



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE  
CENTRO DE TECNOLOGIA  
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO  
CURSO DE ENGENHARIA DE COMPUTAÇÃO

**Relatório da Tarefa 01 - Localidade Temporal e Espacial de Cache**  
**DCA3703 - PROGRAMAÇÃO PARALELA - T01 (2025.2)**

WERBERT ARLES DE SOUZA BARRADAS

20250070655

Docente: Professor Doutor SAMUEL XAVIER DE SOUZA

Natal, 20 de agosto de 2025

# Lista de Figuras

Figura 2 – Gráficos: Impacto do Padrão de Acesso à Memória na Performance (Escala Logarítmica) e Crescimento do Fator de Lentidão vs. Tamanho da Matriz. . . . .	9
--	---

# Sumário

	<b>Lista de Figuras</b>	<b>2</b>
	<b>Sumário</b>	<b>3</b>
<b>1</b>	<b>INTRODUÇÃO</b>	<b>4</b>
<b>2</b>	<b>METODOLOGIA DO EXPERIMENTO</b>	<b>5</b>
2.1	Função 1: Acesso por Linhas (Cache-Friendly)	5
2.2	Função 2: Acesso por Colunas (Cache-Unfriendly)	5
2.3	Hipótese e Mecanismo de Cache	6
2.4	Procedimentos de Teste	6
<b>3</b>	<b>RESULTADOS</b>	<b>8</b>
<b>4</b>	<b>GRÁFICOS</b>	<b>9</b>
<b>5</b>	<b>CONCLUSÃO</b>	<b>10</b>
	<b>Anexo A: Código Completo</b>	<b>11</b>

# 1 Introdução

O objetivo deste estudo é demonstrar e quantificar o impacto da localidade de dados no desempenho de algoritmos computacionais. O foco é identificar o ponto a partir do qual o padrão de acesso à memória — sequencial (linha a linha) versus não sequencial (coluna a coluna) — causa uma divergência significativa no tempo de execução. Este fenômeno está diretamente relacionado à arquitetura da hierarquia de cache do processador.

A tarefa consistiu em implementar duas versões da multiplicação de matriz por vetor ( $M \times V$ ) em C: uma com acesso à matriz por linhas e outra por colunas. O tempo de execução de cada versão foi medido para matrizes de diferentes tamanhos, a fim de identificar a partir de que ponto os tempos divergem significativamente e explicar a causa, relacionando-a com o uso da memória cache.

## 2 Metodologia do Experimento

Para isolar o efeito do padrão de acesso, foram implementadas duas versões funcionalmente idênticas de uma função de multiplicação de matriz por vetor. A única diferença entre elas é a ordem de iteração dos laços de repetição.

### 2.1 Função 1: Acesso por Linhas (Cache-Friendly)

Nesta implementação, o laço interno percorre as colunas de uma determinada linha. Este padrão resulta em acessos a elementos de memória contíguos (ex:  $A[i][0]$ ,  $A[i][1]$ ,  $A[i][2]$ , ...), o que se alinha com a forma como a linguagem C organiza matrizes na memória (layout **row-major**).

```
void multiply_matrix_vector(int rows, int cols, double **A, double *x,
    ↪ double *y) {
    for (int i = 0; i < rows; i++) {
        y[i] = 0.0;
        // O laço externo itera sobre as LINHAS (i) da matriz A.
        for (int j = 0; j < cols; j++) {
            // O laço interno itera sobre as COLUNAS (j).
            y[i] += A[i][j] * x[j];
        }
    }
}
```

Listing 1 – Código com acesso otimizado para cache (por linhas)

