# Relatório do Trabalho 02 - Otimização de Desempenho

Lucas Gabriel Batista Lopes - GRR20220062
Rafel Munhoz da Cunha Marques - GRR20224385

*Universidade Federal do Paraná – UFPR*
Introdução à Computação Científica
Professor: Guilherme Derenievicz
29 de Novembro de 2023
Curitiba, Brasil

***Resumo*—O objetivo desse trabalho foi otimizar o Trabalho 01 e comparar os resultados da versão original e da otimizada. O propósito do Trabalho 01 foi implementar o método dos Mínimos Quadrados em C, que calcula um polinômio de grau N que se ajusta a uma curva descrita por K pontos, utilizando aritmética intervalar para representação rigorosa dos valores reais.**

***Index Terms*—otimização, ajuste de curva, mínimos quadrados, programação, comparação.**

## I. Introdução

O desempenho das duas versões foi comparada sobre três critérios: Geração do Sistema Linear pelo Método dos Mínimos Quadrados, Solução do Sistema Linear pelo Método da Eliminação de Gauss e Cálculo de Resíduo. Para cada uma dessas funções foram comparados os seguintes parâmetros performáticos: Teste de Tempo, Operações aritméticas (MFLOPs/s), Banda de Memória (MBytes/s) e Cache Miss L2.

## II. Otimizações Gerais

Primeiramente, vale ressaltar as otimizações gerais realizadas da segunda versão, que impactam todas as funções positiviamente.

1) Adicionado ”restrict”no parâmetro de ponteiros das funções.
Ao usar restrict em um ponteiro, o compilador é informado que durante a vida útil desse ponteiro, não haverá nenhuma outra referência ao mesmo bloco de memória. Isso permite ao compilador gerar código mais otimizado, eliminando certas verificações de redundância.

2) Uso de ”inline”na declaração de algumas funções
As funções da biblioteca ”interval-analysis.h”foram declaradas como inline, já que são simples, compactas e chamadas com frequência. O principal ganho dessa otimização é evitar a sobrecarga de chamadas de função, substituindo o código da função diretamente no local onde ela é chamada, em vez de fazer uma chamada real para a função.

3) Matriz alocada por meio de um único bloco contínuo de memória
A estrutura de dados da matriz foi otimizada com a implementação de uma struct Matriz, que aloca a memória de forma contígua, otimizando o acesso a cache - evitando ”cache misses”, já que o conteúdo da matriz não está fragmentado na memória. Além disso, a quantidade de chamadas de funções de alocação de desalocação de memória é reduzida.

## III. Geração do Sistema Linear

Essa é a principal funcionalidade do código, é onde o método dos Mínimo Quadrados foi implementado e, portanto, é a parte que mais exige processamento.

- Versão Não Otimizada: A função foi implementada seguindo a fórmula do método. Ou seja, ele calcula todos os somatórios de todas as posições da matriz, realizando muito cálculo repetido e não aproveitando de maneira eficiente a memória cache.

- Otimizações realizadas:
Baseamos a parte principal da otimização no fato de a matriz ser simétrica. Nesse sentido, calculamos apenas a primeira linha e ultima coluna da matriz, depois, para cada elemento restante, copiamos o valor da diagonal superior direita. Dessa forma, o número de FLOPS realizadas pelo programa diminui significativamente, como demonstrado pelo grafico a seguir:

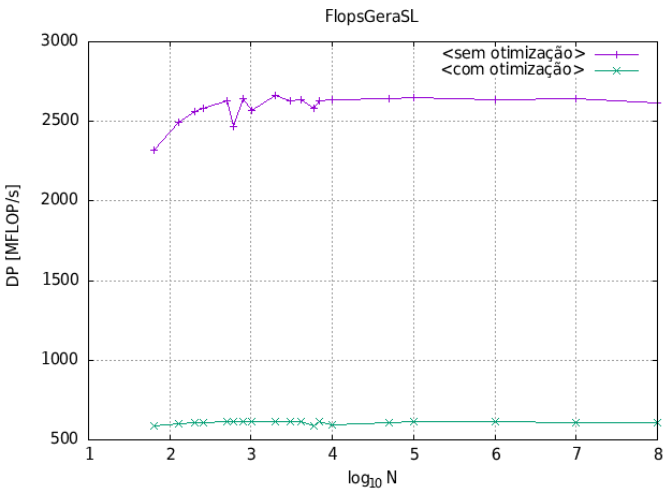


Figura 1. Comparação de MFLOPS/s

Além disso, para melhorar o acesso a cache, fizemos a versão otimizada de modo que os vetores x e y fossem percorridos apenas uma vez, de modo a tentar minimizar a taxa de erros na cache, o que era um ponto crítico da primeira versão. No entanto, para fazer essa modificação, acabamos inserindo acessos com stride ao sistema linear que não ocorriam antes, de modo que essas alterações não tiveram o efeito desejado. Os gráfico a seguir ilustra essa conjuntura:

