Memória Cache, Localidade Espacial e Temporal, e Row Major vs Column Major em C

Memória Cache: A Memória Rápida da CPU

A memória cache é uma memória de pequena capacidade, extremamente rápida e localizada fisicamente próxima ao processador, que armazena cópias de dados e instruções frequentemente acessados da memória principal (RAM). Sua existência é uma solução para o descompasso de velocidade entre a CPU, que é muito rápida, e a RAM, que é comparativamente lenta. Esse desequilíbrio é conhecido como **"Gap de Velocidade"** ou **"Von Neumann Bottleneck"**.

A latência de acesso à memória cache L1 é tipicamente de 1-4 ciclos de clock, enquanto o acesso à RAM pode requerer 200-400 ciclos. Esta diferença substancial justifica a implementação de múltiplos níveis de cache para otimizar o desempenho do sistema através da redução do número médio de acessos à memória principal.

A hierarquia de memória é organizada em níveis (L1, L2, L3), com a L1 sendo a menor e mais rápida, localizada dentro do próprio núcleo do processador, e a L3 sendo maior e um pouco mais lenta, porém compartilhada entre vários núcleos. O princípio fundamental por trás da eficácia da cache é a **localidade**, que se divide em dois tipos: espacial e temporal.

Localidade Espacial e Temporal

Localidade Temporal

A localidade temporal é um princípio fundamental que estabelece que **dados acessados recentemente têm alta probabilidade de serem acessados novamente em um intervalo de tempo próximo**. Este comportamento é explorado pelos algoritmos de substituição de cache, como LRU (Least Recently Used).

Este princípio se manifesta principalmente em estruturas de controle como loops, onde variáveis de iteração, condições de parada e dados processados repetidamente são mantidos na cache. A eficácia da localidade temporal é quantificada pela taxa de cache hits em acessos subsequentes ao mesmo endereço de memória.

Algoritmos que processam os mesmos dados múltiplas vezes (como operações em matrizes densas) beneficiam-se significativamente da localidade temporal, especialmente quando o working set de dados cabe nos níveis superiores da hierarquia de cache.

> **Cache Hit vs Cache Miss:** A cache age com base nesse princípio: quando um dado é buscado da memória principal, ele é copiado para a cache. Se esse mesmo dado for solicitado novamente e ainda estiver lá (um *acerto de cache*, ou cache hit), a CPU o obtém imediatamente. Se não estiver (uma *falha de cache*, ou cache miss), ocorre a custosa viagem à RAM.

Localidade Espacial

A localidade espacial é um princípio que explora a proximidade física dos dados na memória. Este princípio estabelece que **após acessar um endereço de memória, existe alta probabilidade de acessos subsequentes a endereços adjacentes**.

A implementação prática deste princípio baseia-se na organização da memória e cache em **linhas de cache** (cache lines), blocos contíguos de dados tipicamente de 64 bytes em arquiteturas x86-64. Quando ocorre um cache miss, toda a linha de cache contendo o endereço solicitado é transferida da memória principal para a cache, não apenas o byte individual.

Esta estratégia é altamente eficaz para padrões de acesso sequencial, como iteração sobre arrays, onde elementos adjacentes na memória são acessados consecutivamente. O resultado é uma alta taxa de cache hits para acessos subsequentes dentro da mesma linha de cache.

Row Major vs Column Major em C

Em C, arrays multidimensionais são armazenados em **row major order**, ou seja, elementos de uma mesma linha estão em posições consecutivas na memória. Quando acessamos os elementos linha por linha, aproveitamos a localidade espacial, pois os dados já estão próximos na memória, tornando o acesso mais rápido devido ao cache.

Já no **column major order** (usado em outras linguagens como Fortran e MATLAB), os elementos de uma mesma coluna estão próximos. Em C, acessar coluna por coluna resulta em saltos maiores na memória, reduzindo o aproveitamento da cache e tornando o acesso mais lento.