### 2.2 Função 2: Acesso por Colunas (Cache-Unfriendly)

Nesta versão, a ordem dos laços é invertida: o laço interno percorre as linhas de uma determinada coluna. Isso leva a um padrão de acesso não sequencial, acessando elementos de memória distantes entre si (ex:  $A[0][j]$ ,  $A[1][j]$ ,  $A[2][j]$ , ...).

```
void multiply_matrix_vector_cols_outer(int rows, int cols, double **A,
↪ double *x, double *y) {
    for (int i = 0; i < rows; i++) {
        y[i] = 0.0;
    }
    // O laço externo itera sobre as COLUNAS (j) da matriz A.
    for (int j = 0; j < cols; j++) {
        // O laço interno itera sobre as LINHAS (i).
        for (int i = 0; i < rows; i++) {
            y[i] += A[i][j] * x[j];
        }
    }
}
```

Listing 2 – Código com acesso não otimizado para cache (por colunas)

## 2.3 Hipótese e Mecanismo de Cache

A hipótese é que o acesso por linhas terá um desempenho superior devido ao princípio da **localidade espacial**. A CPU carrega dados da RAM em blocos contíguos chamados "linhas de cache"(geralmente 64 bytes).

- **Acesso por Linhas:** Ao solicitar  $A[i][0]$ , a CPU carrega também os elementos vizinhos ( $A[i][1]$ ,  $A[i][2]$ , etc.) na mesma linha de cache. As iterações seguintes encontram os dados necessários no cache ultrarrápido, resultando em um *cache hit*.
- **Acesso por Colunas:** O acesso a  $A[1][j]$  após  $A[0][j]$  requer um bloco de memória completamente diferente. Isso força a CPU a descartar a linha de cache anterior e buscar uma nova na RAM, causando um *cache miss* e degradação de performance.

## 2.4 Procedimentos de Teste

Os testes foram realizados com matrizes quadradas ( $N \times N$ ), com  $N$  crescendo de 32 a 16384.

- **Compilação:** O código foi compilado com GCC, utilizando o nível de otimização `-O2` e a flag `-fopenmp`.
- **Medição de Tempo:** Para garantir robustez contra flutuações do sistema, o tempo de execução registrado para cada tamanho de matriz é a **mediana** de múltiplas execuções.

- **Coleta de Dados:** Os resultados foram gerados em formato CSV, incluindo o tamanho da matriz (N), os tempos de execução e o "Fator de Lentidão".
- **Fator de Lentidão:** Esta métrica normaliza os resultados para análise e é calculada como:

$$\text{Fator de Lentidão} = \frac{\text{Tempo do método por Colunas}}{\text{Tempo do método por Linhas}}$$

Um fator de 5.0, por exemplo, significa que o acesso por colunas foi 5 vezes mais lento.

### 3 Resultados

Os dados de cache da máquina de teste são:

- **Cache L1:** 96 KB
- **Cache L2:** 2.5 MB
- **Cache L3:** 6 MB

A tabela abaixo resume os tempos de execução medianos e o Fator de Lentidão calculado para cada tamanho de matriz.

Tabela 1 – Tempos de execução e Fator de Lentidão.

Tamanho (N x N)	Memória	Tempo Linhas (s)	Tempo Colunas (s)	Fator Lentidão
32 x 32	~8 KB	0.000000	0.000000	1.000
64 x 64	~32 KB	0.000000	0.000000	1.000
128 x 128	~128 KB	0.000000	0.000000	1.000
256 x 256	~512 KB	0.000000	0.000000	1.000
512 x 512	~2 MB	0.000000	0.000000	1.000
1024 x 1024	~8 MB	0.000999	0.002000	2.000
2048 x 2048	~32 MB	0.003999	0.013999	3.500
4096 x 4096	~128 MB	0.017999	0.123000	6.833
8192 x 8192	~512 MB	0.086999	0.629000	7.230
16384 x 16384	~2 GB	0.291000	4.239000	14.567



