Universidade Federal do Paraná

Bacharelado em Ciência da Computação

Disciplina:
Introdução à Computação Científica
[CI1164]

Relatório referente a:
Trabalho 2: Otimização de Desempenho

Mateus Kater Pombeiro - GRR20190366
João Pedro Vicente Ramalho - GRR20224169

Novembro
2023

1. Introdução

Este relatório tem como objetivo detalhar o Trabalho 2 da disciplina de Introdução à Computação Científica (CI1164), bem como sua implementação, limitações e testes realizados. O Trabalho 2 consiste na otimização do código desenvolvido no Trabalho 1, o qual consiste na implementação de ajuste polinomial de curvas utilizando cálculo intervalar.

Dessa forma, o Trabalho 2 representa a versão otimizada do Trabalho 1. Assim, nosso intuito é analisar o resultado das otimizações implementadas. Para isso, foram adotados alguns parâmetros de medição para avaliar a melhoria de desempenho, tais como: MFLOP/s, banda de memória, tempo médio de execução e cache miss.

2. Organização

O trabalho foi organizado de forma que o diretório principal contém:

- “v1”: pasta contendo o código fonte do trabalho Original em linguagem C;
- “v2”: pasta contendo o código fonte do trabalho 2 (versão otimizada);
- “rush.sh”: compila e executa cada uma das versões do trabalho com parâmetros diferentes, armazena os dados obtidos em pastas no diretório principal para então executar o programa “gera_graficos.gp” - que gera gráficos a partir dos dados coletados nas execuções e armazenados nas pastas;
- “README.md”: arquivo contendo descrição geral dos módulos e qualquer outro esclarecimento sobre o programa;

3. Otimizações

No trabalho 1, desenvolvemos um programa capaz de criar um sistema linear utilizando o método numérico dos Mínimos Quadrados. A solução desse sistema resulta em um polinômio que melhor se ajusta à curva correspondente ao conjunto de pontos passados como parâmetro do programa. Para isso, implementamos uma estrutura de dados (ajustePol) que armazena:

- o grau do polinômio;
- a quantidade de pontos recebidas como parâmetro;
- a matriz do sistema linear;
- o vetor com os resultados das equações do sistema;
- o vetor para armazenar os coeficientes do polinômio;
- o vetor contendo o conjunto de pontos [x0,y0,x1,y1,x2,y2,...]

Otimização na Alocação de Estruturas

Com o objetivo de tornar o acesso à memória o mais contínuo possível, optou-se por alterar o modo de alocação da matriz de forma que todos os seus elementos estejam alocados de maneira sequencial na memória. Para atingir esse objetivo, utilizou-se o seguinte método:

```c
// Aloca uma matriz de structs operandos* de tamanho NxN e retorna um ponteiro para ela
struct operandos** alocarMatriz(int N) {

    // Alocação dinâmica da matriz
    struct operandos** matriz = (struct operandos**)malloc(N * sizeof(struct operandos*));

    if (!(matriz)) {
        fprintf(stderr, "Erro falha ao alocar vetor.\n");
        exit(1);
    }

    matriz[0] = malloc(N * N * sizeof(struct operandos));

    for (int i = 1; i < N; i++) {
        matriz[i] = matriz[0] + i * N;
        if (!(matriz[i])) {
            fprintf(stderr, "Erro falha ao alocar vetor.\n");
            exit(1);
        }
    }

    return matriz;
}
```

Além disso, observou-se que ao longo do programa os valores de “x” eram acessados com mais frequência em comparação aos de “y”. Portanto, alocá-los em um único vetor não é a maneira mais eficiente de armazenamento. Por essa razão, optou-se por alocar um vetor exclusivamente para os valores de “x” e outro para “y”.