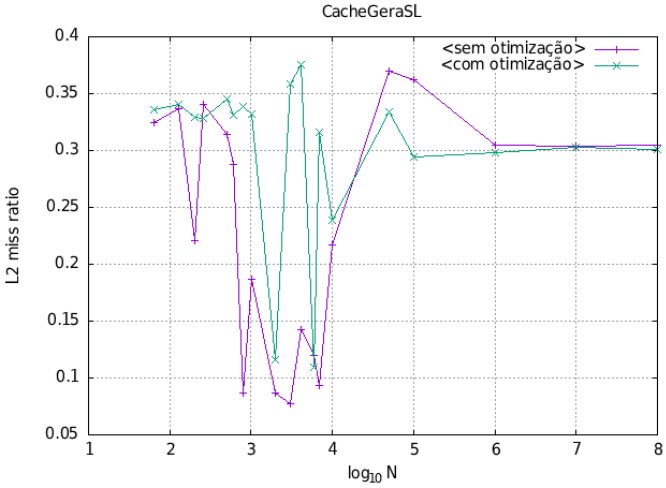


Figura 2. Comparação entre Cache Misses

## IV. Solução do Sistema Linear pelo Método da Eliminação de Gauss

Essa seção não apresenta grande ganho performático, visto que nesse contexto não há muita margem de otimização, pois a matriz calculada é pequena, de 5 linhas e 5 colunas.
Mesmo assim, a função de troca de linhas foi otimizada por meio de uma troca de ponteiros, ao contrário da versão inicial, que realizava a troca de cada elemento da linha por meio de um loop que iterava em todas as colunas. Dessa maneira, além de passos computacionais, foram economizados acessos a memoria, que refletiram na taxa de miss da cache:

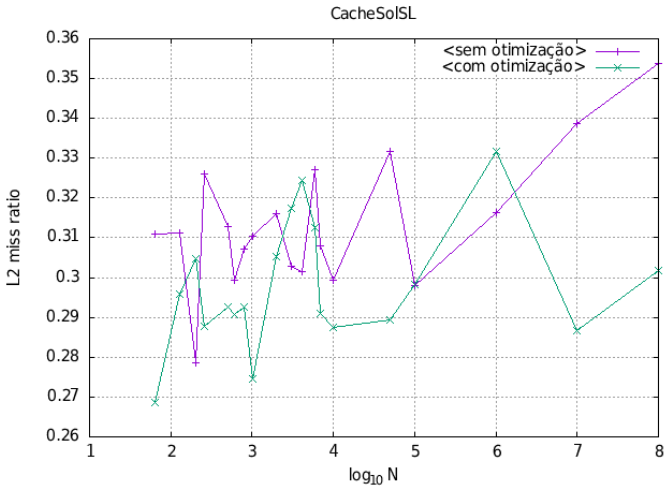


Figura 3. Comparação de L2 miss ratio

Além disso, para evitar erros numéricos e um cálculo repetitivo, na Eliminação de Gauss, ele calcula o inverso do pivô e o reutiliza em cada iteração para os calculos, multiplicando ele pelo número necessário. Na versão antiga, utilizávamos a função divide para realizar esse processo, a qual calculava o inverso do pivô toda vez que era chamada, dessa maneira economizamos FLOPS, como mostra o seguinte grafico:

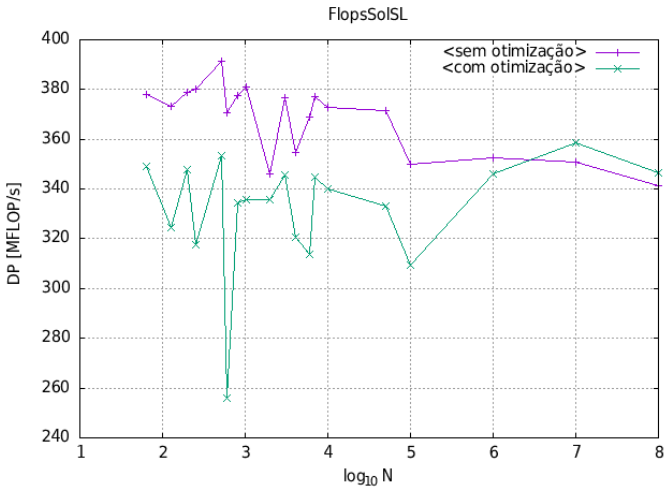


Figura 4. Comparação de MFLOPS/s

A função de retrossubstituição foi otimizada com um

unroll de fator 2. Isso faz com que o funcionamento do pipeline seja aproveitado, de modo que, enquanto uma iteração está executando algumas operações, a próxima iteração já está começando, ou seja, ocorre uma paralelização e um aproveitamento das operações aritméticas disponíveis.

