Universidade Federal do Paraná - UFPR
Setor de Ciências Exatas
Departamento de Informática - DInf
Disciplina: ci164 - Introdução à Computação Científica

Relatório do trabalho prático de Otimização do Desempenho para Sistemas Lineares Esparsos com Pré-condicionantes

Gabriel de Souza Barreto - GRR20166812

Resumo

Este documento irá apresentar os resultados obtidos durante as atividades da segunda parte do trabalho de Introdução à Computação Científica. O objetivo é mostrar as alterações que foram feitas no código, explicando cada uma e justificando as razões para cada alteração. Além disso, será apresentada a topologia da máquina usada para medir os resultados e será feita uma comparação entre a performance das duas versões do código, levando em consideração os critérios definidos pelos professores.

Introdução

As otimizações foram feitas a partir da versão do código **v1** entregue na primeira parte da disciplina, com a exigência que a estrutura de dados armazenando a matriz de coeficientes não incluísse nenhuma diagonal nula. Como isso já era atendido, as únicas alterações necessárias foram a inclusão das marcações do **LIKWID** e a correção da função de *timestamp( )* dada na especificação. Na versão **v2** entregue e utilizada para as comparações, foram utilizadas estratégias vistas em sala e no livro "*Introduction to High Performance Computing for Scientists and Engineers*" dos autores Georg Hager e Gerhard Wellein. A melhora de desempenho foi restrita à duas partes específicas, que consistem em:

