Análise Empírica do Impacto da Localidade de Referência e Latência de Memória no Processamento de Matrizes

Samuel Francisco Ribas

Resumo: Este trabalho apresenta uma análise experimental sobre o impacto dos padrões de acesso à memória no desempenho de sistemas computacionais. Utilizando a linguagem C em ambiente Linux, foram comparados algoritmos de processamento matricial com acesso sequencial (por linhas) e não-sequencial (por colunas). A metodologia empregou alocação dinâmica de memória para evidenciar a latência da RAM (*Random Access Memory*). Os resultados demonstram que a violação do princípio da Localidade Espacial impede o aproveitamento eficiente da Memória Cache, resultando em degradação exponencial de desempenho. O estudo confirma a existência do gargalo de memória e a necessidade de desenvolvimento de software orientado à arquitetura de hardware.
Palavras-Chave: Cache Miss, Desempenho, Hierarquia de Memória, Localidade Espacial.

1 Introdução

A arquitetura de computadores moderna de von Neumann baseia-se, em hierarquia de memória, para equilibrar desempenho e custo. Conforme explicado em [1], a hierarquia de memória é uma estrutura que usa múltiplos níveis de memórias, organizada em uma forma de piramide de acesso á memória, onde o topo é ocupado por memórias de baixa capacidade, porém com altíssima velocidade (memória cache), e a base por memórias de alta capacidade e menor velocidade (RAM e o armazenamento em disco) conforme a distância para o processador aumenta, o tamanho das memórias e o tempo de acesso também aumentam. A memória cache, atua como um intermediário entre o processador e a memória RAM, e funciona como um subconjunto de dados de alto uso armazenados temporariamente em memória de alta velocidade (static RAM) para evitar acesso a memória principal lenta (dynamic RAM). Seu funcionamento eficaz depende dos princípios de localidade: **a *localidade espacial*, que é o acesso de dados vizinho e *localidade temporal* que diz que um dado acessado uma vez pode ser acessado outras vezes.** Quando o processador encontra o dado desejado na cache, ocorre um **Cache Hit**, resultando em acesso rápido. Caso contrário, ocorre um **Cache Miss**, forçando o sistema a buscar o dado na memória RAM, o que penaliza o desempenho de um programa que está diretamente ligado com a eficiência que um processador busca dados nessa hierarquia.

Este relatório aborda o problema de latência de memória através de um experimento empírico. O objetivo é comparar o tempo de execução de dois algoritmos similares, que realizam a tarefa de processamento de uma matriz, mas com padrões de acesso á memória distintos (otimizado versus não-otimizados). A análise demonstra como a violação da localidade espacial gera um excesso de **Cache Misses**, degradando severamente o desempenho do sistema comparado ao acesso sequencial que favorece a ocorrência de **Cache Hits**.

2 Metodologia

Para analisar o impacto da hierarquia de memória e dos princípios de localidade no desempenho de software, foram desenvolvidos dois programas em linguagem C e um script para coleta de dados.

Foram implementadas duas versões de um algoritmo de processamento de imagens (conversão para escala de cinza), operando sobre uma matriz quadrada de pixels com cada pixel ocupando 12 bytes.

O problema do processamento de imagens envolve grandes volumes de dados (GONZALEZ; WOODS, 2000). E é o que torna crítico o uso eficiente da cache. A diferença fundamental entre as versões está no padrão de acesso a memória:

I. **Acesso Otimizado**: Percorre a matriz no sentido das linhas. Este método respeita a forma como a linguagem C organiza vetores multidimensionais na memória, acessando endereços contíguos sequencialmente. Vale ressaltar que ambientes voltados para computação científica histórica, como Fortran, MATLAB e R, utilizam o padrão de ordem por Colunas. Nessas linguagens, a lógica se inverte: a memória é alocada contiguamente coluna a coluna. Portanto, se este mesmo algoritmo fosse portado para Fortran, o acesso matriz[j][i] (por colunas) seria o otimizado, enquanto o acesso por linhas causaria os **Cache Misses.**

II. **Acesso Não-Otimizado:** Percorre a matriz no sentido das colunas. Este método força saltos de memória a cada iteração, pois os elementos de uma mesma coluna estão distantes uns dos outros no endereçamento físico.

2.1 Estratégia de Alocação de Memória

Diferentemente de alocação estática, que garante um bloco único e contínuo de memória, a alocação dinâmica fragmenta a matriz: cada linha pode ser alocada em páginas de memória física dispersas na RAM. Essa escolha metodológica visa acentuar a penalidade de desempenho no algoritmo não-otimizado.

