Laboratório sobre Cache

Alessandra Fialla, Maite Aska, Waldomiro Ottmann
Departamento de Informática
Universidade Federal do Paraná – UFPR
Curitiba, Brasil

***Resumo*—Este trabalho explora o funcionamento da memória cache em processadores modernos. Através de simulações e experimentos, exploramos o funcionamento prático da cache e das ferramentas de avaliação de desempenho. Esse estudo tem como objetivo um maior entendimento sobre os mecanismos da cache.**

***Index Terms*—Memória Cache, Hierarquia de Memória, Desempenho Computacional.**

I. Introdução

O Lab Cache [1] é um roteiro de simulações e experimentos para serem realizados com o objetivo de visualizar o funcionamento da memória cache e avaliar seu desempenho. No roteiro foram propostos três exercícios, "Por que a Cache Importa", " Como o Cache do Processador Funciona" e "Optimização de Código para Cache". Cada um desses exercícios tem o objetivo de observar um ponto do funcionamento da cache ou avaliar o desempenho em diferentes cenários.

II. Exercícios

A. Exercício I - Por que o cache é importante

A cache é essencial para aumentar a eficiência e a velocidade de processamento, uma vez que possibilita o acesso mais rápido de dados e instruções pelo processador. Isso ocorre porque a cache armazena temporariamente as informações, o que reduz significativamente a quantidade de operações de E/S, o impacto do acesso ao disco e o overhead de sistemas. Neste exercício, realizamos um teste de desempenho que mede a velocidade que um sistema lê e escreve dados em um dispositivo de armazenamento. Para isso,executamos três programas que gravam 5 MB de dados no disco rígido do computador de modos diferentes.

Assim, no primeiro programa (w01-byte), teremos o seguinte código:

```
#include "iobench.h"

int main() {
    // opens a file called `data` with
        the O_SYNC
    int fd = open("data", O_WRONLY |
        O_CREAT | O_TRUNC | O_SYNC, 0666);

    if (fd < 0) {
        perror("open");
        exit(1);
    }
    // writes the characer '6' to the file
    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);
        }
        // with some frequency (defined
            in iobench.h), prints out
        // how long it takes to complete
            the write.
        n += r;
        if (n % PRINT_FREQUENCY == 0) {
            report(n, tstamp() - start);
        }
    }
    close(fd);
    report(n, tstamp() - start);
    fprintf(stderr, "\n");
}
```

Nele, podemos observar a presença da flag `OSYNC`, que está relacionada à abertura do arquivo para a gravação com a chamada:

```
    int fd = open("data", O_WRONLY |
        O_CREAT | O_TRUNC | O_SYNC, 0666);
```

Essa ação garante que cada operação de escrita seja sincronizada, ou seja, os dados são gravados fisicamente no disco antes que o sistema retorne a chamada do comando `write`. Fato que introduz latência ao processamento, pois o programa precisa esperar a conclusão da escrita física antes de continuar sua execução. Além disso, ocorre a gravação de um byte por vez. Isso significa que ele escreve 1 byte de cada vez repetidamente, o que é ineficiente, pois a cada chamada de escrita requer que o programa chame o sistema operacional, resultando um aumento significativo da sobrecarga de operações de E/S. Logo, ao executarmos o programa, obtemos os seguintes resultados:

Taxa média de escrita: 1680 byte/sec Total de dados escritos: 4784128 bytes Tempo total: 2845 sec

Tabela I: Dados de escrita

Esses dados comprovam a baixa eficiência do programa, uma vez que a taxa média de escrita de 1680 bytes por segundo é extremamente lenta em comparação a sistemas modernos. Além disso, o tempo total de 47 minutos para gravar aproximadamente 5MB de dados evidencia o impacto significativo da escrita síncrona de um byte por vez, devido a sua demora.

