Relatório do Trabalho 2

Bruno Krügel
GRR20206874

February 23, 2023

1 Introdução

Nesse relatório será apresentado uma avaliação das melhorias feitas no programa do trabalho 1 para melhorar o seu desempenho.

2 Mudanças no Código

Essa segunda versão corrige diversos problemas de operações duplicadas, acessos à memória com stride e instruções que causam branch dentro de loops.

Também tentei usar *looping unrolling* e *blocking* em algumas operações de matrizes, mas por causa da matriz não ser representada inteira na memória (sem as diagonais nulas), essas operações ficaram mais lentas que as versões normais. Isso será mostrado melhor na seção 3.

2.1 Mudanças no Armazenamento da Matriz k-diagonal

No trabalho 1, a matriz era armazenada que nem é demonstrado a seguir. Cada diagonal se torna uma coluna e como A é simétrico, apenas uma metade é representada.

$$A = \begin{bmatrix} 3.7916 & 0.12129 & 0 & 0 & 0 \\ 0.950542 & 1.4836 & 0.0701283 & 0 & 0 \\ 0 & 0.822375 & 0.0899469 & 0.403666 & 0 \\ 0 & 0 & 0.844203 & 4.01547 & 0.887116 \\ 0 & 0 & 0 & 0.738209 & 3.91318 \end{bmatrix}$$

$$A(Compacto) = \begin{bmatrix} 0 & 3.7916 & 0.12129 \\ 0.950542 & 1.4836 & 0.0701283 \\ 0.822375 & 0.0899469 & 0.403666 \\ 0.844203 & 4.01547 & 0.887116 \\ 0.738209 & 3.91318 & 0 \end{bmatrix}$$

Ao não armazenar nenhuma diagonal nula, é possivel economizar espaço de memória, porém, os indices precisavam ser arrumados para a matriz simplificada. Por causa disso, tinha que ter diversos *ifs* dentro dos loops de linhas/colunas, assim como no exemplo abaixo.

```
/* Calcula residuo r */
for(i=0; i<SDSimetrico->n; i++){
    r[i] = 0.0;
    for(j=0; j<SDSimetrico->n; j++){
        k = j - i + (SDSimetrico->k - 1);  // Coluna onde o numero est
        if(k >= (SDSimetrico->k*2 - 1) || k < 0) continue; //    zero.
        if(k <= (SDSimetrico->k - 2)){     // Esta em uma diagonal inferior, entao
             os indices precisam ser ajustado para sua equivalencia nas diagonais
            superiores.
            i_1 = i + k - (SDSimetrico->k - 1);
            j_1 = SDSimetrico->k - k - 1;
            r[i] -= SDSimetrico->A[i_1][j_1] * x[j];
        }
        else{   // O numero esta em uma das diagonais superiores.
```

```
                j_1 = k - (SDSimetrico->k - 1);
                r[i] -= SDSimetrico->A[i][j_1] * x[j];
            }
        }
        r[i] += SDSimetrico->b[i];
    }
```

Por causa desse problema, cheguei a conclusão que seria uma boa idea mudar como as matrizes são armazenadas para um jeito que eu poderia acessar os valores da matriz simplificada com os mesmos i e j que na original.

A nova forma de armazenar a matriz aloca um vetor com a quantidade de números que existem na matriz k-diagonal [1] e outro vetor de ponteiros que são organizados considerando números de outras linhas como os 0's. Assim você pode acessar com os mesmos i e j, mas um acesso a um espaço que teria zero retornará algum valor não esperado, ou seja, os intervalos dos *for-loops* não devem incluir esses valores. O código abaixo faz o mesmo que o exemplo anterior, calcula o resíduo, porém não tem todos aqueles ifs.

```
    /* Calcula residuo r */
    int q = (SD->k - 1) / 2;
    for(int i=0; i < SD->n; i++){
        int j_start = i - q;
        j_start = IS_POSITIVE(j_start) * j_start;
        r[i] = 0.0;
        // Apenas percorre j's que nao sao de diagonais nulas
        for(int j=j_start; j < SD->n && j<=i+q; j++)
            r[i] -= SD->A[i][j] * x[j];
        r[i] += SD->b[i];
    }
```

Em geral, todas as partes do programa são beneficiadas por não precisarem processar cada condional $n \times n$ vezes.

