UNIVERSIDADE FEDERAL DO PARANÁ

GABRIEL MARCZUK THÁ

JOÃO PEDRO PICOLO

# OTIMIZAÇÃO DE FUNÇÕES: TRIANGULARIZAÇÃO E CÁLCULO DE COEFICIENTES

CURITIBA - PR

2021

GABRIEL MARCZUK THÁ
JOÃO PEDRO PICOLO

# OTIMIZAÇÃO DE FUNÇÕES: TRIANGULARIZAÇÃO E CÁLCULO DE COEFICIENTES

Relatório apresentado ao curso de Graduação em Bacharelado da Ciência da Computação, Setor de Exatas, Universidade Federal do Paraná, como requisito à obtenção parcial da aprovação na disciplina de Iniciação à Computação Científica.

Orientador: Prof. Dr. Guilherme Alex Derenievicz

CURITIBA - PR
2021

# SUMÁRIO

# 1 INTRODUÇÃO

Este trabalho tem por objetivo descrever a otimização realizada no código implementado sobre a função de **cálculo de coeficientes** de um sistema linear utilizado no Ajuste de Curvas para uma determinada entrada e sobre a função de **triangularização** aplicada sob este mesmo sistema a fim de realizar a fatoração LU.

No Capítulo 2 deste trabalho iremos discorrer brevemente sobre a arquitetura do computador utilizado para a realização dos testes bem como o passo a passo necessário para a execução dos mesmos. Em seguida, no Capítulo 3 iremos discorrer sobre cada otimização aplicada às funções analisadas no escopo deste trabalho e por fim, no Capítulo 4, utilizaremos uma análise gráfica como forma de nos auxiliar a entender o impacto das mudanças propostas.

# 2 METODOLOGIA

Neste capítulo descreveremos brevemente a arquitetura do computador utilizado para os testes bem como o passo a passo para efetuá-los.

## 2.1 Descrição do Computador Utilizado

Os testes foram realizados de forma remota no computador i27 disponibilizado pelo Departamento de Informática da Universidade Federal do Paraná, cuja arquitetura pode ser acessada ao utilizar o comando **likwid-topology -g -c**. A saída deste comando é transcrita abaixo.

```
--------------------------------------------------------------------------------
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:             5666.48 MB
Total memory:            7858 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                     | |
| +---------------------------------------------+ |
+-------------------------------------------------+
```

## 2.2 Experimentos Realizados

Foi utilizado um *script* em *bash* para obter o tempo de execução das funções ***preencheSistemaAjuste*** e ***triangularizacaoSistema*** bem como de suas versões

otimizadas. Para isso, é usada a função *timestamp(),* provida pelos docentes da disciplina, antes e depois de cada função, de forma a obter o tempo exato de sua execução. Em seguida, o *script* utiliza o *likwid-perfctr* para captar as informações sobre o uso dos grupos L3 (*Memory bandwidth* [MBytes/s]), L2CACHE (*data cache miss ratio*), FLOPS_DP e FLOPS_AVX (MFLOP/s). O *script* insere todas as informações em um arquivo de extensão **.txt** que é utilizado para gerar os gráficos de desempenho.

Para executar os testes da versão otimizada do código basta executar o *script* com os seguintes comandos:

**chmod +x script.sh**
**./script.sh**

A saída do *script* será para o arquivo “análise_de_desempenho.txt” que irá conter um compilado de todas as informações requeridas.

Para executar a versão sem otimização, é necessário alterar a função chamada no código nas linhas 141 e 147 do arquivo “tabela.c”.

Linha 141: **preencheSistemaAjusteOtimizada** para **preencheSistemaAjuste**
Linha 147: **triangularizacaoSistemaOtimizada** para **triangularizacaoSistema**

Após isso, basta compilar o código novamente com o comando ***make*** e então repetir o processo do *script*.

# 3 OTIMIZAÇÕES

Este capítulo será dividido em dois subcapítulos nos quais descreveremos as otimizações realizadas sobre as funções propostas. Primeiramente no Subcapítulo 3.1 discutiremos as otimizações realizadas na função de cálculo de coeficientes do sistema linear utilizado no Ajuste de Curvas e no Subcapítulo 3.2 discutiremos as otimizações realizadas na função de triangularização do sistema linear proposto.

## 3.1 Otimizações Sobre o Cálculo de Coeficientes

Nas seguintes seções descreveremos as otimizações feitas sobre o trecho de código disponibilizado na **Figura 1**, correspondente a função de cálculo de coeficientes sem otimização.

```
91  void preencheSistemaAjuste(Sistema_t *sistema, real_t *valoresTabelados, real_t *funcoesTabeladas) {
92      unsigned int n = sistema->n;
93
94      for(int i = 0; i < n; i++) {
95          real_t soma = 0.0;
96          for(int j = 0; j < n; j ++) {
97              soma += funcoesTabeladas[j] * pow(valoresTabelados[j], i);
98          }
99          sistema->b[i] = soma;
100     }
101
102     // Monta a primeira linha
103     for(int i = 0; i < n; i++) {
104         // Cuida do somatório
105         real_t soma = 0.0;
106         for(int j = 0; j < n; j++) {
107             soma += pow(valoresTabelados[j], i);
108         }
109
110         sistema->U[i] = soma;
111     }
112
113     // Monta linhas seguintes
114     // Só precisa calcular o último valor, o resto repete da anterior
115     for(int i = 1; i < n; i++) {
116         // Preenche até n - 1 colunas
117         // Preenche com linha anterior e coluna seguinte
118         for(int j = 0; j < n - 1; j++) {
119             sistema->U[i*n + j] = sistema->U[(i-1)*n + (j+1)];
120         }
121
122         // Calcula a última coluna
123         real_t soma = 0.0;
124         for(int j = 0; j < n; j++) {
125             soma += pow(valoresTabelados[j], n-1) * pow(valoresTabelados[j], i);
126         }
127         sistema->U[i*n + (n-1)] = soma;
128     }
129 }
```

Figura 1 - Função não otimizada de cálculo de coeficientes.

