TRABALHO 2 DE INTRODUÇÃO À COMPUTAÇÃO CIENTÍFICA - CI1164

Gabriel N. H. do Nascimento, Allan Cedric G. B. A. da Silva

1. Introdução

No primeiro trabalho da disciplina, foram desenvolvidos algoritmos capazes de implementar a Fatoração LU - um método eficaz para resolução de múltiplos sistemas lineares de mesma matriz de coeficientes. A aplicação desse método no primeiro trabalho, foi para o problema de descobrir a matriz inversa de uma matriz quadrada dada como entrada.

Neste trabalho 2, vamos aplicar a Fatoração LU para solucionar problemas de Interpolação e Ajuste de Curvas. E com isso, vamos estar usando tanto a implementação do algoritmo usada no trabalho 1, quanto uma nova versão, otimizada com técnicas apresentadas durante a disciplina.

Este trabalho acadêmico tem como objetivo comparar a performance das versões não otimizadas e otimizadas das funções que implementam o algoritmo de Fatoração LU; além de avaliar o desempenho da função que calcula e gera a matriz de coeficientes do sistema linear utilizado no problema de Ajuste de Curvas.

Informações adicionais sobre os arquivos utilizados neste trabalho estão presentes em um arquivo **`README`** adjunto.

2. Sobre o computador utilizado

O computador utilizado para a execução do experimento tem as seguintes especificações.

Configurações da CPU:

- Nome: Intel(R) Core(TM) i7-7700HQ
- Família: Kaby Lake
- Frequência de clock: 2.80GHz ~ 3.8GHz
- N° de núcleos físicos: 4
- N° de núcleos lógicos (Threads): 8 (2 por núcleo físico)
- Cache L1i: 128 KiB (32 KiB por núcleo físico)
- Cache L1d: 128 KiB (32 KiB por núcleo físico)
- Cache L2: 1 MiB (256 KiB por núcleo físico)
- Cache L3: 6 MiB (Compartilhada)
- Consumo médio (TDP): 45 W

- Litografia: 14 nm

Configurações de memória do computador:
- RAM: 16 GiB
- Tipo: SODIMM DDR4 Synchronous Unbuffered
- Largura de operação: 64 Bits
- Frequência de clock: 2400 MHz

Configurações do Sistema Operacional:
- Distribuição GNU/Linux: Ubuntu 20.04.2 LTS
- Kernel: 5.11.0-25-generic

3. Detalhes sobre o experimento

O experimento então é o seguinte: a cada iteração do laço do script **`performance.sh`**, são geradas, através de **`gera_entrada.py`**, M+1 sequências de N valores em ponto flutuante, onde N é um valor inteiro na lista [10, 32, 50, 64, 100, 128, 200, 256, 300, 400, 512, 1000], para cada iteração do laço; e M é um número inteiro aleatório em [2,6].

Essa sequência de inteiros N, M e os números em ponto flutuante é dada como entrada para os programas **`main_unoptimized,`** programa com as funções de fatoração LU oriundas do trabalho 1, e para **`main_optimized`**, programa cujas funções são otimizadas usando técnicas detalhadas no tópico 4.

Estes 2 programas contém LIKWID markers, para que possamos usar a ferramenta LIKWID e medir as métricas de performance nos procedimentos críticos alvos da corrente avaliação. Para cada execução de cada programa, obtemos as seguintes medidas:
- Teste de tempo [ms]
- Banda de Memória [MBytes/s]
- L2 miss ratio
- Operações de ponto flutuante (FLOPS) [MFLOPS/s]

Obtendo estes valores, usa-se o script **`analise.py`** para organizá-los em suas respectivas tabelas, para cada entrada gerada aleatoriamente. Com estas tabelas, no final do experimento, podemos usar outro script **`gera_plot.py`** para gerar os gráficos para cada função marcada pelo LIKWID. Os gráficos podem ser vistos no tópico 5.

Para executar o experimento, tendo o pacote de software descompactado e estando em sua *workspace*, basta usar os comandos:

```
$ make
$ ./performance.sh
```

Ao final da execução do script, um diretório de nome **`Resultados/`**, criado previamente, vai estar preenchido com os gráficos e tabelas geradas pela análise de performance.

4. Otimizações implementadas

As otimizações implementadas no programa **`main_optimized`**, em relação ao programa **`main_unoptimized`**, foram aplicadas na função `LU_decomp()`, sendo a sua versão otimizada chamada de `LU_decomp_optimized()`.

Ademais, em ambos os programas, foram aplicadas técnicas de otimização na geração da matriz de coeficientes, através da função `gen_curve_matcoef()`, para o sistema linear que resolve o problema de Ajuste de Curvas.

4.1 Fatoração LU

4.1.1 Cálculo prévio dos índices da matriz

Na `LU_decomp()`, uma vez que sabemos, em uma iteração do loop, qual é o índice da matriz em (linha, coluna), podemos calculá-lo uma só vez e armazenar o seu resultado em uma variável, como é o caso do seguinte exemplo utilizando a variável `diag_index`:

```
if (fabs(U->A[i * U->n + i]) <= DBL_EPSILON)
    return INV_MAT_ERROR;
L->A[i * L->n + i] = 1.0;
```

Na função não otimizada, o cálculo do índice (i, i) é independente

```
int diag_index = i * ls->U.n + i;
if (fabs(ls->U.A[diag_index]) <= DBL_EPSILON)
    return INV_MAT_ERROR;
ls->L.A[diag_index] = 1.0;
```

Na função otimizada, `diag_index` é calculado previamente

Muitos outros índices de matriz da função não otimizada, como o visto acima, tiveram seus cálculos refatorados para função otimizada, a fim de economizar o custo computacional extra de operações aritméticas.

4.1.2 Loop unroll (Fase de escalonamento)

Ainda na fatoração LU, quando geramos os coeficientes da matriz U, ou seja, escalonando a matriz U, temos um loop possível de desenrolar. Veja como ficou, usando um passo `UNROLL_STEP` de número 4:

```
for (int j = i + 1; j < ls->U.n; j++)
    ls->U.A[line_k + j] -= (ls->U.A[line_i + j] * mp);
```

Sem *loop unroll*

```
int end_step = ls->U.n - ((ls->U.n - (i + 1)) % UNROLL_STEP);
for (int j = i + 1; j < end_step; j += UNROLL_STEP)
{
    int kj = line_k + j;
    int ij = line_i + j;

    ls->U.A[kj] -= (ls->U.A[ij] * mp);
    ls->U.A[kj + 1] -= (ls->U.A[ij + 1] * mp);
    ls->U.A[kj + 2] -= (ls->U.A[ij + 2] * mp);
    ls->U.A[kj + 3] -= (ls->U.A[ij + 3] * mp);
}
for (int j = end_step; j < ls->U.n; j++)
    ls->U.A[line_k + j] -= (ls->U.A[line_i + j] * mp);
```

Com *loop unroll*

Apesar do código mais extenso, esse tipo de técnica utiliza as linhas de memória cache de forma mais eficiente, contribuindo de maneira significativa para a performance do algoritmo. No caso do código com *loop unroll* acima, estão sendo gerados a cada iteração 4 termos escalonados de uma linha k a partir de uma linha i.