- A operação **op1** de iteração do método de Gradiente Conjugado com Pré-condicionador de Jacobi;
- A operação **op2** de cálculo do resíduo a partir da matriz de coeficientes e vetor de termos independentes originais;

Vale observar que, embora seja a parte mais demorada e custosa do código, a preparação da matriz para a obtenção de uma matriz de coeficientes simétrica **não** foi otimizada, já que não era parte da especificação proposta. Essa preparação consiste em fazer uma multiplicação da matriz A pela sua transposta At e também multiplicar essa transposta pelo vetor b, transformando o sistema **Ax = b** em **AtAx = Atb**.

Topologia

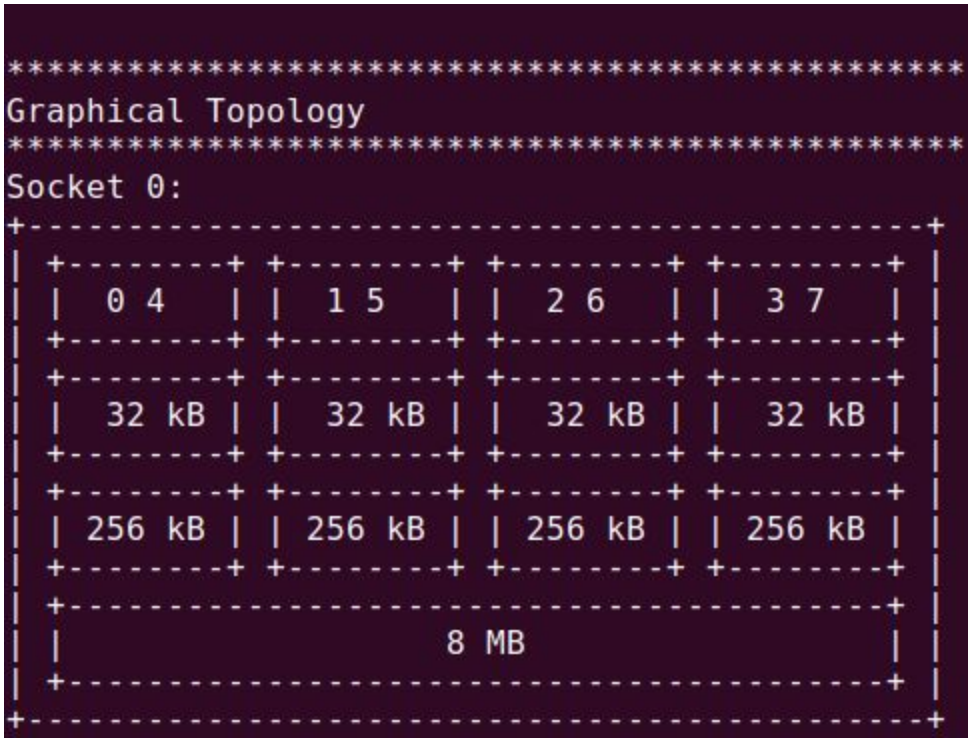
Para o desenvolvimento do trabalho, foi utilizada a máquina de trabalho do aluno no C3SL pela disponibilidade e total acesso para quaisquer instalações ou configurações necessárias. As especificações obtidas por **$ likwid-topology -c -g** são:

```
--------------------------------------------------------------------------------
CPU name:    Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
CPU type:    Intel Skylake processor
CPU stepping:    3

********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:            1
Cores per socket:    4
Threads per core:    2
--------------------------------------------------------------------------------
HWThread    Thread      Core    Socket          Available
0           0           0       0       *
1           0           1       0       *