### 3.1.1 Alocação de espaço na memória

A primeira otimização realizada está mais ligada ao desenvolvimento do Trabalho 1 do que a função propriamente dita. No primeiro Trabalho a matriz U do sistema linear era obtida através da utilização da técnica de ponteiro para ponteiros de forma a armazenar uma matriz de N x M valores. Neste trabalho optou-se por implementar um único ponteiro apontando para um local de memória de tamanho N x M como forma de otimizar o tempo de alocação da matriz e garantir a proximidade destes valores na memória.

### 3.1.2 Cálculo único de endereços na memória

Esta é a otimização mais simples utilizada, ao invés de calcularmos o endereço de acesso da matriz U na memória como nas linhas 119 e 127 da Figura 1 optou-se por calcular este endereço apenas uma única vez a cada iteração reduzindo o número de operações realizadas pelo processador.

### 3.1.3 Cálculo apenas da última coluna das linhas da matriz

Esta se trata de uma otimização já presente no código sem otimização, mas optou-se por descrevê-la aqui por haver outras formas menos eficientes de realizá-la. Ao observarmos a fórmula para o cálculo da matriz de coeficientes para o sistema linear utilizado no Ajuste de Curvas disponibilizada na Aula 12 da disciplina de Introdução à Computação Científica é possível perceber que a partir da segunda linha da matriz o valor de uma coluna K já foi calculado na coluna K + 1 da linha anterior. Desta forma calculamos apenas a primeira linha da matriz no trecho de código presente entre as linhas 103 e 111 na Figura 1, e reutilizamos nas linhas seguintes os valores calculados. Isto faz necessário que para cada linha seja calculada apenas a última coluna como podemos ver no trecho de código presente entre as linhas 124 e 127 da Figura 1.

### 3.1.4 Cálculo da primeira linha separada do laço

Novamente iremos descrever uma otimização já presente no código não otimizado. Baseado na seção anterior percebemos que é possível calcular a primeira linha dentro do laço presente entre as linhas 115 e 128 da Figura 1, porém para que a reutilização dos valores calculados fosse possível o programador teria que incluir neste laço uma verificação de forma a saber qual linha da matriz está sendo calculada. Desta forma, ao fazermos o cálculo da primeira linha fora do laço evitamos desvios internos que acabariam por interferir negativamente no tempo de execução do código.

### 3.1.5 Fusão de laços independentes

Ao analisarmos o código proposto na Figura 1, pudemos perceber a possibilidade de fusão dos dois primeiros laços da função visto que não há dependência de dados entre eles. Fizemos então esta fusão como forma de calcular uma única vez a função de potência comum aos dois laços e, desta forma, reduzir o número de operações realizadas. Posteriormente fizemos mais uma otimização a esta função que será descrita no subcapítulo seguinte.

### 3.1.6 Cálculo único da potência

Ao unirmos os laços percebemos ainda que o cálculo da potência realizado sobre o valores tabelados era também repetido a cada iteração do cálculo da última coluna das linhas da matriz, como pode ser visto entre as linhas 124 e 126 da Figura 1. Desta forma optamos por criar dentro da função um vetor de tamanho I x J que representa a matriz responsável por armazenar o valor da cada potência I do valor tabelado J. Esta alteração pode ser vista na **Figura 2**.

```
149     real_t *lookupPow = (real_t*) malloc(n * n *sizeof(real_t));
150     for(int i = 0; i < n; i++) {
151         real_t soma_b = 0.0;
152         real_t soma_U = 0.0;
153         for(int j = 0; j < n; j ++) {
154             potencia = pow(valoresTabelados[j], i);
155             lookupPow[(i*n) + j] = potencia;
156             soma_b += funcoesTabeladas[j] * potencia;
157             soma_U += potencia;
158         }
159         sistema->b[i] = soma_b;
160         sistema->U[i] = soma_U;
161     }
```

Figura 2 - Fusão de laços com armazenamento das potências.

Com esta otimização é possível reduzir ainda mais o número de operações realizadas, reduzindo o tempo de execução do código. Embora ganhemos no número de operações realizadas, veremos no Subcapítulo 4.1 também que esta alteração acaba por aumentar o *cache miss ratio*.

### 3.1.7 Reaproveitamento da potência calculada

Podemos notar ainda na linha 154 da Figura 2 que as mesmas potências serão recalculadas a cada iteração de *i*. Dessa forma otimizamos ainda mais este trecho com a finalidade de reaproveitarmos as potências já calculadas.

