Relatório: Trabalho 2 – Otimização de Desempenho

Gabriel Lisboa Conegero – GRR20221255
Pedro Folloni Pesserl – GRR20220072
Departamento de Informática
Universidade Federal do Paraná – UFPR
Curitiba, Brasil
glc22@inf.ufpr.br, pfp22@inf.ufpr.br

Resumo

Este relatório documenta o processo de otimização de um programa que realiza ajuste polinomial de curvas, utilizando o método dos **Mínimos Quadrados** e **Eliminação de Gauss**. Também apresenta a comparação entre as duas versões do programa, obtida a partir da ferramenta LIKWID.

1 Metodologia da análise

A análise do programa de ajuste polinomial de curvas foi feita considerando três seções principais do código, que realizam, respectivamente:

1. Geração do sistema linear pelo método dos Mínimos Quadrados;
2. Solução do sistema linear pelo método da Eliminação de Gauss;
3. Cálculo de resíduos do polinômio encontrado.

Tanto a seção de geração do sistema linear quanto a de cálculo dos resíduos do polinômio foram avaliadas com as seguintes métricas: tempo de execução, número de operações aritméticas de ponto flutuante por segundo (FLOP/s), com e sem uso de SIMD, banda de memória utilizada e taxa de *miss* na *cache* de dados. A seção de solução do sistema linear teve seu desempenho avaliado em tempo de execução e FLOP/s, apenas.

2 Otimizações realizadas

2.1 Geração do sistema linear

Originalmente, a versão 1 do programa utilizava uma tabela de *lookup* para armazenar as potências, de 0 a $2m$, dos pontos de entrada, onde m é o grau do polinômio a ser ajustado. Porém, isso precisou ser modificado, porque a entrada agora pode conter até 10^8 pontos, o que impossibilita o armazenamento das potências de todos os pontos na memória. Então, a versão 1 agora utiliza a função `pow_inter()` para calcular as potências dos x's a cada iteração.

A otimização empregada na versão 2 foi inverter a ordem dos laços no cálculo das potências, de forma que iteramos sobre o vetor de pontos de entrada apenas uma vez, possibilitando que ele seja mantido (ainda que em parte) em *cache*. Dessa maneira, percorremos a primeira linha e última coluna da matriz do sistema linear e o vetor de termos independentes múltiplas vezes, calculando a próxima potência do ponto atual que estamos somando a cada iteração com apenas uma multiplicação sobre a potência anterior.

Como o vetor de pontos é muito maior do que a matriz do sistema linear, que é sempre de ordem 5, espera-se que isso diminua a taxa de *miss* na *cache* por ser necessário recarregá-lo em *cache* menos vezes.

2.2 Cálculo de resíduos

Pela mesma questão apresentada na subseção 2.1, a versão 1 do programa foi modificada para usar a função `pow_inter()` na computação das potências dos pontos de entrada, para que fossem aplicadas no cálculo do polinômio com os coeficientes encontrados.

A versão 2 utiliza a mesma estratégia de multiplicar iterativamente cada x_i pela potência já calculada na iteração anterior, de forma a substituir a função custosa `pow_inter()` por uma multiplicação a cada passo.

2.3 Outras

Além do que já foi mencionado, demais otimizações incluem a criação de uma estrutura `struct Ponto_t`, para que os pontos de entrada fossem armazenados em um vetor de estruturas em vez de em dois vetores separados. Essa decisão foi tomada porque sempre que o valor x de um ponto é acessado no programa, o valor de y correspondente também é usado. Dessa forma, quando um ponto é usado em algum cálculo, ambas as suas coordenadas já estão carregadas em *cache*. Essa otimização tem impacto tanto na função que gera o sistema linear quanto na do cálculo dos resíduos.

Além disso, também foi feita a substituição das funções de manipulação de intervalos por suas contrapartes `inline`, o que possibilita redução na taxa de *miss* na *cache* de instruções por não ser necessário desviar o fluxo de execução para a área de definição das funções.

3 Gráficos

Os gráficos aqui apresentados foram gerados por meio da ferramenta gnuplot.