Já no segundo programa(w02-byte), teremos essa alteração no código,em que há a remoção do `OSYNC`:

```
int fd = open("data", O_WRONLY |
    O_CREAT | O_TRUNC, 0666);
```

Essa mudança torna a escrita assíncrona, ou seja, a gravação no disco passa a ser intermediada por uma cache de disco gerenciado pelo sistema operacional para gravar os dados quando for necessário. Desse modo, o programa realiza chamadas ao `write`, solicitando que o sistema operacional escreva um byte. Assim, em vez de gravar diretamente no disco, o sistema operacional armazena o dado no buffer de cache da memória principal e retorna imediatamente para o programa, sem esperar pela gravação física no disco. Isso permite que os dados sejam acumulados no `buffer` e gravados no disco de forma mais eficiente, uma vez que reduz a quantidade de operações físicas no disco, melhorando significativamente o desempenho.

Logo, ao executarmos o programa, obtemos os seguintes dados:

Taxa média de escrita: 1.03475e+06 byte/sec Total de dados escritos: 5120000 bytes Tempo total: 4.948 sec

Tabela II: Dados de escrita

A partir dessas informações, podemos observar que a taxa média de escrita é cerca de 615,92 vezes mais rápida que o programa anterior que demorava cerca de 1680 byte/seg. Além de gravar 5,12 MB em 4.948 segundos. Esse resultado evidencia uma taxa de escrita maior em um tempo menor, sendo assim bem mais eficiente que o programa anterior. Contudo, pode-se destacar que apesar da melhoria, o programa ainda realiza chamadas frequentes do sistema operacional para cada byte escrito. Dessa forma, a cada chamada de write exige que o processador interrompa as operações atuais, salve estado do programa do usuário, para então dar o controle para o sistema operacional executar a operação de escrita.

Por fim, temos o programa(w03-byte), em que houve a substituição do uso da API(Application Programming Interface) de um sistema POSIX(Portable Operating System Interface) [2] para utilização das funções open, write e close pela API da biblioteca padrão C que utiliza funções fopen, fwrite e fclose. Assim, teremos as seguintes alterações no código:

1) Modificação na abertura do arquivo:

```
#include "iobench.h"
int main() {
    FILE* file = fopen("data", "w");
    if (file == NULL) {
        perror("fopen");
        exit(1);
```

Essa alteração consiste em abrir o arquivo com fopen utilizando o “w” de escrita, que corresponde ao 0WRONLY— 0CREAT— 0TRUNC no open. E ao invés de retornar um descritor de arquivo inteiro (fd) , fopen retorna para um ponteiro para o file.

2) Substituição da operação de escrita:

```
size_t r = fwrite(buf, 1, 1, file);
        if (r != 1) {
            perror("fwrite");
            exit(1);
        }
```

Nesse caso há a mudança de write para fwrite, que é a função da biblioteca padrão para escrever dados em arquivos. Sendo o segundo parâmetro, o tamanho de cada elemento e o terceiro parâmetro o número de elementos a escrever.

3) Alteração do fechamento do arquivo:

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

Nessa mudança, teremos a substituição do close, que fecha o descritor de arquivo para o fclose, que fecha o ponteiro file.

Assim, com o uso do fwrite, uma função da biblioteca padrão do C, é possível escrever um byte por vez de forma eficiente, uma vez que ela utiliza um buffer interno para acumular múltiplos bytes antes de gravá-los no disco, o que reduz significativamente o número de operações de escrita física. Logo, o número de chamadas ao sistema operacional também é minimizado, aumentando o desempenho do programa. Além disso, após gravar os dados no buffer de cache, a função retorna imediatamente ao programa, permitindo que ele recupere o controle sem precisar aguardar a conclusão da gravação física no disco, o que torna o processo de escrita mais eficiente. Com isso, ao executarmos o programa,obtemos os seguinte dados:

Taxa média de escrita: 2.51269e+07 byte/sec Total de dados escritos: 5120000 bytes Tempo total: 0.204 sec

Tabela III: Dados de escrita

A partir da análise deles, podemos comprovar que o uso da fwrite é mais eficiente que os dois programas anteriores. Comparativamente, o uso da biblioteca torna-se aproximadamente 14957 vezes mais rápido que o primeiro programa, que utilizava uma escrita síncrona e aproximadamente 24,2 vezes mais rápida que o segundo programa que utilizava uma escrita a assíncrona byte a byte. Com esse resultado, pode-se evidenciar os benefícios do buffering na redução de operações de E/S físicas e no aumento da taxa de escrita.