4.2 Ajuste de Curvas

4.2.1 Cálculo prévio de índices

Essa técnica de otimização aplicada na função `gen_curve_matcoef()`, apresenta motivação semelhante quando foi utilizada na função `LU_decomp_optimized()`, como mostrado anteriormente.

4.2.2 Cálculo da primeira linha da matriz de coeficientes

Como visto na disciplina, essa matriz possui um padrão no que diz respeito aos coeficientes que são gerados. No caso da primeira linha, ela apresenta termos que se repetem ao longo das outras linhas, podemos abusar desse fator para reduzir a quantidade de operações de ponto de flutuante.

O primeiro termo da primeira linha é igual a um somatório de n termos de valor 1, sendo assim podemos simplificar a expressão do somatório para um valor constante n.

```
ls->A.A[0] = (real_t)(ls->n);
```

$$\sum_{i=1}^{n} (1) = n$$

Ademais, cada termo restante é calculado através de um laço, que implementa o somatório, e que utiliza a técnica de *loop unrolling*.

```
for (int j = 1; j < ls->n; j++)
{
    ls->A.A[j] = 0.0;

    int end_step = ls->n - (ls->n % UNROLL_STEP);
    for (int i = 0; i < end_step; i+=UNROLL_STEP)
        ls->A.A[j] += pow(x[i], j) + pow(x[i + 1], j) +
                      pow(x[i + 2], j) + pow(x[i + 3], j);

    for (int i = end_step; i < ls->n; i++)
        ls->A.A[j] += pow(x[i], j);
}
```

loop unrolling para $\sum_{i=1}^{n} (x_i^j)$

4.2.3 Cálculo das linhas restantes da matriz

Utilizando-se do fato anteriormente citado, após à primeira linha da matriz, podemos copiar (n-1) termos, a partir da segunda coluna de uma linha (k-1) para a linha k em sua primeira coluna, havendo necessidade apenas de calcular o último termo da linha k, o qual é igual a um somatório de n termos. A resolução desse somatório também provém da técnica de *loop unrolling*.

```c
for (int k = 1; k < ls->n; k++)
{
    memcpy(ls->A.A + k*ls->n,
        ls->A.A + (k-1)*ls->n + 1,
        (ls->n - 1)*sizeof(real_t));

   int last_index = k * ls->n + (ls->n - 1);
   real_t exp = ls->n + (k - 1);

   ls->A.A[last_index] = 0.0;

   int end_step = ls->n - (ls->n % UNROLL_STEP);
   for (int i = 0; i < end_step; i += UNROLL_STEP)
       ls->A.A[last_index] += pow(x[i], exp) +
                              pow(x[i + 1], exp) +
                              pow(x[i + 2], exp) +
                              pow(x[i + 3], exp);

   for (int i = end_step; i < ls->n; i++)
       ls->A.A[last_index] += pow(x[i], exp);
}
```

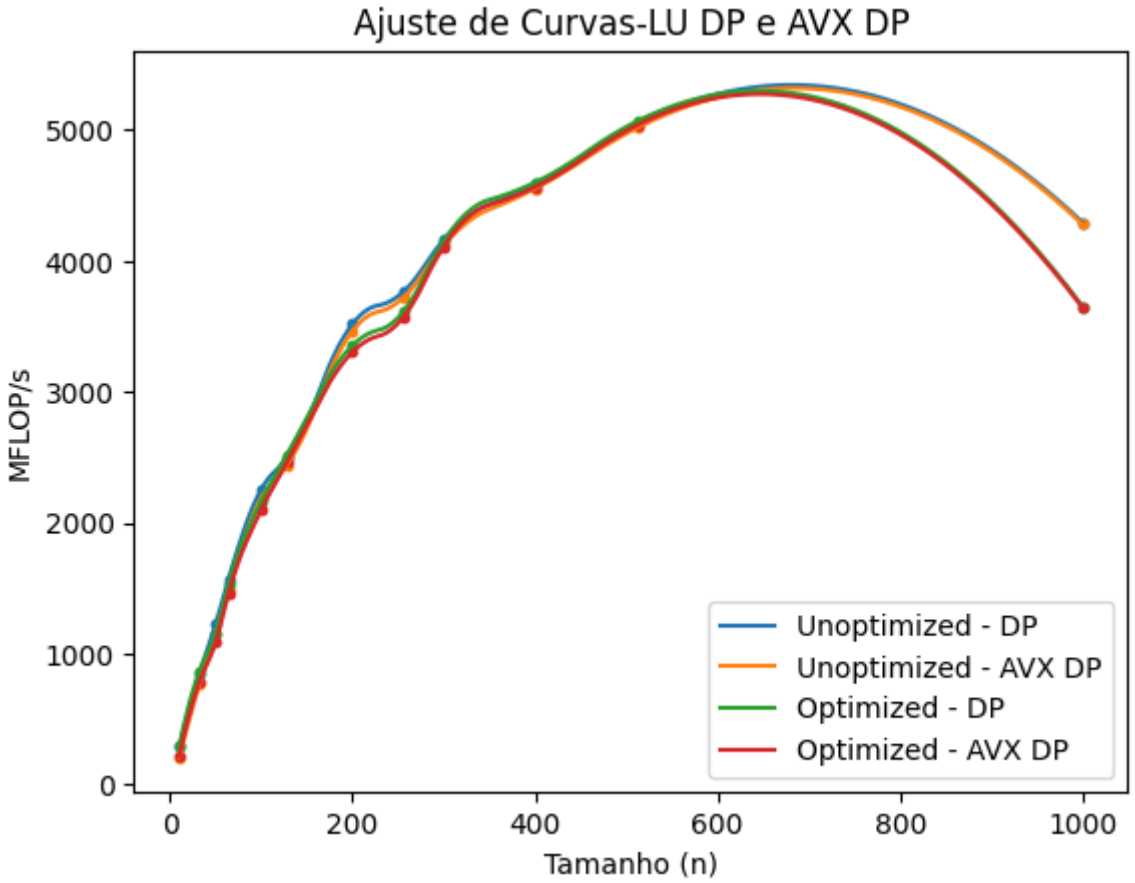
loop unrolling para $\sum_{i=1}^{n} (x_i^j)$ novamente, mas apenas para um termo

