PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

Luiz Fernando Oliveira Maciel[1]

Resumo

Este trabalho consiste na análise e consequente otimização de um código que opera sobre uma matriz n x m. Nos tópicos a seguir, serão analisados tanto o código original, quando cada melhoria realizada, o speedup ocasionado por esta melhoria, e comentários a respeito dessas mudanças.

Palavras-chave: Projeto e Análise de Algoritmos. C. Matrizes.

[1]Aluno de Ciência da Computação, Brasil – lfomaciel@sga.pucminas.br.

1 REPOSITÓRIO

Os arquivos citados estão disponíveis no link: <https://github.com/lfnand0/Trab1-PAA>. Também estão inclusos no .zip da entrega, na pasta "codigo", juntamente com este pdf.

2 DESCRIÇÃO DO SISTEMA

2.1 Compilação

Os códigos utilizados foram feitos em C e compilados utilizando o GCC 12.2.1. Além da flag -lm necessária na compilação por causa da biblioteca math.h, nenhuma outra opção adicional de otimização foi usada.

2.2 Hardware

CPU: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.100GHz

GPU: AMD ATI Radeon Vega Series / Radeon Vega Mobile Series

Memória: 9885MiB

2.3 Sistema Operacional

O Sistema Operacional usado no trabalho foi o Linux (kernel 6.2.5-arch1-1).

3 GERADOR DE MATRIZES

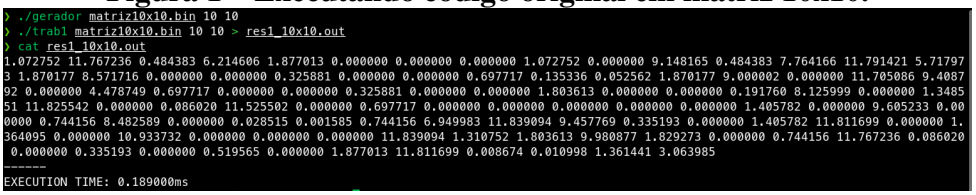
O arquivo responsável pela geração de matrizes é o "gerador.c". Ele recebe 3 argumentos na execução, o nome do arquivo para output, o número de linhas e o número de colunas respectivamente.

4 ANÁLISE DO PROGRAMA ORIGINAL

4.1 Teste inicial

Utilizando o gerador, foi criada uma matriz 10 x 10 para testar o programa original. A matriz está salva no diretório como "matriz10x10.bin". O tempo de execução mostrado em baixo dos resultados é uma adição ao programa - o código fornecido pelo professor não inclui essa parte. Os resultados obtidos foram salvos no arquivo "res1_10x10.out"

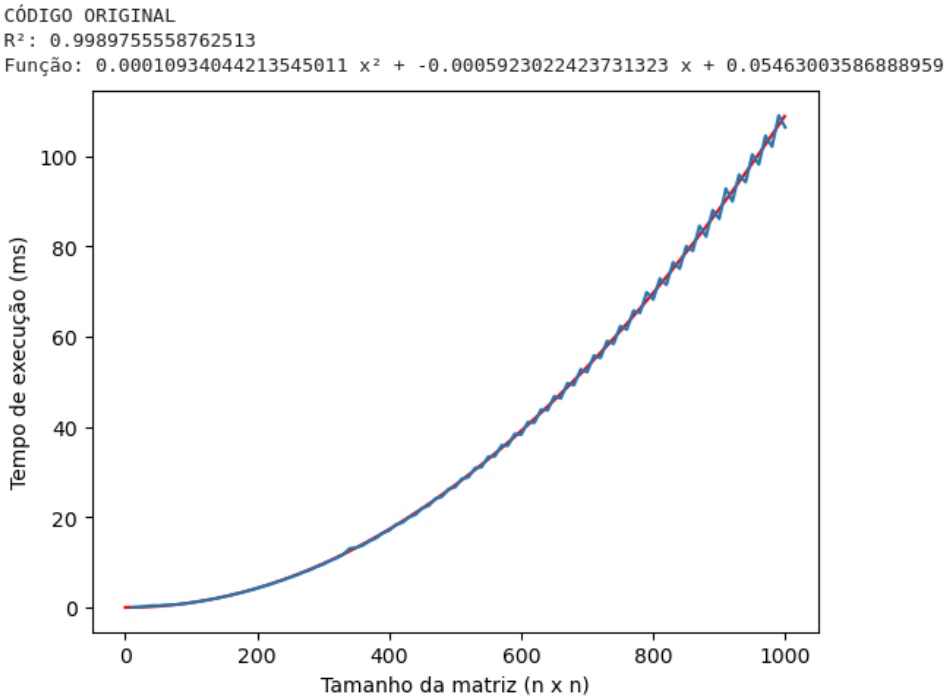
Figura 1 – Executando código original em matriz 10x10.