Layout de Memória em Row-Major

Em sistemas row-major, uma matriz bidimensional A[m][n] é mapeada linearmente na memória onde o elemento A[i][j] está localizado no endereço: **base_address + (i × n + j) × sizeof(element)**

Esta organização implica que elementos consecutivos de uma linha (j, j+1, j+2...) estão em endereços de memória adjacentes, maximizando a eficiência da cache para acessos sequenciais por linha. Conversamente, elementos de uma coluna (A[i][j], A[i+1][j], A[i+2][j]...) estão separados por um stride de n×sizeof(element) bytes, potencialmente causando cache miss em cada acesso.

Exemplo em C (Row Major):

```
int matriz[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};

Memória: [1][2][3][4][5][6][7][8][9][10][11][12]

Acesso eficiente (por linhas):
for(i=0; i<3; i++)
    for(j=0; j<4; j++)
        printf("%d ", matriz[i][j]); // Acesso sequencial

Acesso ineficiente (por colunas):
for(j=0; j<4; j++)
    for(i=0; i<3; i++)
        printf("%d ", matriz[i][j]); // Saltos na memória
```

Métricas de Performance

- **Row-major access:** Taxa de cache miss ≈ 1/(elementos_por_cache_line)
- **Column-major access:** Taxa de cache miss ≈ 100% para matrizes grandes
- **Bandwidth utilization:** Row-major utiliza ~100% da bandwidth de memória, column-major ~1-10%
- **Latência efetiva:** Diferença pode alcançar 2-3 ordens de magnitude

Resultados Experimentais

Os resultados a seguir foram obtidos através da execução do programa de teste de multiplicação matriz-vetor, comparando o desempenho entre acesso por linhas (row-major) e acesso por colunas (column-major) em diferentes tamanhos de matriz.

Resultados dos Testes de Performance

```
==================================================
          TESTE COM MATRIZ 200x200
==================================================
Acesso por linhas   | 0.000194 s
Acesso por colunas  | 0.000194 s
Performance: colunas 1.00x (equivalente)

==================================================
          TESTE COM MATRIZ 400x400
==================================================
Acesso por linhas   | 0.000581 s
Acesso por colunas  | 0.000811 s
Performance: linhas 1.39x mais rápido

==================================================
          TESTE COM MATRIZ 600x600
==================================================
Acesso por linhas   | 0.001395 s
Acesso por colunas  | 0.002440 s
Performance: linhas 1.75x mais rápido

==================================================
          TESTE COM MATRIZ 800x800
==================================================
Acesso por linhas   | 0.003347 s
Acesso por colunas  | 0.004035 s
Performance: linhas 1.21x mais rápido

==================================================
          TESTE COM MATRIZ 1000x1000
==================================================
Acesso por linhas   | 0.004375 s
Acesso por colunas  | 0.007084 s
Performance: linhas 1.62x mais rápido

==================================================
          TESTE COM MATRIZ 1500x1500
```

```
======================================================
Acesso por linhas   | 0.009737 s
Acesso por colunas  | 0.017701 s
Performance: linhas 1.82x mais rápido

======================================================
           TESTE COM MATRIZ 2000x2000
======================================================
Acesso por linhas   | 0.014221 s
Acesso por colunas  | 0.035507 s
Performance: linhas 2.50x mais rápido

======================================================
           TESTE COM MATRIZ 2500x2500
======================================================
Acesso por linhas   | 0.022652 s
Acesso por colunas  | 0.060502 s
Performance: linhas 2.67x mais rápido

======================================================
           TESTE COM MATRIZ 3000x3000
======================================================
Acesso por linhas   | 0.031533 s
Acesso por colunas  | 0.088898 s
Performance: linhas 2.82x mais rápido

======================================================
           TESTE COM MATRIZ 3500x3500
======================================================
Acesso por linhas   | 0.043085 s
Acesso por colunas  | 0.128384 s
Performance: linhas 2.98x mais rápido

======================================================
           TESTE COM MATRIZ 4000x4000
======================================================
Acesso por linhas   | 0.055976 s
Acesso por colunas  | 0.182275 s
Performance: linhas 3.26x mais rápido

======================================================
           TESTE COM MATRIZ 5000x5000
======================================================
Acesso por linhas   | 0.087609 s
Acesso por colunas  | 0.282290 s
Performance: linhas 3.22x mais rápido
```

Análise da Divergência Crítica de Performance

Identificação do Ponto de Divergência Significativa

A análise detalhada dos resultados experimentais revela que **2000×2000 (32 MB de working set)** representa o ponto crítico onde os tempos de execução passam a divergir significativamente entre os dois padrões de acesso à memória. Este ponto marca uma transição fundamental no comportamento de performance.

