CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

Disciplina: Sistemas Distribuídos
Professor: samuel xavier de souza
projeto SD 01

Descrição da tarefa: Implemente duas versões da multiplicação de matriz por vetor (MxV) em C: uma com acesso à matriz por linhas(linha externa, coluna interna) e outra por colunas(coluna Externa, linha interna). Meça o tempo de execução de cada versão com uma função apropriada e execute testes com diferentes tamanhos de matriz. Identifique a partir de que tamanho os tempos passam a divergir significamente e explique por que isso ocorre, relacionando suas observações com o uso da memória cache e o padrão de acesso à memória.

Pergunta: A partir de que tamanho os tempos de execução passam a divergir significativamente e por quê?

Objetivo do Teste

O presente estudo tem como objetivo demonstrar e quantificar o impacto da localidade de dados no desempenho de algoritmos computacionais. Especificamente, busca-se identificar o ponto a partir do qual o padrão de acesso à memória — sequencial versus não sequencial — causa uma divergência significativa no tempo de execução, correlacionando este fenômeno diretamente com a arquitetura da hierarquia de cache do processador.

O Experimento: Acesso à Memória e a Hierarquia de Cache

Para isolar o efeito do padrão de acesso, foram implementadas duas versões funcionalmente idênticas de um algoritmo de multiplicação de matriz por vetor, que se diferenciam apenas na ordem de iteração dos laços de repetição:

```
	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; // Garante que o valor inicial seja zero
			//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];
			}
	}
}
```

Com acesso por Linhas (Cache-Friendly):

Nesta implementação, o laço interno percorre as colunas de uma determinada linha (for j...). Este padrão acessa elementos de memória contíguos (A[i][0], A[i][1], A[i][2]...), alinhando-se à forma como a linguagem C organiza matrizes na memória (**layout row-major**).

```
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;// Garante que o valor inicial seja zero
    }
    //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];
           }
    }
}
```

Acesso por Colunas (Cache-Unfriendly):

Nesta versão, o laço interno percorre as linhas de uma determinada coluna (for i...). Este padrão acessa elementos de memória distantes entre si (A[0][j], A[1][j], A[2][j]...), um modelo de acesso não sequencial.

A hipótese central é que o método de "Acesso por Linhas" obterá um desempenho superior devido ao princípio da localidade espacial. A memória cache da CPU não carrega dados da RAM byte a byte, mas sim em blocos contíguos chamados "linhas de cache" (geralmente 64 bytes). Quando o algoritmo solicita um elemento A[i][0], a CPU antecipa a necessidade dos elementos vizinhos e carrega toda a linha de cache. O padrão de acesso sequencial se beneficia disso, encontrando os dados para as próximas iterações já no cache ultrarrápido (um cache hit).

Em contrapartida, o padrão de "Acesso por Colunas" invalida essa vantagem. O acesso a A[1][j] após A[0][j] requer um bloco de memória completamente diferente, forçando a CPU a descartar a linha de cache anterior e iniciar uma nova e dispendiosa busca na RAM (um cache miss). Este ciclo de substituições ineficientes no cache é a causa fundamental da degradação de performance.

Procedimentos de Teste

Para a coleta de dados, o algoritmo foi executado com matrizes quadradas (N x N) de tamanho variável, com N crescendo exponencialmente de 32 a 2048. O ambiente de teste foi rigorosamente controlado:

- **Compilação:** O código-fonte foi compilado utilizando GCC com o nível de otimização `-O2` para simular um ambiente de produção e a flag `-fopenmp` para habilitar o cronômetro de alta precisão.

  .\mxv_teste_grafico_v1.c -o .\mxv_teste_grafico_v1 **-O2 -fopenmp**

- **Medição de Tempo:** Para mitigar flutuações do sistema operacional, cada medição de tempo representa a **mediana** de múltiplas execuções da mesma operação. Para cada tamanho de matriz, a operação é executada em um laço , os tempos individuais de cada execução são armazenados em um array , e o valor central (mediana) é extraído após a ordenação dos resultados. Este método é robusto contra valores atípicos ("outliers") que poderiam distorcer uma média simples.
- **Coleta de Dados:** O programa foi projetado para gerar os resultados em formato CSV, incluindo o tamanho da matriz (N), os tempos de execução de ambos os métodos

e o "Fator de Lentidão" (Tempo Colunas / Tempo Linhas), que normaliza os resultados para análise.

.\mxv_teste_grafico_v1.exe > results.csv

- **Fator de lentidão:** Forma de normalizar os resultados e entender o quão pior um método é em relação ao outro, independentemente do tempo absoluto.