2.2 Mudanças na Iteração do método de Gradiente Conjugado

A primeira coisa que foi modificada foi no calculo do alfa. A multiplicação de matriz-vetor $A \cdot p$ era feita com acesso com stride na matriz A porque outra multiplicação de vetor-vetor era feito no mesmo *for-loop*. Essas duas multiplicações foram separadas em dois loops e os valores de $A \cdot p$ são armazenados em um vetor de tamanho n, com isso, não só o stride foi resolvido como também ocorre uma multiplicação de matriz-vetor a menos na interação, já que os valores desse vetor são reaproveitados no cálculo do resíduo.

Além disso, operações que não era necessárias de serem feitas em cada interação foram removidas do loop, por exemplo a inversa do pré-condicionador usado no cálculo de z é agora calculada na inicialização do pré-condicionador.

A operação para deixar o sistema linear simétrico [2] foi movida para o arquivo Metodos.c e foi implementada direto na função conjGradient. E agora ela também é beneficiada pela operações sem *ifs*.

Também foi resolvido o problema de divisões por 0 nos cálculos de alfa e beta por meio de condicionais.

3 Comparação

Foram feita a comparação do programa do trabalho 1 com duas versões do trabalho 2, uma usando *looping unrolling + jam* e *blocking* e na outra não. Você vai ver que a versão que não usa consegue entregar a resposta muito mais rápido, mesmo tendo mais problema de cache miss.

Para comparar o desempenho dos programas foi usado a ferramenta LIKWID. Foram comparadas as categorias L2 cache miss, FLOPs (DP e AVX) e tempo de execução. Como eu usei os computadores

[1] Quantidade de números na matriz k-diagonal de tamanho n = $n + 2 \sum_{i=n}^{\frac{k-1}{2}} (n - i)$

[2] $A^\top A\mathbf{x} = A^\top \mathbf{b}$

do DINF para rodar a ferramenta, não tive permissões suficiente para comparar a cache L1 e a banda de memória.

3.1 L2 Cache Miss

3.1.1 Sem *looping unrolling + jam e blocking*

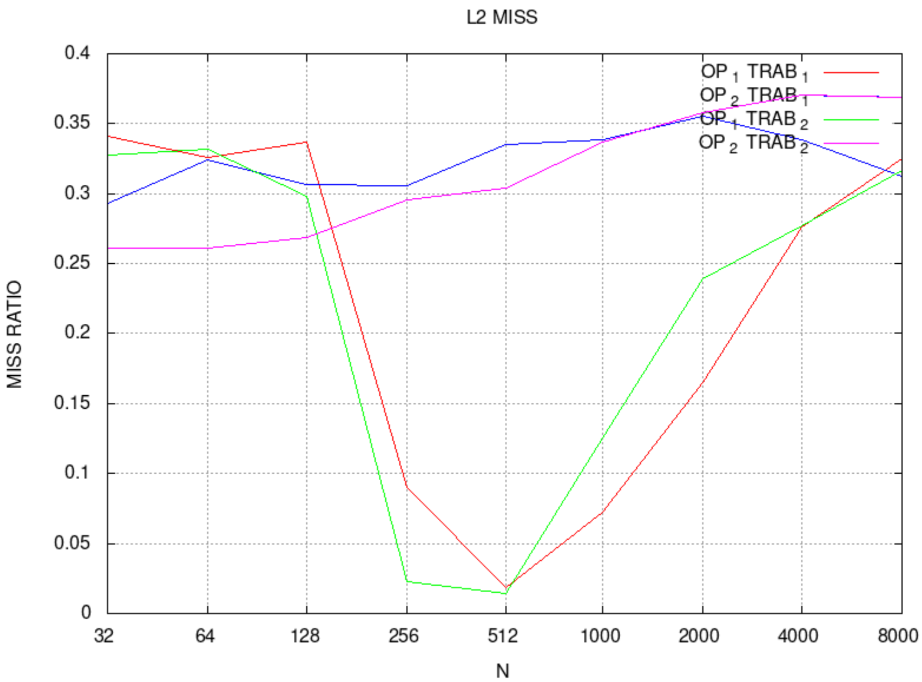

Figure 1: Comparação - L2 Cache Miss sem *looping unrolling + jam*

Como o esperado, o cache miss na L2 ficou praticamente identico entre as versões de cada trabalho.