2           0           2       0       *
3           0           3       0       *
4           1           0       0       *
5           1           1       0       *
6           1           2       0       *
7           1           3       0       *
--------------------------------------------------------------------------------
Socket 0:           ( 0 4 1 5 2 6 3 7 )
--------------------------------------------------------------------------------

********************************************************************************
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: 2
Cache groups: (0 4) (1 5) (2 6) (3 7)

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: 2
Cache groups: (0 4) (1 5) (2 6) (3 7)

Level: 3
Size: 8 MB
Type: Unified cache
Associativity: 16
Number of sets: 8192
Cache line size: 64
Cache type: Inclusive
Shared by threads: 8
Cache groups: (0 4 1 5 2 6 3 7)



Estimativa de Desempenho

Considerando uma máquina com 4GB de memória RAM disponível para o programa e sistemas com 7 diagonais (k=7), levando em consideração as estratégias utilizadas no código, podemos estimar que:

Maior sistema linear que pode ser resolvido:
Com k = 7, a matriz AtA simétrica fica com k = 13;
Cada double ocupa 8 bytes e são alocados no programa:
10 vetores de (n + 1) + 2 vetores de 7n + 1 vetor de 13n;
Dessa forma, o n máximo seria cerca de 13 500 000.

Tempo para uma iteração neste sistema (em função de N):
$O((n + 1)^2) + O(4.(n + 1))$ operações de ponto flutuante double;