## 4 Gráficos

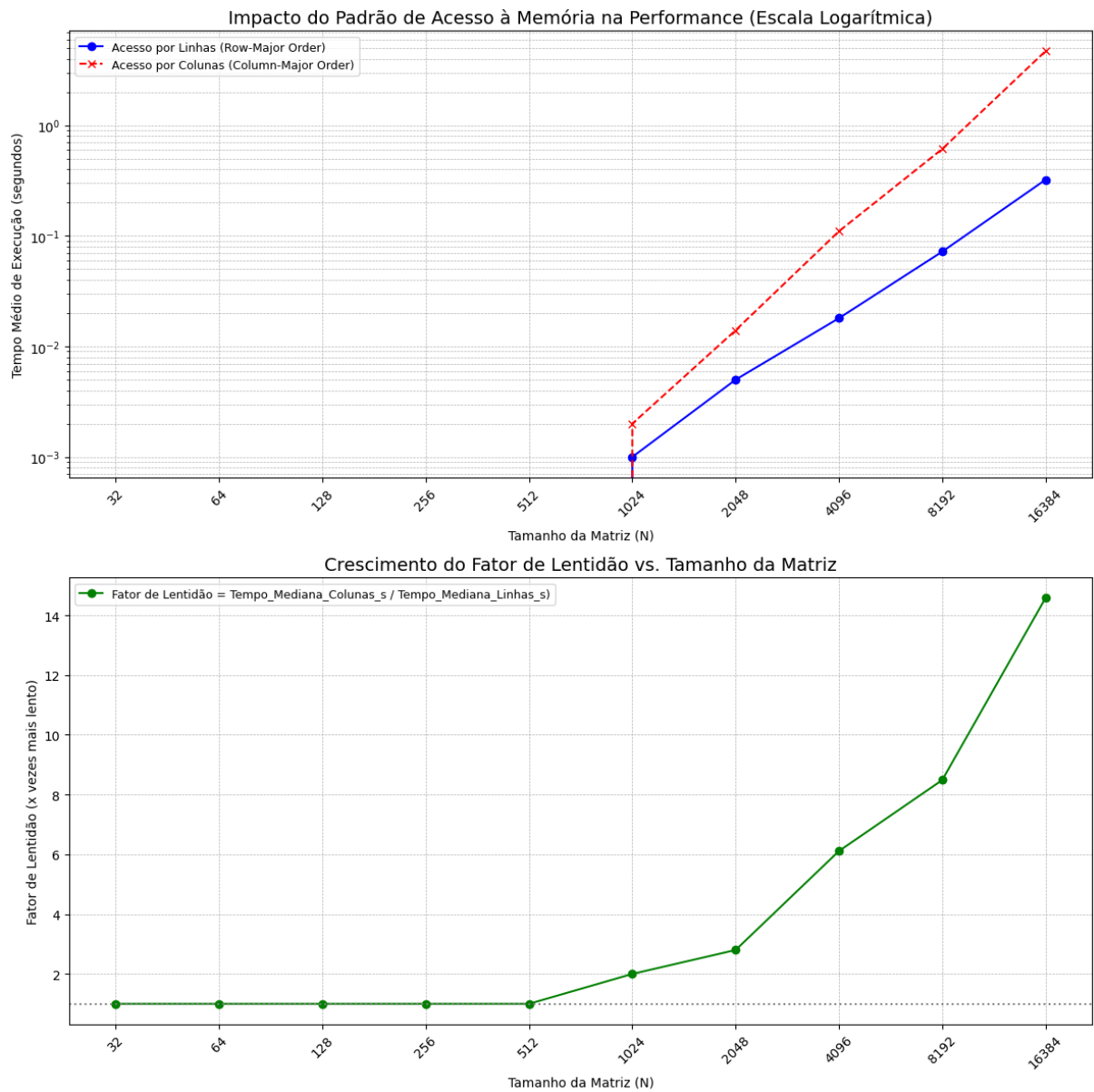


Figura 2 – Gráficos: Impacto do Padrão de Acesso à Memória na Performance (Escala Logarítmica) e Crescimento do Fator de Lentidão vs. Tamanho da Matriz.

