Insper

Atividade 3: Otimização de Cache

Igor dos Santos Montagner

Data de entrega: 31/10

Estudamos na aula 14 a hierarquia de memória e o funcionamento do Cache em baixo nível. Nesta atividade prática vamos mensurar o impacto de técnicas de programação *cache-friendly* no tempo de execução de um programa. Em particular, iremos:

- Testar diferentes padrões de acesso à memória e seus impactos no tempo de execução de um programa.
- Usar a ferramenta *valgrind* para medir a eficiência de cache de um programa.
- Quantificar a diferença de desempenho entre versões *cache-friendly* e não.

Otimizações de uso de Cache

Vimos nas aulas que a memória *cache* tem um desempenho muitíssimo superior à memória RAM, mas também é *muitíssimo mais cara*!. O principal fator que afeta a eficiência do *cache* é a *localidade*:

- *temporal*: o programa usa os mesmos endereços de memória repetidamente (por exemplo em loops ou quando utilizamos variáveis locais). Isto inclui evitar o acesso desnecessário a memória usando variáveis locais.
- *espacial*: o programa acessa a memória de maneira previsível, quase que sequencialmente. (muito comum quando tratamos matrizes e outras estruturas deste tipo).

Os dois tipos de localidade não são disjuntos e código com boa performance tipicamente explora ambos tipos de localidade. Dizemos que um código é *cache-friendly* se ele leva em conta a utilização da memória *cache* em seu design.

Trabalharemos um exemplo clássico na otimização de cache: multiplicação de matrizes. O arquivo com o código está disponível no blackboard. As matrizes de teste podem estão na pasta `entradas`.

Abra o arquivo matmul.c. Dentro deste arquivo definimos uma função `void matmult(float *A, float *B, float *C,` contendo uma implementação ingênua da multiplicação de matrizes. Reescreveremos esta função diversas vezes e compararemos seu desempenho de cache usando *valgrind* e mediremos o tempo de execução usando *time*. Antes de continuar, leia o código em `matmul.c` e entenda como chamar o programa.

Medindo desempenho de cache usando *cachegrind*

Podemos medir o desempenho de memória cache de um programa usando a ferramenta *cachegrind*.

```
$ valgrind –tool=cachegrind ./prog arg1 arg2
```

A execução gerará um arquivo `cachegrind.out.(pid)`, onde `pid` é o número do processo executado (isto é útil para não apagar os detalhes de uma execução anterior). Este arquivo pode ser explorado usando a ferramenta visual *kcachegrind* ou o programa de linha de comando *cg_annotate*. Além da demonstração feita em sala de aula, você pode encontrar mais detalhes na documentação oficial (http://valgrind.org/docs/manual/cg-manual.html)

Estamos interessados nas medidas *Data Read Access*, *Data write Access*, *L1 Cache Read Miss*, *L1 Cache Write Miss*, *LL Cache Read Miss* e *LL Cache Write Miss*. As medidas começando por *LL* medem o desempenho do cache *L2*.

Pergunta: Seu código deverá minimizar quais medidas? Qual o impacto de cada uma delas na velocidade de execução deste código?

Como fazer os próximos itens

É possível habilitar a compilação condicional de código usando diretivas do pré-processador. Veja o exemplo abaixo:

```
#ifdef INGENUO
matriz *matmul(....) {

}
#endif
```

Este código só é compilado quanto `#define INGENUO` está presente no código. Para não ficarmos modificando o arquivo toda vez podemos fazer defines na linha de comando do `gcc`.

> '$gcc -DINGENUO -O3 -Wall -pedantic matmult.c -o matmult_ingenuo

As três versões do código que geraremos deverão ser feitas usando `#ifdef` e compiladas da maneira mostrada acima. Coloque, antes da nova versão da função, um comentário contendo a linha de comando do `gcc` usada para compilá-la.

Note que, fazendo isto, teremos 4 (ingênuo mais três otimizações) executáveis compilados a partir do mesmo arquivo `.c`.

Otimizando a função `matmult`

Iremos utilizar uma série de técnicas para modificar a função `matmult` e verificar se seu desempenho melhora ou piora. Para os benchmarks abaixo estão disponiveis três conjuntos de matrizes (arquivos An, Bn e Cn). Avalie as medidas descritas nas seções anteriores para cada uma das execuções. Utilize o arquivo Cn para verificar se suas otimizações continuam calculando a multiplicação corretamente. Para começar, vamos avaliar a versão ingênua do `matmult`. Preencha a tabela abaixo usando as ferramentas `cachegrind`. **Não se esqueça de medir o tempo de execução separadamente usando o comando** `time`.

