

Análise de código e eficiência do método do Gradiente

Aryane Ast dos Santos
Kevin Katzer

23 de novembro de 2014

Sumário

1	Introdução	2
2	Verificação de uso de memória com Valgrind	2
3	Arquitetura do computador	2
4	Comparação de desempenho geral	3
5	Análise dos cálculos do fator lambda e resíduo	5
5.1	Medidas de operações em ponto flutuante, memória utilizada e cache misses	5
5.2	Total de operação em ponto flutuante de dupla precisão	5

1 Introdução

Motivação...

2 Verificação de uso de memória com Valgrind

Ao executar a ferramenta Valgrind para se obter informações sobre vazamento de memória no programa gradSolver, foi possível observar 5 erros, todos em contextos diferentes, além de 16 alocações e apenas 2 liberações de memória.

Os resultados da execução do programa são parcialmente apresentados na figura 1.

```
==29599== Command: ./gradSolver -r 5
==28949== HEAP SUMMARY:
==28949== in use at exit: 560 bytes in 14 blocks
==28949== total heap usage: 16 allocs, 2 frees, 800 bytes allocated
==28949== LEAK SUMMARY:
==28949== definitely lost: 560 bytes in 14 blocks
==28949== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 0 from 0)
```

Figura 1: Saída do Valgrind

Aqui o gradSolver foi executado com uma matriz quadrada de ordem 5, porém os mesmos problemas listados na figura 1 são encontrados em execuções de matrizes de qualquer dimensão. E de maneira análoga, ao resolver os problemas apresentados, numa execução com matriz maior, eles ficam também automaticamente resolvidos.

Para contornar os vazamentos de memória encontrados, foi necessário liberar a memória dos vetores alocados explicitamente como o vetor x na função main, o vetor aux em calcGrad e o vetor r de resíduo na função gradSolver. Além disso, no main, foram adicionados frees para os ponteiros para char das flags do getopt.

3 Arquitetura do computador

Utilizando a ferramenta likwid-topology, é possível obter as seguintes informações sobre a arquitetura do computador utilizado para os testes de performance.

Como pode-se notar na figura 2, nas servidoras do DInf, há uma CPU Magny Cours, fabricada pela AMD, com 4 socket e 32 cores (8 por socket).

Existem 3 níveis de cache, sendo o primeiro (L1) com 64kB de memória, o segundo (L2) com 512kB e o terceiro (L3) com 5MB. As caches L1 e L2 são separadas em 32 grupos, sendo

```

CPU type:  AMD Magny Cours processor
Hardware Thread Topology
Sockets:  4
Cores per socket:  8
Threads per core:  1
Socket 0:  ( 0 1 2 3 4 5 6 7 )

NUMA Topology
NUMA domains:  2
Domain 0:
Processors:  0 1 2 3 4 5 12 13 14 15 16 17
Relative distance to nodes:  10 21
Memory:  2403.31MB free of total 24103.8MB

```

Cache Topology	Level: 2	Level: 3
Level: 1	Size: 512 kB	Size: 5 MB
Size: 64 kB	Type: Unified cache	Type: Unified cache
Type: Data cache	Associativity: 16	Associativity: 96
Associativity: 2	Number of sets: 512	Number of sets: 512
Number of sets: 512	Cache line size:64	Cache line size:64
Cache line size:64	Non Inclusive cache	Non Inclusive cache
Non Inclusive cache	Shared among 1 threads	Shared among 4 threads
Shared among 1 threads		

Figura 2: Saída resumida do likwid-topology

que cada grupo é destinado a um core diferente, e o último nível de cache, L3, é separado em 8 grupos, cada grupo destinado a 4 cores.

Há 8 domínios NUMA, e cada domínio correspondendo a uma cache L3. Como apenas o Socket 0 será usado, apenas o primeiro domínio NUMA é de interesse para análise de memória disponível. O domínio 0 possui 16047.3MB de memória RAM, e no momento de execução do likwid-topology, havia 10269.2MB de memória livre.

Dada a especificação acima, o maior sistema linear passível de ser resolvido pela arquitetura descrita é aproximadamente 36600, pois, dada a memória RAM disponível, e sabendo que o programa aloca $n^2 + 3n$ doubles, temos que $64(n^2 + 3n) = 10269.2 \times 2^{23}$.

4 Comparação de desempenho geral

Para a execução dos testes de desempenho, foi utilizada a ferramenta likwid-pin, que afixa a execução do programa à um core da máquina em uso dedicado. Mas como as caches continuam sendo compartilhadas, o que é possível notar na figura (likwid-topology -g), analisar o desempenho de diferentes execuções se torna um problema, pois é necessário minimizar o uso de cache pelos outros programas. A solução encontrada foi executar o gradSolver em single user mode.

