COC472 - Trabalho 2

Universidade Federal do Rio de Janeiro

Nome: Gabriel do Amaral Ruas
DRE: 119054685

Repositório do Github: https://github.com/RuasGAR/COC472-T2

Introdução

Este trabalho tem como objetivo o estudo e a experimentação relacionados às técnicas de otimização de código, principalmente as de cunho automático. Vamos utilizar um programa de resolução de sistemas lineares pelo método de Jacobi - confeccionado pela Universidade de Boston e disponível em https://github.com/UoB-HPC/intro-hpc-jacobi/blob/master/jacobi.c - junto da ferramenta de perfilagem **gprof** para tal tarefa. O sistema a ser resolvido encontra-se no formato Ax = b, onde **A** é uma matriz quadrada de n elementos, e **x** e **b** são vetores de uma coluna e n linhas.

Especificações

Componente	Modelo
Processador	Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz (configuração padrão)
Disco Rígido	(1TB) WDC WD10JPVX-22J
SSD (rodando Ubuntu 20.04)	(480GB) CT480BX500SSD1 (SATA)
Memória RAM	(8GiB) SODIMM DDR3 Synchronous 1600 MHz (0,6 ns)

Testes Iniciais

De maneira a obter um tempo de processamento razoável para basear as análises de desempenho, foi realizado um teste com 3 diferentes tamanhos da matriz A. Seus valores são apresentados na tabela a seguir junto com os tempos de parede e de CPU da respectiva execução. Por se tratar de um teste de abordagem, consideramos o tempo do

programa inteiro, incluindo alocação de memória e demais operações que tenham relação indireta com os cálculos.

Qtde de Elementos da Matriz A (n)	Tempo de Parede	Tempo de CPU
2000	2m44,712s	2m44,634s
3000	**14m5,413s**	**13m53,672s**
4000	26m34,263s	26m33,587s

Em função da necessidade de utilização do computador para outras atividades, vamos selecionar o tamanho de 3000 elementos para a matriz A.

Otimização Automática

Nas aulas, aprendemos que os compiladores podem conter algumas funcionalidades adicionais, especialmente considerando ambientes de execução, estágio de maturidade do software (desenvolvimento, produção, etc) e os requisitos de performance do produto. Vimos também que dentre essas features adicionais, geralmente estão alguns mecanismos de otimização que podem interferir nos códigos de instruções gerados a partir de um programa escrito em linguagem compatível. Dessa maneira, o compilador pode, por exemplo, ter algum nível de inteligência e definir que uma operação a mais estaria sendo feita de maneira onerosa, ou mesmo que duas operações provavelmente serão melhor aproveitadas caso feitas sequencialmente. As possibilidades são inúmeras, cabendo à robustez oferecida por cada compilador.

No nosso caso, o compilador utilizado para o código em C é o gcc, comum em toda distribuição Linux-like. Nele, podemos utilizar até 4 níveis de otimização:

1. O0(o-zero): é a opção padrão, que não faz quase nenhuma modificação a fim de preservar o conteúdo original do código;
2. O1: há propagação de constantes ao longo de todo o tempo de execução, além de eliminação de sub-tasks desnecessárias, caso existam;
3. O2: chamadas de função podem ser substituídas pelo corpo da função, o que reduz dificuldades com sua stack; além disso, também ocorrem “unrolls” de loops, que nada mais são do que desmembramento de loops para que possam ser processadas paralelamente (o que automaticamente não se aplica a loops com dependências estritas do passo anterior);
4. O3: é a categoria mais agressiva, na qual ocorre vetorização para utilização de paralelismo com instruções SIMD, junto de “function cloning”, uma ideia parecida com a de funções inline, mas aplicadas a contextos específicos dentro do código.

Como a ideia deste trabalho é que conheçamos o impacto da otimização de forma prática, vamos fazer testes utilizando todos eles. Antes, porém, cabe ressaltar que a escolha de categoria de otimização está proporcionalmente relacionada a quantidade de tempo que o processo de compilação acarreta: quanto mais nível de otimização for

aplicado, maior será o tempo necessário para o compilador averiguar todos os cenários em que pode atuar e enfim produzir um binário.

Flag de Otimização	Tempo de CPU (média de 3 tomadas)
O0	14m5,413s
O1	10m28,145s
O2	8m9,326s
O3	7m25,881s

Obs: o tempo de compilação mostrou-se bastante irrisório, sem nenhuma diferença considerável a ponto de justificar comparação nesse ponto.

