
34      # Calcula endereco de A[i,j]
35      mul $t3, $t0, $s2
36      add $t3, $t3, $t1
37      sll $t4, $t3, 2
38      add $t5, $s0, $t4
39
40      # Carrega A[i,j] como float
41      l.s $f0, 0($t5)
42
43      # Calcula endereco de B[j,i]
44      mul $t3, $t1, $s2
45      add $t3, $t3, $t0
46      sll $t4, $t3, 2
47      add $t7, $s1, $t4
48
49      # Carrega B[j,i] como float
50      l.s $f2, 0($t7)
51

```

A implementação foi realizada com dois loops aninhados, usando os registradores \$t0 para o índice i e \$t1 para o j. Os endereços base das matrizes A e B foram mantidos em \$s0 e \$s1.

O núcleo da lógica está no LOOP\_j. Para acessar A[i,j], o *offset* em bytes foi calculado como  $((i * MAX) + j) * 4$ . Para o acesso já transposto a B[j,i], a lógica dos índices foi invertida, calculando o *offset* como  $((j * MAX) + i) * 4$ .

Finalmente, as instruções de ponto flutuante l.s (load), add.s (soma) e s.s (store) foram usadas para realizar a operação  $A[i,j] = A[i,j] + B[j,i]$ .

## Questão 2: Soma matriz transposta (com cache blocking)

Exemplo do funcionamento da soma das matrizes (A+B):

```
A:      .float 1.0, 1.0, 1.0, 1.0
        .float 1.0, 1.0, 1.0, 1.0
        .float 1.0, 1.0, 1.0, 1.0
        .float 1.0, 1.0, 1.0, 1.0

B:      .float 22.0, 0.0, 0.0, 0.0
        .float 0.0, 2.0, 0.0, 0.0
        .float 0.0, 0.0, 2.0, 0.0
        .float 0.0, 0.0, 0.0, 2.0
```

```
23.0 1.0 1.0 1.0
1.0 3.0 1.0 1.0
1.0 1.0 3.0 1.0
1.0 1.0 1.0 3.0

-- program is finished running --
```

```

LOOP_i:
    bge $t0, $s2, PRINT_A # if (i >= MAX) vai para PRINT_A

    li $t1, 0 # j = 0
LOOP_j:
    bge $t1, $s2, END_j # if (j >= MAX) vai para END_j

    # Limites dos blocos
    # limit_i = min(i + BLOCK_SIZE, MAX)
    add $t4, $t0, $s3 # $t4 = i + BLOCK_SIZE
    blt $t4, $s2, set_limit_i
    move $t4, $s2 # limit_i = MAX
    j set_ii
set_limit_i:
    move $t4, $t4 # limit_i = i + BLOCK_SIZE