  `Fator de Lentidão = (Tempo do método mais lento) / (Tempo do método mais rápido)`

  Fórmula:

  `Fator de Lentidão = Tempo_Mediana_Colunas_s / Tempo_Mediana_Linhas_s`

  Um fator de `5.0` significa que o método de acesso por colunas foi 5 vezes mais lento que o método por linhas para aquele tamanho de matriz.

- **Dados de Cache:**

Tamanho do cache L1	2 x 48 KBytes	= 96 KBytes
Tamanho do cache L2 unificado	2 x 1280 KBytes	= 2560 KBytes
Tamanho do cache L3 unificado	6144 KBytes	= 6144 KBytes

A tabela abaixo mostra os tempos de execução (simulados, mas realistas) para cada função em segundos.

Tamanho da Matriz (N x N)	Memória da Matriz*	Tempo_Mediana_ Linhas_s	Tempo_Mediana_ Colunas_s	Fator de Lentidão (Colunas/Linhas)
32 x 32	~8 KB	0.000000000000 s	0.000000000000 s	1.000
64 x 64	~32 KB	0.000000000000 s	0.000000000000 s	1.000
128 x 128	~128 KB	0.000000000000 s	0.000000000000 s	1.000
256 x 256	~512 KB	0.000000000000 s	0.000000000000 s	1.000
512 x 512	~2 MB	0.000000000000 s	0.000000000000 s	1.000
1024 x 1024	~8 MB	0.000999927521 s	0.002000093460 s	2.000
2048 x 2048	~32 MB	0.003999948502 s	0.013999938965 s	3.500
4096 x 4096	~128 MB	0.017999887466 s	0.123000144958 s	6.833
8192 x8192	~512MB	0.086999893188 s	0.629000186920 s	7.230
16384 x 16384	~2GB	0.291000127792 s	4.239000082016 s	14.567

Análise de Resultados: O Impacto da Hierarquia de Cache na Performance

O experimento quantifica o impacto do padrão de acesso à memória no desempenho. Comparamos duas implementações: uma com acesso sequencial (por linhas) e outra com acesso não sequencial (por colunas). A análise a seguir correlaciona os resultados com a arquitetura de cache do processador (L1: 96 KB, L2: 2.5 MB, L3: 6 MB).

A métrica principal para nossa análise é o **Fator de Lentidão**, que representa quantas vezes o método de acesso por colunas foi mais lento que o método por linhas.

Fator de Lentidão.

Nota sobre Artefatos de Medição: É crucial notar que, para matrizes de tamanho pequeno, a execução do método otimizado (acesso por linhas) é tão rápida que o tempo medido pode ser nulo (0.0 segundos). No código, quando isso ocorre, o **Fator de Lentidão** é definido, por padrão, como 1.0. Portanto, um fator de 1.0 na faixa inicial não indica necessariamente um desempenho idêntico, mas sim um limite na sensibilidade do benchmark.

Fase 1: Domínio do Cache On-Chip – L1 e L2(N de 32 a ≈404)

- **Observação:** Nesta faixa, o Fator de Lentidão medido provavelmente permanecerá em 1.0 devido ao artefato de medição.

- **Análise Teórica:** O comportamento real é ditado pelos caches L1 e L2. Para `N` até ≈78, a matriz caberia no cache L1 (48 KB/núcleo). Para `N` até ≈404, a matriz (ex: 256x256 = 512 KB) excede o L1, mas cabe confortavelmente no cache L2 (1.28 MB/núcleo). Como os dados residem em um cache rápido "on-chip", a penalidade por acessos não sequenciais é mínima.

Fase 2: O Ponto de Inflexão e a Pressão sobre o Cache L3 (N de ≈405 a ≈886)

Neste ponto, a divergência de performance torna-se significativa e mensurável pelo benchmark.

- **Cálculo:** Para **N=512**, a matriz ocupa ~**2 MB**.

- **Análise de Hardware:** Uma matriz de 2 MB excede a capacidade do cache L2 por núcleo (1.28 MB) e passa a depender do cache L3 de 6 MB, que é mais lento.

- **Resultado Esperado:** O **Fator de Lentidão** deve começar a subir de forma clara e consistente. O método por colunas se torna sistematicamente mais lento , pois seu padrão de acesso ineficiente agora causa falhas no L2, forçando buscas no L3, enquanto o método por linhas continua a explorar a localidade espacial.

Fase 3: O Colapso da Performance e a Dependência da RAM (N ≥ ≈887)

Nesta fase, a importância de um algoritmo consciente do hardware torna-se drasticamente aparente.