Para isso, calculamos separadamente a primeira coluna da primeira linha da matriz, o que nos permite colocar o valor 1.0 nas *n* primeiras posições da nossa matriz auxiliar de potências calculadas e reaproveitamos este valor nas colunas seguintes. Podemos observar essa alteração na **Figura 3**.

```
155     // Preenche primeira coluna da primeira linha
156     // assim como o primeiro elemento do vetor de termos independentes
157     real_t soma_b = 0.0;
158     real_t soma_U = 0.0;
159     real_t potencia = 1.0;
160     for(int j = 0; j < n; j ++) {
161         lookupPow[j] = potencia;
162         soma_b += funcoesTabeladas[j];
163         soma_U += potencia;
164     }
165     sistema->b[0] = soma_b;
166     sistema->U[0] = soma_U;
167 
168     // Preenche colunas restantes da primeira linha
169     // assim como os elementos seguintes do vetor de termos independentes
170     for(int i = 1; i < n; i++) {
171         soma_b = 0.0;
172         soma_U = 0.0;
173         for(int j = 0; j < n; j ++) {
174             potencia = lookupPow[(i-1)*n + j] * valoresTabelados[j];
175             lookupPow[(i*n) + j] = potencia;
176             soma_b += funcoesTabeladas[j] * potencia;
177             soma_U += potencia;
178         }
179         sistema->b[i] = soma_b;
180         sistema->U[i] = soma_U;
181     }
```

Figura 3 - Reaproveitamento das potências já calculadas.

Percebemos então que a função *pow()* utilizada anteriormente a cada iteração de *i* tornou-se uma simples multiplicação, conforme vemos na linhas 174 da Figura 3, reduzindo consideravelmente o número de operações a serem executadas pelo processador.

## 3.2 Otimizações Sobre a Triangularização

Nos seguintes subcapítulos descreveremos as otimizações feitas sobre o trecho de código disponibilizado na **Figura 4**, correspondente a triangularização da matriz U utilizada no cálculo da fatoração LU.

```c
int triangularizacaoSistema(Sistema_t *sistema) {
    unsigned int n = sistema->n;

    // Preenche a primeira diagonal de L
    sistema->L[0] = 1.0;

    // Execução Método de Gauss
    for(int i = 0; i < n; ++i) {
        unsigned int iPivo = encontraMax(sistema->U, n, i);
        if(i != iPivo) {
            trocaLinha(sistema, n, i, iPivo);
        }

        for(int k = i + 1; k < n; ++k) {
            real_t denominador = sistema->U[i*n + i];

            real_t m = sistema->U[k*n + i] / denominador;
            sistema->U[k*n + i] = 0.0;

            int tamanho = ceil((k*k + k) / 2);
            sistema->L[tamanho + i] = m;
            sistema->L[tamanho + i + 1] = 1.0;

            for(int j = i + 1; j < n; ++j) {
                sistema->U[k*n + j] -= sistema->U[i*n + j] * m;
            }
        }
    }

    return 0;
}
```

Figura 4 - Função não otimizada de triangularização.

### 3.2.1 Alocação de espaço na memória

A primeira otimização realizada está, novamente, mais ligada ao desenvolvimento do Trabalho 1 do que a função propriamente dita. No primeiro Trabalho as matrizes U e L do sistema linear eram obtidas através da utilização da técnica de ponteiro para ponteiros de forma a armazenar os valores das respectivas matrizes. Neste trabalho optou-se por implementar um único ponteiro apontando para um local diferente e único de memória para cada uma das matrizes como forma de otimizar o tempo de alocação e garantir a proximidade destes valores na memória.

### 3.2.2 Cálculo único de endereços na memória

Esta é a otimização mais simples utilizada, ao invés de calcularmos o endereço de acesso da matriz U na memória como nas linhas 279, 281, 282 e 289 da Figura 4 optou-se por calcular este endereço apenas uma única vez a cada iteração reduzindo o número de operações realizadas pelo processador.

### 3.2.3 Utilização de *Unroll* no acesso a colunas

Por fim, podemos perceber que é possível utilizar a técnica de *loop unroll* no trecho de código presente entre as linhas 288 e 290 da Figura 4. Desta forma optou-se por utilizar um *stride* de tamanho 8 que permite a manipulação de até 8 colunas dentro de uma única iteração do laço fazendo com que o tempo de execução do programa seja reduzido consideravelmente, porém aumentando o número de operações realizadas pelo processador, conforme veremos na análise gráfica do Subcapítulo 4.2.

# 4 ANÁLISE GRÁFICA DAS OTIMIZAÇÕES REALIZADAS

Neste capítulo descreveremos sucintamente os gráficos gerados para cada teste realizado para as funções sob o escopo deste trabalho. Dividiremos ele então em dois subcapítulos, no Subcapítulo 4.1 discutiremos as otimizações sobre a função de cálculo de coeficientes da matriz do sistema linear utilizada no Ajuste de Curvas e no Subcapítulo 4.2 discutiremos as otimizações realizadas sobre a função de triangularização da matriz U utilizada na fatoração LU.

## 4.1 Análise Gráfica do Cálculo de Coeficientes

O primeiro teste realizado diz respeito ao tempo da execução da função, como pode ser visto na **Figura 5**. É possível então ver o impacto positivo das otimizações realizadas sob o tempo de execução da função analisada dado o menor tempo necessário para o cálculo das mesmas operações.