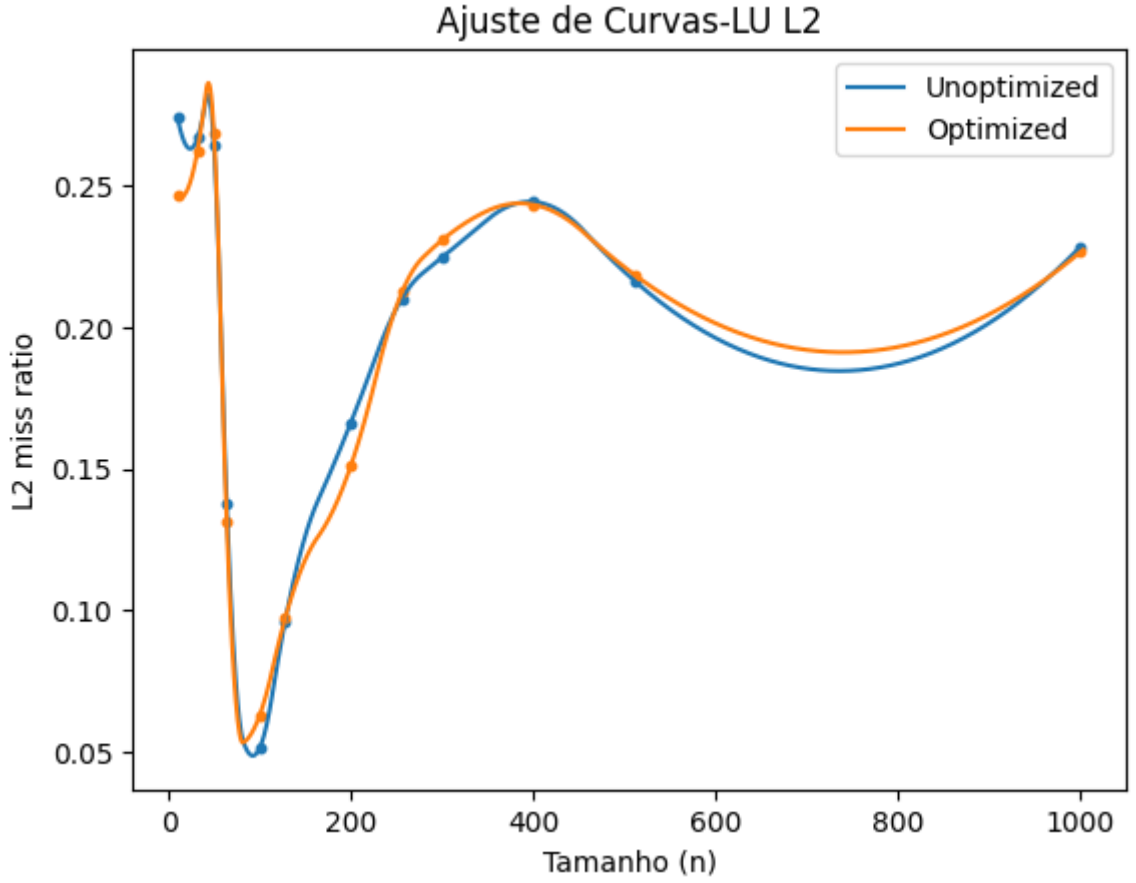
Note que a função `gen_curve_matcoef()` é a mesma para os dois programas **`main_unoptimized`** e **`main_optimized`**, e não se espera mudança nos resultados do ajuste de curva nas duas execuções. Nos gráficos do tópico 5, estes valores foram agrupados e extraiu-se a sua média.