3 Experimentos

Os algoritmos foram executados em um computador equipado com um Intel Core i3-10110U, possuindo 4 MB de Cache L3 e 4GB de RAM, rodando o sistema operacional Linux (ZorisOs). Foram ralacionados os seguintes parâmetros:

- **Medição**: O tempo foi medido utilizando a função *gettimeofday()* da biblioteca <*sys/time.h*> , que oferece a medição do tempo em microssegundos;
- **Variação de TAM:** O tamanho da matriz variou de 0 a 12.000. O limite superior foi estabelecido em 12.000 para evitar esgotamento da memória física (4GB) e consequentemente uso de Swap (disco), o que invalidaria a medição de latência de memória;
- **Visualização:** Os resultados foram exportados para um arquivo CVS e plotados automaticamente utilizando a ferramenta Gnuplot, gerando curvas comparativas de desempenho como veremos na figura 1;
- **Script:** Para garantir a precisão e a otimização, foi desenvolvido um Shell Script que automatiza o processo de compilação, execução e coleta de tempo. A coleta de dados implementa uma média aritmética das execuções para filtrar ruídos causados por processos de fundo do sistema operacional.

Foram variados de 5 a 20 testes, com incrementos de 50 a 200 no tamanho da matriz, porém percebeu-se que em incrementos com valores baixos (como 50), o gráfico ficava visualmente ruidoso (linhas tremem excessivamente e não formam uma curva suave, talvez pelo método que foi plotado). Portanto, foi optado por teste com uma quantidade de dados menores porém com mais qualidade visual. O gráfico gerado ao fim dos testes foi o seguinte:

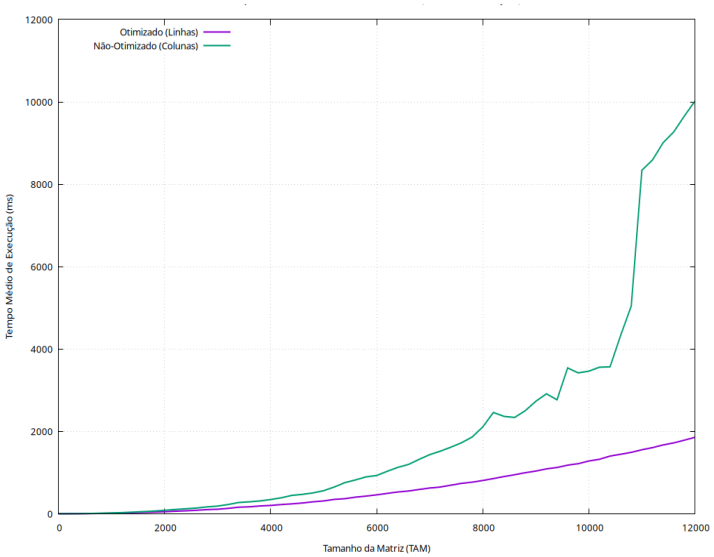

Figura 1: Gráfico da execução do script executando 5 testes, incrementando o tamanho da matriz de 100 em 100, começando em 0 até 12.000.

A eficiência do programa otimizado deve-se ao princípio da Localidade Espacial. A memória cache opera transferindo blocos da RAM. Ao acessar o primeiro pixel de um linha, o processador traz automaticamente os pixels vizinhos no mesmo bloco. Consequentemente, os acessos seguintes resultam em cache hits. No programa que acessa a matriz por colunas a violação da localidade espacial faz com que cada acesso busque um endereço de memória não presente na cache atual, resultando em frequentes **Cache Misses.** O processador é forçado a buscar dados na memória principal (RAM) repetidamente, descartando o restante do bloco carregado anteriormente sem utilizá-lo.

4 Conclusão

Os experimentos que foram realizados testam o impacto crítico de hierarquia de memória no desempenho de algoritmos computacionais . A análise dos métodos de acesso por linha e por coluna demonstrou que a eficiência do software não depende somente da complexidade algorítmica, mas fundamentalmente do padrão de acesso aos dados. Observou-se na figura 1, uma diferença exponencial nos tempos de execução conforme o aumento do tamanho da matriz. O algoritmo de acesso por linhas manteve o tempo linear, validando o principio de localidade espacial, tirando proveito da transferência de dados em blocos, aumentando a ocorrência de cache hits. Já o acesso por colunas mostrou a degradação do desempenho. Isso ocorre porque cada acesso a *matriz[j][i]*, solicita um endereçamento distante do anterior, inutilizando o restante do bloco, fazendo com que o processador solicite novas buscas ocasionando em **Cache Miss**. Por fim a tabela gerada ao fim dos testes:

Tamanho	Linhas_ms
0	0
1000	14,59
2000	63,35
3000	148,43
4000	268,11
5000	423,8
6000	636,87
7000	782,76
8000	1086,99
9000	1350,27
10000	1490,91
11000	2077,83
12000	2479,02

Tabela 1: Gerada a partir dos testes de execução. Para fins de visualização, o script executou 20 testes com incremento de tamanho da matriz em 1000.

5 Referências Bibliográficas

[1] PATTERSON, D. A.; HENNESSY, J. L. *Organização de computadores: a interface hardware/software*. 3. ed. Rio de Janeiro: Campus, 2005.

[2] MONTEIRO, M. A. Introdução à organização de computadores. Rio de Janeiro: LTC, 2000.

[3] GONZALEZ, R. C.; WOODS, R. E. *Processamento de imagens digitais*. São Paulo: Editora Blucher, 2000.S