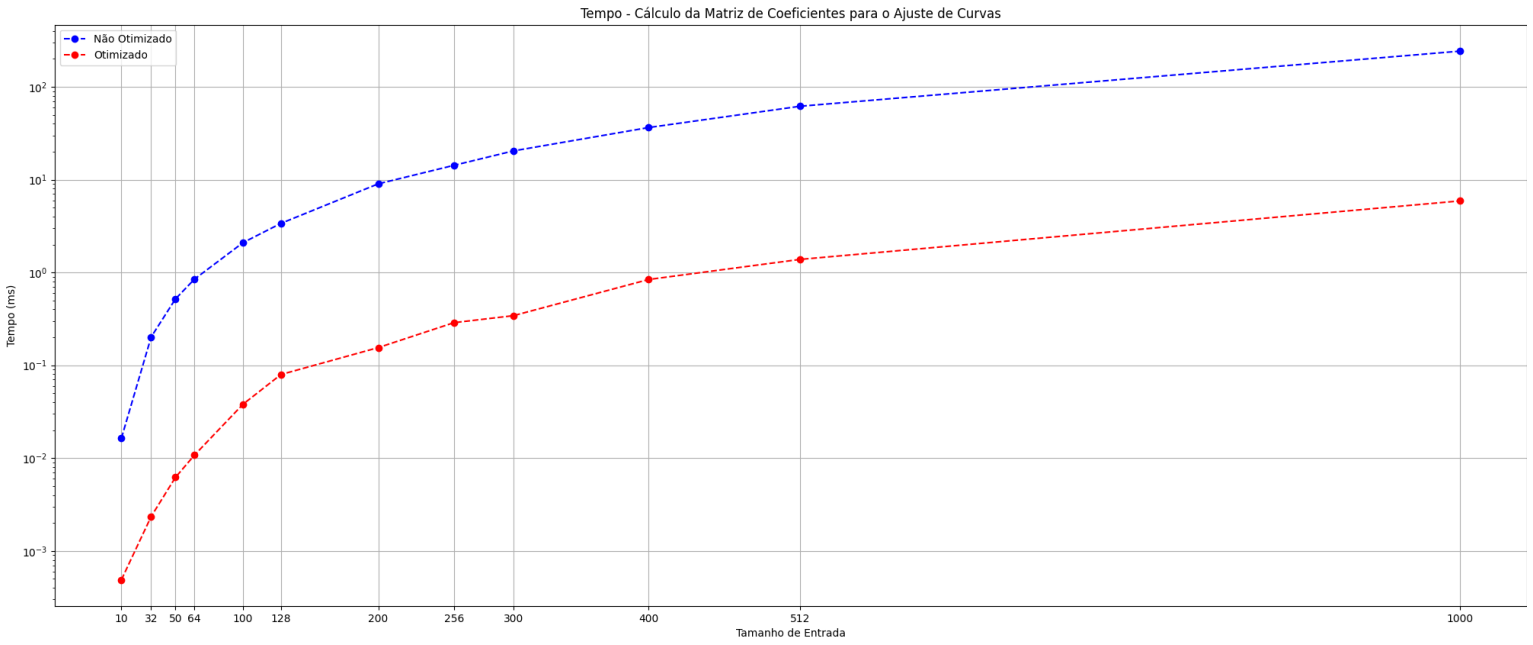


Figura 5 - Gráfico de tempo da função não otimizada e otimizada, em escala logarítmica.

Logo em seguida, na **Figura 6**, verificamos a alteração na largura de banda da memória. É possível então perceber que a capacidade de transmissão dos dados aumentou, o que indica uma melhora na quantidade de dados que podem ser processados.

