

# Relatório de Desempenho: Multiplicação Matriz-Vetor e Memória Cache

Aluno: Cristovão Lacerda Cronje

## 1. Introdução

Este relatório avalia o impacto dos padrões de acesso à memória no desempenho da multiplicação matriz-vetor (MxV), considerando a arquitetura de cache do processador Intel i5-3210M (Ivy Bridge) com as seguintes características:

- Cache L1: 64 KB (dados)
- Cache L2: 256 KB
- Cache L3: 3 MB (compartilhada)

O **objetivo** é comparar duas implementações: Acesso por linhas e Acesso por colunas.

## 2. Metodologia

### 2.1 Código

```
001_mulmat.c 1,0 001_mulmat.c 1,0 X
001_mulmat.c > 90 | methodo_tempo_por_op
1 #include <stdio.h>
2 #include <time.h>
3 #include <stdint.h>
4
5 #define LINHAS 2000
6 #define COLUNAS 2000
7 #define TESTES 1000
8
9 // Variável global para armazenar o tempo por operação do método mais eficiente
10 double melhor_tempo_por_op = 0.0;
11
12 // Versão eficiente: acesso por linhas
13 void multiplicarPorLinhas(int matriz[LINHAS][COLUNAS], int vetor[COLUNAS], int resultado[LINHAS]) {
14     for (int i = 0; i < LINHAS; i++) {
15         resultado[i] = 0;
16         for (int j = 0; j < COLUNAS; j++) {
17             resultado[i] += matriz[i][j] * vetor[j];
18         }
19     }
20 }
21 // Versão ineficiente: acesso por colunas
22 void multiplicarPorColunas(int matriz[LINHAS][COLUNAS], int vetor[COLUNAS], int resultado[LINHAS]) {
23     for (int j = 0; j < COLUNAS; j++) {
24         for (int i = 0; i < LINHAS; i++) {
25             resultado[i] += matriz[i][j] * vetor[j];
26         }
27     }
28 }
29 // Função para medir o tempo de execução
30 void medirTempo(const char* nome, void (*funcao)(int[LINHAS][COLUNAS], int[COLUNAS], int[LINHAS]),
31                int matriz[LINHAS][COLUNAS], int vetor[COLUNAS], int resultado[LINHAS]) {
32     struct timespec inicio, fim;
33     int64_t tempo_total = 0;
34
35     /*
36      * Medição de tempo usando clock_gettime() com CLOCK_MONOTONIC:
37      * - CLOCK_MONOTONIC: Mede tempo absoluto, não é afetado por ajustes do sistema
38      * - tv_sec: segundos
39      * - tv_nsec: nanosegundos (0-999999999)
40      */
41     for (int t = 0; t < TESTES; t++) {
42         // Zerar o vetor resultado antes de cada teste
43         for (int i = 0; i < LINHAS; i++) resultado[i] = 0;
44
45         clock_gettime(CLOCK_MONOTONIC, &inicio); // o identificador "CLOCK_MONOTONIC" não está definido
46         funcao(matriz, vetor, resultado);
47         clock_gettime(CLOCK_MONOTONIC, &fim);
48
49         // Calcula o tempo decorrido em nanosegundos
50         tempo_total += (fim.tv_sec - inicio.tv_sec) * 1000000000LL + (fim.tv_nsec - inicio.tv_nsec);
51     }
52
53     // Calcula médias
54     double tempo_medio = (double)tempo_total / TESTES;
55     double tempo_por_op = tempo_medio / (LINHAS * COLUNAS);
56
57     // Atualiza o melhor tempo se for o primeiro método ou se for mais rápido
58     if (melhor_tempo_por_op == 0.0 || tempo_por_op < melhor_tempo_por_op) {
59         melhor_tempo_por_op = tempo_por_op;
60     }
61
62     // Calcula eficiência relativa ao melhor método
63     double eficiencia = (melhor_tempo_por_op / tempo_por_op) * 100;
64
65     printf("| %-16s | %12.2f ns | %-8.2f ns/op | %8.2f%% |\n",
66           nome, tempo_medio, tempo_por_op, eficiencia);
67 }
68
69 void imprimirCabecalho() {
70     printf("\n");
71     printf("Método | Tempo Total | Tempo por Op | Eficiência |\n");
72     printf("-----|-----|-----|-----|\n");
73 }
74
75 void imprimirRodape() {
76     printf("\n");
77 }
78
79 void imprimirParametros() {
80     printf("\n");
81     printf("PARAMETROS DO TESTE\n");
82     printf("-----\n");
83     printf("• Dimensões da matriz: %d x %d\n", LINHAS, COLUNAS);
84     printf("• Número de testes: %d\n", TESTES);
85     printf("• Tipo de dados: int (%zu bytes)\n", sizeof(int));
86     printf("• Tamanho total da matriz: %.2f MB\n", (double)(LINHAS * COLUNAS * sizeof(int)) / (1024 * 1024));
87     printf("• Tamanho do vetor: %.2f KB\n", (double)(COLUNAS * sizeof(int)) / 1024);
88     printf("-----\n");
89 }
90
91 int main() {
92     // ...
93 }
```

```
93
94 int main() {
95     // Alocar matriz e vetor grandes na heap para evitar stack overflow
96     static int matriz[LINHAS][COLUNAS];
97     static int vetor[COLUNAS];
98     static int resultado[LINHAS];
99
100    // Inicializar matriz e vetor com valores simples
101    for (int i = 0; i < LINHAS; i++) {
102        for (int j = 0; j < COLUNAS; j++) {
103            matriz[i][j] = i + j;
104        }
105        vetor[i] = i % 10;
106    }
107
108    imprimirParametros();
109
110    printf("(Obs: 194)");
111    printf("\n");
112
113    imprimirCabecalho();
114    medirTempo("Por Linhas", multiplicarPorLinhas, matriz, vetor, resultado);
115    medirTempo("Por Colunas", multiplicarPorColunas, matriz, vetor, resultado);
116    imprimirRodape();
117
118    printf("\n LEGENDA:\n");
119    printf("■ Tempo Total: Tempo médio para multiplicação completa\n");
120    printf("■ Tempo por Op: Tempo médio por operação (matriz[i][j]*vetor[j])\n");
121    printf("■ Eficiência: Percentual em relação ao método mais eficiente\n");
122    printf("\n OBSERVAÇÕES:\n");
123    printf("■ Tempo medido com clock_gettime(CLOCK_MONOTONIC)\n");
124    printf("■ 1000 testes realizados para cada método\n", TESTES);
125    printf("■ Eficiência calculada como: (melhor_tempo / tempo_atual) * 100\n");
126
127    return 0;
128 }
```