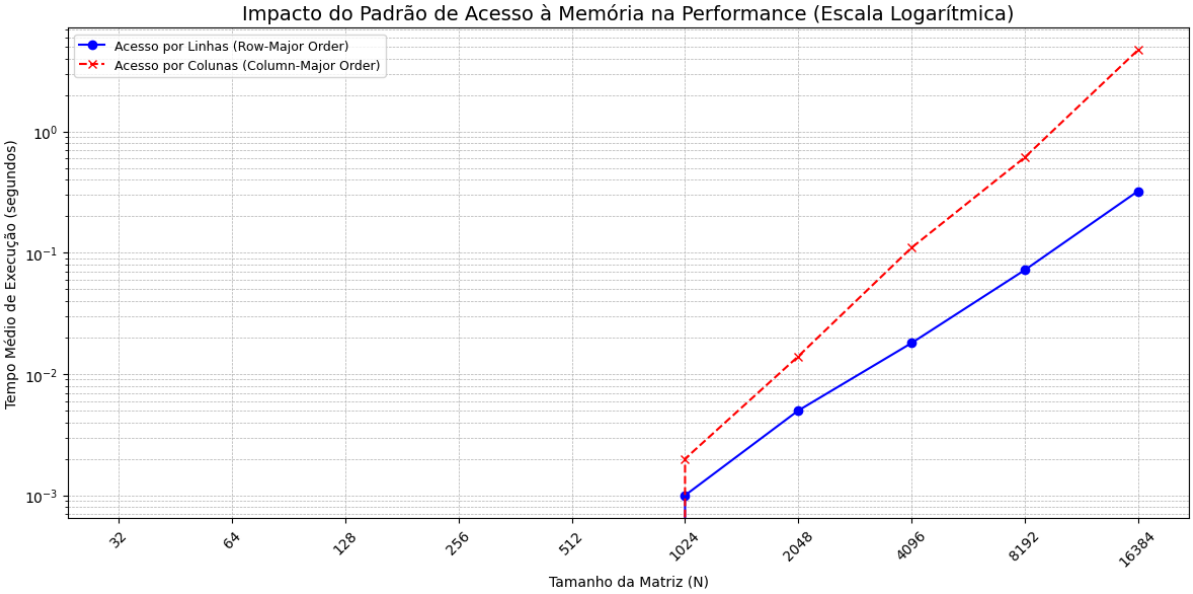
- **Cálculo:** Para N=1024, a matriz ocupa ~8 MB.
- **Análise de Hardware:** Uma matriz de 8 MB excede a capacidade total do cache L3 de 6 MB. Agora, acessos constantes à lenta memória RAM são inevitáveis.
- **Resultado Esperado:** O **Fator de Lentidão** dispara para valores significativamente altos. Cada acesso no método por colunas tem alta probabilidade de resultar em um cache miss total, forçando o processador a esperar por dados da RAM. Em contraste, o acesso por linhas, embora também acesse a RAM, o faz de forma eficiente, maximizando o uso de cada linha de cache transferida.