Otimização nos Cálculos Intervalares

No Trabalho 1, as funções “calcularMultiplicacao()” e “calcularDivisao()”, referentes aos cálculos de multiplicação e divisão intervalar, chamavam as funções “max()” e “min()” para encontrar o número máximo e mínimo de um vetor, respectivamente. Considerando a recorrência dessas funções de cálculo intervalar no programa, optou-se por transcrever as implementações de “max()” e “min()” para dentro das funções de cálculo intervalar, pois assim evita-se “jumps” para procedimentos fora das funções.

```
// Realizando multiplicações para o limite inferior
fesetround(FE_DOWNWARD);
aux[0] = x.anterior * y.anterior;
aux[1] = x.anterior * y.posterior;
aux[2] = x.posterior * y.anterior;
aux[3] = x.posterior * y.posterior;

resultado.anterior = min(aux);
```

```
// Realizando multiplicações para o limite inferior
fesetround(FE_DOWNWARD);
aux[0] = x.anterior * y.anterior;
aux[1] = x.anterior * y.posterior;
aux[2] = x.posterior * y.anterior;
aux[3] = x.posterior * y.posterior;

// Inicializa 'menor' com o valor do primeiro elemento
menor = aux[0];
for (int i = 1; i < 4; i++) {
    dif = fabs(aux[i] - menor);

    // Verificando se o valor do vetor é o menor
    // levando em consideração a margem de erro (DBL_EPSILON)
    if ((aux[i] < menor) && (dif > DBL_EPSILON)) {
        menor = aux[i];
    }
}

resultado.anterior = menor;
```

Além disso, observou-se que na geração da matriz do sistema linear (SL) os valores de “x” são sempre positivos. Assim, eles sempre caem no mesmo caso de exponenciação ([a,b]^p = [a^p, b^p]). Diante disso, foi possível criar uma versão otimizada da função “calcularExpo()”, nomeada de “calcularExpoOtimizada()”, que aplica diretamente o caso de exponenciação mencionado acima, eliminando a necessidade de percorrer condicionais aninhadas presentes no Trabalho 1.

Otimização no Mínimos Quadrados

Após análise da função “minimosQuadrados()”, verificou-se que a matriz do SL do Trabalho 1 era simétrica no eixo da diagonal principal. Dessa forma, seria possível obter a matriz completa apenas calculando os valores presentes no triângulo superior e posteriormente aplicando-os na região do triângulo inferior. Para implementar essa otimização, foram introduzidos loops que aplicam o método dos mínimos quadrados à região triangular superior da matriz, seguido pela transposição desses valores para a região inferior.

Além disso, a fim de aproveitar o paralelismo desses loops, foi aplicado o método de “Unroll & Jam” sobre o loop que itera sobre os pontos (x,y), sendo o cofator de unroll correspondente a 4. As implementações dessa otimização ficaram extensas, e por esse motivo, não incluímos imagens nesta explicação, mas elas estão disponíveis no arquivo “minimosQuadrados.c”.