3.1 Geração do sistema linear

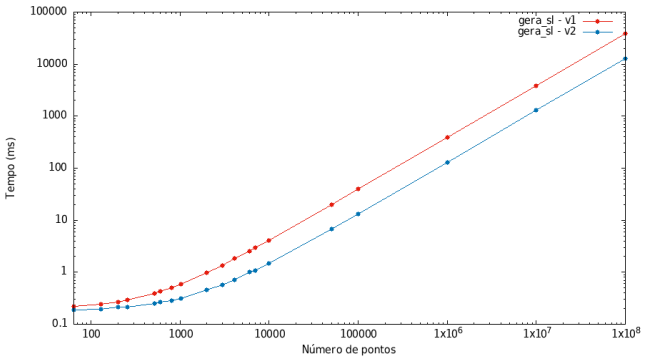

Figura 1.1: Tempo de execução.

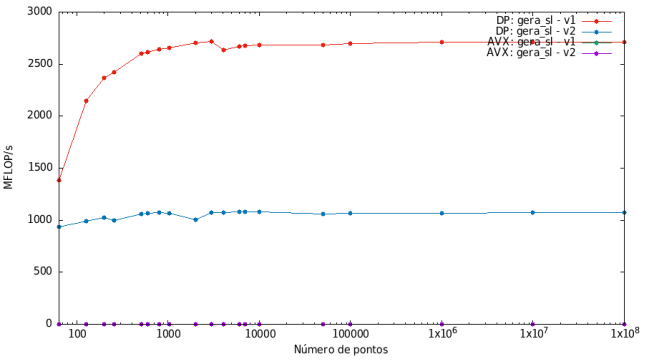

Figura 1.2: Número de operações de ponto flutuante por segundo.

Percebe-se pela figura 1.1 uma redução de aproximadamente 4 vezes no tempo de execução da geração do sistema; isso se deve às otimizações de uso da *cache* e cálculo das potências dos valores de entrada.

A figura 1.2 demonstra que a quantidade de FLOP/s é muito maior na versão 1, por causa do uso da função `pow_inter()`. A versão 2 tem menos FLOP/s devido ao reaproveitamento dos cálculos para exponenciação do intervalo.

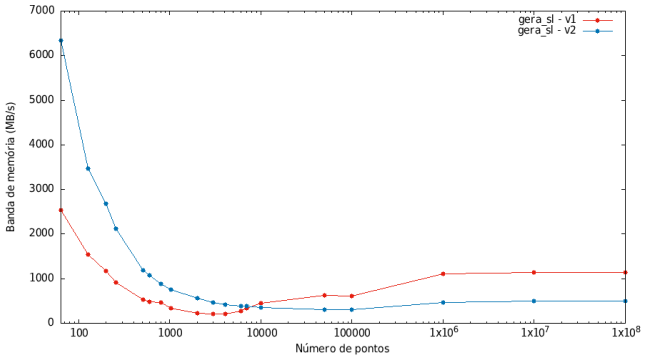

Figura 1.3: Banda de memória utilizada.

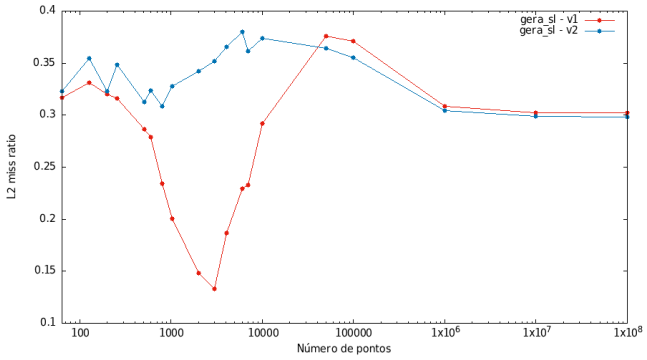

Figura 1.4: Taxa de *miss* na *cache* L2.

Como relatado em 1.3, a banda de memória utilizada é cerca de uma uma ordem de magnitude maior na versão 2, para um número pequeno de pontos. A origem desse aumento são as otimizações que mantêm os dados na *cache*. Contudo, conforme o número de pontos aumenta, as taxas vão diminuindo e se igualando, devido ao aumento de *cache miss*.

O comportamento apresentado na figura 1.4 foi inesperado: com às otimizações de uso da cache, o resultado esperado era que a versão 2 tivesse um *cache miss* menor que a versão 1.