Profiling

Como o intuito final é encontrar a melhor versão do programa em termos de performance, **vamos considerar a versão compilada com a flag "O3" para a continuidade do estudo.**

Ao rodar o perfilador gprof, são construídas tabelas de chamada e tempo de execução de cada função. Dada a simplicidade do programa, a saída do processo é pouco surpreendente: a função 'run' concentra a maior participação nas instruções executadas pela CPU.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.32    719.45   719.45        1   719.45   719.45  run
  0.03    719.66     0.21                             main
  0.00    719.66     0.00        4     0.00     0.00  get_timestamp
  0.00    719.66     0.00        1     0.00     0.00  parse_arguments
  0.00    719.66     0.00        1     0.00     0.00  parse_int
```

Dessa maneira, se há alguma alteração manual capaz de otimizar o código, é dentro dessa função que poderemos encontrá-la.

Otimização Manual

Em primeiro lugar, é importante notarmos que a matriz A é percorrida (e construída) de forma um pouco diferente do que estamos acostumados: embora tenhamos os tradicionais loops aninhados, a atribuição/leitura de valores acontece em espaçamentos de N posições de memória, dado o passo unitário no loop das colunas.

Em termos de desempenho, essa configuração do programa é prejudicial porque não aproveita os mecanismos de cache de forma otimizada: quando acessamos um dado na memória, os processadores modernos entendem que existe alta probabilidade de utilização de unidades de dados próximas espacialmente. Dessa forma, no caso de um array, por exemplo, é comum que os próximos índices contíguos a determinado índice selecionado sejam alocados nos caches da CPU - que, vale lembrar, são memórias menores mas extremamente velozes, também em função da proximidade com o núcleo.

No caso do código original, estamos sempre pulando N unidades de armazenamento até recuperar um novo dado, quando poderíamos simplesmente inverter o passo unitário, deixando-o relativo ao loop mais externo. Com essa alteração, vamos iterar sobre espaços estritamente sequenciais (e.g., 1,2,3 …, sem saltos), levando ao aprimoramento de uso do cache e a consequente ascensão de desempenho.

Realizando as mudanças na função “run” e na função main - que constrói a matriz - A, temos o seguinte paralelo:

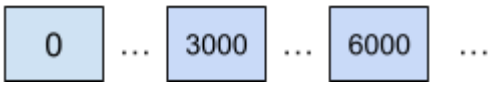
Antes

```
// Perfom Jacobi iteration
for (row = 0; row < N; row++)
{
  dot = 0.0;
  for (col = 0; col < N; col++)
  {
    if (row != col)
      dot += A[row + col*N] * x[col];
  }
  xtmp[row] = (b[row] - dot) / A[row + r
}
```

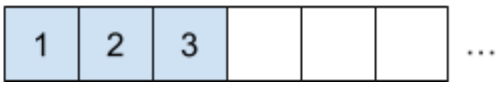
Depois

```
// Perfom Jacobi iteration
for (row = 0; row < N; row++)
{
  dot = 0.0;
  for (col = 0; col < N; col++)
  {
    if (row != col)
      dot += A[col + row*N] * x[col];
  }
  xtmp[row] = (b[row] - dot) / A[row + r
}
```

Pulos de 3000 (valor de N) posições:



Posições são acessadas sequencialmente:



Analisando o programa após tais mudanças, observamos uma redução de aproximadamente 65% do tempo de execução da função gargalo (“run”). Para tornar o estudo mais completo, vamos averiguar os cache-misses ao longo da execução dos programas com e sem a otimização manual, com auxílio do programa **perf**, disponível nativamente em várias distribuições Linux.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.30    253.69   253.69        1   253.69   253.69  run
  0.05    253.81     0.12                             main
  0.00    253.81     0.00        4     0.00     0.00  get_timestamp
  0.00    253.81     0.00        1     0.00     0.00  parse_arguments
  0.00    253.81     0.00        1     0.00     0.00  parse_int
```

Tempo de execução (CPU) após mudança é de "apenas" ***4m13,283s.***

```
Performance counter stats for './jacobi_o3_sem_otm_pg.out --norder 3000':

   970.229.114.195      L1-dcache-loads
   128.983.578.726      L1-dcache-load-misses     #   13,29% of all L1-dcache accesses
```

X

```
Performance counter stats for './jacobi_o3_com_otm_pg.out --norder 3000':

   969.994.843.926      L1-dcache-loads
    17.370.076.019      L1-dcache-load-misses     #    1,79% of all L1-dcache accesses
```

Confirmamos a expectativa: o ***número de cache-misses no programa otimizado*** *é aproximadamente* ***8 vezes menor*** *do que o programa original*

Conclusão

Não há nada mais fundamental na programação do que experimentação de ferramentas e conceitos teóricos na prática. Embora seja um exemplo simples, o método iterativo de Jacobi mostrou-se uma boa alternativa para o estudo de otimizações e seus impactos na performance do produto final, bem como para a familiarização de utilitários nativos de fácil utilização.

Por fim, como síntese do relatório geral, vamos encerrar o documento com uma comparação gráfica dos tempos de execução (CPU) em todos os casos testados.