Arquitetura de Computadores
Trabalho 1

Professora:
Simone de Lima Martins

Alunos:
Leonardo Brasil de Oliveira Cerne
Victor Teles Araujo Speorin

Novembro 2024

Conteúdo

Lista de Tabelas

1 Introdução

Neste trabalho, iremos analisar o comportamento de memórias cache utilizando um simulador e executando programas em C.

O trabalho é dividido em 2 partes, a primeira parte sendo a simulação de uma cache utilizando o Venus, onde iremos verificar as taxas de acerto usando diferentes parâmetros e tipos de associatividade, já a segunda parte é composta de um programa em C que multiplica matrizes, que vai servir para analisarmos os tempos de execução de diferentes implementações.

2 Experimentos da parte 1

2.1 Funcionalidade

Primeiramente tivemos que entender como o simulador Venus funciona. Foi disponibilizado, para realização do trabalho, um código escrito na linguagem RISC-V, o simulador consegue traduzir esse código para binário e assim o executa.

Dependendo dos parâmetros utilizados dentro do código (Array Size, Step Size, Rep Count e Option) e das configurações da cache simulada (Níveis da Cache, Tamanho do Bloco, Número de Blocos, Associatividade e Tipo de Mapeamento) temos diferentes taxas de acerto e partir disso conseguimos responder os questionários disponibilizados.

2.2 Questionário 1

Parâmetros do programa:

- arraysize (a0): 256
- stepsize (a1): 16
- repcount (a2): 4
- option (a3): 0

Parâmetros da cache:

- Cache Levels: 1
- Block Size: 8
- Number of Blocks: 4
- Placement Policy: Direct Mapped
- Associativity: 1
- Block Replacement Policy: LRU

Note que por ser uma política de mapeamento direto, teremos uma linha por conjunto, portanto obrigatoriamente a associatividade será 1.

2.2.1 Questão 1

Como a Cache possui 4 linhas, com 1 bloco em cada, e o tamanho da informação é de 4 bytes, já que é um inteiro, conseguimos colocar 2 informações em cada linha. Porém, o stepsize é de 16, logo, teremos falta em todo acesso a cache, já que os elementos 0, 16, 32, 48 são mapeados para o mesmo conjunto, então sempre terá uma falta por conflito. Ou seja, o hit rate é de 0.

2.2.2 Questão 2

Ao executarmos a simulação, o hit rate observado é de 0.

2.2.3 Questão 3

Calculando o número de faltas na questão 1 e observando o número de faltas no simulador Venus, o resultado é o mesmo, hit rate de 0

2.2.4 Questão 4

Antes de executar a simulação, já era possível notar que somente ocorreriam faltas na cache, já que, devido ao tipo de mapeamento, número de linhas e ao valor do stepsize, sempre teria uma sobreposição das informações.

2.2.5 Questão 5

Independente do valor do repcount, o hit rate continuará sendo 0, visto que a cache sempre tentará acessar a mesma linha, ocorrendo conflito em qualquer iteração.

2.2.6 Questão 6

Como visto na questão 5, o valor do repcount não irá interferir no hit rate, então ao observar os resultados do simulador, utilizando diferentes valores de repcount, o hit rate sempre será 0.

Repcount	Acessos	Acertos	*Hit Rate*
4	16	0	0
5	20	0	0
6	24	0	0
7	28	0	0
8	32	0	0
9	36	0	0
10	40	0	0
11	44	0	0
12	48	0	0
13	52	0	0
14	56	0	0

Tabela 1: *Repcount* x *Hit Rate* no cenário 1

2.2.7 Questão 7

- Arraysize, 64: Diminuindo o tamanho do array, é possível ter uma melhor alocação e utilização do espaço da cache, já que agora trabalhamos com menos elementos, ao fazermos isso o hit rate vai depender do repcount, já que o primeiro acesso sempre vai ser falta e depois todos os outros acessos vao ser acertos, ou seja, com um repcount de 4 teremos um hit rate de 75%, já com um repcount de 100 teremos um hit rate de 99%. Essa mudança também é válida para valores de Arraysize menores que 64 (32, 16, 8, 4).