3.2 Solução do sistema linear

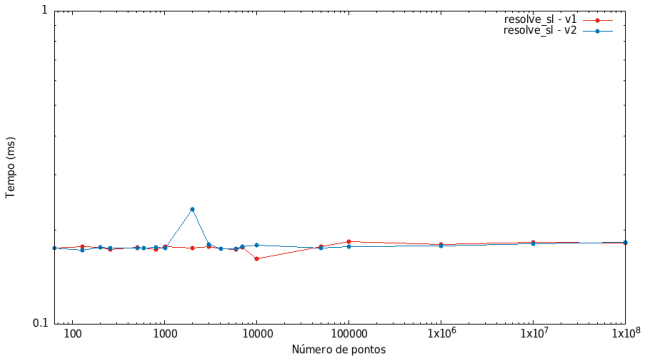

Figura 2.1: Tempo de execução.

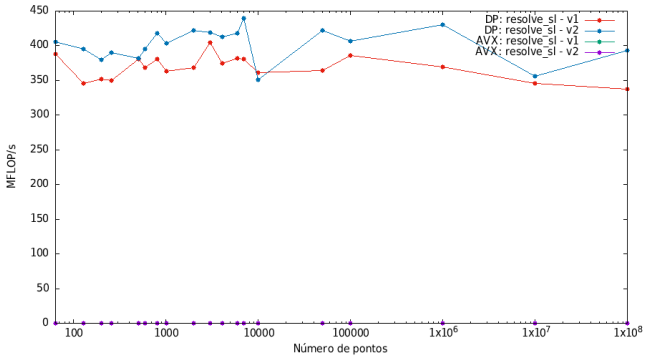

Figura 2.2: Número de operações de ponto flutuante por segundo.

Percebe-se pela figura 2.1 que o tempo de execução da solução do sistema linear foi aproximadamente constante, indepentendemente do número de pontos da entrada. Isso é devido ao fato de que a matriz sempre possui ordem 5, e os cálculos não utilizam a função `pow_inter()`.

Essa relativa constância se repetiu na análise de FLOP/s, como apresentado na figura 2.2, pelos mesmo motivo descritos acima.

3.3 Cálculo de resíduos

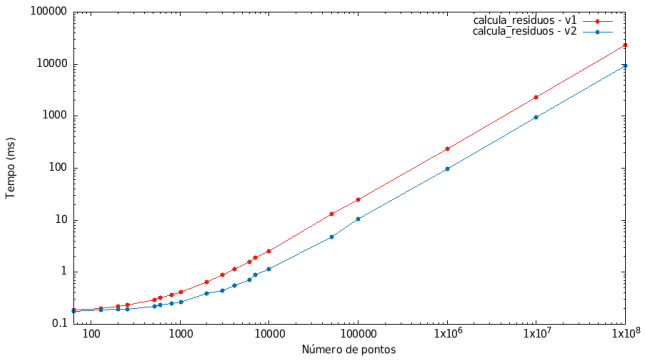

Figura 3.1: Tempo de execução.

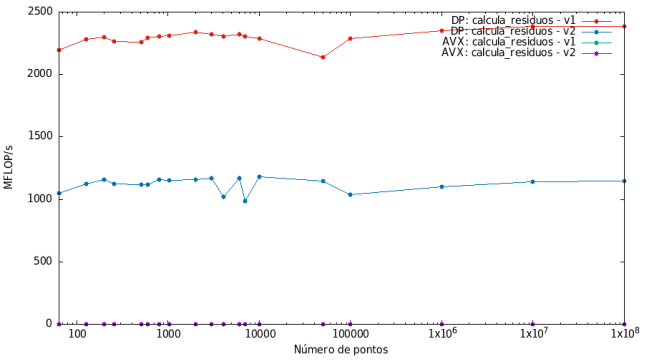

Figura 3.2: Número de operações de ponto flutuante por segundo.

Como na função geradora do sistema linear, é possível perceber pela figura 3.1 que o tempo de execução da função que calcula resíduos foi reduzido em cerca de 4 vezes. Isso se deve à otimização do uso da função `pow_inter()` e do uso de dados na *cache*.

O número de FLOP/s foi reduzido pela metade na função de cálculo de resíduos, como apresentado na figura 3.2. Isso porque a versão 2 otimiza o código e reaproveita a potência x_i^k, calculada em uma iteração, para calcular a potência x_i^{k+1} na próxima, em lugar de utilizar a função `pow_inter()`.