## 5 Conclusão

O presente estudo validou experimentalmente a hipótese de que o desempenho de operações computacionais intensivas é profundamente influenciado pela maneira como os dados são acessados na memória. Ao comparar duas implementações da multiplicação de matriz por vetor — uma alinhada com o layout *row-major* da memória (acesso por linhas) e outra que o contraria (acesso por colunas) — foi possível quantificar o custo de um padrão de acesso não otimizado em termos de tempo de execução.

Respondendo à questão central do projeto, **a partir de que tamanho os tempos de execução passam a divergir significativamente e por quê?**, a análise dos dados revelou uma correlação direta com a hierarquia de cache do sistema:

A divergência de performance inicia-se de forma mensurável quando a matriz de dados excede a capacidade do cache L2. A partir deste limiar, o método de acesso por colunas, que não explora a localidade espacial, sofre uma penalidade crescente devido a *cache misses* no nível L2, forçando buscas no cache L3, que possui maior latência.

Posteriormente, a degradação de desempenho se agrava drasticamente quando o volume de dados ultrapassa a capacidade do cache L3. Neste ponto, o acesso não sequencial resulta em falhas de cache constantes em todos os níveis, tornando a latência da memória RAM o fator limitante da velocidade. O Fator de Lentidão, que mede a ineficiência do método não otimizado, dispara para valores superiores a 14x, ilustrando um colapso na performance.

## mxv\_teste\_grafico\_v1.c

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
5
6  // Função de comparação para usar com qsort em um array de doubles.
7  int compare_doubles(const void *a, const void *b) {
8      double da = *(const double *)a;
9      double db = *(const double *)b;
10     if (da > db) return 1;
11     if (da < db) return -1;
12     return 0;
13 }
14
15 // Aloca dinamicamente uma matriz 2D.
16
17 double **create_matrix(int rows, int cols) {
18     // Aloca um array de ponteiros (um para cada linha)
19     double **matrix = (double **)malloc(rows * sizeof(double *));
20     if (matrix == NULL) return NULL;
21
22     // Para cada ponteiro de linha, aloca a memória para as colunas
23     for (int i = 0; i < rows; i++) {
24         matrix[i] = (double *)malloc(cols * sizeof(double));
25         if (matrix[i] == NULL) {
26             // Se falhar no meio, libera o que já foi alocado
27             for(int k = 0; k < i; k++) free(matrix[k]);
28             free(matrix);
29             return NULL;
30         }
31     }
32     return matrix;
33 }
34
35 // Libera a memória de uma matriz alocada dinamicamente.
36
37 void free_matrix(int rows, double **matrix) {
38     if (matrix == NULL) return;
39     // Primeiro, libera a memória de cada linha
40     for (int i = 0; i < rows; i++) {
41         free(matrix[i]);
42     }
43     // Finalmente, libera o array de ponteiros
44     free(matrix);
45 }
46
47 // Preenche uma matriz e um vetor com valores aleatórios.
48
49 void fill_random_data(int rows, int cols, double **matrix, double *vector) {
50     // Usa o tempo atual como semente para o gerador de números aleatórios
51     srand(time(NULL));
```