B. Exercício II - Como funciona o cache do processador

Nesta simulação, observamos o funcionamento do cache do processador. O cache do processador é parte integrante do hardware e está localizado no chip do próprio processador. Para este experimento, foi utilizada a ferramenta `Venus` [3], um simulador de cache, com o objetivo de analisar alterações no desempenho.

A configuração da simulação inclui o uso da estratégia `write-through`, na qual os dados são gravados simultaneamente na cache e na memória principal. Além disso, serão empregadas a política de falta `write-allocate` e a política de substituição `LRU (Least Recently Used)`, para avaliar o impacto dessas configurações no desempenho do sistema.

O Roteiro utilizado disponibiliza um arquivo "cache.s" que contém um código em assembly para RISC-V para ser modificado em diferentes cenários.

O código executa o seguinte Pseudocódigo, também fornecido pelo roteiro:

```
int array[];  // Assume sizeof(int) == 4
    for (k = 0; k < repcount; k++) { //
        repeat the loop repcount times
    // Step through the selected array
        segment with the given step size.
    for (index = 0; index < arraysize;
        index += stepsize) {
        if(option==0)
            // Option 0: One cache access
                - write
            array[index] = 0;
        else
            // Option 1: Two cache
                accesses - read AND write
            array[index] = array[index] +
                1;
    }
}
```

O programa simula o comportamento de acesso à cache, permitindo a realização de diversas simulações para observar o desempenho em diferentes condições.

1) Variáveis Utilizadas:

- `Array[]`: Representa o espaço de memória que está sendo manipulado.
- `repcount`: Conta a quantidade de repetições em que os acessos à cache serão realizados.
- `stepsize`: Determina o intervalo entre os acessos consecutivos no `Array[]`.

2) Comportamento do Código: Se a variável `stepsize` tiver um valor pequeno, a cache se beneficia da localidade espacial, uma vez que seriam realizados acessos mais próximos dentro do vetor. Em oposição, um valor grande pode ocasionar `cache misses`, dependendo do tamanho da linha da cache.

O código possui dois loops. O primeiro loop, mais externo, repete o processo interno `repcount` vezes. O segundo loop, mais interno, realiza operações de escrita e leitura de acordo com a opção fornecia na variável `option` , que pode simular apenas um acesso a cache através da escrita de um valor do vetor, ou 2 acessos, através da leitura seguida de uma operação que atualiza o valor, ou seja, realiza uma escrita.

Cenário 1:

No cenário 1, os parâmetros utilizados estão descritos na Tabela IV.

Cenário 1	**Dados**
Parâmetros do Programa	
ArraySize	128 Bytes
Stepsize	1
Repcount	2
Option	0
Parâmetros da Cache	
Cache Levels	1
Block Size	8 Bytes
Number of Blocks	1
Block Replacement Policy	LRU

Tabela IV: Especificações do Cenário 1

Como podemos observar, temos um array de 128 Bytes (32 inteiros de 4 bytes) e faremos acessos consecutivos nos elementos, uma vez que o `stepsize` está em 1. O loop externo ocorrerá 2 vezes de acordo com a variável `repcount` e será simulado um acesso a cache para cada elemento do array em cada repetição. A cache utilizada possui 1 bloco de 8 bytes, equivalente a 2 inteiros de 4 bytes por bloco. Isso significa que a cada execução do loop, 32 acessos a cache são realizados, totalizando 64 acessos no total. E dois inteiros cabem em um único bloco da cache, sendo assim, o primeiro acesso causará `miss`, mas como serão acessados consecutivamente, o próximo acesso será um `hit`(o segundo acesso ao mesmo bloco já esta na cache).