Arraysize	*Hit Rate*
32	0.75
16	0.75
8	0.75
4	0.75

Tabela 2: *Hit Rate* x *Arraysize* menores ou iguais a 32

- Repcount (valor depende): Ao modificarmos somente o repcount, o hit rate continua sendo 0 para qualquer valor, mas se mudarmos o tamanho do arraysize para valores iguais ou menores que 64, o repcount vai definir o hit rate, já que com mais repcount, maior será o hit rate.

- Stepsize, 256: Quando o Stepsize é configurado para o mesmo tamanho do Arraysize, acessamos o array em saltos que correspondem ao comprimento total do array. Isso significa que em cada iteração, é acessada uma posição única do array uma vez antes de reiniciar, maximizando a reutilização dos dados que já estão na cache.

- Option, 1: Quando o valor de Option é 1, temos dois acessos a cache por iteração, um para leitura e outro para gravação. Na leitura, o elemento nunca está na cache anteriormente (utilizando os parâmetros originais), porém na gravação o elemento já está na cache, o que resulta em um hit rate de 50%.

2.3 Questionário 2

Parâmetros do programa:

- arraysize (a0): 128
- stepsize (a1): 2
- repcount (a2): 1
- option (a3): 1

Parâmetros da cache:

- Cache Levels: 1
- Block Size: 16
- Number of Blocks: 8
- Placement Policy: N-Way Set Associative
- Associativity: 4
- Block Replacement Policy: LRU

2.3.1 Questão 1

Com as configurações determinadas, existem dois acessos à memória por interação do loop interno. A primeira sendo para Load (linha 52), e a segunda para Store (linha 54), o que totaliza 32 acessos na execucão do programa.

2.3.2 Questão 2

O padrão será "faaa", sendo a primeira falta compulsória por encontrar a cache sem o bloco correspondente, e depois disso acessando novamente para gravação, ocorrerá um acerto. O próximo valor ainda estará neste mesmo bloco, dando acerto nos dois acessos.

2.3.3 Questão 3

Quando aumentamos o *repcount*, apenas o primeiro acesso a cada linha da cache terá falta, todos os outros serão acertos, pois os blocos já estarão na cache. Assim, aumentamos o nosso *hit rate* conforme aumentamos *repcount*.

Repcount	*Hit Rate*
1	0.750
7	0.964
15	0.983
30	0.991
45	0.994
75	0.996

Tabela 3: *Repcount* x *Hit Rate* no cenário 2

2.4 Questionário 3

- Parâmetros do programa:
  - arraysize (a0): 128 (bytes)
  - stepsize (a1): 1
  - repcount (a2): 1
  - option (a3): 0
- Cache Levels: 2
- Parâmetros da cache L1:
  - Block Size: 8
  - Number of Blocks: 8
  - Placement Policy: Direct Mapped
  - Associativity: 1
  - Block Replacement Policy: LRU
- Parâmetros da cache L2:
  - Block Size: 8
  - Number of Blocks: 16
  - Placement Policy: Direct Mapped
  - Associativity: 1
  - Block Replacement Policy: LRU

2.4.1 Questão 1

Nessas configurações, foram feitos 32 acessos à L1, com 16 acertos. Já em L2, ocorreram 16 acessos, com 0 acertos. Assim percebe-se que as faltas de L1 são, na verdade, os acessos de L2.

Para calcularmos as taxas pedidas faremos o número de acertos sobre o número total de acessos respectivos:

- L1: $\frac{16}{32} = 0.5$ de *hit rate*.
- L2: $\frac{0}{16} = 0$ de *hit rate*.
- Geral: $\frac{16}{16+16} = \frac{16}{32} = 0.5$ de *hit rate* (já que os acessos de L2 já são as faltas de L1).

2.4.2 Questão 2

Ocorrem 32 acessos à L1 e 16 deles dão falta, porque quando acessamos o primeiro elemento de cada bloco, ele ainda não está copiado na cache, o que resulta em falta.

2.4.3 Questão 3

Ocorrem 16 acessoas à L2, pois só é acessado esse nível da cache quando ocorre falta na L1. Baseado na questão anterior, percebemos que temos 16 faltas na L1, logo realizamos 16 acessos à L2.