O valor fica alto quando N ainda é baixo porque a quandade de acessos à dados ainda é pequena então as poucas vezes o dado não esta na cache aumentam muito esse valor. [3]

[3] Só lembrando que o ratio é calculado por

$$\frac{Quantidadedeacessosnãosucedidos}{Quantidadedeacessosnototal} \quad (1)$$

3.1.2 Com *looping unrolling + jam e blocking*

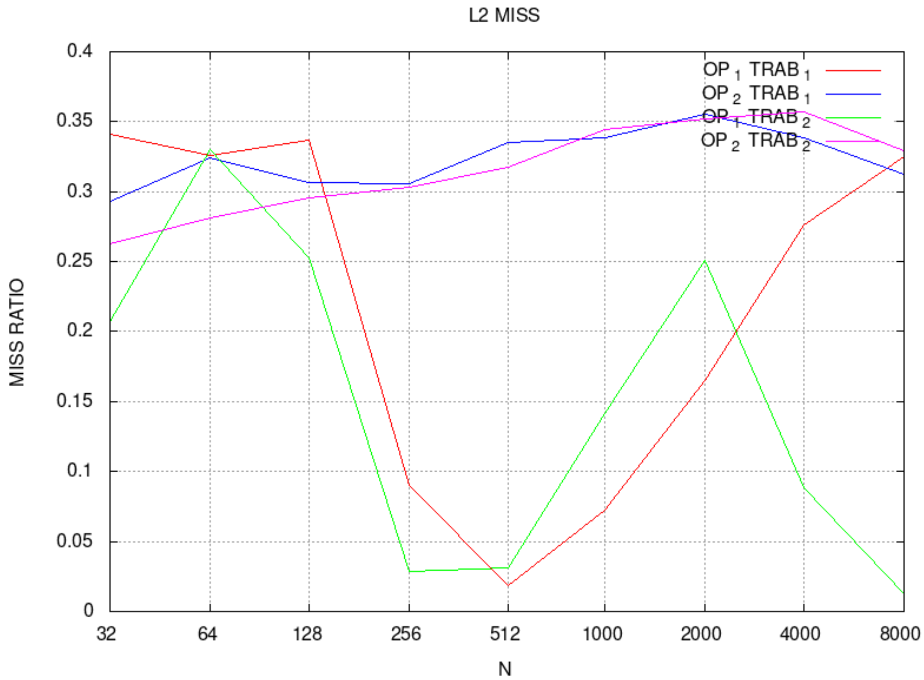

Figure 2: Comparação - L2 Cache Miss com *looping unrolling + jam e blocking*

A versão com *blocking* melhora muito o uso da cache quando o N é muito alto (observe o valor de 8000), por reutilizar a colunas já carregadas da segunda matriz da operação. O valor alto quando $N = 2000$ indica a possibilidade de *cache thrashing*.