4.2 Teste em matrizes maiores

Para essa etapa, um programa alterado foi feito: "media_trab1.c". A única diferença dele para o original é que ele recebe dois limites a e b na chamada e gera 5 matrizes de tamanhos incrementais, de a x a até b x b, aumentando de 10 em 10 (ex.: 10x10 , 20x20, 30x30, ... , b x b), e retorna o tempo de execução médio para cada tamanho. Esse programa foi utilizado para encontrar o tempo de execução para matrizes de 10x10 até 1000x1000 elementos (resultados salvos no arquivo "res_media_original.out") - a seguir encontra-se um gráfico com os resultados obtidos (curva azul), além de uma função de tempo aproximada calculada com regressão polinomial (curva vermelha) e o quoeficiente R^2 (que representa o quão próxima a função é dos valores verdadeiros):

Figura 2 – Tempo de execução médio do programa original



4.3 Complexidade

Observando o código, podemos chegar em uma função aproximada de custo f(n, m) = 13*n*m + 512 , para uma matriz de n linhas e m colunas. Com isso, podemos concluir que a complexidade do programa é O(n*m).

5 MELHORIAS

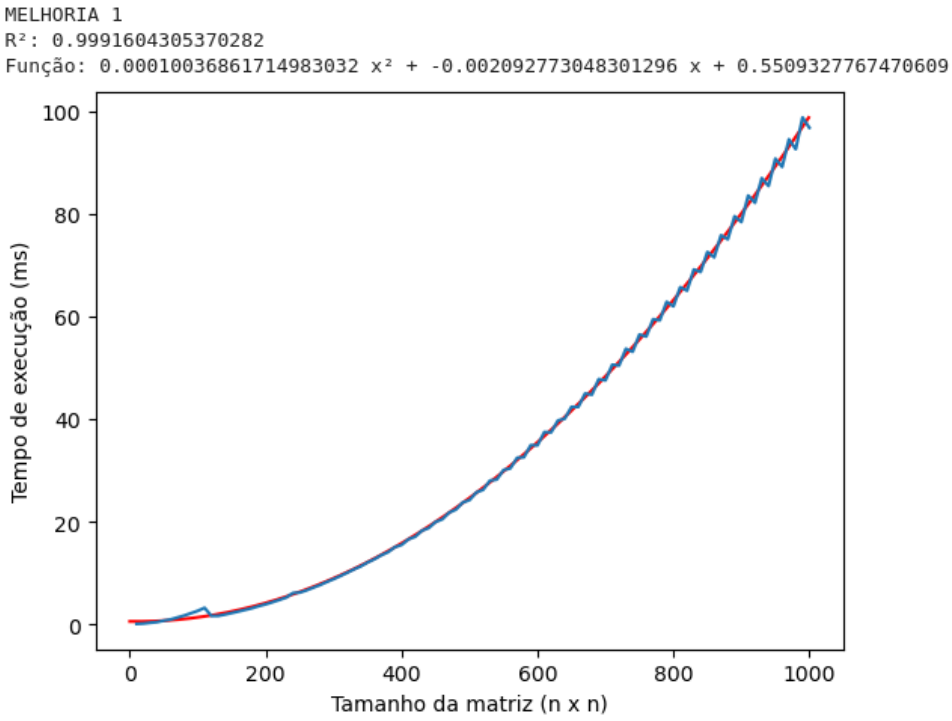
5.1 Melhoria 1

O programa original gera, para cada elemento da matriz, dois valores diferentes, "out_even"(para elementos pares) e "out_odd"(para elementos ímpares), e depois checa se o elemento é par ou ímpar e salva o que foi calculado em um array. Essa melhoria consiste em, ao invés de calcular tanto "out_even"quanto "out_odd"e depois salvar o valor, checar anteriormente se o número é par ou ímpar e fazer apenas um cálculo. O arquivo com o programa que implementa essa melhoria é o "melhoria1.c".

5.1.1 Tempo de Execução Médio

A seguir encontra-se o gráfico do tempo de execução médio do programa para matrizes de tamanhos diferentes:

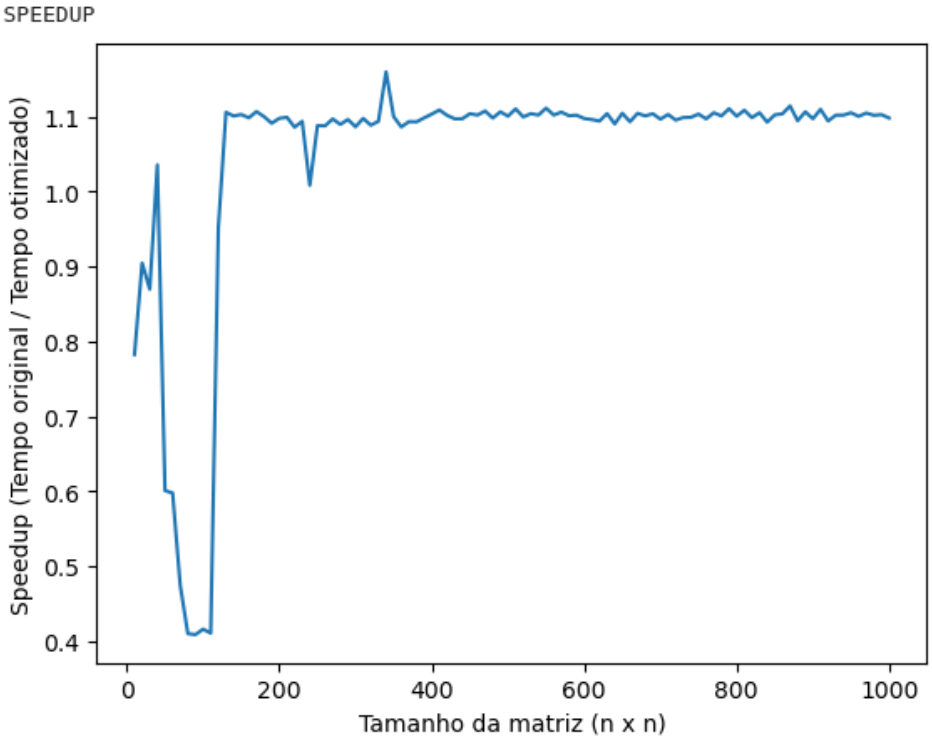
Figura 3 – Tempo de execução médio - melhoria 1



5.1.2 Speedup

O seguinte gráfico representa o speedup do programa melhorado em relação ao programa original, para tamanhos de matrizes diferentes.

Figura 4 – Speedup da melhoria 1




```
Speedup médio:  1.0459454077638772
Speedup matriz 1000x1000:  1.097862718374427
```

O speedup médio foi de aprox. 4,5945%, e o speedup para uma matriz 1000x1000 de aprox. 9,7863%. Podemos observar no gráfico que o comportamento para matrizes pequenas é pouco previsível, e que se torna mais estável ao aumentarmos o tamanho.

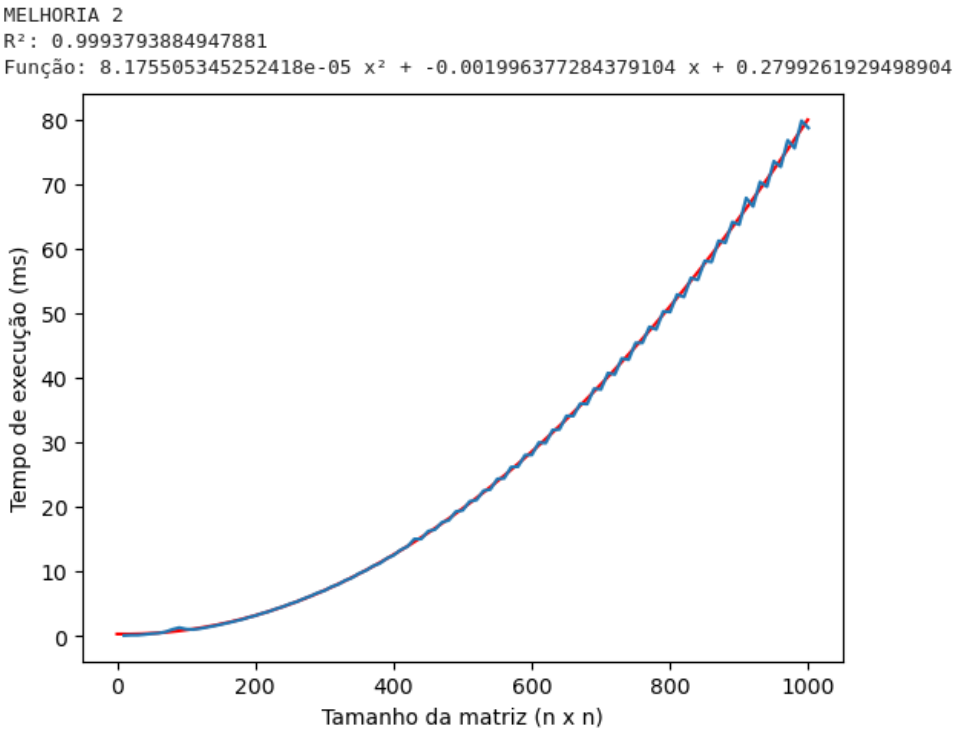
5.2 Melhoria 2

Essa melhoria está relacionada ao array que conta a quantidade de cada elemento na matriz. Originalmente, se era feita a leitura da matriz em um loop, e depois um outro loop iterava sobre a matriz incrementando os contadores. A segunda melhoria consiste em incrementar os contadores após a leitura do elemento, evitando acessos repetidos à memória.