2.4.4 Questão 4

- Parâmetros L2:
  - *Block size*: Ao aumentarmos o *block size* da L2, haverá maior taxa de acerto na L2 sem alterar a taxa de acerto na L1.
  - Número de blocos: Já se aumentarmos o número de blocos, nada mudará nem em L1 nem em L2.
  - Associatividade: Ao aumentar a associatividade, continuará sem mudar nada tanto em L1 quanto em L2.

- Parâmetros L1:
  - *Block size*: Ao aumentarmos o *block size* da L1, a taxa de acerto da mesma aumentará, mas a de L2 continuará a mesma.
  - Número de blocos: Se aumentarmos o número de blocos, nada mudará nem em L1 nem em L2.
  - Associatividade: Ao aumentar a associatividade, continuará sem mudar nada tanto em L1 quanto em L2.

Com isso percebe-se que o único parâmetro que consegue aumentar a taxa de acerto de L2 mantendo a de L1, é o *block size* de L2 ao ser aumentado.
Para visualizar isso, nós duplicamos o valor do *block size* da L2. Assim, O primeiro acesso de cada conjunto da L2 será faltoso, mas o segundo dará acerto, resultando um *hit rate* de 0.5 na L2, que podemos observar na segunda linha da tabela (4).

Blocksize L2 (*bytes*)	*Hit Rate*	
	L1	L2
8	0.5	0
16	0.5	0.5
32	0.5	0.75
64	0.5	0.87
128	0.5	0.93
256	0.5	0.93

Tabela 4: *Blocksize* L2 x *Hit Rate*.

2.4.5 Questão 5

i Aumento do número de blocos da L1:

Como calculado na questão 4, fazendo esta alteração, as taxas de acerto se mantêm, visto que para acessar o primeiro elemento de cada bloco, dará uma falta. O que mudou é que agora não precisamos sobrescrever os blocos em cada conjunto, já que temos mais linhas na cache L1. Contudo, elas estão inválidas, o que implica na falta do acesso ao primeiro valor de cada linha. Quanto a L2, a taxa de acerto se manteve, já que a quantidade de acessos depende diretamente da quantidade de faltas da L1, e já que essa se manteve, logo a taxa de acerto da L2 também.

ii Aumento do tamanho do bloco:

Com o aumento do número de blocos, a taxa de falta da L1 diminui, já que uma linha da cache armazenará mais elementos por bloco. Portanto, teremos falta somente no primeiro elemento de cada bloco e todos os seguintes serão acertos. Assim, quanto maior o tamanho do bloco, menor o número de faltas, visto que teremos mais elementos em uma mesma linha da cache, diminuindo a necessidade de cópia do bloco. Já a taxa de acerto da L2 irá se manter, uma vez que não alteramos o tamanho dos seus blocos, então ela continua com a mesma disposição de elementos.

2.4.6 Questão 6

Number of blocks L1 (*bytes*)	*Hit Rate*	
	L1	L2
4	0,5	0
16	0.5	0
32	0.5	0
64	0.5	0

Tabela 5: *Number os Blocks* L1 x *Hit Rate*.

A previsão estava correta, já que a taxa de acerto de L1 e de L2 se mantiveram iguais independente da alteração do número de blocos. Portanto, ocorreu conforme previsto e explicado nas questões 4 e 5.

2.4.7 Questão 7

Block size L1 (*bytes*)	*Hit Rate*	
	L1	L2
4	0.5	0
16	0.75	0
32	0.87	0
64	0.93	0

Tabela 6: *Block size* L1 x *Hit Rate*.

Essa previsão também estava correta, visto que é possível enxergar que a taxa de acerto de L1 aumentou conforme o aumento do tamanho do bloco e a de L2 se manteve igual. Portanto, ocorreu conforme previsto e explicado na questões 4 e 5.

2.5 Conclusão do experimento

Esse primeiro experimento mostrou como as configurações da memória cache são importantíssimas para a execução de programas, e ter conhecimento sobre elas e entender como alterá-las para se beneficiar é fundamental. Percebe-se também que aumentar o tamanho do bloco e o número de blocos diminui a quantidade de faltas, o que melhora o desempenho da cache. Já diminuir a associativade ou utlizar mapeamento direto, pode resultar numa alta taxa de falta.