3.2 FLOPs AVX

3.2.1 Sem *looping unrolling + jam e blocking*

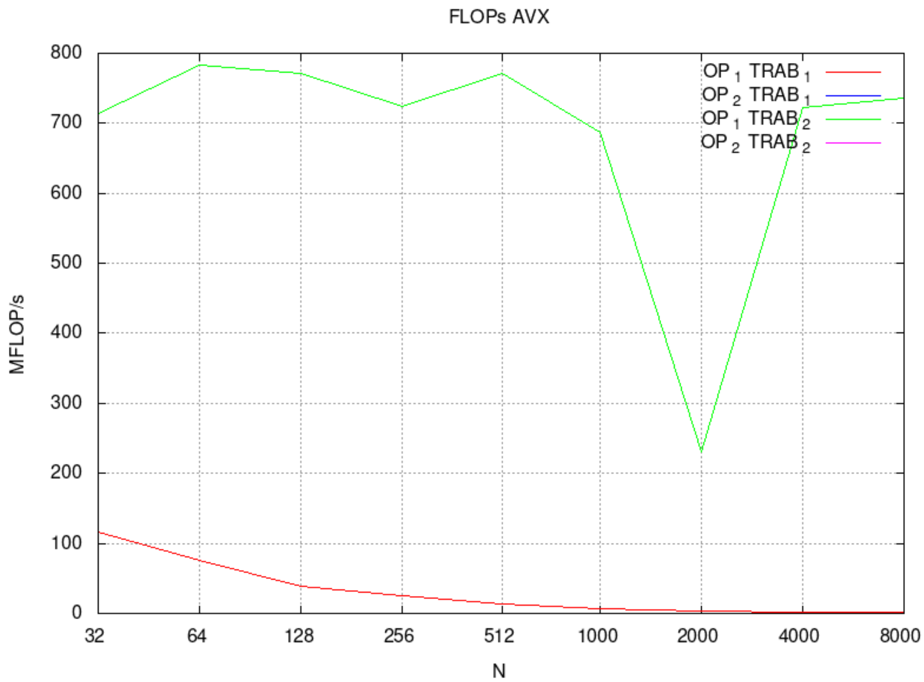

Figure 3: Comparação - FLOPs AVX sem *looping unrolling + jam e blocking*

A operação de gradiente conjudado do trabalho 2 é a única que está organizada de modo a permitir o uso de AVX. Isso pode até ser provado ao desmontar o arquivo objeto Metodos.o e procurar pelo uso de registradores %ymm ou %zmm, eles só estaram presentes na função do OP1.

3.2.2 Com *looping unrolling + jam e blocking*

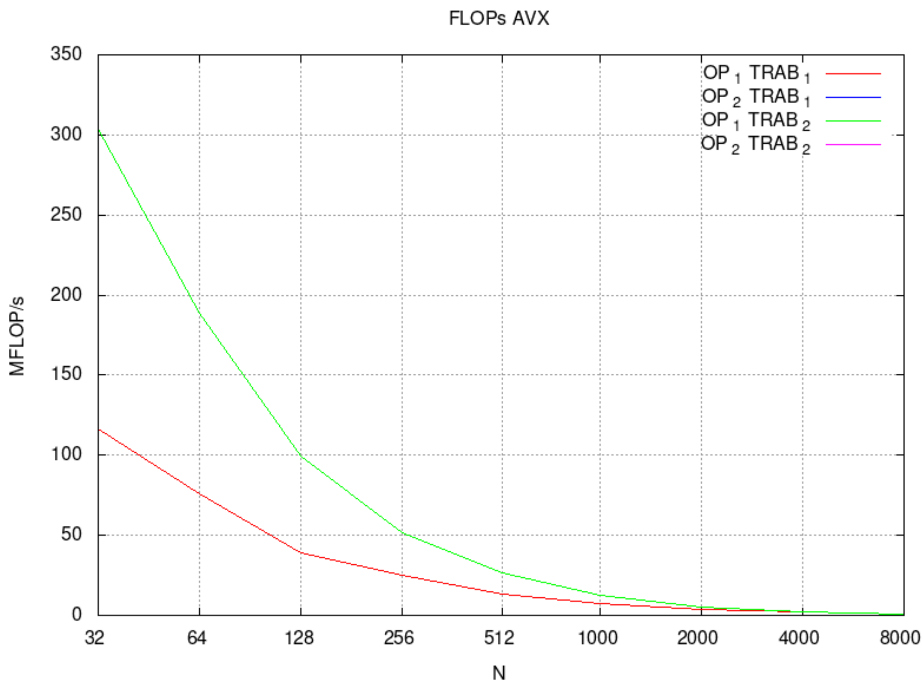

Figure 4: Comparação - FLOPs AVX com *looping unrolling + jam e blocking*

Por causa dos branchs causados pelas instruções *ifs* que tive que colocar nas funções com *looping unrolling ou blocking*, é impossível para o programa conseguir usar AVX quando N é muito grande. Se eu conseguisse remover essas instruções essa versão do trabalho 2 também conseguiria usar AVX.