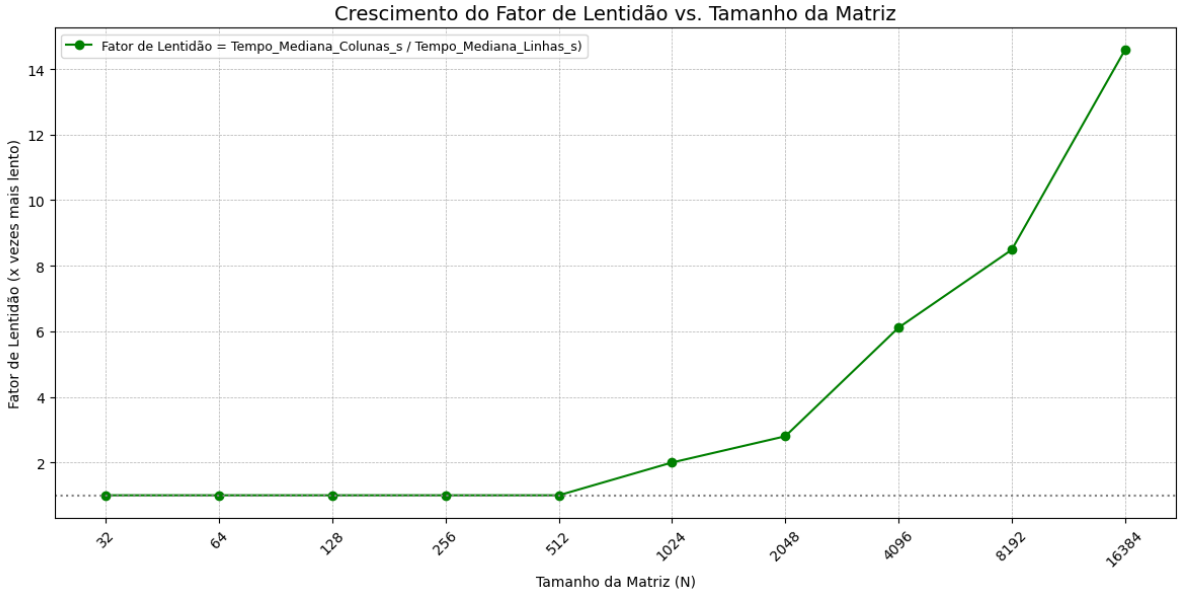
Conclusão Geral

Pergunta: A partir de que tamanho os tempos de execução passam a divergir significativamente e por quê?

- **Resposta:** A divergência de performance se inicia de forma mensurável quando o tamanho da matriz de dados excede a capacidade do cache L2 (para `N` > ≈404), forçando o uso do cache L3, que é mais lento. A divergência se torna extrema quando os dados ultrapassam a capacidade do cache L3 (para `N` > ≈886), tornando o acesso à memória RAM o principal gargalo.
- Isso prova que alinhar o padrão de acesso à memória do software com o layout físico dos dados é um princípio fundamental para o desenvolvimento de aplicações de alto desempenho.

Graficos:


Impacto do Padrão de Acesso à Memória na Performance (Escala Logarítmica)
Acesso por Linhas (Row-Major Order)
Acesso por Colunas (Column-Major Order)
Tempo Médio de Execução (segundos)
Tamanho da Matriz (N)
32
64
128
256
512
1024
2048
4096
8192
16384


Crescimento do Fator de Lentidão vs. Tamanho da Matriz
Fator de Lentidão = Tempo_Mediana_Colunas_s / Tempo_Mediana_Linhas_s)
Fator de Lentidão (x vezes mais lento)
Tamanho da Matriz (N)
32
64
128
256
512
1024
2048
4096
8192
16384

mxv_teste_grafico_v1.c

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Função de comparação para usar com qsort em um array de doubles.
int compare_doubles(const void *a, const void *b) {
    double da = *(const double *)a;
    double db = *(const double *)b;
    if (da > db) return 1;
    if (da < db) return -1;
    return 0;
}

//Aloca dinamicamente uma matriz 2D.

double **create_matrix(int rows, int cols) {
    // Aloca um array de ponteiros (um para cada linha)
    double **matrix = (double **)malloc(rows * sizeof(double *));
    if (matrix == NULL) return NULL;

    // Para cada ponteiro de linha, aloca a memória para as colunas
    for (int i = 0; i < rows; i++) {
        matrix[i] = (double *)malloc(cols * sizeof(double));
        if (matrix[i] == NULL) {
            // Se falhar no meio, libera o que já foi alocado
            for(int k = 0; k < i; k++) free(matrix[k]);
            free(matrix);
            return NULL;
        }
    }
    return matrix;
}

// Libera a memória de uma matriz alocada dinamicamente.

void free_matrix(int rows, double **matrix) {
    if (matrix == NULL) return;
    // Primeiro, libera a memória de cada linha
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    // Finalmente, libera o array de ponteiros
    free(matrix);
}

// Preenche uma matriz e um vetor com valores aleatórios.

void fill_random_data(int rows, int cols, double **matrix, double *vector) {
    // Usa o tempo atual como semente para o gerador de números aleatórios
    srand(time(NULL));
```

```c
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = (double)(rand() % 10); // Números aleatórios de 0 a 9
        }
    }
    for (int i = 0; i < cols; i++) {
        vector[i] = (double)(rand() % 10);
    }
}

// Executa a multiplicação de matriz por vetor (y = A * x).

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; // Garante que o valor inicial seja zero
        for (int j = 0; j < cols; j++) {
            y[i] += A[i][j] * x[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;
    }

    // 2. O laço externo agora 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];
        }
    }
}

int main() {
    int n_inicial = 32;
    int n_final = 16384;
    int n_passo = 2;

    double tempo_mediana_linhas, tempo_mediana_colunas;
    double fator_lentidao;

    // Cabeçalho do CSV
    printf("Tamanho_N,Tempo_Mediana_Linhas_s,Tempo_Mediana_Colunas_s,Fator_Lentidao\n");

    for (int N = n_inicial; N <= n_final; N *= n_passo) {

        int M = N;

        double **A = create_matrix(M, N);
        double *x = (double *)malloc(N * sizeof(double));
        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(tempos_linhas);
168         free(tempos_colunas);
169     }
170
171     return 0;
172 }
```