Otimização no Cálculos dos Resíduos

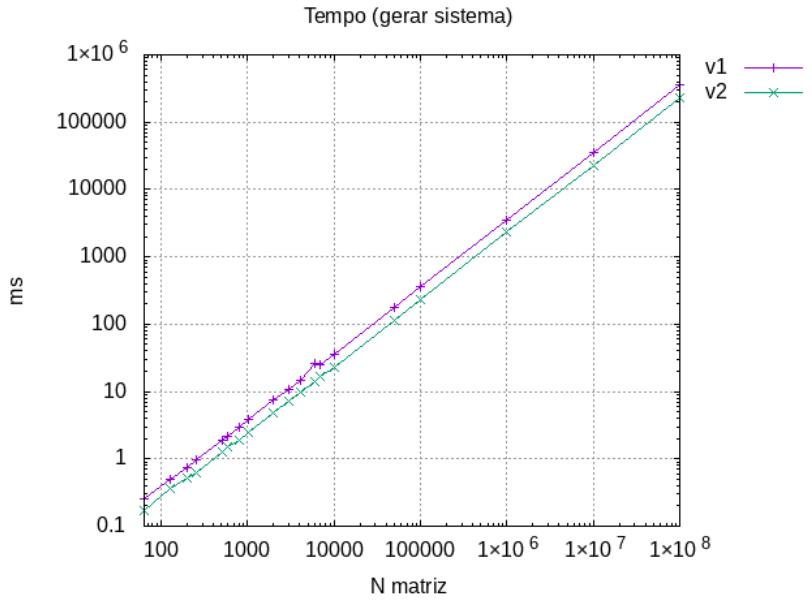
Para otimizar a função referente ao cálculo dos resíduos, implementou-se o método de “Unroll & Jam”, sendo o cofator de unroll correspondente a 4. Essa abordagem foi escolhida para aproveitar mais eficientemente os recursos de hardware, como os processadores multi-core. Abaixo está um print da implementação:

```c
void calculaResiduos(struct ajustePol* sistema) {

    struct operandos *residuos = malloc(sistema->qntdPontos*(sizeof(struct operandos)));
    struct operandos FXi, Yi;

    // Percorre pela tabela de pontos calculando o resíduo de cada f(Xi)
    for (int i = 0; i < sistema->qntdPontos - sistema->qntdPontos%4; i+=4) {
        Yi = sistema->pontos_y[i];
        FXi = polinomio(sistema, sistema->pontos_x[i]);
        residuos[i] = calcularSubtracao(Yi, FXi);
        Yi = sistema->pontos_y[i+1];
        FXi = polinomio(sistema, sistema->pontos_x[i+1]);
        residuos[i+1] = calcularSubtracao(Yi, FXi);
        Yi = sistema->pontos_y[i+2];
        FXi = polinomio(sistema, sistema->pontos_x[i+2]);
        residuos[i+2] = calcularSubtracao(Yi, FXi);
        Yi = sistema->pontos_y[i+3];
        FXi = polinomio(sistema, sistema->pontos_x[i+3]);
        residuos[i+3] = calcularSubtracao(Yi, FXi);
    }
    for (int i = sistema->qntdPontos - sistema->qntdPontos%4; i < sistema->qntdPontos; i++) {
        Yi = sistema->pontos_y[i];
        FXi = polinomio(sistema, sistema->pontos_x[i]);
        residuos[i] = calcularSubtracao(Yi, FXi);
    }
}
```

4. Gráficos

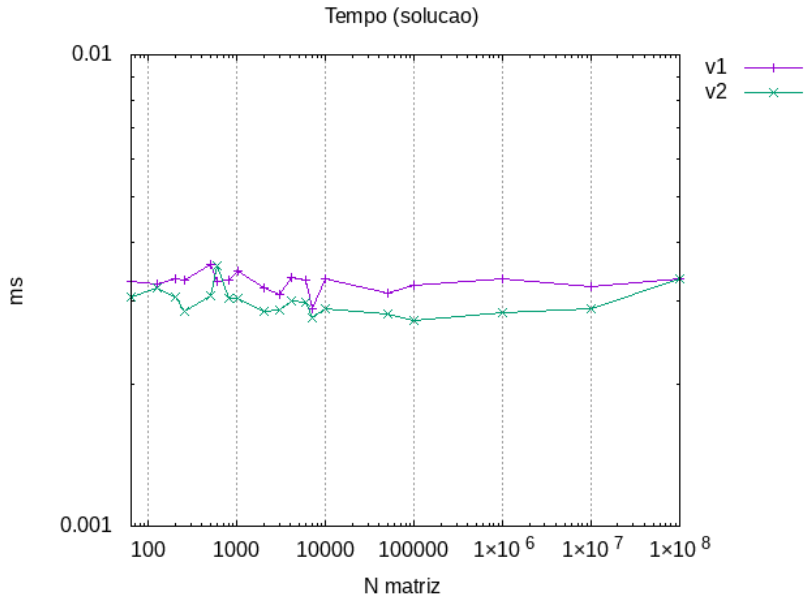
A seguir, apresentaremos os gráficos gerados pelo programa com o objetivo de realizar comparações entre o desempenho de ambas as versões. Foram gerados 10 gráficos utilizando as métricas extraídas do programa. Para visualizá-los, basta executar o script “gera_graficos.gp” (gnuplot) com os dados exportados do programa no devido local (localizados na pasta “dados”).