De forma resumida, para cada miss, teremos também um hit. Como temos 64 acessos, 32 serão misses e 32 serão hits. A `Hit Rate` é calculada dividindo a `hit count` pelo total de acessos a cache e multiplicando por 100.

Então temos:

$$\text{Hit Rate} = \left(\frac{32}{64}\right) \times 100 = 50\%$$

Com os parâmetros dessa simulação, analisamos o comportamento da hit hate com a variação da quantidade de repetições do programa, e pudemos observar que a taxa não varia com o aumento das repetições. Embora aumentem os hits pela localidade temporal, o número de misses permanece constante, uma vez que os misses nos primeiros acessos aos blocos que não foram carregados na cache.

Esses resultados foram validados pela simulação com o `Venus`, como podemos observar nos valores da Tabela V.

Cenário 1	Resultado
Hit Count	32
Accesses	64
Hit Rate	0,5 (50%)

Tabela V: Resultados do Cenário 1

Cenário 2:

Para o cenário 2, a variável de `stepsize` foi aumentada para 27, isso significa que após acessar um elemento do vetor, o próximo a ser acessado estará a 27 índices de distância. O restante dos parâmetros continuou o mesmo que o cenário 1, como pode ser observado na Tabela VI.

Cenário 2	Dados
Parâmetros do Programa	
ArraySize	128 Bytes
Stepsize	27
Repcount	2
Option	0
Parâmetros da Cache	
Cache Levels	1
Block Size	8 Bytes
Number of Blocks	1
Block Replacement Policy	LRU

Tabela VI: Especificações do Cenário 2

Como nossa cache possui apenas um bloco com capacidade para 2 elementos do vetor, esse espaçamento implica que não serão utilizados dados presentes na cache. Isso ocasiona a obtenção de `miss` em todos os acessos, já que cada novo acesso mapeia para um bloco diferente, mas como temos apenas 1 bloco, ele é sempre substituído Sendo assim, o `Hit Rate` fica em 0%.

Além disso, por estarmos usando o espaçamento de 27 índices e nosso vetor ser composto por 32 inteiros, significa que a cada repetição do loop temos 2 acessos a cache, totalizando 4 acessos.

O aumento no número de blocos da cache é uma forma de aumentar a `Hit Rate` obtida. Isso porque com mais blocos disponíveis, com o espaçamento do `stepsize`, teriam mais chances de encontrar o dado já na cache, reduzindo o `miss`. Por exemplo, se a cache desse cenário tivesse pelo menos com 2 blocos, seria possível armazenar os dados dos 2 elementos acessados na primeira iteração e na segunda iteração do loop, por não ter alteração nos índices acessados em repetições diferentes, esses dois dados seriam `hits`. De um modo mais amplo, se a cache tivesse 32 blocos (tamanho do vetor), seria possível que todos os elementos sejam mapeados para a cache e em outras iterações contribuírem para a elevação da `hit Rate`.

Para esse programa em específico, notamos que o cache ter 2 ou 32 blocos não interfere na Hit Hate, uma vez que os índices acessados são os mesmos em cada iteração do loop, e são realizados apenas 2 acessos por iteração. Isto pode ser observado na Tabela VII.

Cenário 2	Resultado
1 Bloco de Cache	
Hit Count	0
Accesses	4
Hit Rate	0 (0%)
2 Blocos de Cache	
Hit Count	2
Accesses	4
Hit Rate	0,5 (50%)
32 Blocos de Cache	
Hit Count	2
Accesses	4
Hit Rate	0,5 (50%)

Tabela VII: Resultados do Cenário 2

Cenário 3:

Já no cenário 3, os parâmetros utilizados estão descritos na Tabela VIII. Como podemos observar, temos um array de 256 Bytes (64 inteiros de 4 bytes) e iremos acessar dois elementos por vez, visto que o `stepsize` está em 2. O loop externo ocorrerá 2 vezes de acordo com a variável `repcount`.