No gráfico 3, são mostrados os tempos de execução em segundos, que foram obtidos com a função timestamp, para matrizes de dimensões 32, 256, 1024 e 2048. Na escala horizontal do gráfico, as diferenças entre as potências de 2 e $2^n + 1$ não aparecem claramente. O eixo

das abscissas está em escala logarítmica.

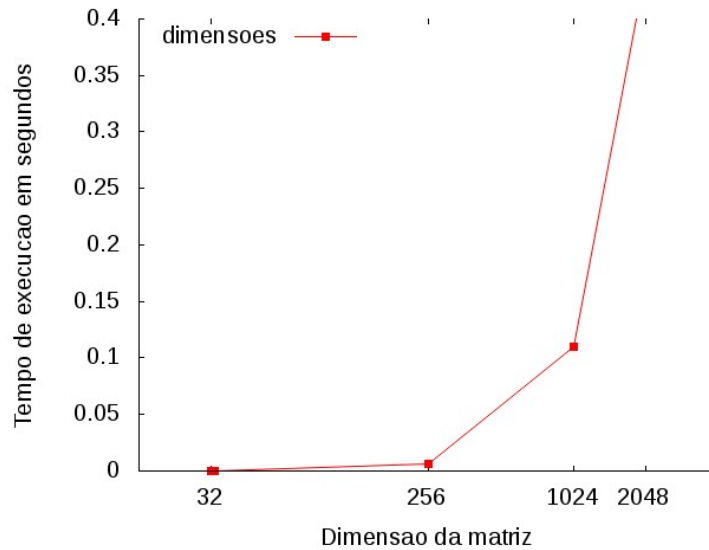


Figura 3: Tempo de execução por dimensão da matriz

Em teoria, as execuções do gradSolver com matrizes de dimensões que não são potência de 2 seriam ligeiramente melhores, por causa de um melhor uso da associatividade da cache. Porém, isso não pode ser verificado nas execuções para os tamanhos de cache exibidos acima. Uma melhor visualização dos tempos de execução pode ser observado na tabela 4, onde a primeira coluna são as dimensões da matriz e a segunda os tempos de execução em segundos.

32	0.00014233589172363
33	0.00027585029602051
256	0.00672078132629395
257	0.00610208511352539
1024	0.10956859588623047
1025	0.10990786552429199
2048	0.43970751762390137
2049	0.43948912620544434

Figura 4: Tempo de execução por dimensão da matriz

5 Análise dos cálculos do fator lambda e resíduo

5.1 Medidas de operações em ponto flutuante, memória utilizada e cache misses

Utilizando a ferramenta likwid-perfctr, foi possível obter informações nos trechos de código referentes ao cálculo do fator lambda e do resíduo sobre memória, cache e operações em ponto flutuante de dupla precisão.

Nos gráficos, nas figuras 5, 6 e 7, podem ser observadas as taxas de cache miss, MBytes/s e MFlops/s, respectivamente, de uma iteração do cálculo de lambda, cálculo do resíduo e as versões de lambda e resíduo com compilação otimizada com -O3 em relação às dimensões da matriz.

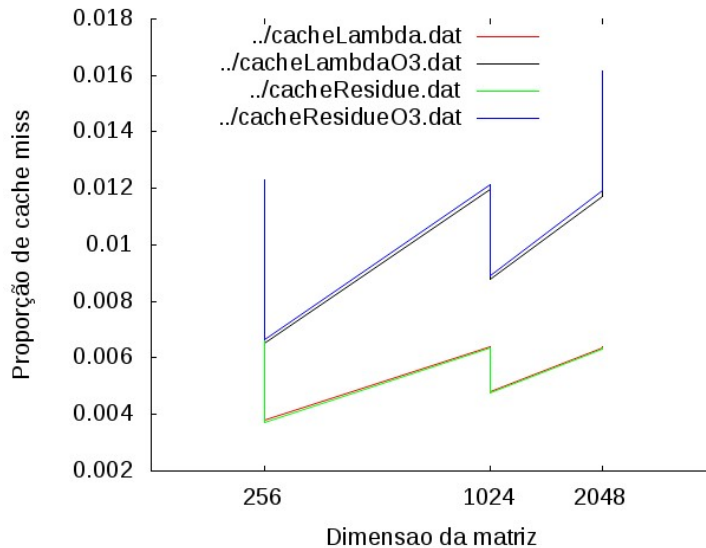


Figura 5: Taxa de cache miss por dimensão da matriz

Faz sentido que o gráfico na figura 5 mostre que a taxa de cache miss aumentou na compilação otimizada, pois.