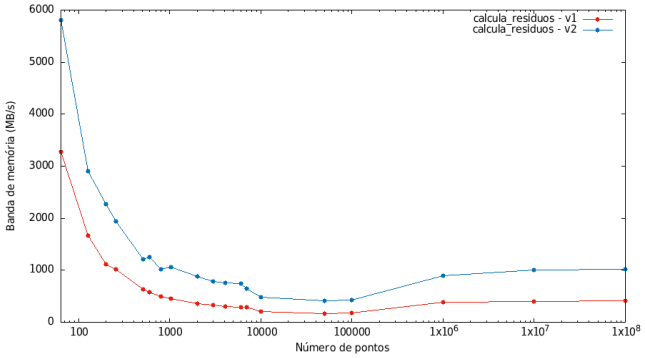

Figura 3.3: Banda de memória utilizada.

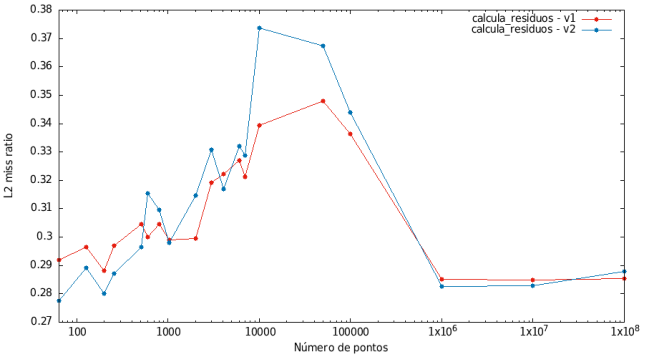

Figura 3.4: Taxa de *miss* na *cache* L2.

Na figura 3.3, pode-se perceber que a função usa, aproximadamente, o dobro de banda de memória na segunda versão quando comparada com a primeira, por conta das otimizações de cache e CPU mencionadas.

Porém, novamente, constatamos que a taxa de *cache miss* foi aproximadamente igual ou maior na versão 2 do que na versão 1, como descrito na figura 3.4, que é um resultado inesperado.

4 Arquitetura do processador utilizado

Saída do comando `likwid-topology -g -c`.

```
--------------------------------------------------------------------------------
CPU name: Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
CPU type: Intel Coffeelake processor
CPU stepping: 9
```

```
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets: 1
Cores per socket: 4
Threads per core: 1
--------------------------------------------------------------------------------
HWThread Thread Core Socket Available
0 0 0 0 *
1 0 1 0 *
2 0 2 0 *
3 0 3 0 *
--------------------------------------------------------------------------------
Socket 0: ( 0 1 2 3 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level: 1
Size: 32 kB
Type: Data cache
Associativity: 8
Number of sets: 64
Cache line size: 64
Cache type: Non Inclusive
Shared by threads: 1
Cache groups: ( 0 ) ( 1 ) ( 2 ) ( 3 )
--------------------------------------------------------------------------------
Level: 2
Size: 256 kB
Type: Unified cache
Associativity: 4
Number of sets: 1024
Cache line size: 64
Cache type: Non Inclusive
Shared by threads: 1
Cache groups: ( 0 ) ( 1 ) ( 2 ) ( 3 )
--------------------------------------------------------------------------------
Level: 3
Size: 6 MB
Type: Unified cache
Associativity: 12
Number of sets: 8192
Cache line size: 64
Cache type: Inclusive
Shared by threads: 4
Cache groups: ( 0 1 2 3 )
--------------------------------------------------------------------------------
********************************************************************************
```

```
NUMA Topology
********************************************************************************
NUMA domains: 1
--------------------------------------------------------------------------------
Domain: 0
Processors: ( 0 1 2 3 )
Distances: 10
Free memory: 5171.89 MB
Total memory: 7826.25 MB
--------------------------------------------------------------------------------


********************************************************************************
Graphical Topology
********************************************************************************
Socket 0:
+---------------------------------------------+
| +---------+ +---------+ +---------+ +---------+ |
| |    0    | |    1    | |    2    | |    3    | |
| +---------+ +---------+ +---------+ +---------+ |
| +---------+ +---------+ +---------+ +---------+ |
| |  32 kB  | |  32 kB  | |  32 kB  | |  32 kB  | |
| +---------+ +---------+ +---------+ +---------+ |
| +---------+ +---------+ +---------+ +---------+ |
| | 256 kB  | | 256 kB  | | 256 kB  | | 256 kB  | |
| +---------+ +---------+ +---------+ +---------+ |
| +-----------------------------------------+ |
| |                   6 MB                  | |
| +-----------------------------------------+ |
+---------------------------------------------+
```