Cenário 3	Dados
Parâmetros do Programa	
ArraySize	256 Bytes
Stepsize	2
Repcount	2
Option	1
Parâmetros da Cache	
Cache Levels	1
Placement Policy	Direct Mapped
Associativity	1
Block Replacement Policy	LRU

Tabela VIII: Especificações do Cenário 3

Ao realizar os testes no simulador `Venus` para este exercício, analisamos dois cenários diferentes, variando o tamanho dos blocos da cache. O objetivo era analisar o impacto do tamanho do bloco na taxa de acertos (hit rate), considerando que o programa processa 256 bytes de dados do array.

No primeiro cenário, configuramos o tamanho do bloco como 128 bytes, o que resultou em uma cache com 2 blocos (256 bytes totais divididos em blocos de

128 bytes). Nessa configuração, a taxa de acertos foi de 98,4375%, indicando um desempenho muito bom em termos de localidade espacial.

No segundo cenário, utilizamos um tamanho de bloco menor, de 64 bytes, o que gerou uma cache com 4 blocos (256 bytes totais divididos em blocos de 64 bytes). Nesse caso, a taxa de acertos foi ligeiramente inferior, chegando a 96,875%.

Embora ambos os tamanhos de bloco sejam eficazes neste caso, o bloco de 128 bytes apresenta uma vantagem em termos de `hit rate`, devido ao melhor aproveitamento da localidade espacial. Com blocos de 128 bytes, cada bloco armazena 32 elementos do array (32 inteiros de 4 bytes cada). Como o programa acessa os elementos em passos de 8 bytes (2 elementos por passo), cada bloco cobre 16 passos de acesso. Nesse caso, o primeiro acesso a um bloco resulta em um miss (falha), enquanto os 15 acessos subsequentes são hits (acertos). Assim, para processar os 256 bytes do array, teremos 2 blocos carregados, resultando em 2 misses e 30 acertos no total.

Por outro lado, com blocos de 64 bytes, cada bloco armazena 16 elementos (16 inteiros de 4 bytes cada). Esses blocos cobrem apenas 8 passos de acesso antes de serem esgotados. Assim, para cada bloco, o primeiro acesso é um miss, e os 7 passos subsequentes são hits.

Para processar os mesmos 256 bytes, serão necessários 4 blocos carregados, o que resulta em 4 misses e 28 acertos no total. A vantagem dos blocos de 128 bytes em relação aos de 64 bytes se deve à redução pela metade no número de misses e ao maior número de acertos subsequentes. Blocos maiores armazenam mais dados adjacentes na memória, reduzindo a necessidade de carregar novos blocos frequentemente e aproveitando melhor a localidade espacial.

Com blocos menores, o programa rapidamente esgota os dados armazenados em cada bloco, gerando mais falhas iniciais e menos acertos subsequentes. Isso explica a superioridade do hit rate dos blocos de 128 bytes em comparação aos de 64 bytes. Os resultados obtidos podem ser observados na tabela IX.

Cenário 3	**Resultado**
2 Blocos de Cache de 128 bytes cada	
Hit Count	126
Accesses	128
Hit Rate	0,984375 (98,43%)
Elementos do Vetor por Bloco	32
4 Blocos de Cache de 64 bytes cada	
Hit Count	124
Accesses	128
Hit Rate	0,96875 (96,87%)
Elementos do Vetor por Bloco	16