2.2 Testes: Realizados em matrizes de diferentes tamanhos para identificar em qual ponto o acesso por colunas se torna significativamente mais lento:

Tamanho (N×N)	Tamanho em Memória	Nível de Cache Afetado
100×100	0.04 MB	L1 (64 KB)
300×300	0.34 MB	L2 (256 KB)
600×600	1.37 MB	L3 (3 MB)
800×800	2.44 MB	Limite da L3
2000×2000	15.26 MB	RAM (fora da cache)

- Número de testes por configuração: 1000 execuções para garantir precisão estatística.
- Método de medição: clock\_gettime(CLOCK\_MONOTONIC) para evitar interferência do SO.

2.3 Fatores que Influenciam a Precisão: Interferência do SO( Outros processos podem consumir cache e CPU), Gerenciamento de Cache( O SO pode realocar memória cache para tarefas em segundo plano) e Turbo Boost( Variações na frequência do processador).

### 3. Resultados Obtidos:

PARÂMETROS DO TESTE				
■ Dimensões da matriz: 100 x 100 ■ Número de testes: 1000 ■ Tipo de dados: int (4 bytes) ■ Tamanho total da matriz: 0.04 MB ■ Tamanho do vetor: 0.39 KB				
RESULTADOS DOS TESTES				
Método	Tempo Total	Tempo por Op	Eficiência	
Por Linhas	52284.06 ns	5.22 ns/op	100.00%	
Por Colunas	51830.94 ns	5.10 ns/op	100.00%	
LEGENDA: ■ Tempo Total: Tempo médio para multiplicação completa ■ Tempo por Op: Tempo médio por operação (matriz[i][j]*vetor[j]) ■ Eficiência: Percentual em relação ao método mais eficiente				
OBSERVAÇÕES: - Tempo medido com clock_gettime(CLOCK_MONOTONIC) - 1000 testes realizados para cada método - Eficiência calculada como: (melhor_tempo / tempo_atual) * 100				

