**UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE**
Departamento de Engenharia de Computação e Automação

Programação Paralela
Juliane da Silva Santos

# Relatório 1:
# Memória cache

## 1 Introdução

A memória do computador é organizada em uma hierarquia. No nível mais alto (mais perto do processador), estão os registradores do processador. Em seguida, vêm um ou mais níveis de cache. Quando são usados múltiplos níveis, eles são indicados por L1, L2 e assim por diante. Em seguida, vem a memória principal, que normalmente é uma memória dinâmica de acesso aleatório e dinâmico (DRAM). Todos estes são considerados internos ao sistema de computação. A hierarquia continua com a memória externa. À medida que descemos na hierarquia da memória, encontramos custo/ bit menor, capacidade maior e tempo de acesso mais lento. STALLINGS [1]

Este projeto tem como objetivo analisar o tempo de execução de duas abordagens distintas para a multiplicação de uma matriz por um vetor, com ênfase na influência do padrão de acesso à memória. O código completo pode ser verificado no repositório disposto na plataforma GitHub.

## 2 Metodologia

O projeto teve como ponto de partida a implementação de uma função para gerar números aleatórios e preencher tanto a matriz quanto o vetor.

```
1  // Funcao para inicializar a matriz e o vetor de tamanho N
2  void inicializar() {
3      srand(time(NULL));
4      for (int j = 0; j < N; j++) {
5          for (int i = 0; i < N; i++) {
6              matrix[i][j] = rand() % 100;
7          }
8          vetor[j] = rand() % 100;
9          resultado_linhas[j] = 0;
10         resultado_colunas[j] = 0;
11     }
12 }
```

Para medir o tempo de execução, foi recomendado em sala de aula o uso da função gettimeofday(), que captura o tempo total de execução (wall-clock time), incluindo pausas, esperas e operações de I/O. Essa abordagem difere da função clock(), que mede apenas o tempo de CPU efetivamente utilizado pelo processo, ignorando períodos de inatividade, como espera por I/O ou suspensão do programa. Para facilitar a medição do tempo, foi desenvolvida a função gettimems(), que utiliza gettimeofday() e retorna o tempo atual em milissegundos. Essa função é chamada no inicio do for e no final e para o cálculo fina do tempo de execução realizamos uma subtração do tempo final pelo inicial.

```c
// Funcao para medir o tempo
double get_time_ms() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (tv.tv_sec) * 1000.0 + (tv.tv_usec) / 1000.0;
}
```

Em seguida, para uma melhor compreensão do impacto do padrão de acesso à memória, foram testadas duas abordagens distintas para calcular a multiplicação de uma matriz de tamanho N por um vetor de mesmo tamanho.

## 2.1 Implementação

Dois algoritmos foram testados:

1. Acesso por Linhas

   Percorre a matriz linha por linha. Para cada linha i, multiplica cada elemento matrix[i][j] pelo elemento correspondente do vetor vetor[j] e acumula o resultado em resultadolinhas[i].

```c
// Versao por linhas
double start_linhas = get_time_ms();
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        resultado_linhas[i] += matrix[i][j] * vetor[j];
    }
}
double end_linhas = get_time_ms();
```

2. Acesso por Colunas

   Percorre a matriz coluna por coluna. Para cada coluna j, multiplica cada elemento matrix[i][j] pelo elemento vetor[j] e acumula em resultadocolunas[i].

```c
// Versao por colunas
double start_colunas = get_time_ms();
for (int j = 0; j < N; j++) {
```

```
    for (int i = 0; i < N; i++) {
        resultado_colunas[i] += matrix[i][j] * vetor[j];
    }
}
double end_colunas = get_time_ms();
```

# 3 Resultados e Discussão

Para os resultados fora implementado duas linhas de códigos que apresenta o tempo de execução para ambos os casos.

```
printf("Tempo por LINHAS: %.15f ms\n", end_linhas - start_linhas);
printf("Tempo por COLUNAS: %.15f ms\n", end_colunas - start_colunas);
```