Observa-se um **salto abrupto de 1.82x para 2.50x** entre as matrizes 1500×1500 e 2000×2000, indicando uma mudança qualitativa no regime de operação do sistema de memória.

Análise por Fases de Comportamento

Fase 1: Matrizes Pequenas e Médias (≤ 1500×1500, ≤ 18 MB)

- **Working Set:** 320 KB - 18 MB (cabe parcial/totalmente na cache L3)
- **Speedup:** 1.00x - 1.82x (inconsistente e moderado)
- **Característica:** Competição por recursos de cache entre padrões
- **Comportamento:** Diferenças mascaradas por cache hits ocasionais em ambos os padrões

Fase 2: Matrizes Grandes (≥ 2000×2000, ≥ 32 MB)

- **Working Set:** ≥ 32 MB (excede completamente cache L3 típica)
- **Speedup:** 2.50x - 3.26x (alto e consistente)
- **Característica:** Exposição completa da diferença fundamental entre padrões
- **Comportamento:** Column-major causa cache thrashing, row-major mantém eficiência

Análise Técnica da Causa da Divergência

Cálculo do Working Set Crítico:
Para matriz N×N com elementos double (8 bytes):
• 1500×1500: $1500^2 \times 8 = 18$ MB (ainda cabe parcialmente na L3)
• 2000×2000: $2000^2 \times 8 = 32$ MB ← **Ponto crítico**
• 5000×5000: $5000^2 \times 8 = 200$ MB (muito superior à L3)

Cache L3 Típica: 16-32 MB
Cache Line: 64 bytes (8 elementos double)

Padrões de Stride:
• Row-major: 8 bytes (acesso sequencial)
• Column-major (N=2000): 2000 × 8 = 16,000 bytes
→ 250x maior que cache line (64 bytes)

Miss Rate Estimado para Working Set > L3:
• Row-major: ~12.5% (1 miss a cada 8 acessos)
• Column-major: ~100% (miss em praticamente cada acesso)

Diferença de Latência:
• Cache L1/L2: ~4 ciclos de clock
• RAM: ~300 ciclos de clock
• Ratio teórico: 300/4 = 75x
• Ratio observado: ~3.2x (limitado por prefetching e paralelismo)

Explicação do Fenômeno de Divergência

A divergência significativa a partir de **2000×2000** ocorre devido a múltiplos fatores convergentes:

- **Excesso do Threshold de Cache L3:** Working set de 32 MB excede capacidade típica da cache L3 (16-32 MB)
- **Cache Thrashing Completo:** Column-major passa a causar cache miss em ~100% dos acessos
- **Manutenção da Eficiência Row-Major:** Acesso sequencial continua aproveitando cache lines completas
- **Saturação de Bandwidth:** Column-major satura bus de memória com transferências ineficientes

- **Dominância da Latência:** Diferença entre cache (~4 ciclos) e RAM (~300 ciclos) se torna fator determinante

Visualização Gráfica dos Resultados

Gráfico de Análise de Performance

O gráfico abaixo ilustra visualmente a divergência de performance entre os padrões de acesso row-major e column-major, destacando o ponto crítico de 2000×2000 onde ocorre a transição para o regime de alta divergência.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <math.h>

#ifdef _WIN32
#include <windows.h>
#else
#include <sys/time.h>
#endif

// Macro para indexacao de matriz (mais eficiente)
#define MATRIX_INDEX(matrix, i, j, cols) ((matrix)[(i) * (cols) + (j)])

// Numero de iteracoes para cada teste (primeira descartada)
#define NUM_ITERATIONS 4