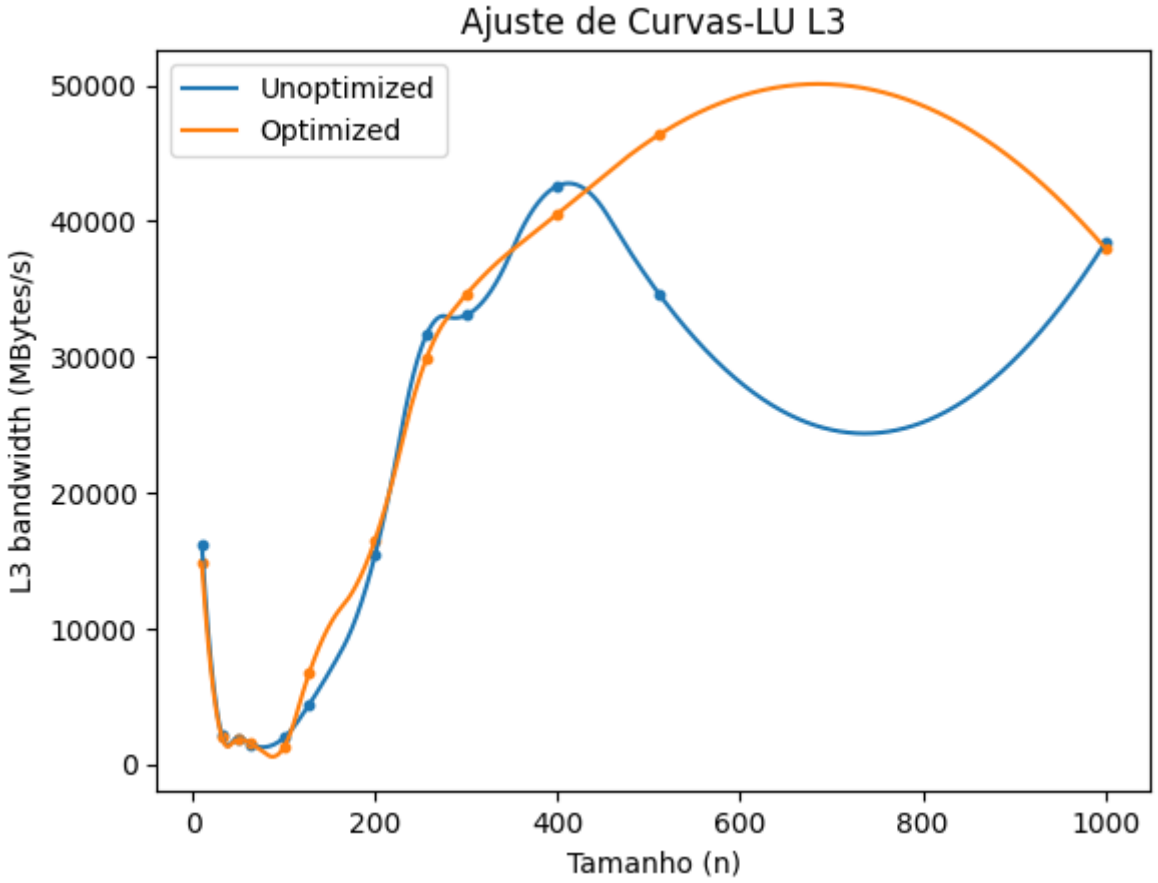
5. Resultados do experimento

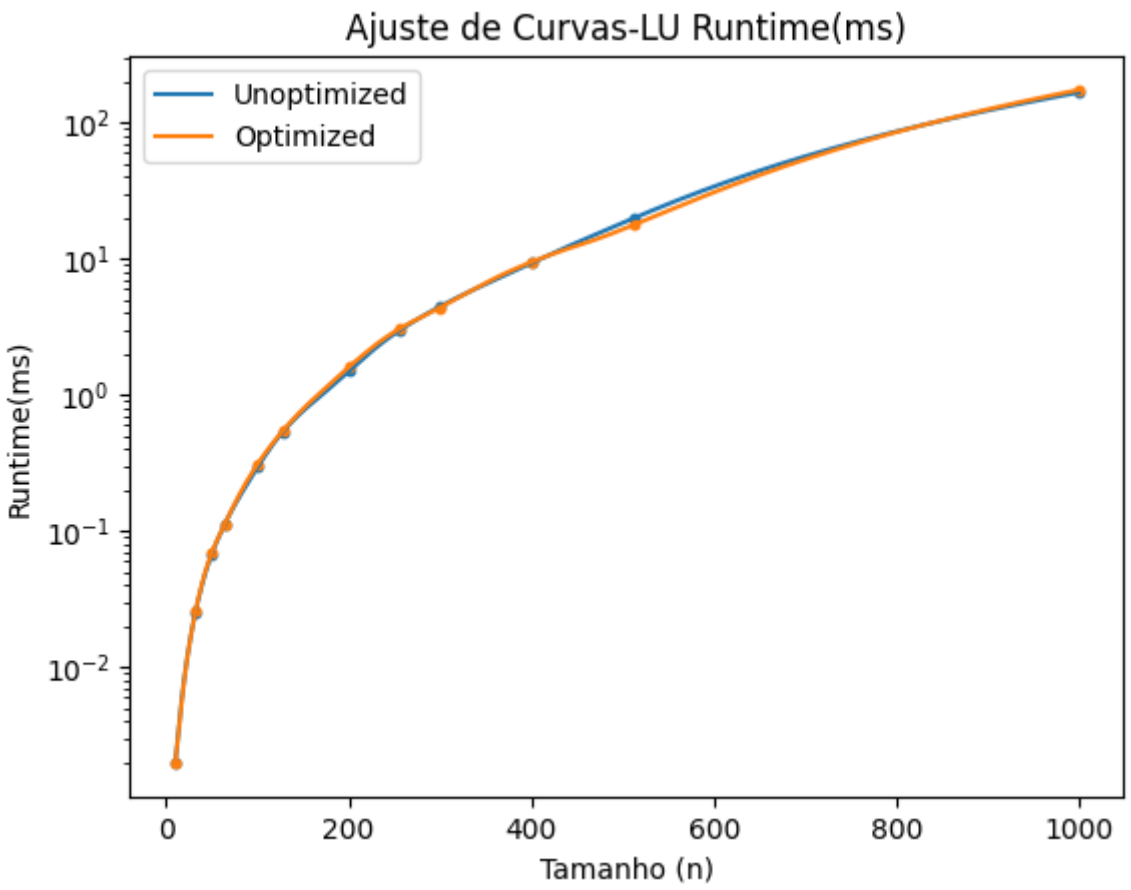
As otimizações não causaram muitas melhorias em questão de tempo de execução, porém é possível perceber que as melhorias mais visíveis foram em sistemas lineares de ordem superior, como 400, 512 e 1000, principalmente nas métricas de memória e de uso de instruções, como L2 Cache miss ratio, L3 Bandwidth e no uso de operações de ponto flutuante simples (DP) e vetorizadas (AVX).

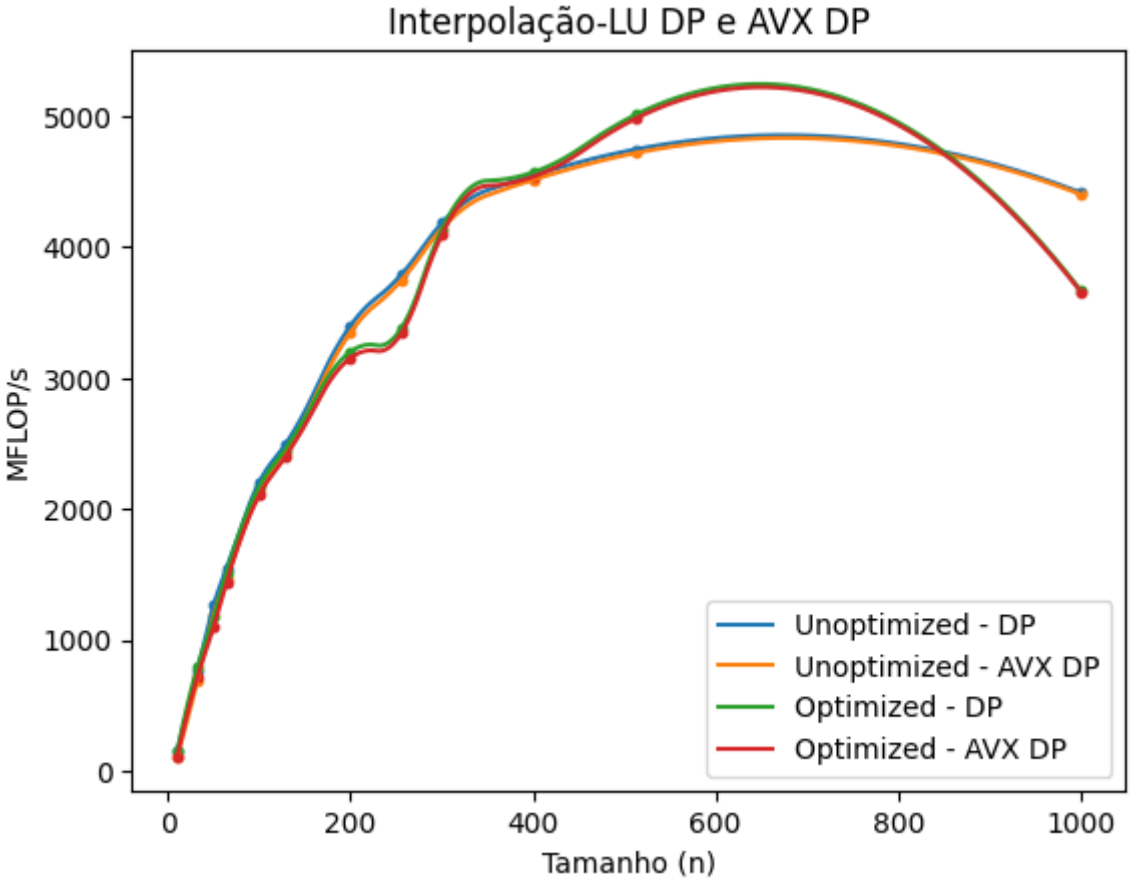
A seguir estão os gráficos, métricas para a fatoração LU usada para solução de ajuste de curvas, LU para interpolação, e desempenho do `gen_curve_matcoef()`.