PARÂMETROS DO TESTE				
■ Dimensões da matriz: 300 x 300 ■ Número de testes: 1000 ■ Tipo de dados: int (4 bytes) ■ Tamanho total da matriz: 0.34 MB ■ Tamanho do vetor: 1.17 KB				
RESULTADOS DOS TESTES				
Método	Tempo Total	Tempo por Op	Eficiência	
Por Linhas	309932.37 ns	3.44 ns/op	100.00%	
Por Colunas	352534.15 ns	3.92 ns/op	87.92%	
LEGENDA: ■ Tempo Total: Tempo médio para multiplicação completa ■ Tempo por Op: Tempo médio por operação (matriz[i][j]*vetor[j]) ■ Eficiência: Percentual em relação ao método mais eficiente				
OBSERVAÇÕES: - Tempo medido com clock_gettime(CLOCK_MONOTONIC) - 1000 testes realizados para cada método - Eficiência calculada como: (melhor_tempo / tempo_atual) * 100				

PARÂMETROS DO TESTE				
■ Dimensões da matriz: 600 x 600 ■ Número de testes: 1000 ■ Tipo de dados: int (4 bytes) ■ Tamanho total da matriz: 1.37 MB ■ Tamanho do vetor: 2.34 KB				
RESULTADOS DOS TESTES				
Método	Tempo Total	Tempo por Op	Eficiência	
Por Linhas	1215856.47 ns	3.38 ns/op	100.00%	
Por Colunas	1652170.20 ns	4.59 ns/op	73.59%	
LEGENDA: ■ Tempo Total: Tempo médio para multiplicação completa ■ Tempo por Op: Tempo médio por operação (matriz[i][j]*vetor[j]) ■ Eficiência: Percentual em relação ao método mais eficiente				
OBSERVAÇÕES: - Tempo medido com clock_gettime(CLOCK_MONOTONIC) - 1000 testes realizados para cada método - Eficiência calculada como: (melhor_tempo / tempo_atual) * 100				

PARÂMETROS DO TESTE			
<ul style="list-style-type: none"> <li>Dimensões da matriz: 800 x 800</li> <li>Número de testes: 1000</li> <li>Tipo de dados: int (4 bytes)</li> <li>Tamanho total da matriz: 2.44 MB</li> <li>Tamanho do vetor: 3.12 KB</li> </ul>			
RESULTADOS DOS TESTES			
Método	Tempo Total	Tempo por Op	Eficiência
Por Linhas	2432091.23 ns	3.80 ns/op	100.00%
Por Colunas	4456203.61 ns	6.96 ns/op	54.58%
<b>LEGENDA:</b> <ul style="list-style-type: none"> <li>Tempo Total: Tempo médio para multiplicação completa</li> <li>Tempo por Op: Tempo médio por operação (matriz[i][j]*vetor[j])</li> <li>Eficiência: Percentual em relação ao método mais eficiente</li> </ul>			
<b>OBSERVAÇÕES:</b> <ul style="list-style-type: none"> <li>Tempo medido com clock_gettime(CLOCK_MONOTONIC)</li> <li>1000 testes realizados para cada método</li> <li>Eficiência calculada como: (melhor tempo / tempo atual) * 100</li> </ul>			