5.2.1 Tempo de Execução Médio

A seguir encontra-se o gráfico do tempo de execução médio do programa para matrizes de tamanhos diferentes:

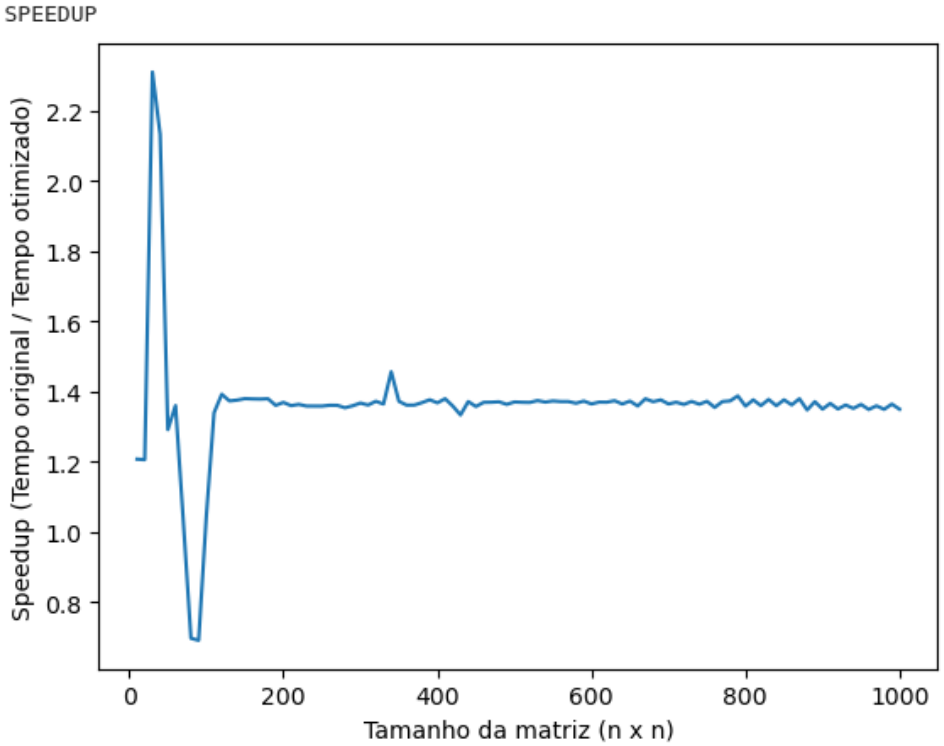
Figura 5 – Tempo de execução médio - melhoria 2



5.2.2 Speedup

O seguinte gráfico representa o speedup do programa melhorado em relação ao programa original, para tamanhos de matrizes diferentes.

Figura 6 – Speedup da melhoria 2




```
Speedup médio:  1.3604647830899785
Speedup matriz 1000x1000:  1.349878507354304
```

O speedup médio foi de aprox. 36,0465%, e o speedup para uma matriz 1000x1000 de aprox. 34,9879%. Houve um aumento de aprox. 31,4520% do speedup médio ao adicionarmos a melhoria 2.