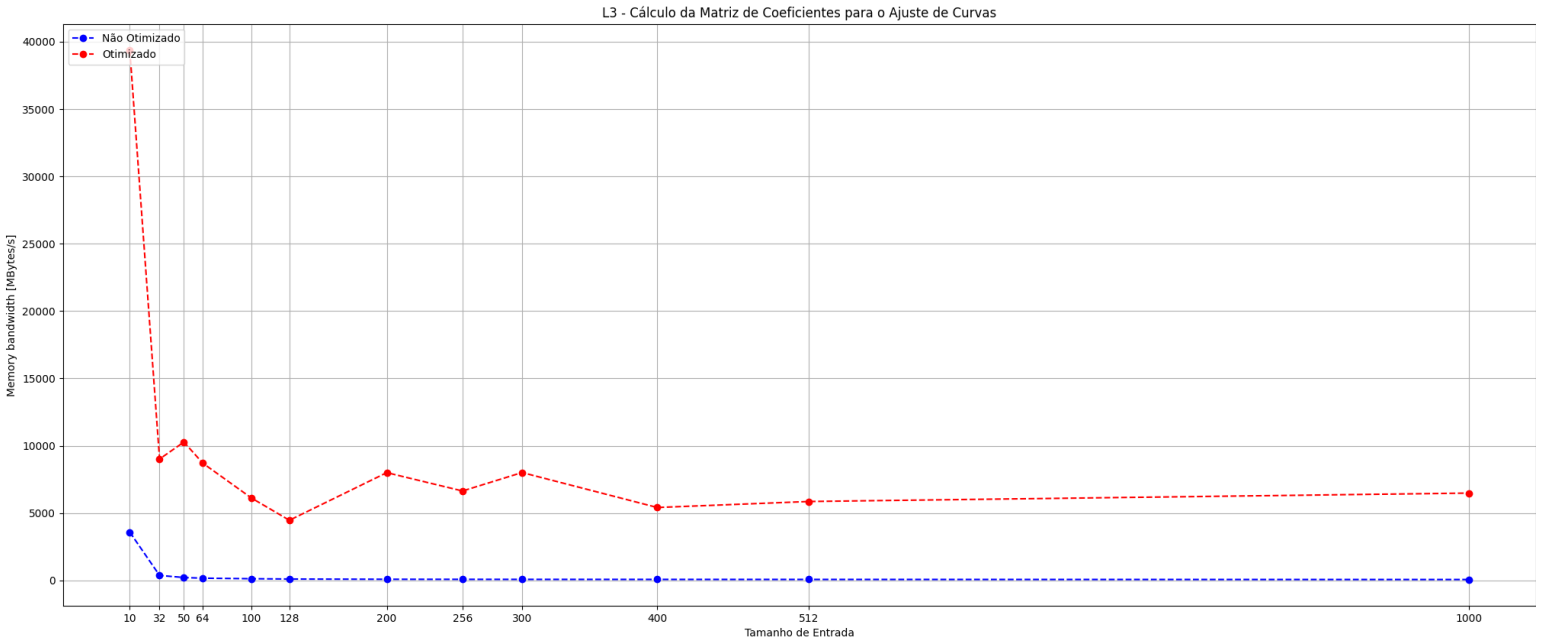


Figura 6 - Gráfico da largura de banda da função não otimizada e otimizada.

Partimos então para a análise do *cache miss ratio*, visível na **Figura 7**. Neste caso é possível perceber que a função otimizada apresenta uma maior quantidade de *cache miss ratio*, provavelmente causada pela implementação da matriz auxiliar de potências descrita no Subcapítulo 3.1.6.

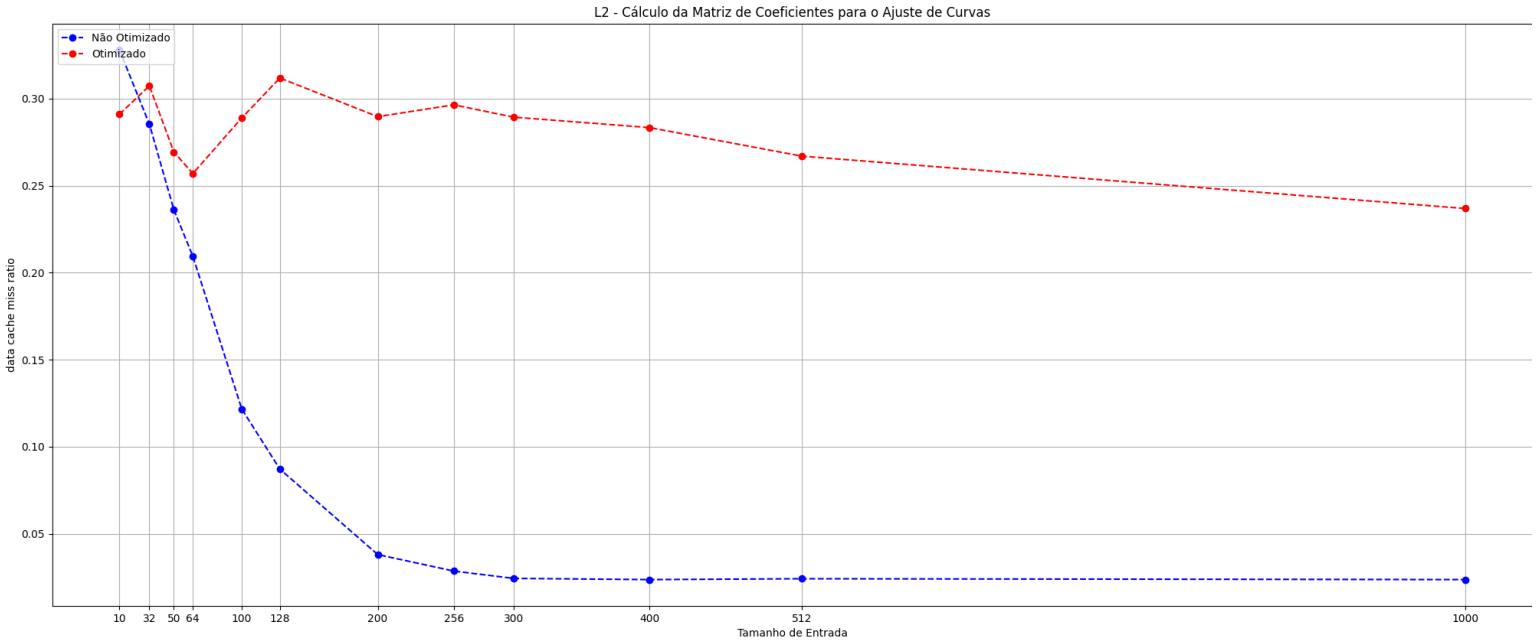


Figura 7 - Gráfico de *cache miss ratio* da função não otimizada e otimizada.

Por último, analisamos as ações aritméticas realizadas, representadas na **Figura 8**.