## V. Cálculo de Resíduo

Essa função realiza a diferença entre os valores de y esperados e os valores calculados. Na primeira versão do código é utilizada a função pow() para calcular a potência dos x's. Na segunda versão otimizamos isso salvando o valor da potência em uma variável auxiliar e a cada iteração chama a função multiplica() com essa variável e com o valor original de x.
Além disso, foi realizado uma técnica de "loop unrolling"de fator 2. Os ganhos performáticos são os mesmos já citados na função de retrossubstituição.
Assim, o programa não repete cálculos e evita utilizar a função pow(), que pode ser bastante custosa. O ganho em flops obitido por essa mudança está ilutrado em:

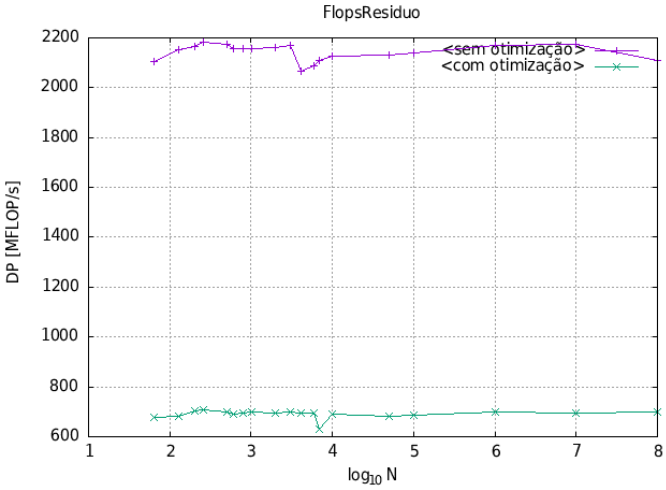


Figura 5. Comparação de MFLOPS/s

## VI. Conclusão

Com essas otimizações, conseguimos com que o método sofresse uma queda de aproximadamente 80% em seu tempo de execução. Uma vez que a parte mais custosa do método é a geração do sistema linear, as otimizaçõpes feitas nessa seção, em especial a melhora na quantidade de flops, foram as grandes reponsáveis por essa queda, e também foram o foco do nosso trabalho. Fizemos tentativas com diferentes versões desse cálculo, algumas com melhoras mais signifcativas de memómia, outros de CPU. A versao apresentada nesse reltório foi a que consideramos mais satisfatória. A seguir, o gráfico de tempo para a gração do sistema linear, que também pode ser entendido como o tempo de execução do programa como um todo, dado que o tempo de execução das outras partes é mínima se comparadas a essa.

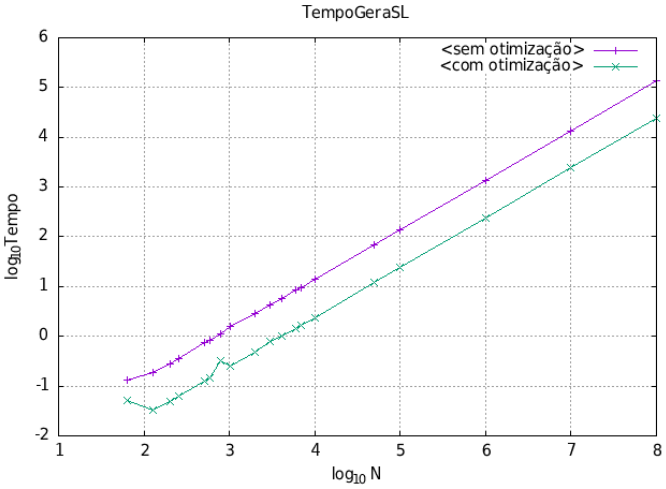


Figura 6. Comparação de tempo para gerar o sistema linear

Os demais gráficos obtidos durante os testes estão no diretório resultadosObtidos/ , bem como os arquivos .out com os pontos desses gráficos.