```
52     for (int i = 0; i < rows; i++) {
53         for (int j = 0; j < cols; j++) {
54             matrix[i][j] = (double)(rand() % 10); // Números aleatórios de 0 a 9
55         }
56     }
57     for (int i = 0; i < cols; i++) {
58         vector[i] = (double)(rand() % 10);
59     }
60 }
61
62 // Executa a multiplicação de matriz por vetor (y = A * x).
63
64 void multiply_matrix_vector(int rows, int cols, double **A, double *x, double *y) {
65     for (int i = 0; i < rows; i++) {
66         y[i] = 0.0; // Garante que o valor inicial seja zero
67         for (int j = 0; j < cols; j++) {
68             y[i] += A[i][j] * x[j];
69         }
70     }
71 }
72
73 void multiply_matrix_vector_cols_outer(int rows, int cols, double **A, double *x, double
*y) {
74     for (int i = 0; i < rows; i++) {
75         y[i] = 0.0;
76     }
77
78     // 2. O laço externo agora itera sobre as COLUNAS (j) da matriz A.
79     for (int j = 0; j < cols; j++) {
80         // O laço interno itera sobre as LINHAS (i).
81         for (int i = 0; i < rows; i++) {
82             y[i] += A[i][j] * x[j];
83         }
84     }
85 }
86
87 int main() {
88     int n_inicial = 32;
89     int n_final = 16384;
90     int n_passo = 2;
91
92     double tempo_mediana_linhas, tempo_mediana_colunas;
93     double fator_lentidao;
94
95     // Cabeçalho do CSV
96     printf("Tamanho_N,Tempo_Mediana_Linhas_s,Tempo_Mediana_Colunas_s,Fator_Lentidao\n");
97
98     for (int N = n_inicial; N <= n_final; N *= n_passo) {
99
100         int M = N;
101
102         double **A = create_matrix(M, N);
103         double *x = (double *)malloc(N * sizeof(double));
104         double *y = (double *)malloc(M * sizeof(double));
```

```
105
106     if (A == NULL || x == NULL || y == NULL) {
107         fprintf(stderr, "Falha ao alocar para N=%d\n", N);
108         continue;
109     }
110
111     fill_random_data(M, N, A, x);
112
113     int repeticoes = 1001;
114     if (N >= 512) repeticoes = 51;
115     if (N >= 1024) repeticoes = 11;
116     if (N >= 2048) repeticoes = 5;
117
118     //Alocar arrays para armazenar os tempos de cada repetição ---
119     double *tempos_linhas = (double *)malloc(repeticoes * sizeof(double));
120     double *tempos_colunas = (double *)malloc(repeticoes * sizeof(double));
121     if (tempos_linhas == NULL || tempos_colunas == NULL) {
122         fprintf(stderr, "Falha ao alocar arrays de tempo para N=%d\n", N);
123         free_matrix(M, A); free(x); free(y);
124         continue;
125     }
126
127     // Teste 1: Acesso por Linhas (Coletando tempos individuais)
128     for(int r = 0; r < repeticoes; r++) {
129         double start_time = omp_get_wtime();
130         multiply_matrix_vector(M, N, A, x, y);
131         double end_time = omp_get_wtime();
132         tempos_linhas[r] = end_time - start_time;
133     }
134
135     // Teste 2: Acesso por Colunas (Coletando tempos individuais)
136     for(int r = 0; r < repeticoes; r++) {
137         double start_time = omp_get_wtime();
138         multiply_matrix_vector_cols_outer(M, N, A, x, y);
139         double end_time = omp_get_wtime();
140         tempos_colunas[r] = end_time - start_time;
141     }
142
143     // Ordena os tempos do acesso por linhas
144     qsort(tempos_linhas, repeticoes, sizeof(double), compare_doubles);
145
146     // Ordena os tempos do acesso por colunas
147     qsort(tempos_colunas, repeticoes, sizeof(double), compare_doubles);
148
149     // Calcula a mediana (pegando o elemento do meio do array ordenado)
150     tempo_mediana_linhas = tempos_linhas[repeticoes / 2];
151     tempo_mediana_colunas = tempos_colunas[repeticoes / 2];
152
153     // Calcula o fator de lentidão
154     if (tempo_mediana_linhas > 0) {
155         fator_lentidao = tempo_mediana_colunas / tempo_mediana_linhas;
156     } else {
157         fator_lentidao = 1.0;
158     }
```

```
159
160     // Imprime a linha de dados CSV
161     printf("%d,%.12f,%.12f,%.3f\n", N, tempo_mediana_linhas, tempo_mediana_colunas,
fator_lentidao);
162
163     // --- MODIFICAÇÃO 3: Liberar a memória dos arrays de tempo ---
164     free_matrix(M, A);
165     free(x);
166     free(y);
167     free(tempo_linhas);
168     free(tempo_colunas);
169 }
170
171 return 0;
172 }
```