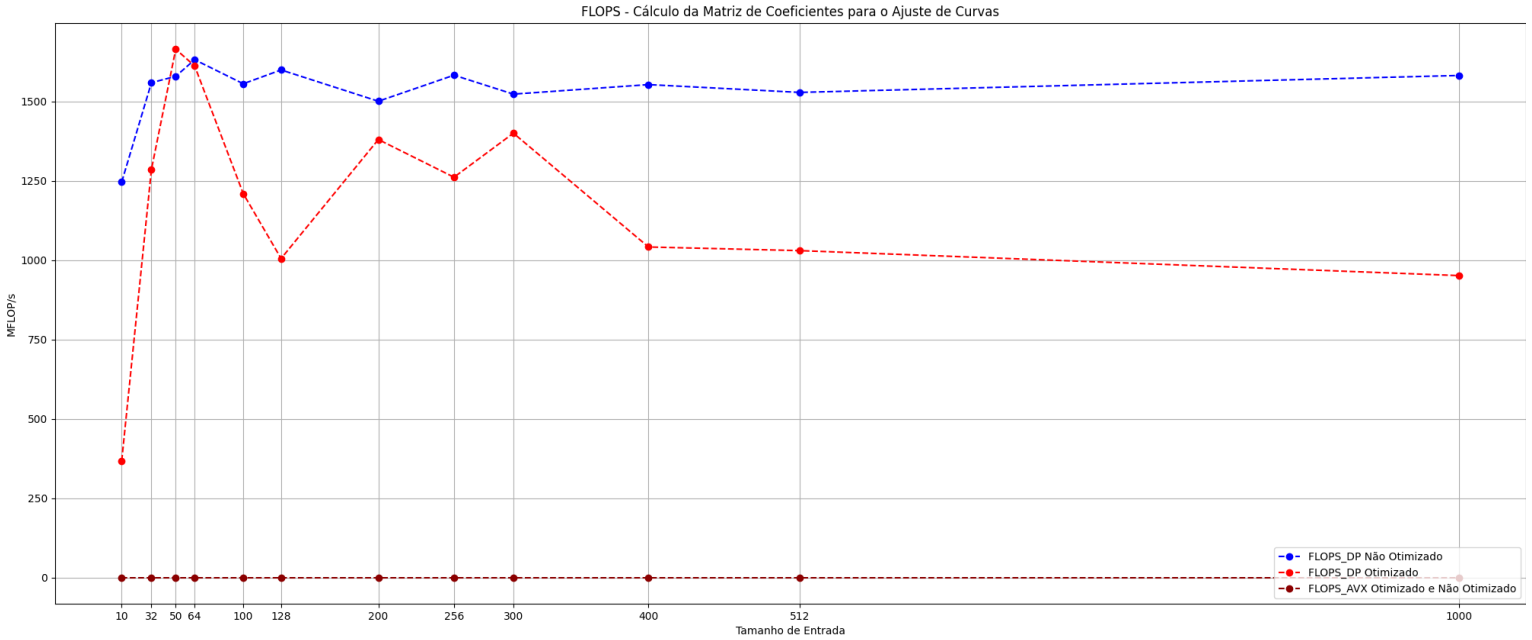


Figura 8 - Gráfico de operações aritméticas da função não otimizada e otimizada.

## 4.2 Análise Gráfica da Triangularização

O primeiro teste realizado diz respeito ao tempo da execução da função, como pode ser visto na **Figura 9**. Podemos ver então o impacto positivo das otimizações efetuadas visto que, de forma geral, a versão otimizada executou em um menor tempo para a mesma entrada de dados.

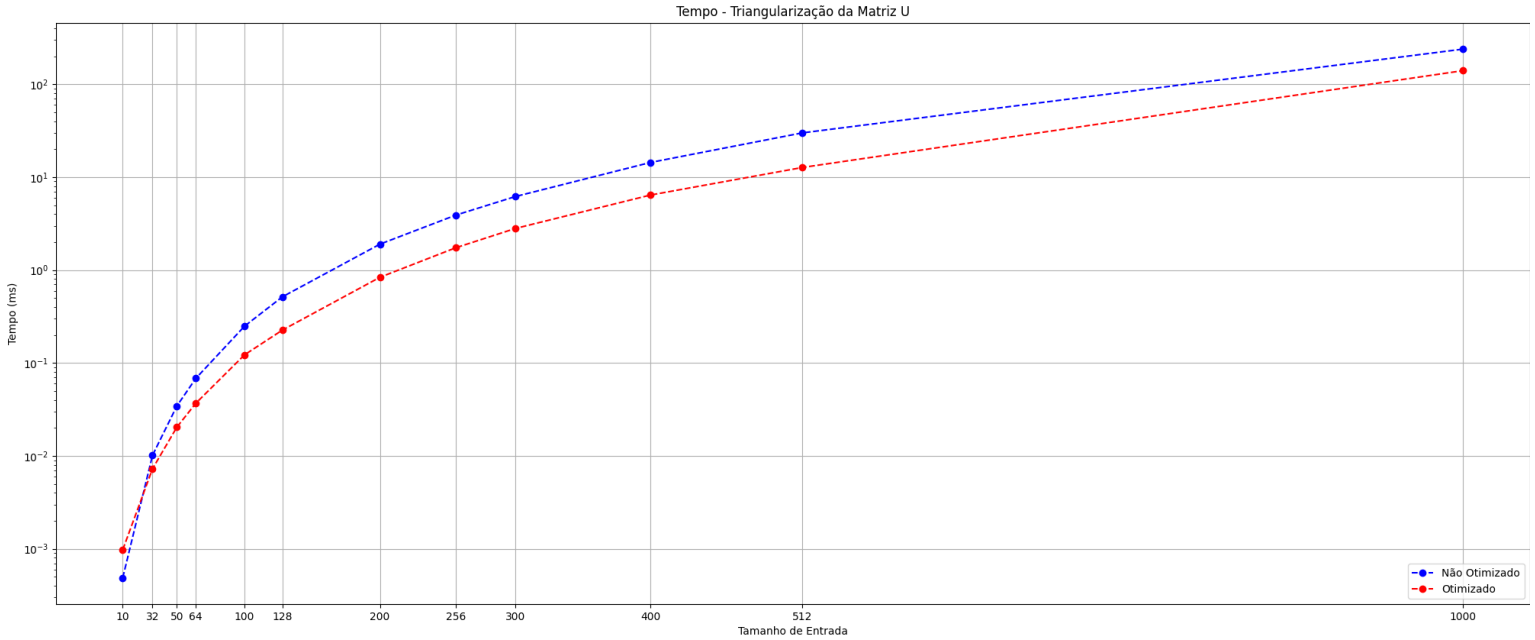


Figura 9 - Gráfico de tempo da função não otimizada e otimizada, em escala logarítmica.

