Na implementação da multiplicação de matriz por vetor (MxV) em C, percebi que a forma como acessamos os elementos da matriz influencia significativamente o tempo de execução. Fiz testes com dois métodos: um que percorre a matriz por linhas e outro que percorre por colunas. Os resultados foram os seguintes:

Tamanho (N)	Tempo (Linhas)	Tempo (Colunas)
100	0.000000s	0.000000s
1000	0.002000s	0.005000s
10000	0.316000s	1.012000s

A partir do momento em que a matriz se tornou 1000x1000, notei que o tempo do método que percorre as colunas começou a aumentar mais do que o método que percorre as linhas. Para uma matriz de tamanho 10000x10000, a diferença se tornou bem significativa.

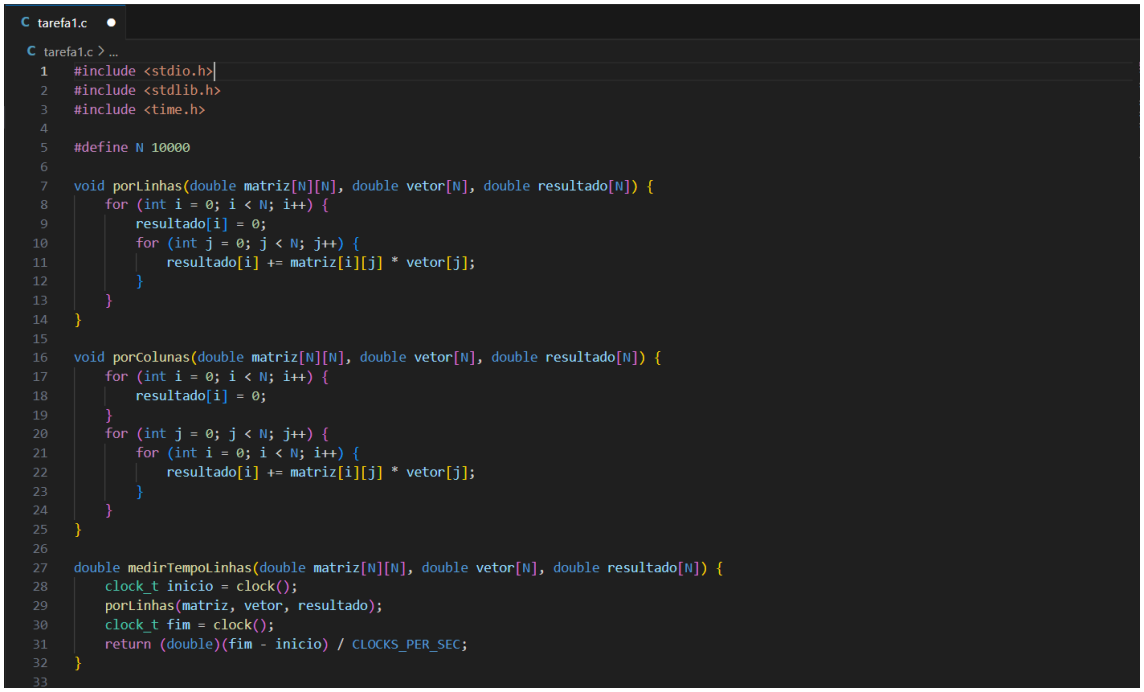
Como a memória cache afeta o desempenho?

Os processadores modernos possuem uma hierarquia de memória que inclui caches L1**,** L2 e L3, que são muito mais rápidas do que a memória RAM. O objetivo da cache é armazenar dados frequentemente acessados para reduzir o tempo de leitura e escrita. No entanto, a cache funciona melhor quando acessamos os dados de forma contínua, ou seja, na ordem em que eles estão armazenados na memória.

Em C, as matrizes são armazenadas por linhas na memória. Isso significa que, quando percorremos a matriz por linhas, os elementos já estão próximos uns dos outros na memória, o que permite um uso eficiente da cache.

Por outro lado, quando percorremos a matriz por colunas, os elementos acessados estão distantes na memória. Isso faz com que a CPU precise buscar os dados na RAM com mais frequência, pois a cache não consegue armazenar informações de tantas linhas ao mesmo tempo. Como o acesso à RAM é mais lento que o acesso à cache, o desempenho da versão por colunas se torna pior.

Código Utilizado para os testes:



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

#define N 10000

void porLinhas(double matriz[N][N], double vetor[N], double resultado[N]) {
    for (int i = 0; i < N; i++) {
        resultado[i] = 0;
        for (int j = 0; j < N; j++) {
            resultado[i] += matriz[i][j] * vetor[j];
        }
    }
}

void porColunas(double matriz[N][N], double vetor[N], double resultado[N]) {
    for (int i = 0; i < N; i++) {
        resultado[i] = 0;
    }
    for (int j = 0; j < N; j++) {
        for (int i = 0; i < N; i++) {
            resultado[i] += matriz[i][j] * vetor[j];
        }
    }
}

double medirTempoLinhas(double matriz[N][N], double vetor[N], double resultado[N]) {
    clock_t inicio = clock();
    porLinhas(matriz, vetor, resultado);
    clock_t fim = clock();
    return (double)(fim - inicio) / CLOCKS_PER_SEC;
}
```

```c
double medirTempoColunas(double matriz[N][N], double vetor[N], double resultado[N]) {
    clock_t inicio = clock();
    porColunas(matriz, vetor, resultado);
    clock_t fim = clock();
    return (double)(fim - inicio) / CLOCKS_PER_SEC;
}

int main() {
    static double matriz[N][N];
    static double vetor[N];
    static double resultado[N];

    for (int i = 0; i < N; i++) {
        vetor[i] = rand() % 10;
        for (int j = 0; j < N; j++) {
            matriz[i][j] = rand() % 10;
        }
    }

    double tempoLinhas = medirTempoLinhas(matriz, vetor, resultado);
    double tempoColunas = medirTempoColunas(matriz, vetor, resultado);

    printf("Tempo (acesso por linhas): %f segundos\n", tempoLinhas);
    printf("Tempo (acesso por colunas): %f segundos\n", tempoColunas);

    return 0;
}
```

Saídas obtidas nos testes:

- N = 100

```
Tempo (acesso por linhas): 0.000000 segundos
Tempo (acesso por colunas): 0.000000 segundos
```

- N = 1000

```
Tempo (acesso por linhas): 0.002000 segundos
Tempo (acesso por colunas): 0.005000 segundos
```

- N = 10000

```
Tempo (acesso por linhas): 0.316000 segundos
Tempo (acesso por colunas): 1.012000 segundos
```