Relatório

Análise de Desempenho da Memória Cache em Diferentes Configurações por Meio de Simulação no Software Venus

Pedro Henrique Marques de Lima
Felipe Gonçalves Pereira

5 de dezembro de 2024

Introdução

A memória cache constitui um componente essencial na arquitetura de sistemas computacionais modernos, desempenhando um papel central na redução da latência de acesso a dados frequentemente utilizados. Essa camada intermediária de memória, posicionada entre o processador e a memória principal, influencia significativamente o desempenho geral de aplicações, especialmente em operações intensivas de processamento e manipulação de dados.

Neste relatório, analisamos o impacto de diferentes configurações de cache no desempenho de sistemas computacionais utilizando simulações no software Venus. Por meio de testes controlados, buscamos compreender como parâmetros como o número de blocos, o tamanho dos blocos e o padrão de acesso à memória afetam a taxa de acertos (*hit rate*) e, consequentemente, a eficiência do sistema. O estudo foi conduzido em três exercícios práticos, abordando desde a avaliação de taxas de transferência de dados até a simulação de cenários complexos envolvendo acesso a matrizes.

Metodologia

Os testes foram realizados em duas etapas principais, combinando análise de desempenho de algoritmos escritos em C e simulações detalhadas utilizando o simulador Venus, amplamente empregado no estudo de arquiteturas computacionais. Os algorítmos podem ser conferidos e clonados por meio do repositório https://github.com/Felipe0Pereira/Cache.git. As execuções foram conduzidas em uma máquina com especificações descritas a seguir:

- **Máquina 1:** Processador Pentium G4560, 12 GB de memória RAM, sistema operacional Linux Ubuntu 22.04.

A metodologia seguiu uma abordagem estruturada para garantir precisão e reprodutibilidade, dividida nos seguintes passos:

Etapa 1: Análise de Desempenho de Algoritmos em C

Três algoritmos foram desenvolvidos em C para explorar diferentes padrões de acesso à memória e suas implicações no desempenho. Todos os códigos foram executados para uma análise geral, e os resultados foram medidos com foco em tempo de execução e taxa de transferência de dados.

Etapa 2: Simulação com o Venus

Na segunda etapa, foram realizadas simulações no simulador Venus, que permitiram emular o comportamento do cache em diferentes configurações. Essa análise focou na avaliação de padrões de acesso à memória e seu impacto no desempenho do cache.

Etapa 3: Multiplicação de Matrizes

O terceiro exercício avaliou um algoritmo de multiplicação de matrizes, considerando diferentes padrões de acesso aos índices, com foco em identificar o impacto no desempenho devido à localidade espacial e temporal.

Resultados e Discussões

Exercício 1: Taxa de Transferência de Dados

Códigos utilizados para os testes

Primeiro Algoritmo: Escrita sequencial de dados em um arquivo utilizando o sistema de chamadas padrão com a flag `O_SYNC`, que força a escrita síncrona a cada operação. O objetivo foi medir o impacto de operações síncronas no desempenho.

```
// Primeiro algoritmo
#include "iobench.h"

int main() {
    int fd = open("data", O_WRONLY | O_CREAT | O_TRUNC |
       O_SYNC, 0666);
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    size_t size = 5120000;
    const char* buf = "6";
    double start = tstamp();

    size_t n = 0;
    while (n < size) {
        ssize_t r = write(fd, buf, 1);
        if (r != 1) {
            perror("write");
            exit(1);
        }
        n += r;
        if (n % PRINT_FREQUENCY == 0) {
            report(n, tstamp() - start);
        }
    }

    close(fd);
    report(n, tstamp() - start);
    fprintf(stderr, "\n");
}
```

Segundo Algoritmo: Similar ao primeiro, porém sem a flag `O_SYNC`. Isso permitiu analisar o impacto da ausência de sincronização forçada.

```c
// Segundo algoritmo
#include "iobench.h"

int main() {
    int fd = open("data", O_WRONLY | O_CREAT | O_TRUNC ,0666);
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    size_t size = 5120000;
    const char* buf = "6";
    double start = tstamp();

    size_t n = 0;
    while (n < size) {
        ssize_t r = write(fd, buf, 1);
        if (r != 1) {
            perror("write");
            exit(1);
        }
        n += r;
        if (n % PRINT_FREQUENCY == 0) {
            report(n, tstamp() - start);
        }
    }

    close(fd);
    report(n, tstamp() - start);
    fprintf(stderr, "\n");
}
```