Logo em seguida, na **Figura 10**, verificamos a alteração na largura de banda da memória. É possível então perceber que a capacidade de transmissão dos dados aumentou novamente, indicando maior capacidade de processamento dos dados por parte do processador.

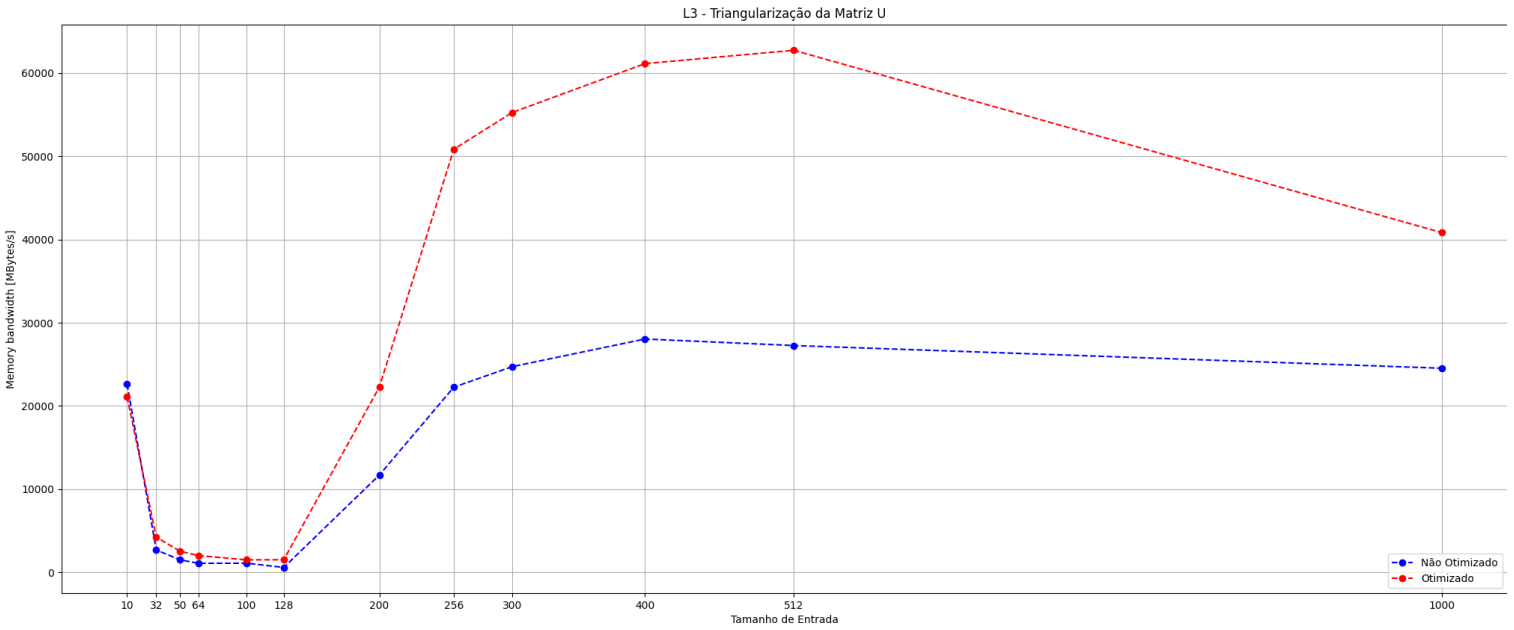


Figura 10 - Gráfico da largura de banda da função não otimizada e otimizada.

Partimos então para a análise de *cache miss ratio*, visível na **Figura 11**. É possível perceber que no geral a técnica de *loop unroll* foi eficiente em aumentar a taxa de acertos na cache, demonstrando boa capacidade do programa em manter localidade espacial e temporal dos dados.