Tabela IX: Resultados do Cenário 3

C. Exercício III - Optimização de Código para Cache

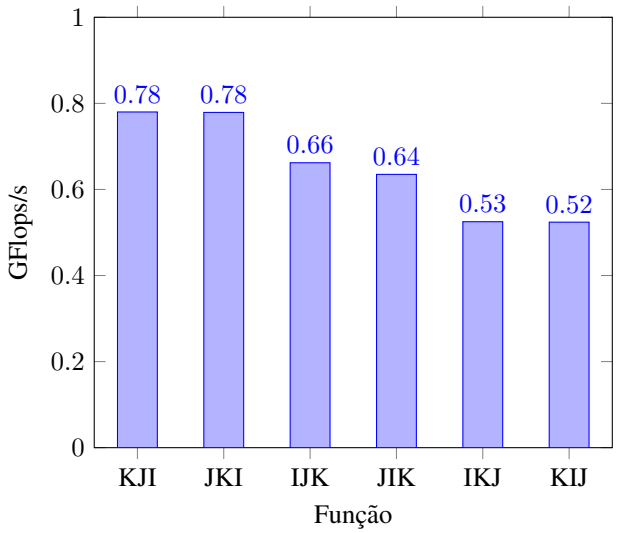
Objetivo do Exercício: O objetivo deste exercício é entender como diferentes padrões de acesso à memória afetam o desempenho de cache e como isso pode ser explorado para otimizar a performance. Para ilustrar esses conceitos, foram analisados os desempenhos de diferentes funções de multiplicação de matrizes, conforme o código base a seguir:

```
// IJK
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] *
                B[k][j];
```

As diversas funções se diferenciam apenas pela ordem em que cada loop é sequenciado, sendo a ordem das variáveis o critério de nomeação de cada função. Portanto, a função IJK tem a variável I no seu loop mais externo, a variável J no loop intermediário e a variável K no seu loop mais interno. A função KIJ, por exemplo, tem no seu loop mais externo a variável K, no loop intermediário a variável I e no loop mais interno a variável J.

```
// KIJ
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] += A[i][k] *
                B[k][j];
```

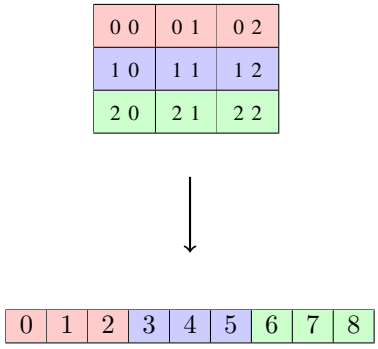
Para obter todas as variações possíveis, as três variáveis necessárias são permutadas, resultando em seis funções diferentes: IJK, IKJ, JIK, JKI, KIJ e KJI. Todas elas calculam corretamente a multiplicação matricial, mas com diferentes níveis de performance. O gráfico abaixo apresenta os resultados médios para cada permutação:



Observação: O código realiza o produto matricial de matrizes $n \times n$ (onde $n = 1000$) e elementos em ponto flutuante. Resultados podem variar.

1) Dados em memória:

Para entender esses resultados, é preciso compreender como os dados são armazenados em memória e, consequentemente, como as matrizes são representadas. As matrizes, naturalmente 2D, são representadas em vetores únicos e unidimensionais. Assim, uma matriz $A[n \times n]$ se torna um vetor A de n^2 elementos. Tomando como base uma matriz 3×3 indexada de 0 a 2, o equivalente ao acessar o elemento $A[1][2]$ seria acessar o elemento $A[5]$, ou seja, $A[i][j] = A[i \times n + j]$. Esse mapeamento significa que o acesso a uma linha da matriz é sequencial na memória, enquanto o acesso a uma coluna envolve saltos entre endereços não consecutivos. Essa diferença é crucial, pois a localidade espacial da memória favorece leituras sequenciais, tornando o acesso linha a linha mais eficiente do que coluna a coluna.