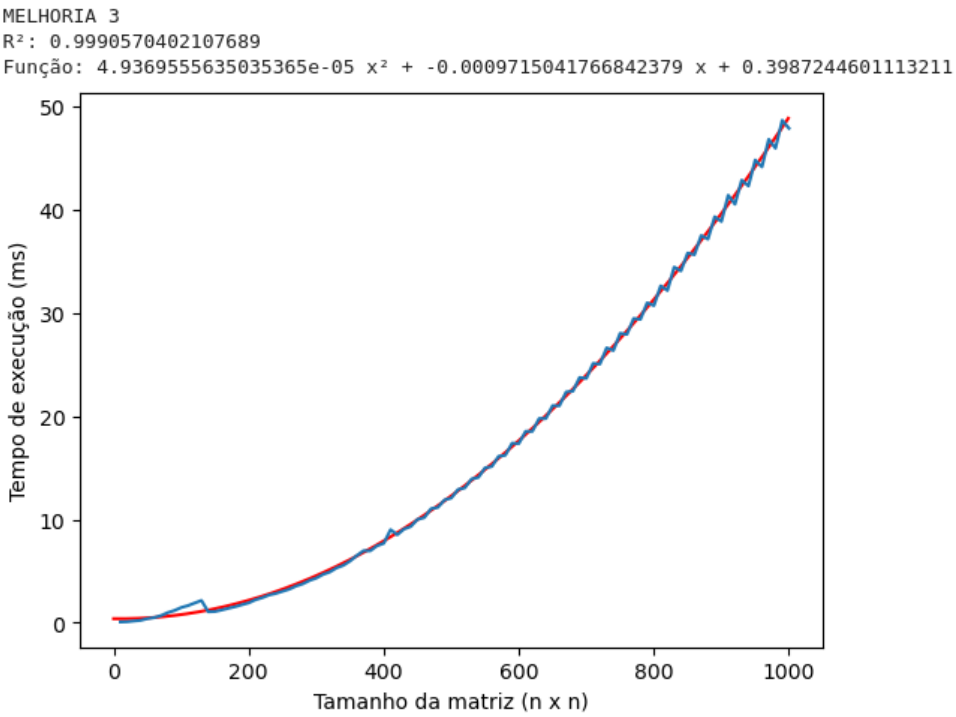
5.3 Melhoria 3

A terceira melhoria implementa um array com os valores do seno e cosseno de 0 a 255º. A probabilidade de existirem dois elementos iguais em um conjunto de 58 números aleatórios entre 0 e 255 é maior que 99% (aproximação obtida com a expansão da equação do Paradoxo do Aniversário utilizando a Série de Taylor) - portanto, para matrizes com mais de 58 elementos, podemos presumir que algum/alguns dos valores de seno e cosseno utilizados na função DetOutput provavelmente iriam ser calculados mais de uma vez. Essa melhoria apenas é aplicada, portanto, quando o número de elementos na matriz for maior que 58 (a função chamada é a DetOutputLargeMatrix nesse caso).

5.3.1 Tempo de Execução Médio

A seguir encontra-se o gráfico do tempo de execução médio do programa para matrizes de tamanhos diferentes:

Figura 7 – Tempo de execução médio - melhoria 3

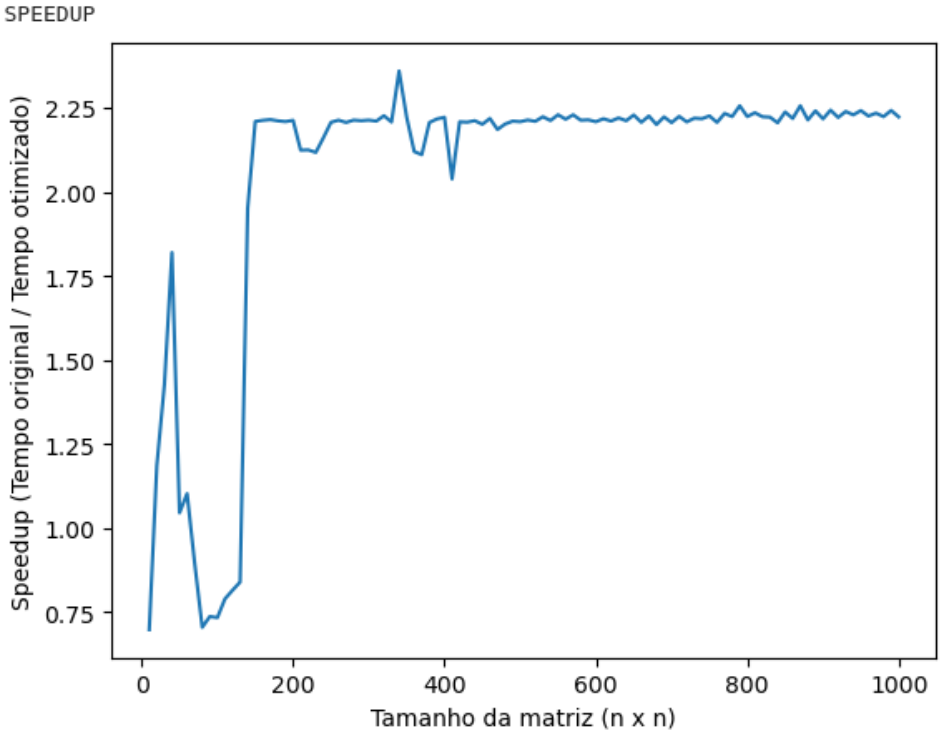


5.3.2 Speedup

O seguinte gráfico representa o speedup do programa melhorado em relação ao programa original, para tamanhos de matrizes diferentes.

O speedup médio foi de aprox. 104,9768%, e o speedup para uma matriz 1000x1000 de aprox. 122,3272%. Houve um aumento de aprox 69,9303% do speedup médio ao adicionarmos a melhoria 3. Essa melhoria é uma solução de compromisso, já que é necessário armazenar os valores calculados em memória - porém a quantidade de memória é ínfima comparada ao que o programa já gasta, e o grande speedup implica um ótimo custo/benefício.

Figura 8 – Speedup da melhoria 3




```
Speedup médio:  2.0497676110423364
Speedup matriz 1000x1000:  2.223271860494292
```