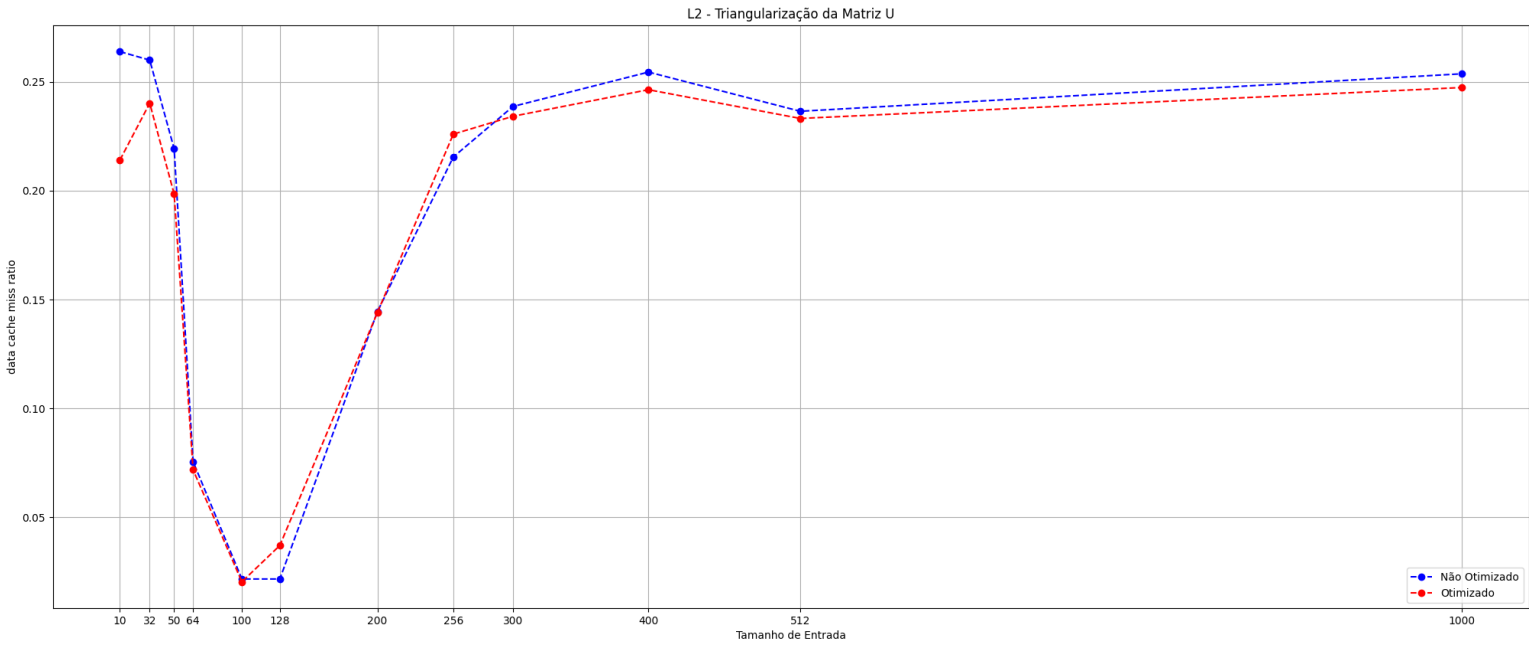


Figura 11 - Gráfico de *cache miss ratio* da função não otimizada e otimizada.

Por fim, analisamos as ações aritméticas realizadas, representadas na **Figura 12**.

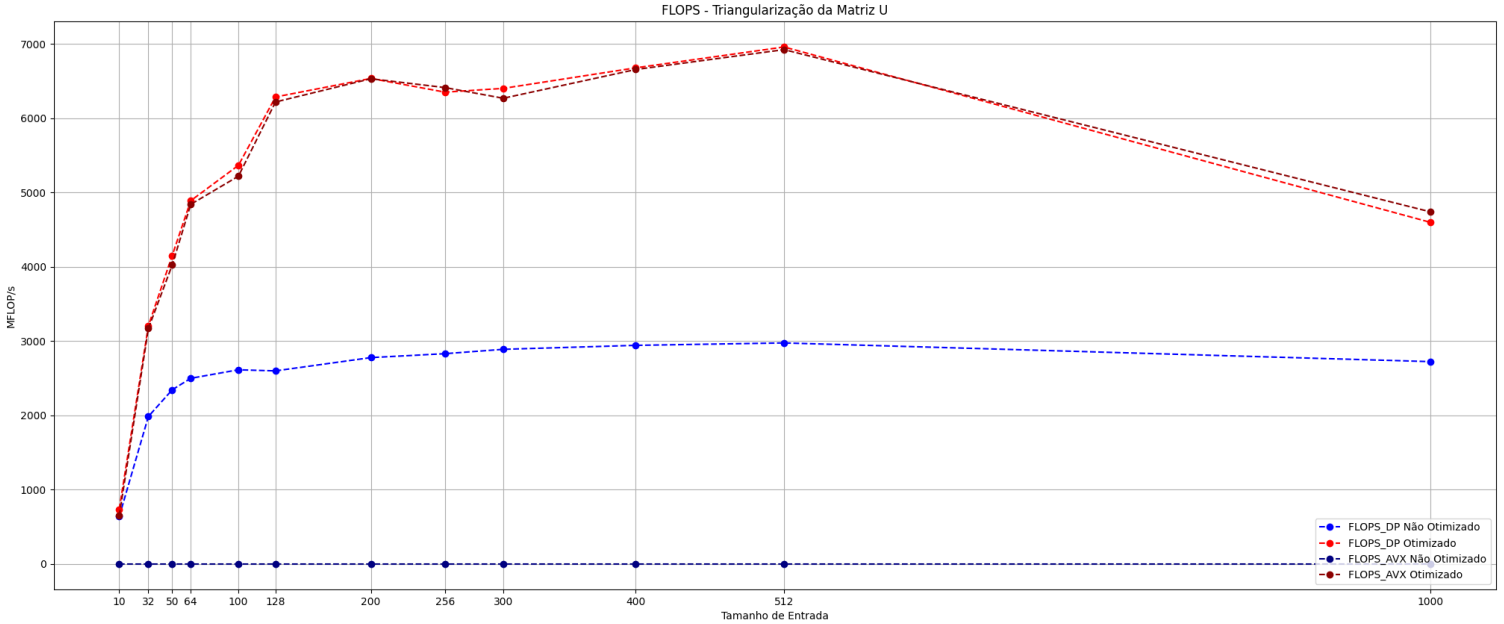


Figura 12 - Gráfico de operações aritméticas da função não otimizada e otimizada.