Em benchmarks, a referência de desktops é de 60GFLOPS;
Com isso, sem considerar a latência da memória, podemos estimar que cada iteração levaria cerca de 0,00025n segundos.

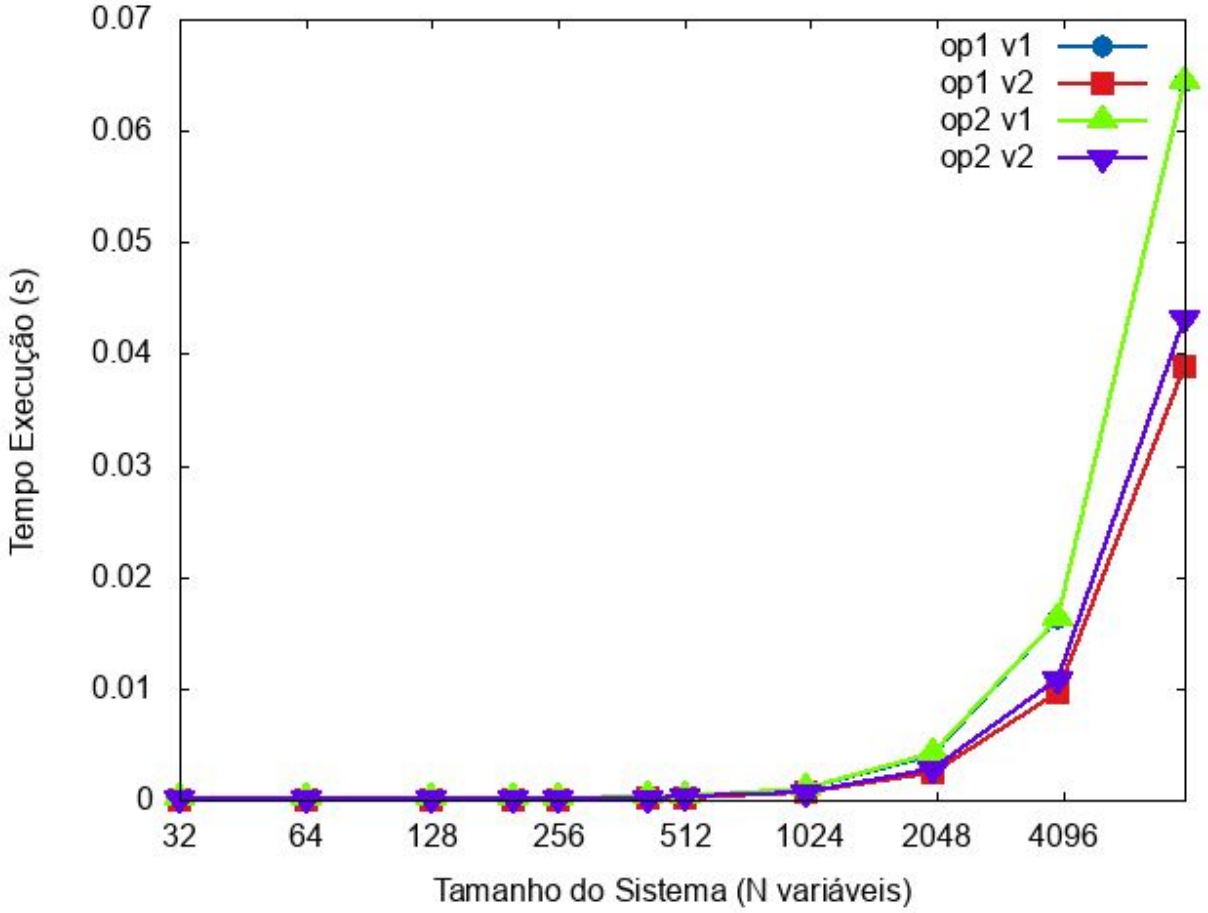
Operações em ponto flutuante executadas em cada iteração:
Cerca de $n^2 + 6n$ operações

Resultados obtidos

O programa foi testado em relação ao tempo, banda de memória, cache miss e MFLOP/s para cada uma das versões do código **v1** e **v2**. Para isso, foi usada a função *timestamp( )* e os grupos **L3**, **L2CACHE** e **FLOPS_DP**. Como não foi utilizado **AVX**, não foram feitos testes nesse quesito.

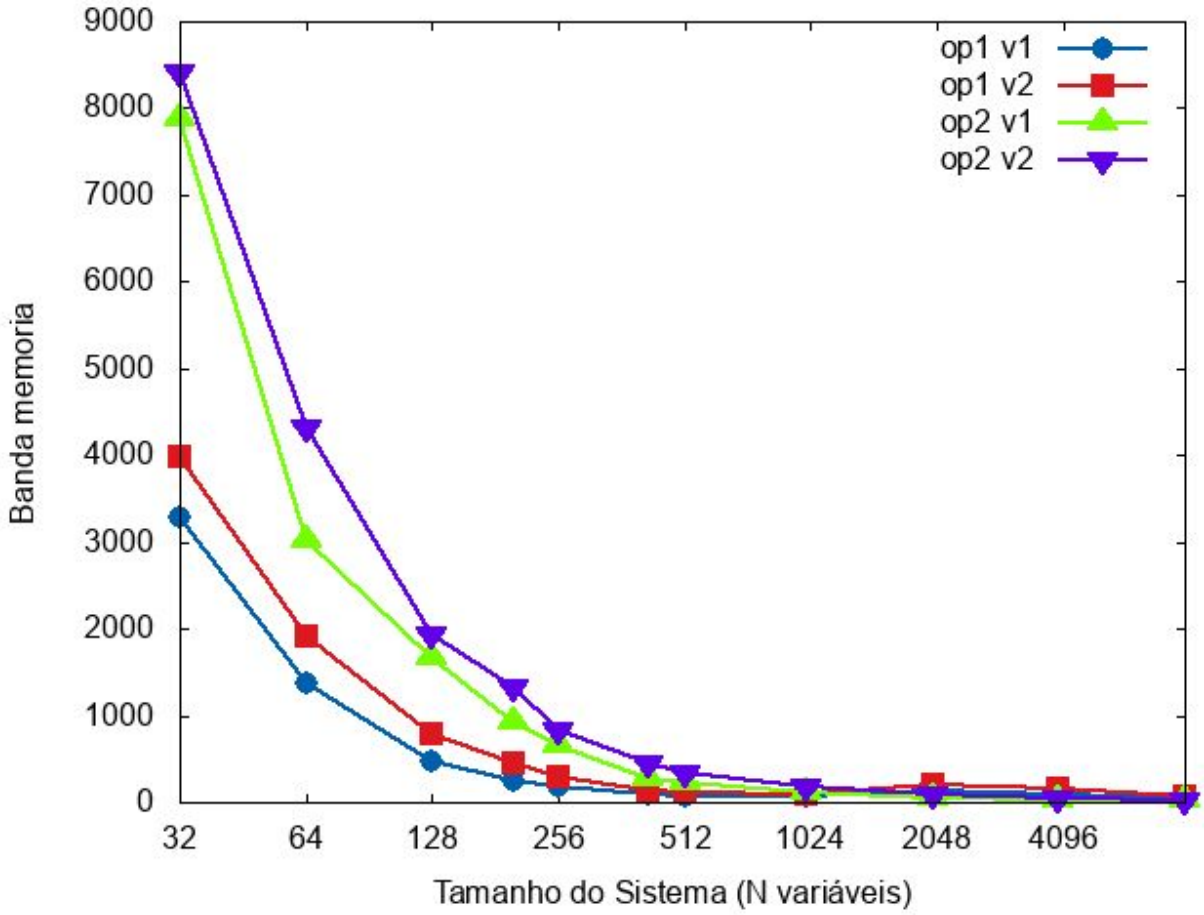
- Tempo de Execução:

  Foi observada uma pequena melhora no tempo, principalmente no cálculo do resíduo. Contudo, o ganho não foi significativo. As estratégias de *Unroll and Jam* e de *Loop Fusion* não surtiram tanto efeito quanto era esperado. Podemos observar isso no gráfico abaixo:



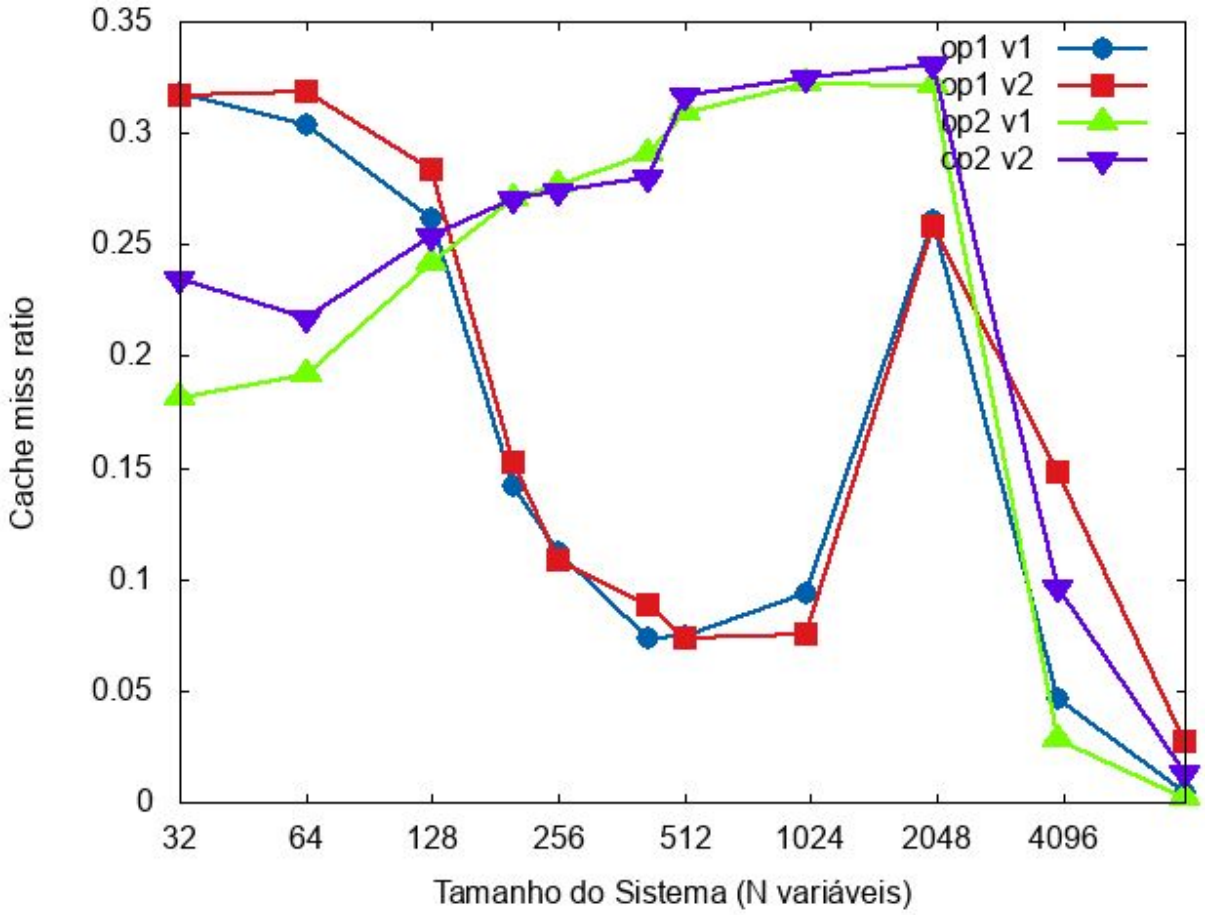
- Banda de Memória:

A banda de memória medida em **L3** foi consistentemente maior em **v2** em relação a **v1**, conforme era esperado pela otimização do uso dos dados. Desde o início, foi buscado um uso eficiente dos dados, evitando assim acessos desnecessários à memória. Segue o gráfico abaixo:



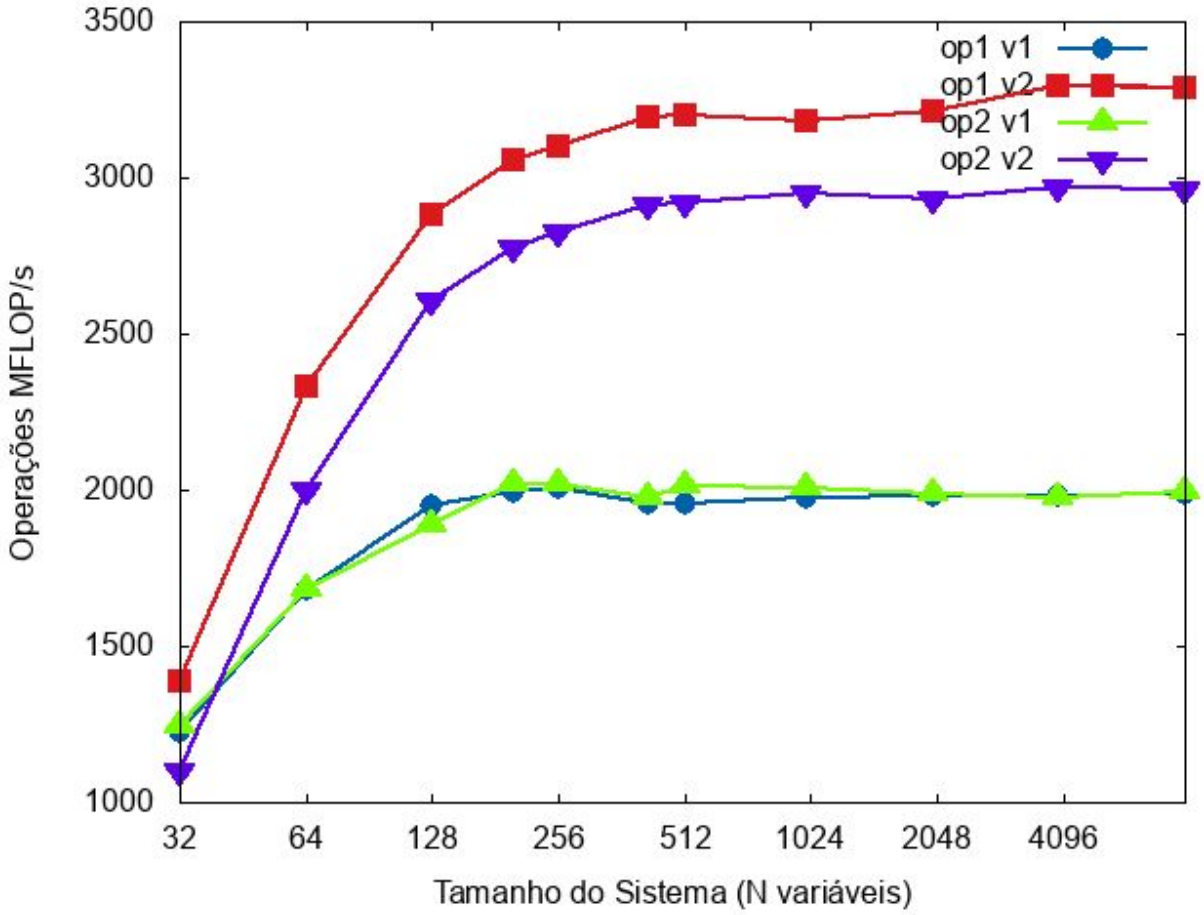
- Cache miss em **L2**