2) Análise de Desempenho:

A função KJI realiza os seguintes acessos sequenciais à memória:

```
C[0] += A[0] * B[0]
C[1] += A[1] * B[0]
C[2] += A[2] * B[0]
C[3] += A[0] * B[3]
C[4] += A[1] * B[3]
C[5] += A[2] * B[3]
C[6] += A[0] * B[6]
C[7] += A[1] * B[6]
C[8] += A[2] * B[6]
C[0] += A[3] * B[1]
C[1] += A[4] * B[1]
C[2] += A[5] * B[1]
C[3] += A[3] * B[4]
C[4] += A[4] * B[4]
C[5] += A[5] * B[4]
C[6] += A[3] * B[7]
C[7] += A[4] * B[7]
C[8] += A[5] * B[7]
C[0] += A[6] * B[2]
C[1] += A[7] * B[2]
C[2] += A[8] * B[2]
C[3] += A[6] * B[5]
C[4] += A[7] * B[5]
C[5] += A[8] * B[5]
C[6] += A[6] * B[8]
C[7] += A[7] * B[8]
C[8] += A[8] * B[8]
```

A função KIJ, por sua vez, realiza os seguintes acessos sequenciais à memória:

```
C[0] += A[0] * B[0]
C[3] += A[0] * B[3]
C[6] += A[0] * B[6]
C[1] += A[1] * B[0]
C[4] += A[1] * B[3]
C[7] += A[1] * B[6]
C[2] += A[2] * B[0]
C[5] += A[2] * B[3]
C[8] += A[2] * B[6]
C[0] += A[3] * B[1]
C[3] += A[3] * B[4]
C[6] += A[3] * B[7]
C[1] += A[4] * B[1]
C[4] += A[4] * B[4]
C[7] += A[4] * B[7]
C[2] += A[5] * B[1]
C[5] += A[5] * B[4]
C[8] += A[5] * B[7]
C[0] += A[6] * B[2]
C[3] += A[6] * B[5]
C[6] += A[6] * B[8]
C[1] += A[7] * B[2]
C[4] += A[7] * B[5]
C[7] += A[7] * B[8]
C[2] += A[8] * B[2]
C[5] += A[8] * B[5]
C[8] += A[8] * B[8]
```

Analisando as funções com melhor e pior desempenho, é possível verificar o impacto da propriedade de localidade temporal e espacial na perfomance da cache. A função KIJ, com pior desempenho, apesar de se beneficiar da localidade temporal pelos acessos a elementos repetidos, não utiliza acessos sequenciais para as matrizes C e B, fazendo uma leitura coluna a coluna. Já a função KJI, que também se aproveita da localidade temporal, utiliza de maneira muito mais eficiente o acesso sequencial linha a linha, resultando em um desempenho superior.

Fica evidente também a semelhança de velocidade entre algumas funções. Isso ocorre porque o loop mais interno, que é executado com maior frequência, exerce a maior influência no desempenho geral. Funções que compartilham a mesma variável no loop mais interno tendem a apresentar tempos de execução semelhantes, pois o padrão de acesso à memória e o aproveitamento da localidade espacial e temporal são determinados principalmente por essa camada do código.

III. CONCLUSÃO

Neste roteiro, exploramos diversos conceitos relacionados à Memória Cache, o que nos permitiu compreender como suas configurações afetam o desempenho do computador. Essa experiência foi fundamental para reforçar os conteúdos aprendidos em aula, além de aprofundar nossos conhecimentos tanto em aspectos técnicos quanto práticos.

Os códigos utilizados nos exercícios I e III estão disponíveis em : https://github.com/alessandrafialla/arquitetura-lab-cache

REFERÊNCIAS

[1] B. University, "Lab 4: Caching," Disponível: https://cs.brown.edu/courses/csci1310/2020/assign/labs/lab4.html, 2020, acessado: Dec. 5, 2024.

[2] R. Q. de Almeida, "O padrão posix: Unificando os sistemas unix-like," Disponível: https://www.dicas-l.com.br/arquivo/o_padrao_posix_unificando_os_sistemas_unix-like.php, 2002, acessado: Dec. 4, 2024.

[3] B. University of California, "Venus: Risc-v simulator for cs61c," Disponível: https://venus.cs61c.org/, 2024, acessado: Dec. 5, 2024.