// Funcao inline para multiplicacao matriz-vetor com acesso por linhas
static inline void matrix_vector_multiply_rows(double *matrix, double *vector, double *result, int rows,
int cols) {
    for (int i = 0; i < rows; i++) { // Percorre linhas da matriz (row-major order)
        result[i] = 0.0; // Inicializa elemento do resultado
        for (int j = 0; j < cols; j++) { // Percorre colunas da linha atual
            result[i] += MATRIX_INDEX(matrix, i, j, cols) * vector[j]; // Acumula produto escalar
        }
    }
}

// Funcao inline para multiplicacao matriz-vetor com acesso por colunas
static inline void matrix_vector_multiply_cols(double *matrix, double *vector, double *result, int rows,
int cols) {
    // Inicializar resultado com zeros
    for (int i = 0; i < rows; i++) { // Zera vetor resultado
        result[i] = 0.0;
    }

    for (int j = 0; j < cols; j++) { // Percorre colunas primeiro (column-major access)
        for (int i = 0; i < rows; i++) { // Percorre linhas para cada coluna (saltos de memória)
            result[i] += MATRIX_INDEX(matrix, i, j, cols) * vector[j]; // Acesso com stride = cols
        }
    }
}

// Funcao para alocar matriz dinamicamente como bloco contiguo
double *allocate_matrix(int rows, int cols) {
    return (double *)malloc(rows * cols * sizeof(double)); // Aloca memória contígua para a matriz
}

// Funcao para liberar matriz
void free_matrix(double *matrix) {
    free(matrix); // Libera memória alocada dinamicamente
}

// Funcao para inicializar matriz com valores aleatorios
void initialize_matrix(double *matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) { // Percorre linhas
        for (int j = 0; j < cols; j++) { // Percorre colunas
            matrix[i * cols + j] = (double)rand() / RAND_MAX; // Preenche com valores aleatórios [0,1]
        }
    }
}

// Funcao para inicializar vetor com valores aleatorios
void initialize_vector(double *vector, int size) {
    for (int i = 0; i < size; i++) { // Percorre elementos do vetor
        vector[i] = (double)rand() / RAND_MAX; // Preenche com valores aleatórios [0,1]
    }
}

// Funcao para medir tempo real de execucao (wall time)
double get_wall_time() {
#ifdef _WIN32
    LARGE_INTEGER time, freq; // Estruturas para high-resolution timer no Windows
    QueryPerformanceFrequency(&freq); // Obtém frequência do contador
    QueryPerformanceCounter(&time); // Obtém valor atual do contador
    return (double)time.QuadPart / freq.QuadPart; // Retorna tempo em segundos
#else
    struct timeval time; // Estrutura para tempo no Unix/Linux
    gettimeofday(&time, NULL); // Obtém tempo atual com precisão de microssegundos
    return time.tv_sec + time.tv_usec * 1e-6; // Converte para segundos
#endif
```

```c
}



// Funcao para medir wall time com multiplas iteracoes
double measure_wall_time_multiple(void (*func)(double*, double*, double*, int, int),
                                  double *matrix, double *vector, double *result, int rows, int cols) {
    double times[NUM_ITERATIONS]; // Array para armazenar tempos de cada iteração

    // Executar multiplas iteracoes
    for (int iter = 0; iter < NUM_ITERATIONS; iter++) { // Loop de iterações para obter média
        double start, end; // Variáveis para tempo inicial e final
        start = get_wall_time(); // Marca tempo inicial
        func(matrix, vector, result, rows, cols); // Executa função a ser medida
        end = get_wall_time(); // Marca tempo final
        times[iter] = end - start; // Calcula tempo decorrido
    }

    // Calcular media ignorando a primeira iteracao (aquecimento)
    double sum = 0.0; // Soma dos tempos (exceto primeira iteração)
    for (int i = 1; i < NUM_ITERATIONS; i++) { // Ignora primeira iteração (warm-up)
        sum += times[i]; // Acumula tempos
    }
    return sum / (NUM_ITERATIONS - 1); // Retorna média dos tempos válidos
}