O cache miss ratio variou bastante, com **v2** mostrando desempenho às vezes melhor e às vezes pior que **v1.** É possível perceber uma piora enorme a partir de n = 1024 no caso da operação de iteração do método, o que indica que o limite da cache **L2** de ter sido excedido. No entanto, nas duas operações avaliadas e em ambas versões do código, ocorre uma melhora significativa após n = 2048, o que indica a utilização de **L3** a partir desse valor. Segue o gráfico abaixo:



- Desempenho em MFLOP/s

Esse foi o setor de maior melhora na otimização. Os resultados em **v2** foram extremamente melhores do que os vistos em **v1**. Isso indica que as estratégias de *Unroll and Jam* e de *Loop Fusion* surtiram um efeito enorme em um ganho de desempenho em operações de pontos flutuantes double. Outro aspecto importante foi o alinhamento da memória com *aligned_alloc( )* , o que possibilita vetorização e uso de instruções **SIMD** por parte do compilador. O ganho de desempenho pode ser visto abaixo no gráfico:



Principais alterações

Como mencionado anteriormente, foi feito *Unroll and Jam* e *Loop Fusion* nos laços mais críticos dentro das operações em que se buscava ganho de desempenho. Além disso, toda a memória foi alocada com alinhamento. O fator utilizado para unroll foi 7, já que assim eram utilizadas as linhas da cache sem exceder o seu limite. Foram testados alguns valores para o STRIDE e para o alinhamento, até encontrar aqueles que proporcionaram os melhores resultados.