- Tempo (gerar sistema)



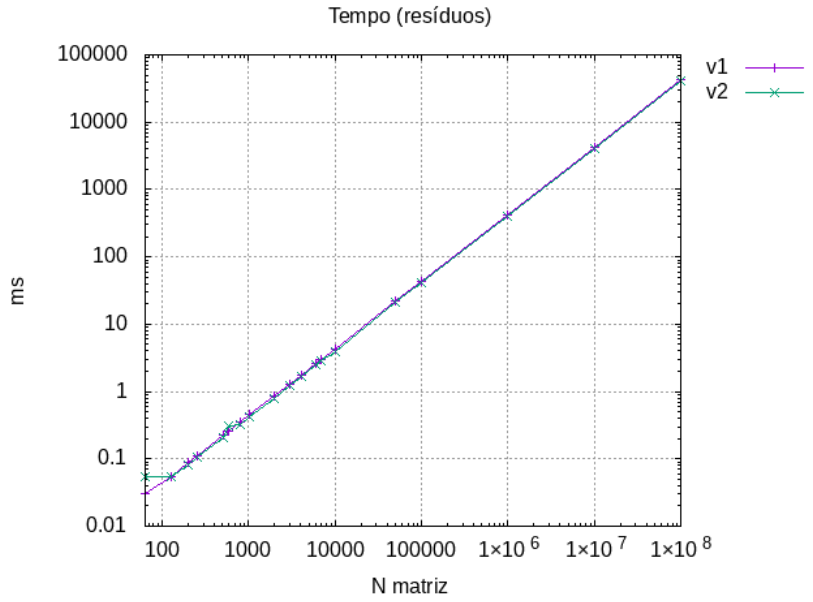
Tendo em vista que o tempo necessário para gerar o sistema foi menor para a versão optimizada do código original (v2), podemos concluir que as otimizações foram eficazes para a redução no tempo de execução desse trecho do programa.

- Tempo (solução)



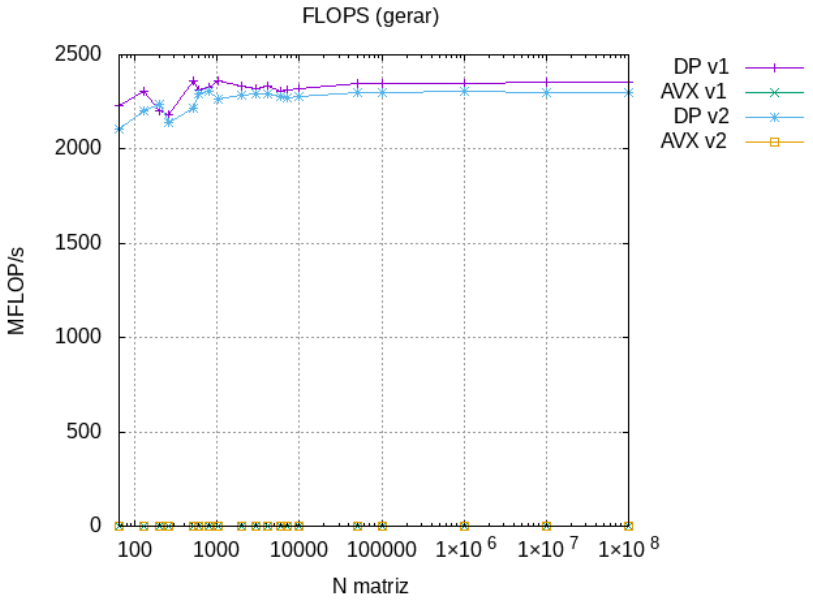
Neste caso, observou-se que para a maioria das matrizes analisadas o tempo da versão 2 foi melhor, o que evidencia a otimização do código.

- Tempo (resíduos)