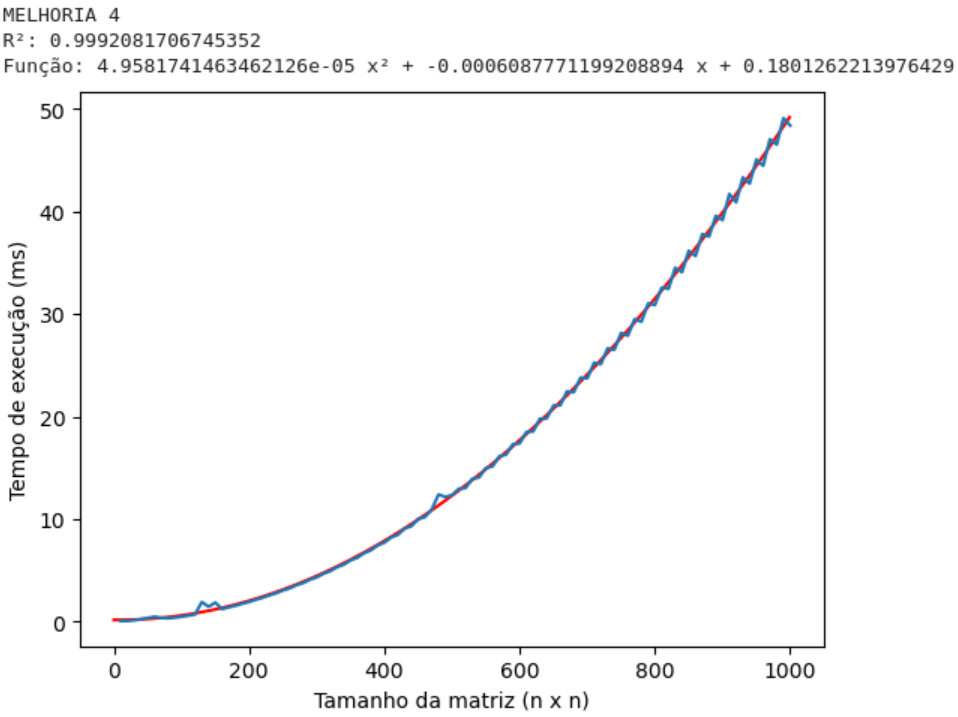
5.4 Melhoria 4

A quarta melhoria consiste na mudança de uma operação aritmética para uma lógica (operações lógicas são executadas com muito mais facilidade no processador). Ao invés de utilizarmos a operação de módulo para ver se um elemento é par ou ímpar (funções DetOutput e DetOutputLargeMatrix), podemos realizar um "and"() entre o elemento e o número 1 - essa operação cria uma "máscara"que retorna apenas o bit menos significante do elemento, se esse bit for 0 o elemento é par e, se ele for 1, ímpar.

5.4.1 Tempo de Execução Médio

A seguir encontra-se o gráfico do tempo de execução médio do programa para matrizes de tamanhos diferentes:

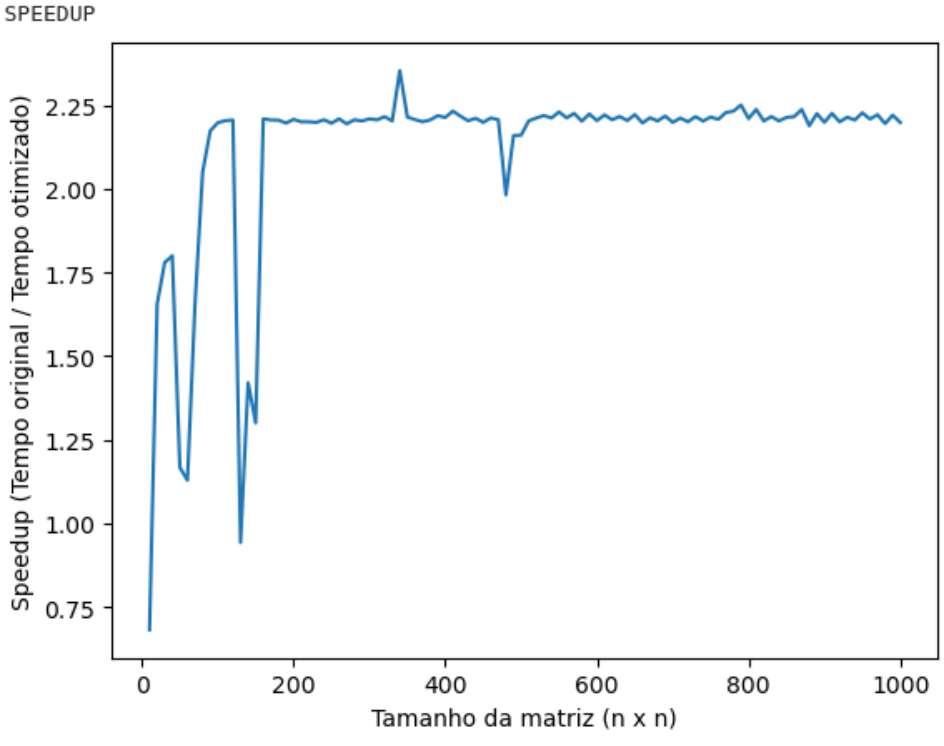
Figura 9 – Tempo de execução médio - melhoria 4



5.4.2 Speedup

O seguinte gráfico representa o speedup do programa melhorado em relação ao programa original, para tamanhos de matrizes diferentes.

Figura 10 – Speedup da melhoria 4



O speedup médio foi de aprox. 112,1649%, e o speedup para uma matriz 1000x1000 de aprox. 119,9095%. Houve um aumento de aprox 7,1881% do speedup médio ao adicionarmos a melhoria 4.