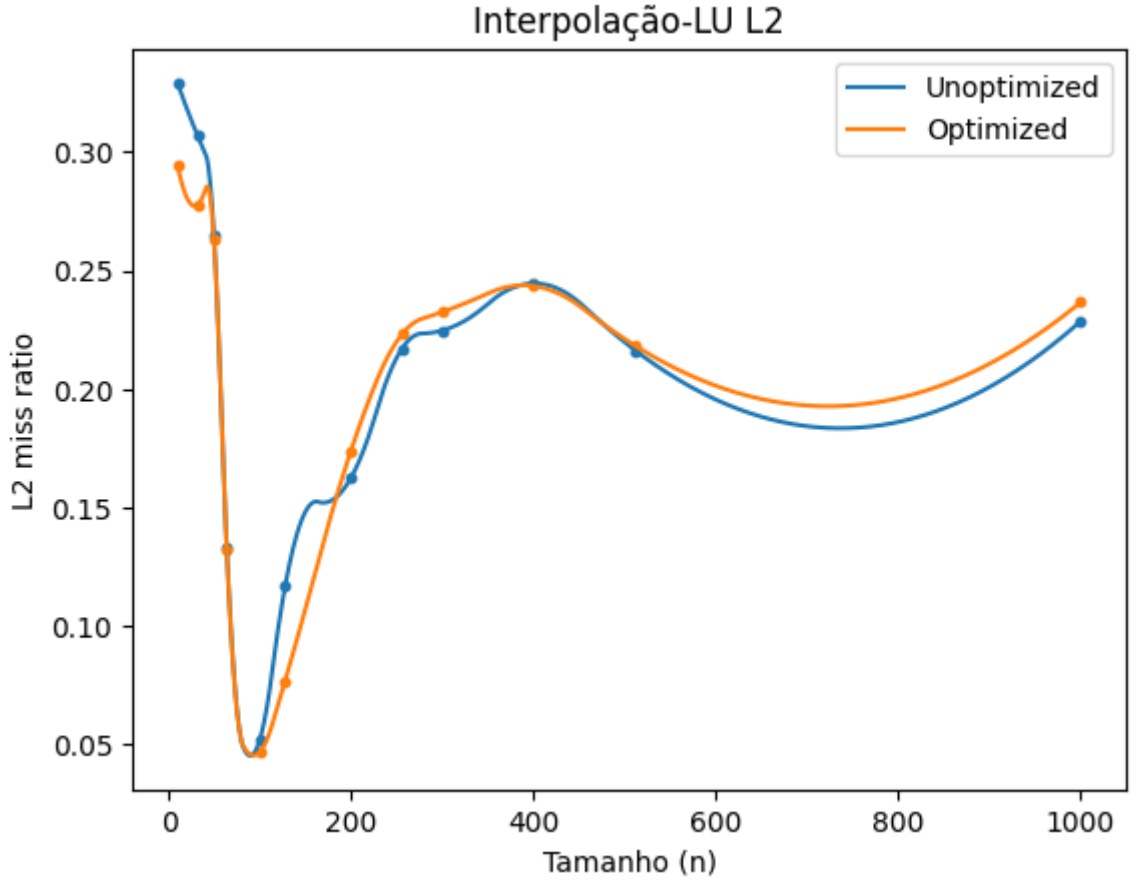

Ajuste de Curvas-LU DP e AVX DP
MFLOP/s
Tamanho (n)
0
1000
2000
3000
4000
5000
0
200
400
600
800
1000
Unoptimized - DP
Unoptimized - AVX DP
Optimized - DP
Optimized - AVX DP