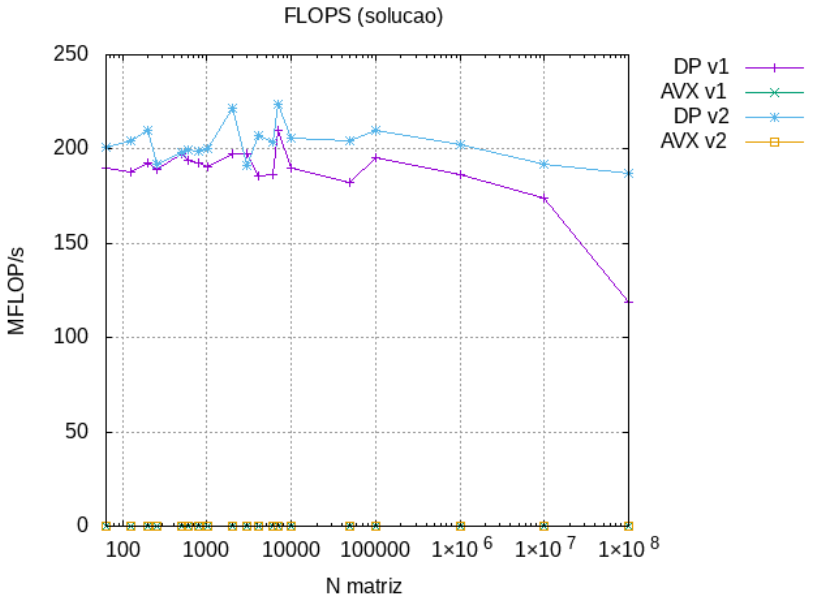
Este gráfico representa o tempo decorrido no cálculo dos resíduos. Nota-se que não houve mudança no desempenho, isso ocorre porque não foram encontradas formas eficazes de otimização de tempo para a função do cálculo do resíduo.

- FLOPS (gerar)



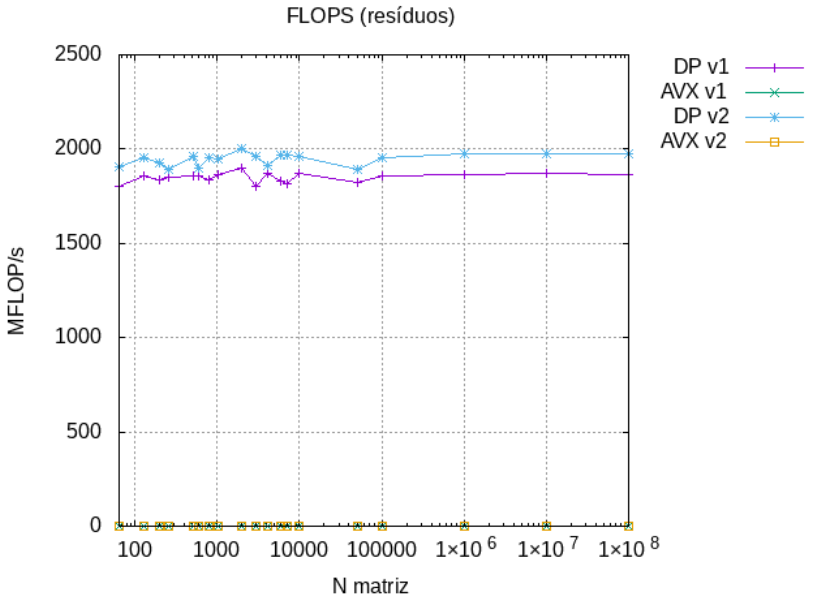
Este gráfico representa a velocidade de operações aritméticas para gerar o sistema linear. Ele evidencia que a abordagem adotada para otimização da função dos mínimos quadrados foi eficaz, pois a velocidade para gerar o sistema linear foi maior.

- FLOPS (solução)



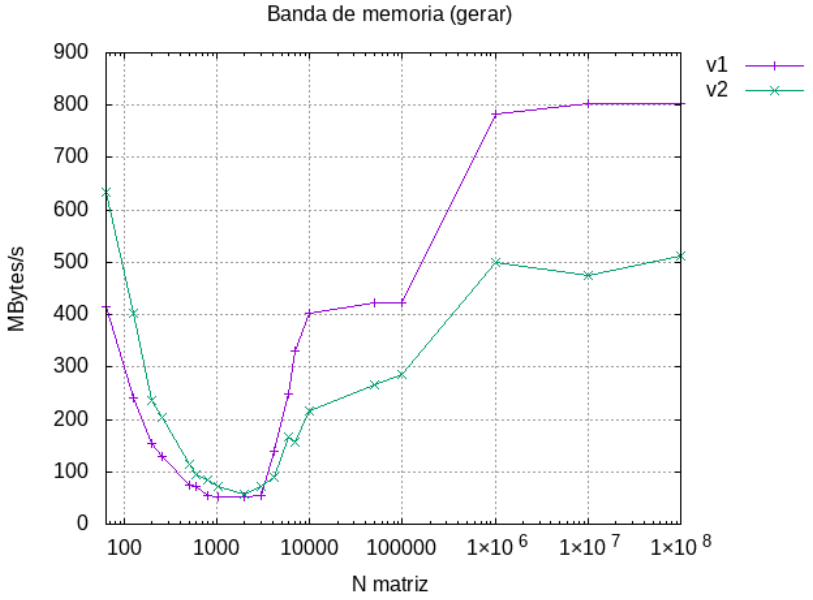
Este gráfico representa a velocidade de operações aritméticas para calcular a solução do sistema pelo método de Gauss. Com base nele, verifica-se a otimização do uso de FLOPs. Não foi realizada nenhuma modificação substancial, pois optamos por otimizar os dados com os quais o método de Gauss trabalhará.