Um fator muito importante que também foi mudado foi a estrutura de dados. Mesmo armazenando apenas as diagonais não nulas desde o início, em **v1** o vetor armazenava em linha uma diagonal após a outra, começando pela inferior esquerda e seguindo até a superior direita. Isso tornava o acesso à estrutura de dados, ao percorrer a matriz em linha, completamente caótico e sem continuidade espacial. Para resolver isso, foi feito o armazenamento linha por linha, permitindo em **v2** mais localidade espacial e temporal no acesso.

Considerações finais

Foi necessário profundo conhecimento sobre o código, o método e a arquitetura do processador para trabalhar em cima de melhorias. Mesmo assim, os resultados obtidos ainda estão muito aquém do potencial mostrado pelas estimativas. O uso de **AVX** poderia ter aumentado ainda mais o desempenho. De qualquer forma, o próprio problema em questão oferecia muitas limitações por conta do seu comportamento.

Outra estratégia com muito potencial que poderia ser adotada é a implementação de paralelismo no código. A multiplicação de matriz por vetor é altamente paralelizável, já que o vetor resultante tem cada índice dependendo apenas de uma linha da matriz.

Finalmente, a análise poderia ter sido mais rica com um espaço amostral maior (aumentando o intervalo e a quantidade de valores de n) e a otimização poderia ter sido mais eficiente se tivesse sido atacado o trecho mais caro do código (a preparação da matriz e do vetor para o método). De qualquer forma, o trabalho proporcionou uma experiência muito rica de aprendizado tanto para a análise de desempenho quanto para implementações futuras melhores.