

MiniEP5

Diogo José Costa Alves (13709881)

27/04/2023

1. **Mostre, com embasamento estatístico, a variação do tempo de execução entre `matrix_dgemm_0` e sua implementação de `matrix_dgemm_1`. Houve melhora no tempo de execução? Explique o porquê.**

Partindo do caso com matrizes de dimensão $N=2048$, rodando 31 vezes cada função e registrando o valor médio, é possível perceber um speed up de mais de **10x** entre as implementações.

`dgemm_0`: 73,3861 segundos.

`dgemm_1`: 7,02896 segundos.

- a. Por que 31 execuções?

Acredito que esse número vem a partir do teorema limite central (TLC), que afirma que a soma variável a aleatória independente se aproxima de uma distribuição normal à medida que o número de amostras aumenta. Como não conhecemos de antemão a distribuição de origem, seguindo a cartilha do TLC, rodamos mais do que 30 experimentos.

- b. Por que o speed up de mais de 10x?

Acredito estar relacionado ao melhor acoplamento entre a forma que definimos a matriz na memória e a sequência de acesso aos dados na memória durante execução. Explicando de outra forma, no código, as matrizes são definidas como arrays contíguos de uma dimensão, onde a linha da matriz equivale a um offset multiplicativo no índice do array e a coluna da matriz equivale a um offset aditivo no índice do array.

Sendo assim, elementos em colunas vizinhas estão localizados sequencialmente no array, já elementos em linhas vizinhas estão distantes $N*8$ bytes (double = 8 bytes).

Olhando para as implementações, no código original (`dgemm_0`), os elementos da segunda matriz são acessados linha a linha. Como disse, esses acessos estão distantes $N*8$ bytes entre si, o que pode dificultar o cache de dados. Durante execução do código original foram contabilizadas 14.991.021.916 **L1-dcache-load-misses**, representando **86,17%** de todos os acessos ao cache L1.

```
(base) diogoalves@servidor:~/work/MAC0219 - Programação Concorrente e Paralela 2023.1/miniEPs/5/src$ sudo perf stat
-d ./main --matrix-size 2048 --algorithm 0
79.345595

Performance counter stats for './main --matrix-size 2048 --algorithm 0':

    79,489.66 msec task-clock                #    1.000 CPUs utilized
         393      context-switches          #    4.944 /sec
          0      cpu-migrations              #    0.000 /sec
        24,639      page-faults             #   309.965 /sec
300,379,591,562      cycles                  #    3.779 GHz
276,058,018,081      stalled-cycles-frontend #   91.90% frontend cycles idle
  60,942,195,990      instructions           #    0.20  insn per cycle
                                   #    4.53  stalled cycles per insn
   8,771,502,656      branches               #   110.348 M/sec
   4,827,018      branch-misses              #    0.06% of all branches
 17,397,639,927      L1-dcache-loads         #   218.867 M/sec
→ 14,991,021,916      L1-dcache-load-misses   #   86.17% of all L1-dcache accesses
   9,160,432,693      LLC-loads              #   115.241 M/sec
<not supported>      LLC-load-misses

 79.509090616 seconds time elapsed

 79.474045000 seconds user
  0.015998000 seconds sys
```

Figura 1 - Contadores de performance da execução do código original

Na função aprimorada, (dgemm_1), alteramos a sequência de acesso a segunda matriz na tentativa de melhorar o acoplamento entre a estrutura de dados da matriz e a sequência de leituras. Nesse caso, passamos a acessar os elementos da segunda matriz, não mais linha à linha mas agora coluna à coluna.

Como tentando explicar anteriormente, esses dados estão guardados sequencialmente no array, o que aumenta a possibilidade de já estarem na cache de dados. Durante execução do código aprimorado, foram contabilizadas 1.079.636.604 **L1-dcache-load-misses**, representando **12,29%** de todos os acessos ao cache L1.

```
(base) diogoalves@servidor:~/work/MAC0219 - Programação Concorrente e Paralela 2023.1/miniEPs/5/src$ sudo perf stat -d ./main
--matrix-size 2048 --algorithm 1
6.896494

Performance counter stats for './main --matrix-size 2048 --algorithm 1':

    7,070.36 msec task-clock                #    1.000 CPUs utilized
         38      context-switches          #    5.375 /sec
          0      cpu-migrations              #    0.000 /sec
        24,638      page-faults             #    3.485 K/sec
26,656,026,683      cycles                  #    3.770 GHz
  9,235,990,664      stalled-cycles-frontend #   34.65% frontend cycles idle
52,264,819,977      instructions           #    1.96  insn per cycle
                                   #    0.18  stalled cycles per insn
   8,756,065,945      branches               #    1.238 G/sec
   4,545,183      branch-misses              #    0.05% of all branches
  8,785,707,843      L1-dcache-loads         #    1.243 G/sec
 1,079,636,604      L1-dcache-load-misses   #   12.29% of all L1-dcache accesses
   552,307,657      LLC-loads              #    78.116 M/sec
<not supported>      LLC-load-misses

 7.071422900 seconds time elapsed

 7.046662000 seconds user
  0.024009000 seconds sys
```

Figura 2 - - Contadores de performance da execução do código aprimorado

Apesar de não saber explicitamente, os detalhes da arquitetura de cache e possível perceber que a reorganização nos acessos à segunda matriz foi capaz de gerar um impacto significativo na redução de L1-dcache-misses e em consequência no tempo total de execução.

2. Execute os mesmos experimentos em outra máquina (por exemplo, a de um colega) e verifique se houve mudança no tempo de execução. Eventuais diferenças podem ocorrer devido a diferenças nas especificações das máquinas (por exemplo, tamanho das linhas de cache)

	dgemm_0:	dgemm_1:
Máquina 1 – Ubuntu Server Intel(R) Core(TM) i5-3570 CPU @ 3.40GHz Bogomips: 6784.99 L1 Data cache 256 KB L2 cache 1MB L3 cache 6 MB	73,3861 segundos	7,02896 segundos
Máquina 2 – Windows WSL Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz Bogomips 6384.00 L1 Data cache 128 KB L2 cache 1MB L3 cache 6 MB	68.0525 segundos	5,69452 segundos

Apesar da máquina 1 ser de uma geração anterior quando comparada a máquina 2 (3ª versus 6ª geração), estava esperando que ela se saísse melhor, já que possui o dobro de cache L1-data-cache (256KB vs 128KB).

Contrariando minhas expectativas, o código executou mais rápido na Máquina 2, mostrando que existem outros componentes da arquitetura que influenciam na performance além do tamanho do cache L1-data.

3. Comentário sobre os desafios encontrados.

Gostaria apenas de deixar um pequeno comentário sobre a experiência de desenvolvimento em C.

É muito bom perceber que uma implementação em C pode ter uma performance 10-100x melhor que a do python. Mas, minha falta de fluência na linguagem e ter compilar executar o código manualmente acabou deixando o processo de experimentação um pouco maçante.

Vou dar uma pesquisada em algo para compilar e rodar o código automaticamente no editor, mas se tiverem alguma dica para melhorar a experiência já agradeço muito.