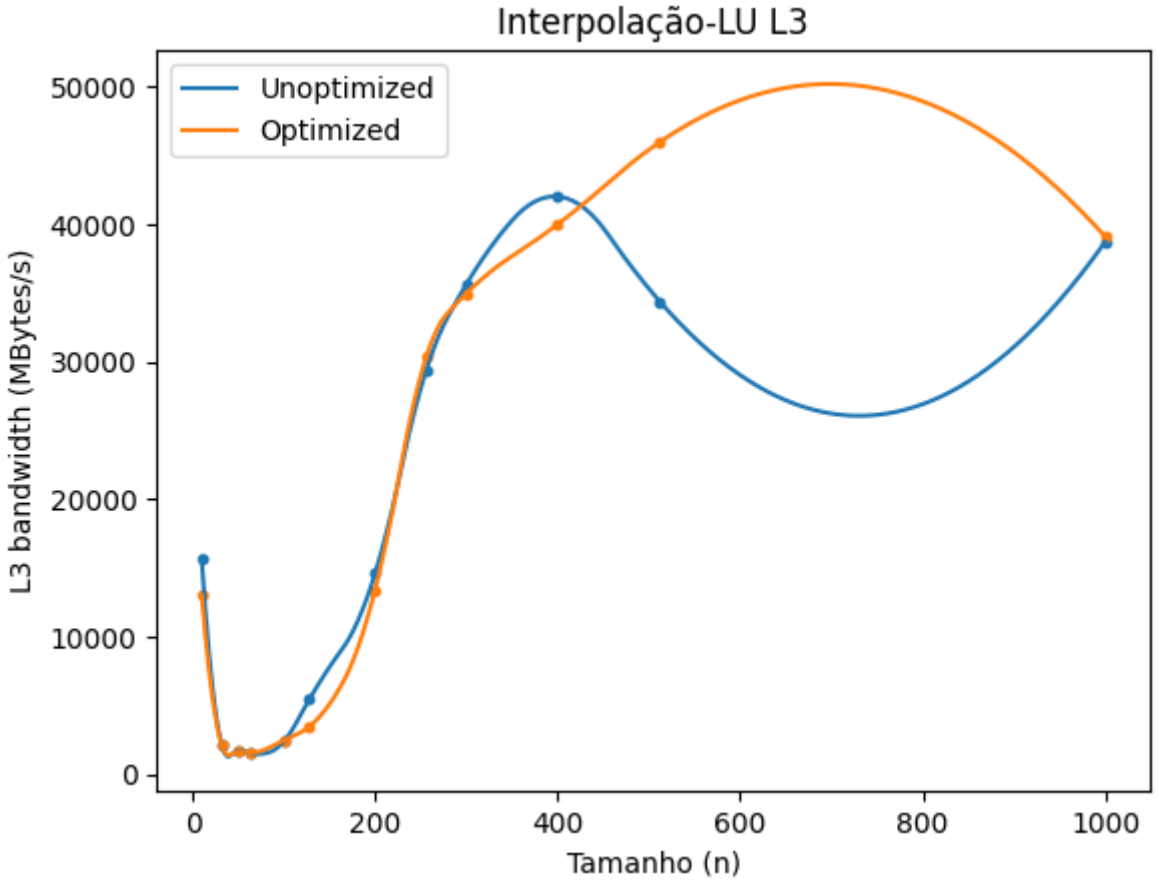

Ajuste de Curvas-LU L2
Unoptimized
Optimized
L2 miss ratio
0.25
0.20
0.15
0.10
0.05
0
200
400
600
800
1000
Tamanho (n)

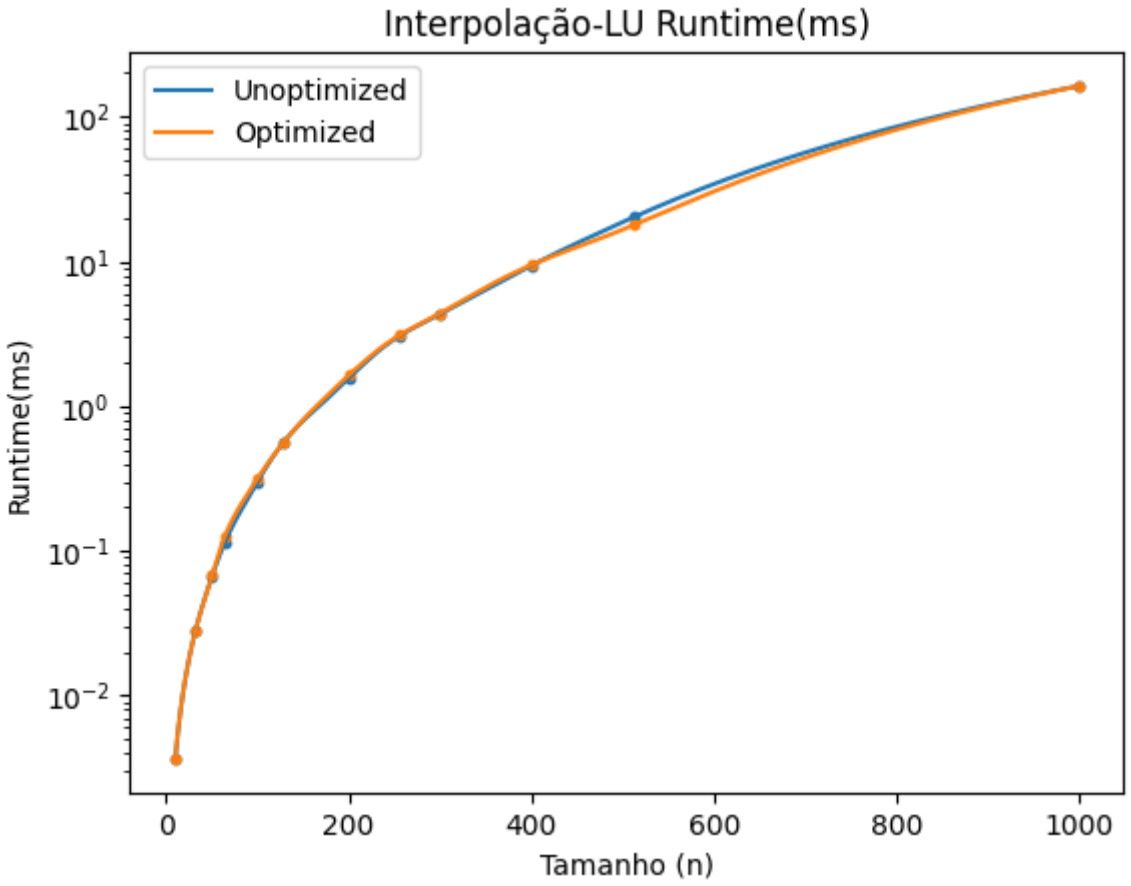

Ajuste de Curvas-LU L3
Unoptimized
Optimized
L3 bandwidth (MBytes/s)
50000
40000
30000
20000
10000
0
0
200
400
600
800
1000
Tamanho (n)