Terceiro Algoritmo: Utiliza funções de biblioteca padrão (`fwrite`) para realizar operações semelhantes às anteriores, com foco na análise de abstrações de alto nível.

```c
// Terceiro algoritmo
#include "iobench.h"

int main() {
    FILE *file = fopen("data", "w");
    if (!file) {
        perror("fopen");
        exit(1);
    }

    size_t size = 5120000;
    const char* buf = "6";
    double start = tstamp();

    size_t n = 0;
    while (n < size) {
        ssize_t r = fwrite(buf, 1, 1, file);
        if (r != 1) {
            perror("fwrite");
            exit(1);
        }
        n += r;
        if (n % PRINT_FREQUENCY == 0) {
            report(n, tstamp() - start);
        }
    }

    fclose(file);
    report(n, tstamp() - start);
    fprintf(stderr, "\n");
}
```

Os testes realizados no primeiro exercício resultaram nas seguintes taxas de transferência para diferentes cenários:

- **w01-byte (Primeiro algorítmo):** 4236 bytes/segundo.
- **w02-byte (Segundo algorítmo):** 929203 bytes/segundo.
- **w03-byte (Terceiro algorítmo):** $2,97994 \times 10^7$ bytes/segundo.

Os resultados evidenciam variações significativas no desempenho, indicando a influência de fatores como a capacidade do cache e o padrão de acesso à memória.

Exercício 2

Algorítmo utilizado: Pseudocódigo referente aos cenários do simulador Venus.

```
    # PSEUDOCODE:
#    int array[];  //Assume sizeof(int) == 4
#    for (k = 0; k < repcount; k++) {
#      for (index = 0; index < arraysize; index += stepsize) {
#        if(option==0)
#          array[index] = 0;
#        else
#          array[index] = array[index] + 1;
#      }
#    }
```

Cenário 1

Perguntas:

Quantos elementos de vetor cabem em um bloco da cache?
2 elementos, pois cada elemento é um inteiro de 4 bytes e cada bloco tem tamanho de 8 bytes.

Que combinação de parâmetros está produzindo o hit rate que você observou? Pense sobre os valores de cada parâmetro.
Dados os parâmetros do cenário:

Parâmetros do Programa:

- Tamanho do Array: 128 bytes
- Step Size: 1
- Rep Count: 2
- Opção: 0 (apenas escrita)

Parâmetros da Cache:

- Níveis de Cache: 1
- Tamanho do Bloco: 8
- Número de Blocos: 1
- Política de Posicionamento: Mapeamento Direto
- Política de Substituição de Bloco: LRU

Nesse cenário, o **Step Size = 1** e **Rep Count = 2** resultaram em um **hit rate = 0,5**, porque cada bloco da cache pode armazenar apenas 2 elementos do vetor e a cache possui somente um bloco. Na primeira varredura pelo vetor, o algoritmo tenta acessar a primeira posição do vetor, mas, como o elemento ainda não está na cache, ocorre um **miss**, e o bloco correspondente é carregado na cache contendo o primeiro e segundo elementos do vetor, devido à política de **write-allocate**.

O primeiro acesso ao bloco gera um miss, e o segundo acesso gera um hit. Esse padrão de 1 miss seguido de 1 hit resulta no **hit rate de 0,5**. No entanto, como há espaço suficiente na cache para armazenar apenas 1 bloco de cada vez (2 elementos do vetor), após cada hit, o próximo acesso resulta em um miss, pois o bloco anterior não possui o elemento buscado.

Na segunda varredura (com **Rep Count = 2**), o bloco da cache contém o penúltimo e último elementos do vetor, mas a varredura começa pelo início do vetor. Por isso, a cache precisa recarregar os elementos buscados, mantendo o mesmo padrão de alternância entre **misses** e **hits** que ocorreu na primeira varredura, preservando o **hit rate = 0,5** mesmo com a repetição.

Qual é o hit rate se aumentarmos o Rep Count arbitrariamente? Por quê?
Com os mesmos parâmetros apresentados acima, o **hit rate** permanecerá em **0,5**, pois a cache possui apenas 1 bloco, capaz de armazenar apenas dois elementos do vetor.

Para qualquer vetor com tamanho maior que 2, cada varredura pelo vetor (**Rep Count**) resultará em um padrão fixo: uma iteração que causa um **miss**, seguida por uma iteração que causa um **hit** (o elemento seguinte foi carregado na memória após o miss). Esse comportamento se mantém independentemente do número de repetições, resultando consistentemente em um **hit rate de 0,5**.