- FLOPS (resíduos)



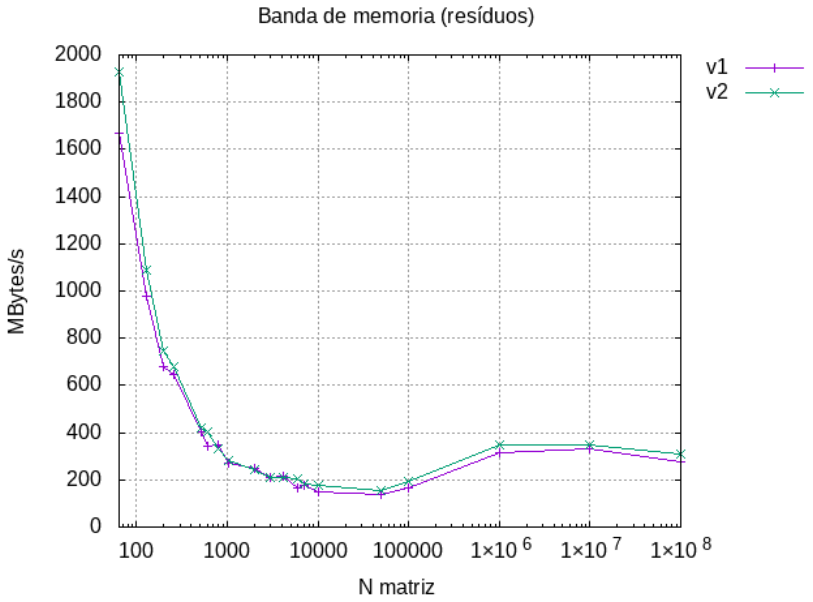
Este gráfico representa a velocidade de operações aritméticas para calcular os resíduos da função. Evidenciou-se uma melhora no desempenho referente à versão otimizada - fruto da implementação de “Unroll & Jam”.

- Memória (gerar)



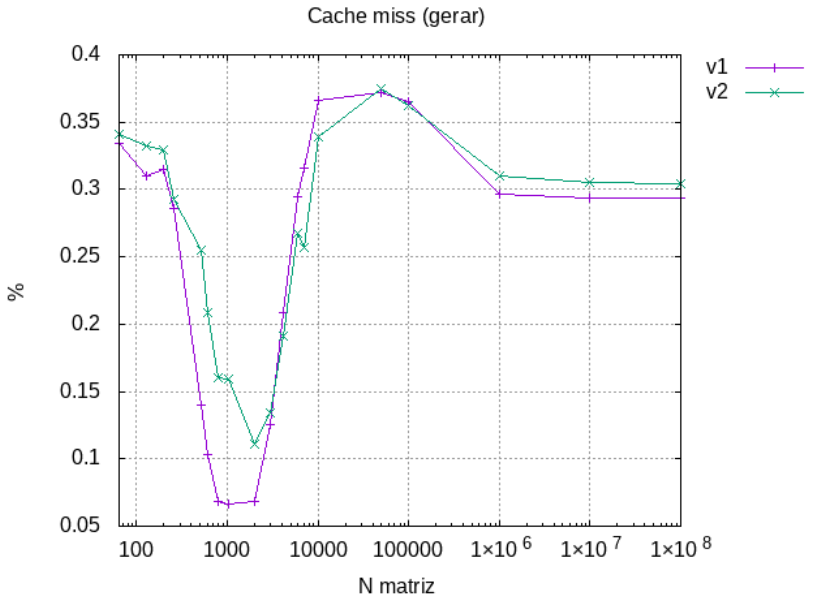
Este gráfico representa a largura de banda de memória durante a geração do sistema linear. Observa-se que a versão 1 passa a apresentar uma banda maior; isso ocorre porque na versão 2 tirou-se vantagem do fato da matriz ser simétrica.