set_ii:
    # limit_j = min(j + BLOCK_SIZE, MAX)
    add $t5, $t1, $s3 # $t5 = j + BLOCK_SIZE
    blt $t5, $s2, set_limit_j
    move $t5, $s2 # limit_j = MAX
    j start_inner_loops
    . . .

```

Esta implementação utiliza a técnica de *cache blocking* para melhorar a localidade de referência. A lógica foi alterada de dois para quatro loops aninhados.

1. Loops Externos (LOOP\_i, LOOP\_j): Estes loops iteram sobre a matriz em "blocos". Em vez de incrementar por 1 (addi), eles avançam pulando o tamanho do bloco (add \$t1, \$t1, \$s3), onde \$s3 armazena o BLOCK\_SIZE.
2. Loops Internos (LOOP\_ii, LOOP\_jj): Estes loops (\$t2 para ii, \$t3 para jj) iteram sobre os elementos *dentro* do bloco atual. Eles vão de i até i + BLOCK\_SIZE e de j até j + BLOCK\_SIZE, respectivamente.

O cálculo do endereço e a soma de ponto flutuante dentro do loop mais interno (LOOP\_jj) são idênticos à versão anterior, mas agora usam os iteradores de bloco ii (\$t2) e jj (\$t3). Isso garante que o programa processe um conjunto menor da matriz (o bloco) de cada vez, mantendo os dados acessados na cache.

## 1. Taxa de acertos (Cache Hit Rate)

Implementação sem cache blocking

Tamanho da matriz	Tamanho da Cache (nº blocos * tamanho do bloco)	Taxa de acertos
2x2	4x4 (64 bytes)	69%
4x4	8x4 (128 bytes)	90%
4x4	16x8 (512 bytes)	95%
8x8	16x8 (512 bytes)	97%

Implementação com cache blocking

Tamanho da matriz	Tamanho do bloco	Tamanho da Cache (nº blocos * tamanho do bloco)	Taxa de acertos
2x2	1	4x4 (64 bytes)	67%
2x2	2	4x4 (64 bytes)	67%
4x4	2	8x4 (128 bytes)	91%
4x4	2	16x8 (512 bytes)	95%
8x8	4	16x8 (512 bytes)	97%

## 2. Análise da tabela

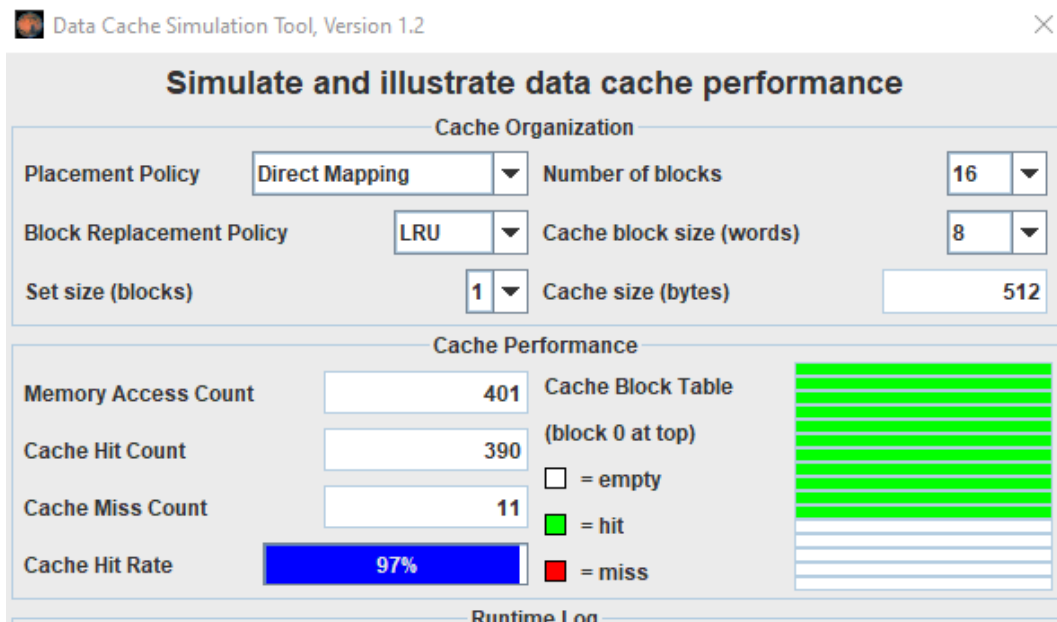
O aumento do **tamanho da cache** melhora a taxa de acertos significativamente nas duas implementações, pois as caches maiores conseguem guardar mais dados e reduzir os acessos à memória principal.

A versão da soma de matrizes com **cache blocking** tem uma taxa de acertos um pouco maior em geral, pois o cache blocking organiza o acesso à memória em blocos menores, assim os dados mais usados ficam na cache por mais tempo. Reduz misses principalmente em matrizes maiores.

O aumento do **block size** também aumenta a taxa de acertos, pois traz mais palavras consecutivas da memória principal por acesso.

## 3. Prints de resultados do Data Cache Simulator

1. Implementação sem cache blocking, matriz 8x8 e cache size de 512 bytes:



2. Implementação com cache blocking, matriz 8x8 e cache size de 512 bytes:

Data Cache Simulation Tool, Version 1.2

### Simulate and illustrate data cache performance

#### Cache Organization

Placement Policy	Direct Mapping	Number of blocks	16
Block Replacement Policy	LRU	Cache block size (words)	8
Set size (blocks)	1	Cache size (bytes)	512

#### Cache Performance

Memory Access Count	402	Cache Block Table (block 0 at top) <input type="checkbox"/> = empty <input checked="" type="checkbox"/> = hit <input type="checkbox"/> = miss	
Cache Hit Count	391		
Cache Miss Count	11		
Cache Hit Rate	97%		