Cenário 2

Perguntas:

Qual combinação de parâmetros está produzindo o hit rate observado? Pense sobre os valores de cada parâmetro.
Parâmetros do Programa:

- Tamanho do Array: 128 bytes
- Step Size: 27
- Rep Count: 2
- Opção: 0 (apenas escrita)

Parâmetros da Cache:

- Níveis de Cache: 1
- Tamanho do Bloco: 8
- Número de Blocos: 1
- Política de Posicionamento: Mapeamento Direto
- Política de Substituição de Bloco: LRU

A cache possui apenas 1 bloco neste cenário, capaz de armazenar apenas 2 elementos do vetor. Com um **Step Size = 27** e um tamanho de vetor de 32 elementos, na primeira varredura pelo vetor, cada iteração resulta em um **miss**, já que o passo é maior do que a quantidade de elementos que a cache pode conter.

Ao acessar o primeiro elemento do vetor, ocorre um **miss**, e os dois primeiros elementos são carregados no bloco da cache devido à política de **write-allocate**. O próximo acesso gera outro **miss**, onde são carregados o vigésimo sétimo e vigésimo oitavo elementos do vetor.

No início da segunda varredura, os dados na cache são o vigésimo sétimo e vigésimo oitavo elementos do vetor, mas os dados buscados terão que ser novamente carregados na cache. Isso recria a mesma situação da primeira varredura, onde todos os acessos também geram **misses**. Portanto, o **hit rate** é **0**, já que não há acessos atendidos pela cache.

O que acontece com o hit rate se aumentarmos o número de blocos e por quê?
Para qualquer cache com mais de 1 bloco, haverá 2 **misses** e 2 **hits**. As faltas ocorrem durante a primeira varredura pelo vetor, quando o início do vetor é acessado, gerando um **miss**. O primeiro e segundo elementos são então carregados na cache.

Na segunda varredura, como os blocos da cache já contêm os valores necessários, cada acesso resulta em um **hit**. O **hit rate** é de **0,5**, já que metade dos acessos foram **hits**.

Cenário 3

Perguntas:

Escolha um número de blocos maior do que 1 e determine o menor tamanho do bloco que usa todos os blocos e maximiza o hit rate dado os parâmetros abaixo. Explique o por quê.

Parâmetros do Programa:

- Tamanho do Array: 256 bytes
- Step Size: 2
- Rep Count: 2
- Opção: 1 (escrita e leitura a cada iteração)

Parâmetros da Cache:

- Níveis de Cache: 1
- Tamanho do Bloco: 64
- Número de Blocos: 4
- Política de Posicionamento: Mapeamento Direto
- Política de Substituição de Bloco: LRU

Usando esses parâmetros, cada bloco pode armazenar **16 elementos** do vetor, pois cada elemento é um inteiro de 4 bytes e cada bloco tem tamanho de 32 bytes ($64/4 = 16$ elementos). Os 4 blocos da cache conseguem conter todos os 64 elementos do vetor neste cenário.

Na **primeira varredura**, cada **miss** carrega um bloco com 16 elementos, mas devido à política de escrita e leitura a cada iteração (**Opção = 1**), ocorre um hit, e devido como os outros elementos foram carregados no bloco ocorrem 2 hits (escrita e leitura) a cada elemento acessado. Na segunda varredura, todos os elementos já estão na cache, resultando em hits em todas as operações.

O **hit rate** neste cenário é **0,96875**, o maior possível para o menor tamanho de bloco (16 elementos) em uma cache de 4 blocos.

Exercício 3: Multiplicação de Matrizes

Código utilizado

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>

void multMat1( int n, float *A, float *B, float *C ) {
    for( int i = 0; i < n; i++ )
        for( int j = 0; j < n; j++ )
            for( int k = 0; k < n; k++ )
                C[i+j*n] += A[i+k*n]*B[k+j*n];
}

\\Os demais multMat seguem o mesmo padrao, apenas modificando
   a ordem dos indices dos loops

int main( int argc, char **argv ) {
    int nmax = 1000;
    float *A = (float *)malloc(nmax*nmax * sizeof(float));
    float *B = (float *)malloc(nmax*nmax * sizeof(float));
    float *C = (float *)malloc(nmax*nmax * sizeof(float));

    for( int i = 0; i < nmax*nmax; i++ ) {
        A[i] = drand48()*2-1;
        B[i] = drand48()*2-1;
        C[i] = drand48()*2-1;
    }

    struct timeval start, end;
    void (*orderings[])(int, float*, float*, float*) =
       {&multMat1, &multMat2, &multMat3};
    for (int i = 0; i < 3; i++) {
        gettimeofday(&start, NULL);
        (*orderings[i])(nmax, A, B, C);
        gettimeofday(&end, NULL);