- Memória (resíduos)



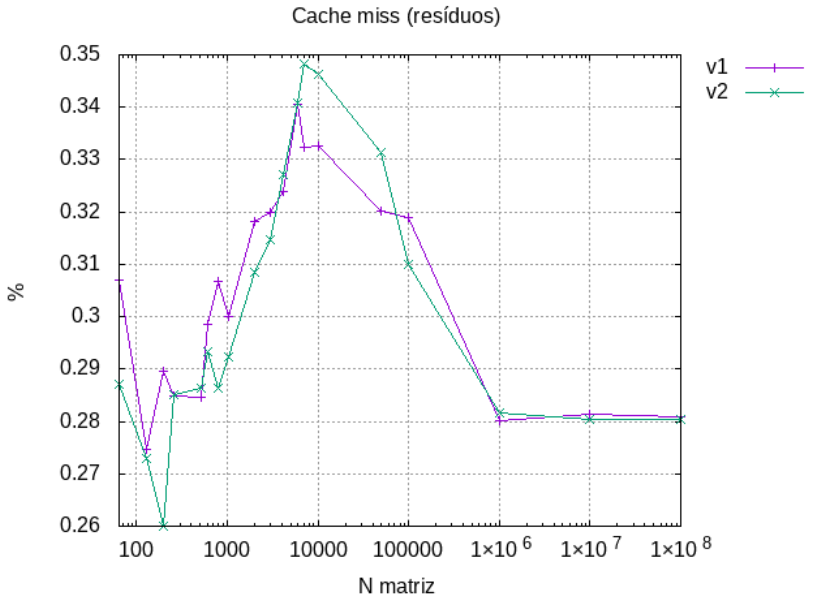
Este gráfico representa a largura de banda de memória durante o cálculo dos resíduos da função. As duas versões apresentaram um comportamento similar, pois não foram feitas otimizações que interferissem na quantidade de dados manipulados, nesta função em específico.

- Cache miss (gerar)



Este gráfico representa a porcentagem do CACHE MISS, ou seja, a frequência de vezes em que o computador procura um dado na memória cache e não encontra durante a geração do sistema linear. Neste caso, a medição foi realizada para o CACHE MISS na geração do sistema linear. Com base no gráfico e nas otimizações implementadas pode-se observar que ambas as versões apresentaram um comportamento similar para os misses de cache - possivelmente em decorrência da quantidade de pontos que não se adequa ao tamanho da cache do hardware.

- Cache miss (resíduos)



Este gráfico representa a porcentagem de CACHE MISS durante o cálculo dos resíduos da função. Acredita-se que as regiões em que a versão 2 apresentou mais misses é em decorrência da aplicação do “Unroll & Jam” para quantidades de pontos que não favorecem ao tamanho de cache do hardware utilizado para testes.

Topologia do hardware

Todos os testes foram executados na máquina h16 do DINF.

```
--------------------------------------------------------------------------------
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:            5259.68 MB
Total memory:           7826.24 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                 | |
| +---------------------------------------+ |
+-------------------------------------------+
```

5. Conclusão

Tendo em vista os resultados obtidos dos testes, podemos concluir que as otimizações de código serial e de acesso à memória foram eficazes. A versão 2 foi mais eficiente em relação ao código original - baseando-se nos parâmetros de velocidade de operações aritméticas, banda de memória, tempo médio de execução e cache miss rate.

O enfoque das otimizações foi em relação ao reaproveitamento do resultado de operações aritméticas e da paralelização em loops. Além disso, optamos por opções mais vantajosas de estruturas de dados e melhoramento nos modos de alocação - com o objetivo de armazenar os elementos das estruturas de maneira sequencial na memória.