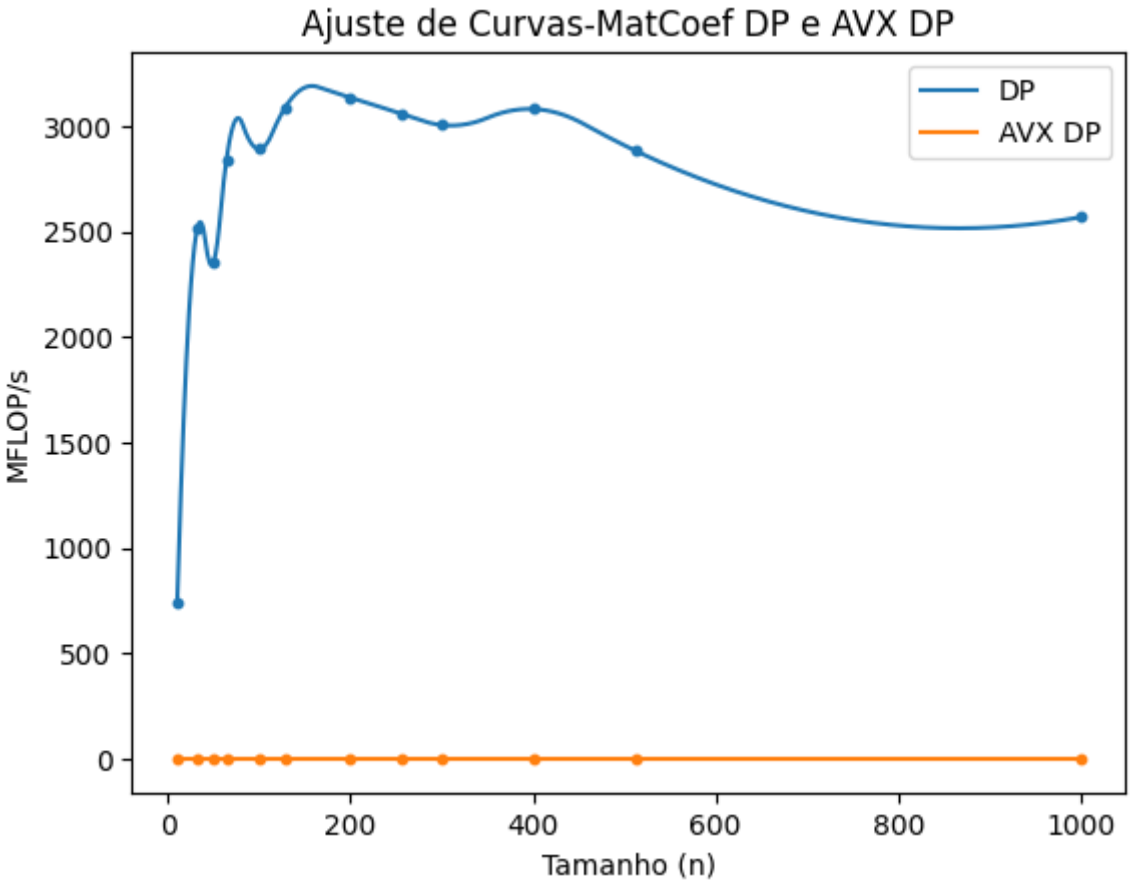

Ajuste de Curvas-LU Runtime(ms)
Unoptimized
Optimized
Runtime(ms)
10^2
10^1
10^0
10^{-1}
10^{-2}
0
200
400
600
800
1000
Tamanho (n)

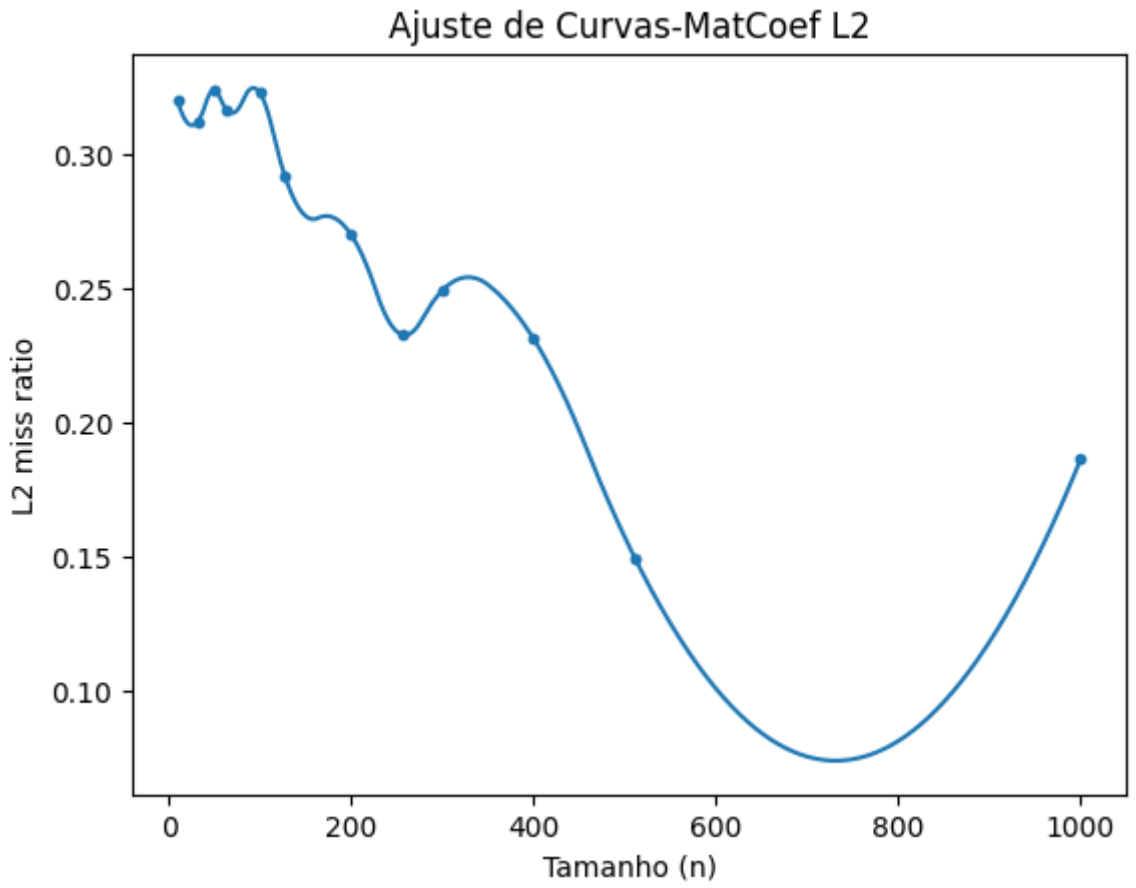

Interpolação-LU DP e AVX DP
MFLOP/s
5000
4000
3000
2000
1000
0
Unoptimized - DP
Unoptimized - AVX DP
Optimized - DP
Optimized - AVX DP
0
200
400
600
800
1000
Tamanho (n)