        double seconds = (end.tv_sec - start.tv_sec) +
                         1.0e-6 * (end.tv_usec -
                            start.tv_usec);
        double Gflops = 2e-9 * nmax * nmax * nmax / seconds;
        printf("%d: %.3f Gflops\n", i+1, Gflops);
    }

    free(A); free(B); free(C);
    return 0;
}
```

As funções foram ordenadas com base no desempenho (GFLOP/s) obtido na execução com $n = 1000$:

1. **jki**: 0.611 GFLOP/s
2. **kji**: 0.595 GFLOP/s
3. **ijk**: 0.513 GFLOP/s

4. **jik**: 0.453 GFLOP/s
5. **kij**: 0.200 GFLOP/s
6. **ikj**: 0.197 GFLOP/s

Explicação do Desempenho

O desempenho de cada função depende diretamente de como os elementos das matrizes A, B e C são acessados na memória. Isso é determinado pela localidade espacial e temporal no uso do cache.

Motivo do Alto Desempenho das Funções Rápidas (jki e kji)

- Em **jki**, o loop externo j garante que os elementos da matriz $B[k + j * n]$ sejam acessados de maneira sequencial (bom uso da localidade espacial). O loop intermediário k mantém $A[i + k * n]$ fixo por mais tempo, aproveitando o cache.
- Em **kji**, o loop externo k varia enquanto j mantém $B[k + j * n]$ acessado sequencialmente. Apesar de ser eficiente, o acesso a $A[i + k * n]$ é menos otimizado em comparação com **jki**.

Desempenho Médio (ijk e jik)

- Em **ijk**, $B[k + j * n]$ é acessado de maneira razoável, mas não tão eficiente quanto as funções mais rápidas.
- Em **jik**, o loop externo j favorece $B[k+j*n]$, mas os acessos à matriz $A[i+k*n]$ são menos eficientes.

Motivo do Baixo Desempenho (kij e ikj)

- Em **kij**, o loop externo k varia com muita frequência, quebrando a continuidade do acesso à matriz $B[k + j * n]$.
- Em **ikj**, o loop intermediário k e o loop interno j causam acessos não sequenciais em $B[k + j * n]$, resultando em muitos *cache misses*.

Impacto do Cache no Desempenho

As funções mais rápidas aproveitam a hierarquia de memória de maneira eficiente, acessando os dados sequencialmente e reduzindo o número de acessos à memória principal. Abaixo, uma tabela resumo com os motivos do desempenho:

Ordem	Motivo do Desempenho
jki	Melhor localidade espacial em $B[k+j*n]$ e $A[i+k*n]$.
kji	Boa localidade espacial em $B[k+j*n]$, mas menos eficiente em $A[i+k*n]$.
ijk	Localidade razoável, mas com limitações para acesso a $B[k+j*n]$.
jik	Localidade razoável para $B[k+j*n]$, mas com menos eficiência em $A[i+k*n]$.
kij	Quebra de localidade em $B[k+j*n]$, resultando em muitos *cache misses*.
ikj	Pouca localidade tanto em $B[k+j*n]$ quanto em $A[i+k*n]$.

As diferenças no desempenho entre as funções são causadas principalmente pelo aproveitamento (ou não) da hierarquia de memória. Funções que exploram a localidade espacial de $B[k+j*n]$ e $A[i+k*n]$ têm desempenho superior. Já as funções com acessos aleatórios à memória sofrem penalidades devido aos altos custos de *cache misses*.

Conclusão

Os resultados obtidos reforçam a relevância da configuração do cache e do padrão de acesso à memória para o desempenho de sistemas computacionais. A capacidade de ajustar parâmetros como o número de blocos, o tamanho de cada bloco e o padrão de acesso foi crucial para demonstrar como esses fatores influenciam diretamente a taxa de acertos (*hit rate*) e, consequentemente, o desempenho geral.

O uso do simulador Venus se mostrou particularmente eficaz para compreender as interações entre o cache e a memória principal em arquiteturas modernas. Ele permitiu a exploração detalhada de cenários representativos, oferecendo uma visão prática e fundamentada sobre a relação entre hardware e software no contexto de gerenciamento de memória.

Como perspectivas futuras, sugere-se a análise de padrões de acesso mais complexos, incluindo acessos não sequenciais e aleatórios, além da simulação de diferentes níveis de cache, como L1, L2 e L3. Tais estudos poderiam aprofundar o entendimento sobre as hierarquias de memória e a eficiência do cache em cenários mais realistas, contribuindo ainda mais para o desenvolvimento de sistemas computacionais otimizados.