// Funcao para verificar se os resultados sao iguais
int compare_results(double *result1, double *result2, int size) {
    double tolerance = 1e-10; // Tolerância para comparação de ponto flutuante
    for (int i = 0; i < size; i++) { // Compara elemento por elemento
        if (fabs(result1[i] - result2[i]) > tolerance) { // Verifica diferença absoluta
            return 0; // Resultados diferentes
        }
    }
    return 1; // Resultados iguais dentro da tolerância
}

// Funcao para executar teste com um tamanho especifico
void run_test(int size) {
    printf("\n"); // Linha em branco para separação visual
    printf("======================================================\n");
    printf("           TESTE COM MATRIZ %dx%d\n", size, size);
    printf("======================================================");

    // Alocar memoria
    double *matrix = allocate_matrix(size, size); // Matriz quadrada NxN
    double *vector = (double *)malloc(size * sizeof(double)); // Vetor de entrada
    double *result_rows = (double *)malloc(size * sizeof(double)); // Resultado row-major
    double *result_cols = (double *)malloc(size * sizeof(double)); // Resultado column-major

    // Inicializar dados
    srand(42); // Seed fixo para resultados reproduziveis entre execuções
    initialize_matrix(matrix, size, size); // Preenche matriz com valores aleatórios
    initialize_vector(vector, size); // Preenche vetor com valores aleatórios

    // Medir tempo da versao por linhas - Wall Time
    double wall_time_rows = measure_wall_time_multiple(matrix_vector_multiply_rows, matrix, vector,
result_rows, size, size); // Acesso row-major

    // Medir tempo da versao por colunas - Wall Time
    double wall_time_cols = measure_wall_time_multiple(matrix_vector_multiply_cols, matrix, vector,
result_cols, size, size); // Acesso column-major

    // Verificar se os resultados sao iguais
    if (compare_results(result_rows, result_cols, size)) { // Validação da correção dos algoritmos
        printf("\n\n");// printf("[OK] Resultados corretos (ambas as versoes produziram o mesmo
resultado)\n\n");
    } else {
        printf("[ERRO] Resultados diferentes entre as versoes!\n\n"); // Indica erro na implementação
    }

    // Exibir tempos de forma organizada
    printf("TEMPOS DE EXECUCAO:\n");
    printf("------------------------------------------------------\n");
    printf("                    | Wall Time\n");
    printf("------------------------------------------------------\n");
    printf("Acesso por linhas   | %.6f s\n", wall_time_rows);
    printf("Acesso por colunas  | %.6f s\n", wall_time_cols);
    printf("------------------------------------------------------\n");

    // Calcular speedups
    printf("\nANALISE DE PERFORMANCE:\n");
    if (wall_time_cols > 0 && wall_time_rows > 0) { // Evita divisão por zero
        double wall_speedup = wall_time_cols / wall_time_rows; // Calcula fator de aceleração
        // printf("Speedup: %.2fx - ", wall_speedup);
        if (wall_speedup > 1.0) { // Row-major mais rápido (caso esperado)
            printf("linhas %.2fx mais rapido\n", wall_speedup);
```

```
        } else { // Column-major mais rapido (caso inesperado)
            printf("colunas %.2fx mais rapido\n", 1.0/wall_speedup);
        }

    }
    // Liberar memoria
    free_matrix(matrix); // Libera matriz
    free(vector); // Libera vetor de entrada
    free(result_rows); // Libera resultado row-major
    free(result_cols); // Libera resultado column-major
}

int main() {
    printf("=== Comparacao de Performance: Multiplicacao Matriz-Vetor ===\n");
    printf("Testando diferentes padroes de acesso a memoria:\n");


    // Tamanhos de teste - valores expandidos para análise mais ampla de performance
    int sizes[] = {200, 400, 600, 800, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 5000};
    int num_tests = sizeof(sizes) / sizeof(sizes[0]); // Calcula número de testes automaticamente

    // Executar testes
    for (int i = 0; i < num_tests; i++) { // Loop através de todos os tamanhos de teste
        run_test(sizes[i]); // Executa teste para tamanho específico
    }

    return 0; // Indica execução bem-sucedida
}
//gcc -O0 -o tarefa1_O0.exe tarefa1.c -lm
// ./tarefa1_O0.exe
```