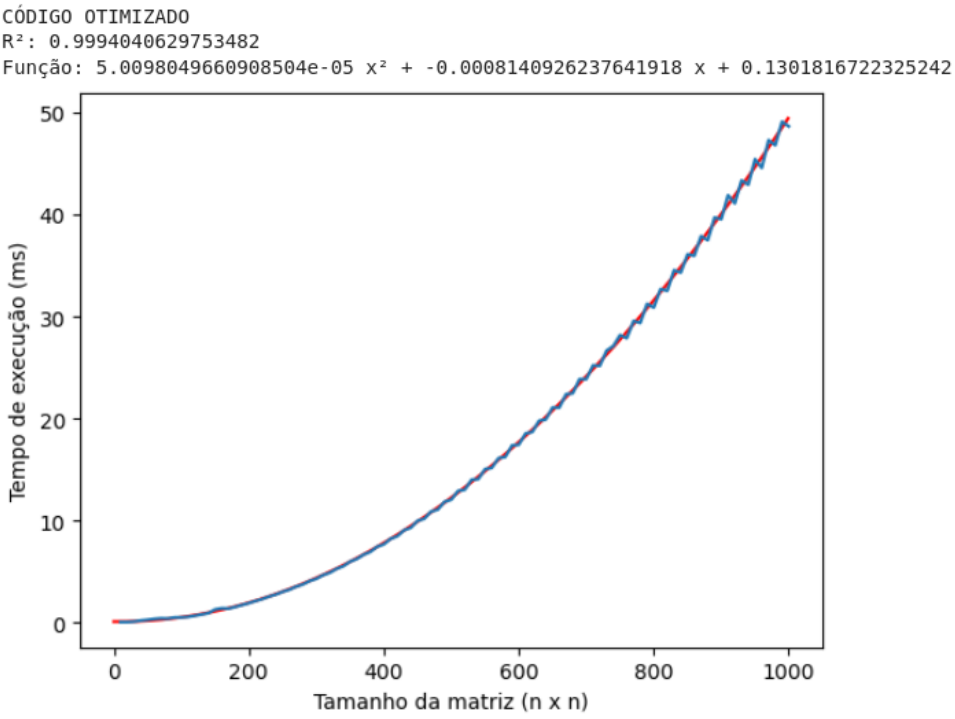
5.5 Melhoria 5 (Final)

Uma operação é executada sobre o array de contadores: caso a quantidade de um elemento seja maior que 0, o valor no array de contadores é alterado para o logaritmo da quantidade. Originalmente, essa operação é feita sobre todos os elementos do array - entretanto, o valor presente no array apenas é utilizado quando estamos determinando o output de um elemento ímpar. Dessa forma, podemos calcular o logaritmo apenas para as posições ímpares do array.

5.5.1 Tempo de Execução Médio

A seguir encontra-se o gráfico do tempo de execução médio do programa para matrizes de tamanhos diferentes:

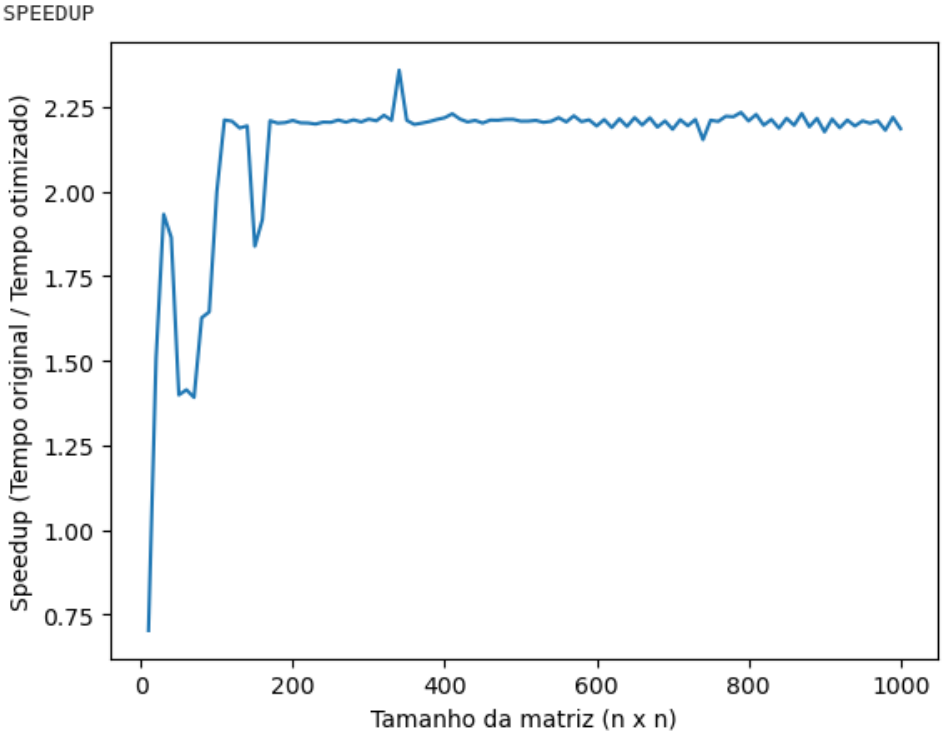
Figura 11 – Tempo de execução médio - código final



5.5.2 Speedup

O seguinte gráfico representa o speedup do programa melhorado em relação ao programa original, para tamanhos de matrizes diferentes.