3 Experimentos da parte 2

3.1 Descrição do experimento

Este experimento tem como objetivo explorar a influência da organização dos *loops* no desempenho da operação de multiplicação de matrizes. Ao analisar múltiplas implementações desse algoritmo, é investigado como diferentes ordens de acesso aos elementos das matrizes impactam o padrão de acesso à memória e, consequentemente, a eficiência da cache.

Durante a execução do experimento, o código fornecido é compilado, registrando-se as velocidades de execução para cada implementação. Em seguida, é proposta uma classificação das implementações da mais rápida para a mais lenta com base na velocidade média de execução.

3.2 Realização do experimento

Para analisarmos o desempenho de cada uma das implementações serão considerados os conceitos de localidade temporal e espacial, que se tratam de formas de acesso contínuo de um mesmo dado, no caso da temporal, ou de forma consecutiva, no caso da espacial.

Com isto em mente, executamos cada uma das implementações 5 vezes e então calculamos a média de cada uma delas. Ao analisarmos os dados obtidos, os representamos nas tabelas (7) e (8),

Implementação	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Vel. Média
ijk (1)	0.543	0.623	0.625	0.630	0.623	0.609
ikj (2)	0.427	0.451	0.453	0.433	0.432	0.439
jik (3)	0.569	0.596	0.597	0.587	0.582	0.586
jki (4)	0.702	0.729	0.740	0.724	0.733	0.725
kij (5)	0.383	0.484	0.486	0.442	0.480	0.455
kji (6)	0.653	0.730	0.732	0.719	0.724	0.711

Tabela 7: Velocidades de Execução e Média por Implementação

Classificação	Implementação	Vel. Média (Gflop/s)
1°	jki (4)	0.725
2°	kji (6)	0.711
3°	ijk (1)	0.609
4°	jik (3)	0.586
5°	kij (5)	0.455
6°	ikj (2)	0.439

Tabela 8: Classificação das implementações.

Desta forma, com os dados em mão e levando em conta os conceitos explicados, conclui-se que:

- jki (4) e kji (6) :Ambas as implementações possuem o loop mais interno sendo o "i", que acessa as linhas de A e C. Assim se aproveitando da localidade espacial de A e C, e da localidade temporal em B, a execução dessas implementações é a mais eficiente. Para complementar, na implementação 4 temos localidade temporal e espacial em C, o que torna levemente mais rápida que a implementação 6.
- ijk (1) e jik (3) : Essas implementações possuem o loop mais interno sendo o "k", que acessa a coluna de A, e a linha de B. Dessa forma, temos localidade espacial em B e temporal em C, contudo, não temos proveito de localidade em A, por isso é menos eficiente que as implementações 4 e 6.
- kij (5) e ikj (2) : Já essas implementações possuem o loop mais interno sendo o "j", que acessa as colunas de C e B, e nesses casos, tem-se pouco proveito da localidade. O que deixa a implementação kij levemente mais eficiente é que os valores da matriz vetor C são acessados para leitura e escrita enquanto os da A são acessados somente para leitura.

3.3 Conclusão do experimento

Durante a execução do experimento, compilamos e executamos o código fornecido, registrando as velocidades de execução para cada implementação. Em seguida, classificamos as implementações da mais rápida para a mais lenta com base na velocidade média de execução. A análise levou em consideração fatores como a quantidade de acessos à memória, o tipo de acesso realizado e a eficiência do padrão de acesso em relação ao funcionamento da cache.

Observamos que as implementações mais rápidas (jki e kji) exploram a localidade espacial e temporal de forma eficiente, realizando acessos contínuos e repetidos aos elementos das matrizes. Por outro lado, as implementações mais lentas (ijk e jik) fazem menos proveito da localidade, resultando em acessos menos eficientes à memória.

Em resumo, as diferenças de desempenho entre as implementações estão relacionadas à forma como os *loops* são organizados e como os acessos aos elementos das matrizes são realizados, destacando a importância da otimização do padrão de acesso à memória para o desempenho geral do algoritmo de multiplicação de matrizes.