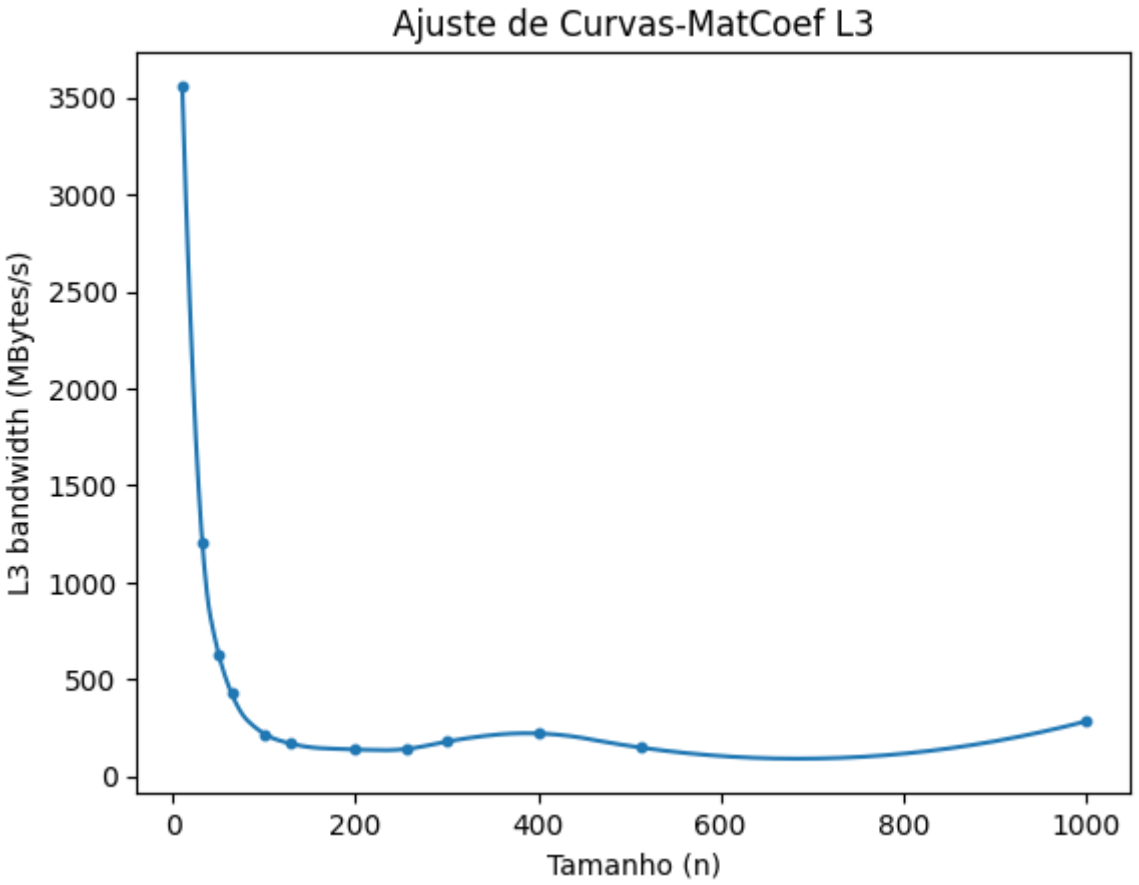

Interpolação-LU L2
Unoptimized
Optimized
L2 miss ratio
0.30
0.25
0.20
0.15
0.10
0.05
0
200
400
600
800
1000
Tamanho (n)

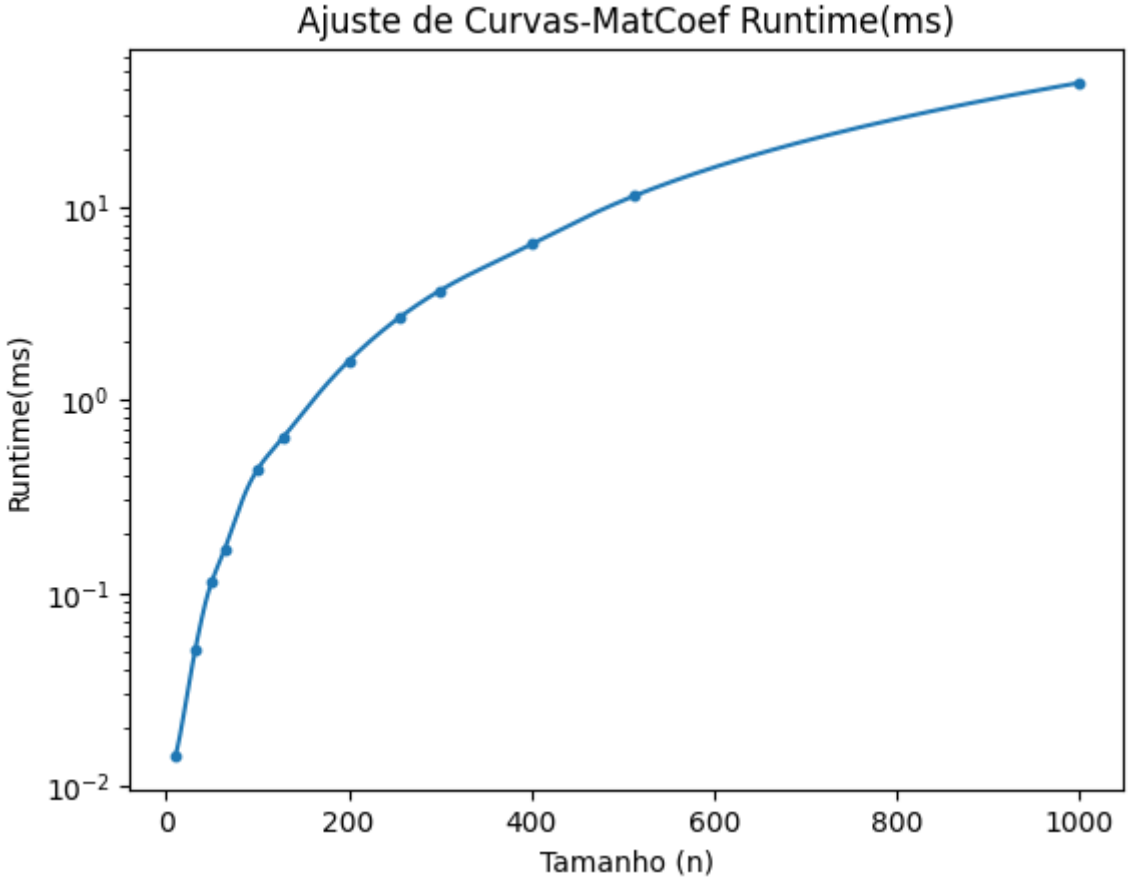

Interpolação-LU L3
Unoptimized
Optimized
L3 bandwidth (MBytes/s)
50000
40000
30000
20000
10000
0
0
200
400
600
800
1000
Tamanho (n)


Interpolação-LU Runtime(ms)
Unoptimized
Optimized
Runtime(ms)
10^2
10^1
10^0
10^{-1}
10^{-2}
0
200
400
600
800
1000
Tamanho (n)


Ajuste de Curvas-MatCoef DP e AVX DP
DP
AVX DP
MFLOP/s
3000
2500
2000
1500
1000
500
0
0
200
400
600
800
1000
Tamanho (n)


Ajuste de Curvas-MatCoef L2
L2 miss ratio
0.30
0.25
0.20
0.15
0.10
0
200
400
600
800
1000
Tamanho (n)


Ajuste de Curvas-MatCoef L3
L3 bandwidth (MBytes/s)
3500
3000
2500
2000
1500
1000
500
0
0
200
400
600
800
1000
Tamanho (n)


Ajuste de Curvas-MatCoef Runtime(ms)
Runtime(ms)
10^{1}
10^{0}
10^{-1}
10^{-2}
0
200
400
600
800
1000
Tamanho (n)