PARÂMETROS DO TESTE			
<ul style="list-style-type: none"> <li>Dimensões da matriz: 2000 x 2000</li> <li>Número de testes: 1000</li> <li>Tipo de dados: int (4 bytes)</li> <li>Tamanho total da matriz: 15.26 MB</li> <li>Tamanho do vetor: 7.81 KB</li> </ul>			
RESULTADOS DOS TESTES			
Método	Tempo Total	Tempo por Op	Eficiência
Por Linhas	13414467.09 ns	3.35 ns/op	100.00%
Por Colunas	36194440.82 ns	9.05 ns/op	37.06%
<b>LEGENDA:</b> <ul style="list-style-type: none"> <li>Tempo Total: Tempo médio para multiplicação completa</li> <li>Tempo por Op: Tempo médio por operação (matriz[i][j]*vetor[j])</li> <li>Eficiência: Percentual em relação ao método mais eficiente</li> </ul>			
<b>OBSERVAÇÕES:</b> <ul style="list-style-type: none"> <li>Tempo medido com clock_gettime(CLOCK_MONOTONIC)</li> <li>1000 testes realizados para cada método</li> <li>Eficiência calculada como: (melhor tempo / tempo atual) * 100</li> </ul>			

## 3.2 Análise dos Resultados e Conclusões

### Acesso por Linhas (Row-Major)

#### Padrão ideal para cache:

- Cache hit > 90% quando a matriz cabe na L3.
- Tempos consistentes, mesmo em matrizes grandes.
- Localidade espacial ótima: Acesso sequencial aproveita as linhas de cache de 64 bytes.

### Acesso por Colunas (Column-Major)

#### Problemas de cache miss:

- Cada acesso a uma nova coluna causa um cache miss (linhas de cache = 64 bytes), carregando dados adjacentes quando um elemento é acessado.
- Degradação extrema em matrizes maiores que a L3(3 MB): Suporta até ~886×886 ints (784.996 elementos).
- Matrizes maiores, os dados são expulsos da cache antes da reutilização e o acesso por colunas torna-se catastrófico, com quase 100% de cache misses.

### Acesso por Linhas (Row-Major)

#### Padrão ideal para cache:

- Cache hit > 90% quando a matriz cabe na L3 (3 MB), pois o acesso sequencial por linhas aproveita a localidade espacial.
- Tempos consistentes, mesmo em matrizes grandes, porque os dados são acessados em ordem contígua na memória, minimizando cache misses.
- Linhas de cache (64 bytes) são totalmente utilizadas antes de serem substituídas, reduzindo a necessidade de buscar dados na RAM.

### Acesso por Colunas (Column-Major)

#### Problemas de cache miss:

- Cada acesso a uma nova coluna causa um cache miss porque:
  - A matriz é armazenada em row-major (linhas contíguas), mas o acesso por colunas salta para posições distantes na memória.
  - A CPU carrega blocos de 64 bytes adjacentes na cache, mas o padrão de acesso por colunas ignora essa otimização.
- Degradação extrema em matrizes maiores que a L3 (886×886):
  - Cache L3 (3 MB) comporta até ~886×886 ints (784.996 elementos).
  - Para matrizes maiores (ex.: 2000×2000 = 15.26 MB):
    - Os dados não cabem na cache e são expulsos antes da reutilização.
    - O acesso por colunas força a CPU a buscar diretamente na RAM.
    - Quase 100% de cache misses, tornando a operação catastroficamente lenta.
- Por Que a Diferença é Tão Grande?

**I.** Hierarquia de Cache(L1 (64 KB): ~4 ciclos (1.6 ns) enquanto a RAM: ~100 ns (60× mais lento que L1)

## **II.** Padrão de Acesso:

- Linhas: 1 cache miss a cada 16 acessos (64 bytes = 16 ints).
- Colunas: 1 cache miss por acesso (cada elemento está a 8 KB de distância).

## **Conclusão**

- Acesso por linhas é ideal para operações matriciais em C/C++ devido ao armazenamento row-major.
- Acesso por colunas deve ser evitado, especialmente em matrizes maiores que a L3, onde a penalidade de desempenho é extrema.
- **Solução possível:** Técnicas como blocking/tiling podem mitigar cache misses em acessos não sequenciais. Em vez de percorrer toda a matriz de uma vez, o algoritmo divide-a em partes menores (ex.: 64x64), processa cada bloco sequencialmente (linha por linha) e repete até completar a operação. Isso reduz acessos lentos à memória RAM, pois os dados ficam temporariamente na cache, acelerando operações não sequenciais, como acesso por colunas. O tamanho ideal do bloco varia conforme a arquitetura do processador.