Medida	Exemplo 1	Exemplo 2	Exemplo 3
Data Read Access	-	-	-
Data Write Access	-	-	-
L1 Read Miss	-	-	-
L1 Write Miss	-	-	-
LL Read Miss	-	-	-
LL Write Miss	-	-	-
Tempo `matmult`	-	-	-

Eliminando acessos de memória desnecessários I (`OPT1`)

No código uma matriz é representada pela struct abaixo. Na função `matmult` atual todo acesso a um campo da struct é feito repetidamente. Porém, não modificamos nenhum dos campos. Escreva uma nova versão de `matmult` que evita esses acessos de memória guardando os valores dos campos em variáveis locais.

```
typedef struct {
    float *data;
    int m, n;
} matriz;
```

Para facilitar as comparações, pesquise como usar a diretiva `#ifdef` para fazer compilação condicional de código. Um exemplo pode ser visto com a definição `INGENUO`, que contém a implementação inicial do código, e que é compilada condicionalmente ao definir esta constante durante a compilação. Para este exercício faça uma versão de `matmult` que é compilada quando definimos `OPT1` e faça uma entrada no seu makefile para compilar um executável chamado `matmult_opt1` que use esta otimização.

Medida	Exemplo 1	Exemplo 2	Exemplo 3
Data Read Access	-	-	-
Data Write Access	-	-	-
L1 Read Miss	-	-	-
L1 Write Miss	-	-	-
LL Read Miss	-	-	-
LL Write Miss	-	-	-
Tempo `matmult`	-	-	-

Pergunta: Esta otimização fez diferença significativa no tempo total? Você consegue explicar o por que usando os valores das outras medidas?

Eliminando acessos de memória desnecessários II (`OPT2`)

A segunda otimização que aplicaremos se refere ao uso de acumuladores para evitar o acesso repetido à uma região de memória. Ao executar a linha

```
C->data[i*C->n + j] += A->data[i*A->n + k] * B->data[k*C->n + j];
```

fazemos um acesso a memória para buscar o conteúdo do elemento i, j de C e depois escrevemos a soma parcial na posição $C_{i,j}$. Fazemos, portanto, `A->n` leituras e `A->n` escritas na matriz C, sendo que só precisamos fazer 1 escrita por elemento!

Podemos substituir estes acessos repetidos por uma variável *soma* que acumula os resultados parciais e escrever na matriz C somente uma vez. Crie uma nova versão da função `matmult` baseado nas otimizações já aplicadas, faça esta modificação no código e meça o desempenho novamente. Verifique se esta modificação pode eliminar outros acessos a memória além do mostrado na linha acima. Reexecute os testes e preencha a tabela abaixo.

Para este exercício faça uma versão de `matmult` que é compilada quando definimos `OPT2` e faça uma entrada no seu makefile para compilar um executável chamado `matmult_opt2` que use esta otimização.

Medida	Exemplo 1	Exemplo 2	Exemplo 3
Data Read Access	-	-	-
Data Write Access	-	-	-
L1 Read Miss	-	-	-
L1 Write Miss	-	-	-
LL Read Miss	-	-	-
LL Write Miss	-	-	-
Tempo `matmult`	-	-	-

Pergunta: Esta otimização fez diferença significativa no tempo total? Você consegue explicar o por que usando os valores das outras medidas?

Otimizando uso do cache (`OPT3`)

Como vimos em aula, matrizes são armazenadas na memória linha a linha. Logo, quando acessamos uma linha de maneira sequencial temos ganhos de desemepenho de cache, pois após acessar um elemento da matriz os próximos acessos serão rápidos. Por outro lado, quando percorremos a matriz por colunas o acesso é lento pois não aproveitamos o cache.

No loop interno de nossa função `matmult` acessamos as matrizes A e C por linha e a matriz B por coluna. Modifique a função criada no tópico `OPT1` para acessar todas as matrizes em ordem de linha.

Dica: mude a ordem dos loops para *kij* e aplique as otimizações do `OPT2`.

Para este exercício faça uma versão de `matmult` que é compilada quando definimos `OPT3` e faça uma entrada no seu makefile para compilar um executável chamado `matmult_opt3` que use esta otimização.

Medida	Exemplo 1	Exemplo 2	Exemplo 3
Data Read Access	-	-	-
Data Write Access	-	-	-
L1 Read Miss	-	-	-
L1 Write Miss	-	-	-
LL Read Miss	-	-	-
LL Write Miss	-	-	-
Tempo `matmult`	-	-	-

Pergunta: a ordem dos loops alterou significativamente os resultados. Você consegue explicar por que?

Conclusões

Seu programa usando as otimizações deve ter obtido desempenho muito superior a versão ingênua. Comente aqui suas conclusões sobre a atividade. Leve em conta os seguintes aspectos:

1. quando vale otimizar?
2. o código otimizado é tão fácil de ler quanto o original?