Como forma de observar o desempenho dos dois casos, fora testado 9 tamanhos diferentes, tendo como resultado os seguintes tempos de execução:

| Tamanho de N | Tempo de Execução por Linha | Tempo de Execução por Coluna |
|---|---|---|
| 100 | 0.000000000000000 ms | 0.000000000000000 ms |
| 250 | 0.000000000000000 ms | 0.000000000000000 ms |
| 500 | 1.004882812500000 ms | 1.000976562500000 ms |
| 800 | 1.000976562500000 ms | 2.996093750000000 ms |
| 1000 | 2.610107421875000 ms | 4.313964843750000 ms |
| 1250 | 4.014892578125000 ms | 8.995117187500000 ms |
| 1500 | 5.843017578125000 ms | 15.605957031250000 ms |
| 1800 | 8.505126953125000 ms | 24.042968750000000 ms |
| 2000 | 13.147949218750000 ms | 46.659179687500000 ms |

Com base na Figura 1 é possível ter uma melhor visualização do tempo de execução das duas implementações, é notório observar que o tempo de execução do acesso por linha é menor do que o tempo de execução do acesso por coluna.

Sabendo que a memória é organizada em diferentes níveis (registradores, cache L1/L2/L3, RAM, disco), de modo a balancear velocidade, capacidade e custo – ou seja, formando uma hierarquia de memória –, a forma como se acessa a memória terá um impacto significativo no desempenho do programa.

No contexto da multiplicação matriz-vetor, observamos dois padrões de acesso distintos: por linhas e por colunas. O primeiro segue uma sequência natural de armazenamento na memória, enquanto o segundo viola esta organização, resultando em diferenças marcantes de desempenho.

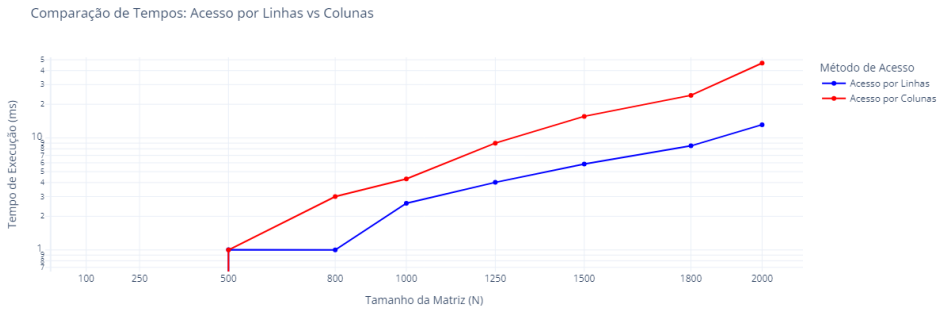


Figura 1: Gráfico de desempenho do tempo de execução para os dois casos estudados.

Quando percorremos uma matriz linha por linha, como no exemplo de uma matriz 3x3 ([0][0] → [0][1] → [0][2] → [1][0] → ...), estamos explorando a localidade espacial, ou seja, à tendência de um programa acessar dados próximos na memória (ex: elementos sequenciais de uma matriz). Este padrão de acesso é altamente eficiente porque pode reduzir drasticamente as falhas de cache (cache misses), que é quando a CPU busca um dado na memória cache, mas ele não está presente, logo a palavra desejada primeiro é lida para a cache e depois transferida da cache para o processador.

Por outro lado, o acesso por colunas ([0][0] → [1][0] → [2][0] → [0][1] → ...) apresenta desafios significativos, como a violação do princípio de localidade espacial, acessando posições de memória distantes entre si, causando dessa forma um maior tempo de execução em comparação ao exemplo anterior.

# 4 Conclusão

Em conclusão, este estudo reforça a importância de alinhar os padrões de acesso de dados com a organização da hierarquia de memória. O acesso por linhas, ao respeitar os princípios de localidade espacial, mostra-se significativamente mais eficiente, especialmente para matrizes de grande porte.

# Referências

[1] William. STALLINGS. *Arquitetura e organização de computadores.* Pearson Education, São Paulo, 8 edition, 2010.