De maneira análoga, o uso de memória aumenta nas versões otimizadas, pois...

Também...

5.2 Total de operação em ponto flutuante de dupla precisão

Na função multMat, há duas operações em ponto flutuante dentro de dois aninhados, e as duas operações são uma soma e multiplicação juntas, de forma que é possível que se

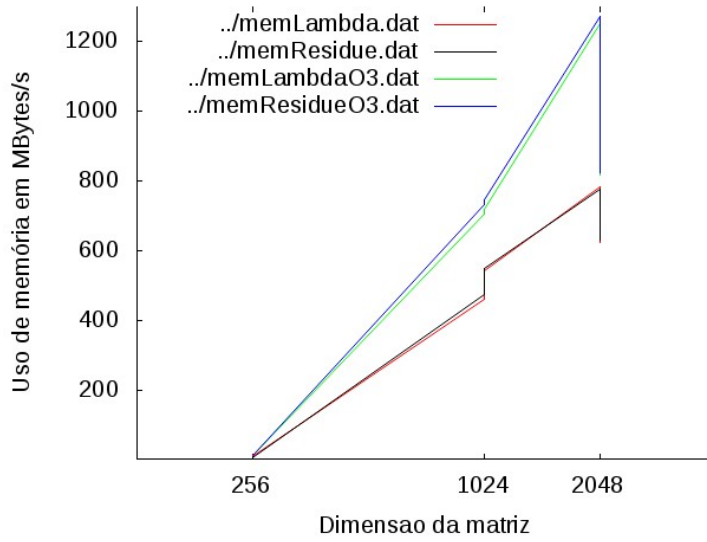


Figura 6: Uso de memória em MBytes/s por dimensão da matriz

conte apenas uma operação. Na função `multVet`, há duas operações de doubles dentro de apenas um laço. Como a função `lambda` usa duas vezes a função `multVet` e uma vez a função `multMat`, flops em função de n é, aproximadamente, $f(n) = n^2 + 2n$, o que pode variar com a otimização do compilador.

Para obter os valores a partir do `likwid`, se usa `likwid-perfctr` medindo o grupo `FLOPS_DP`, pega-se o valor em MFlops/s, multiplica-se pelo tempo em segundos, no caso o `Runtime unhalted`, e por 2^{20} e aí se obtém o total absoluto de flops.

Não foi possível fazer um casamento da complexidade em função da dimensão da matriz com a conta que se faz com os valores do `likwid`, pois saída do `likwid-perfctr` para `FLOPS_DP` sofre muitas variações. Aí seria necessário ler a documentação da ferramenta para descobrir se o programa pega os valores por amostragem de código em execução. Se isso for assumido, os valores ficam mais próximos aos reais a partir de uma amostragem de código maior, e no caso apresentado, somente é levado em consideração uma iteração do cálculo do `lambda`.

A mesma coisa vale para o cálculo do resíduo.

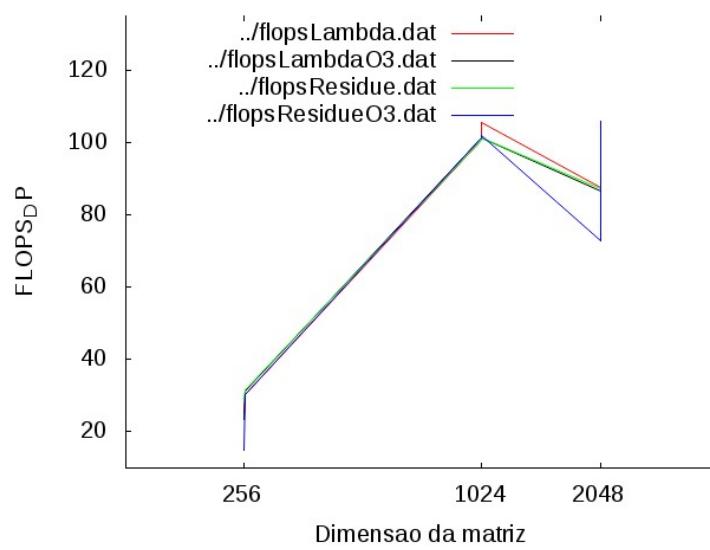


Figura 7: MFlops/s por dimensão da matriz