Figura 12 – Speedup do código final




```
Speedup médio:  2.1361139839039245
Speedup matriz 1000x1000:  2.18636163624176
```

O speedup médio foi de aprox. 113,6114%, e o speedup para uma matriz 1000x1000 de aprox. 118,6362%. Houve um aumento de aprox 1,4465% do speedup médio ao adicionarmos a melhoria 5.

6 CONCLUSÃO

O código otimizado com as 5 melhorias apresentou tempos de execução mais que 2 vezes melhores que do original. Apesar disso, a complexidade permaneceu a mesma: para matrizes com menos de 59 elementos, as funções de custo para o melhor, o médio e o pior caso são, respectivamente, $f1(n,m) = 7*n*m+384$, $f2(n,m) = 7,5*n*m+384$, $f3(n,m) = 8*n*m+384$; já para matrizes com mais de 59 elementos, $f1(n,m) = 6*n*m+1152$, $f2(n,m) = 6,5*n*m+1152$, $f3(n,m) = 7*n*m+1152$. O pior caso é quando todos os elementos forem pares, e o melhor quando todos forem ímpares. A seguir encontra-se a tela monstrando que o output do código otimizado e do código original foram iguais:

```
❯ ./trab1 matriz10x10.bin 10 10 > trab1.out
❯ gcc -o otimizado otimizado.c -lm
❯ ./otimizado arq.bin 1000 1000 > pub2.out
❯ ./otimizado matriz10x10.bin 10 10 > otimizado.out
❯ diff -i trab1.out otimizado.out
3c3
< EXECUTION TIME: 0.149000ms
---
> EXECUTION TIME: 0.121000ms
❯ cat otimizado.out
1.072752 11.767236 0.484383 6.214606 1.877013 0.000000 0.000000 0.000000 1.072752 0.000000 9.14
8165 0.484383 7.764166 11.791421 5.717973 1.870177 8.571716 0.000000 0.000000 0.325881 0.000000
 0.000000 0.697717 0.135336 0.052562 1.870177 9.000002 0.000000 11.705086 9.408792 0.000000 4.4
78749 0.697717 0.000000 0.000000 0.325881 0.000000 0.000000 1.803613 0.000000 0.000000 0.191760
 8.125999 0.000000 1.348551 11.825542 0.000000 0.086020 11.525502 0.000000 0.697717 0.000000 0.
000000 0.000000 0.000000 0.000000 1.405782 0.000000 9.605233 0.000000 0.744156 8.482589 0.00000
0 0.028515 0.001585 0.744156 6.949983 11.839094 9.457769 0.335193 0.000000 1.405782 11.811699 0
.000000 1.364095 0.000000 10.933732 0.000000 0.000000 0.000000 11.839094 1.310752 1.803613 9.98
0877 1.829273 0.000000 0.744156 11.767236 0.086020 0.000000 0.335193 0.000000 0.519565 0.000000
 1.877013 11.811699 0.008674 0.010998 1.361441 3.063985
------
EXECUTION TIME: 0.121000ms
❯ cat trab1.out
1.072752 11.767236 0.484383 6.214606 1.877013 0.000000 0.000000 0.000000 1.072752 0.000000 9.14
8165 0.484383 7.764166 11.791421 5.717973 1.870177 8.571716 0.000000 0.000000 0.325881 0.000000
 0.000000 0.697717 0.135336 0.052562 1.870177 9.000002 0.000000 11.705086 9.408792 0.000000 4.4
78749 0.697717 0.000000 0.000000 0.325881 0.000000 0.000000 1.803613 0.000000 0.000000 0.191760
 8.125999 0.000000 1.348551 11.825542 0.000000 0.086020 11.525502 0.000000 0.697717 0.000000 0.
000000 0.000000 0.000000 0.000000 1.405782 0.000000 9.605233 0.000000 0.744156 8.482589 0.00000
0 0.028515 0.001585 0.744156 6.949983 11.839094 9.457769 0.335193 0.000000 1.405782 11.811699 0
.000000 1.364095 0.000000 10.933732 0.000000 0.000000 0.000000 11.839094 1.310752 1.803613 9.98
0877 1.829273 0.000000 0.744156 11.767236 0.086020 0.000000 0.335193 0.000000 0.519565 0.000000
 1.877013 11.811699 0.008674 0.010998 1.361441 3.063985
------
EXECUTION TIME: 0.149000ms
```