3.3 FLOPs DP

3.3.1 Sem *looping unrolling + jam e blocking*

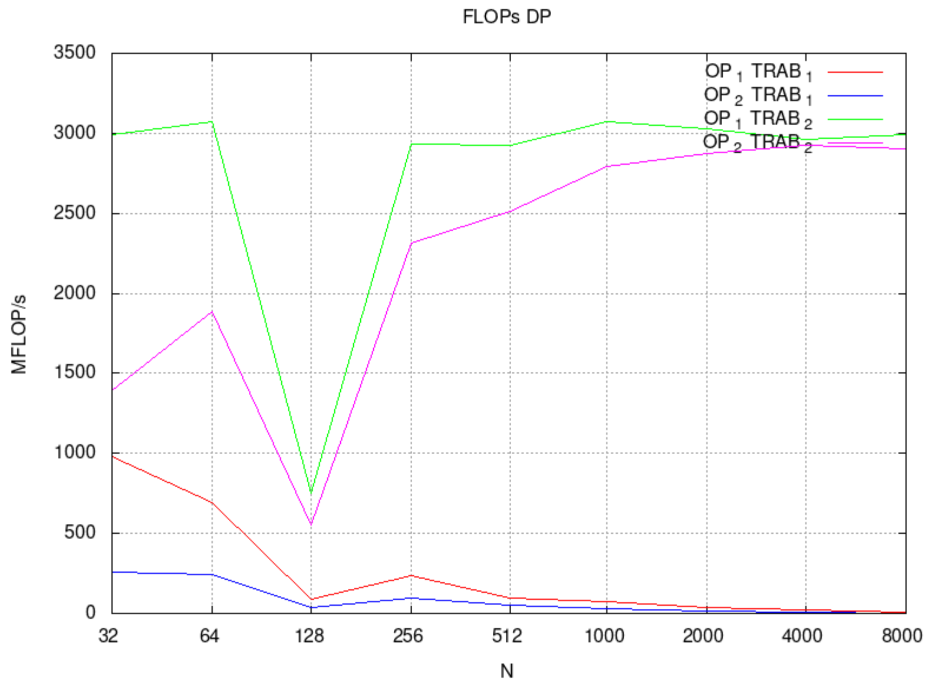

Figure 5: Comparação - FLOPs DP sem *looping unrolling + jam e blocking*

A versão do trabalho 2 consegue fazer muito mais operações de ponto flutuante por segundo que a versão do trabalho 1, isso provavelmente porque a primeira versão ficava limitada pela banda de memória, mas não temos o gráfico do uso da banda, então não da para afirmar com certeza.

3.3.2 Com *looping unrolling + jam e blocking*

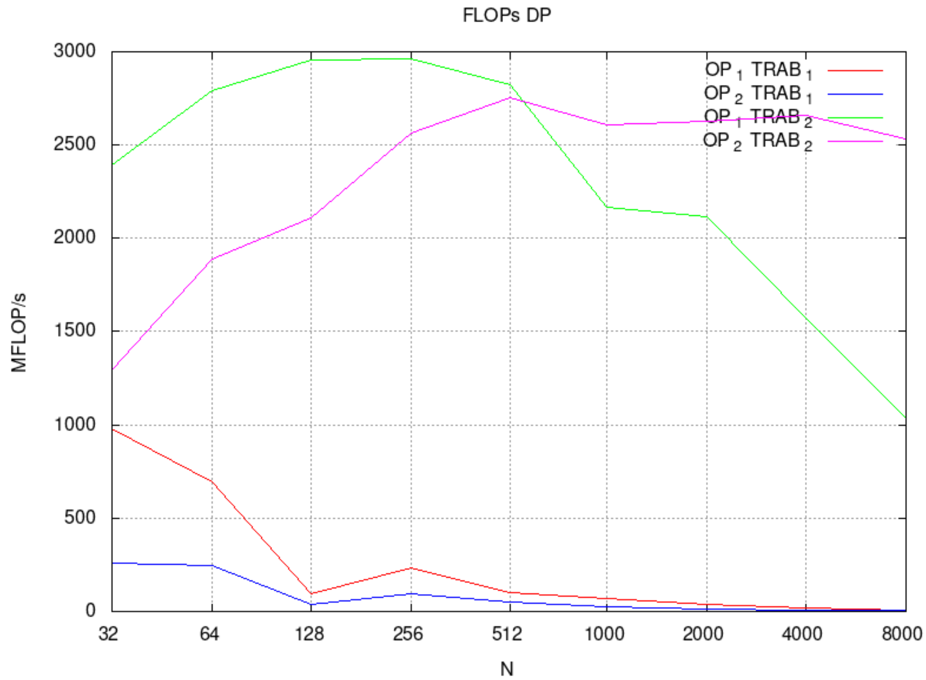

Figure 6: Comparação - FLOPs DP com *looping unrolling + jam e blocking*

A versão com *looping unrolling* usa mais FLOPs DP por que ela não consegue usar AVX e ao mesmo tempo ela não fica limitada pela banda igual ao programa do trabalho 1.

3.4 Tempo de Execução

3.4.1 Sem *looping unrolling + jam e blocking*

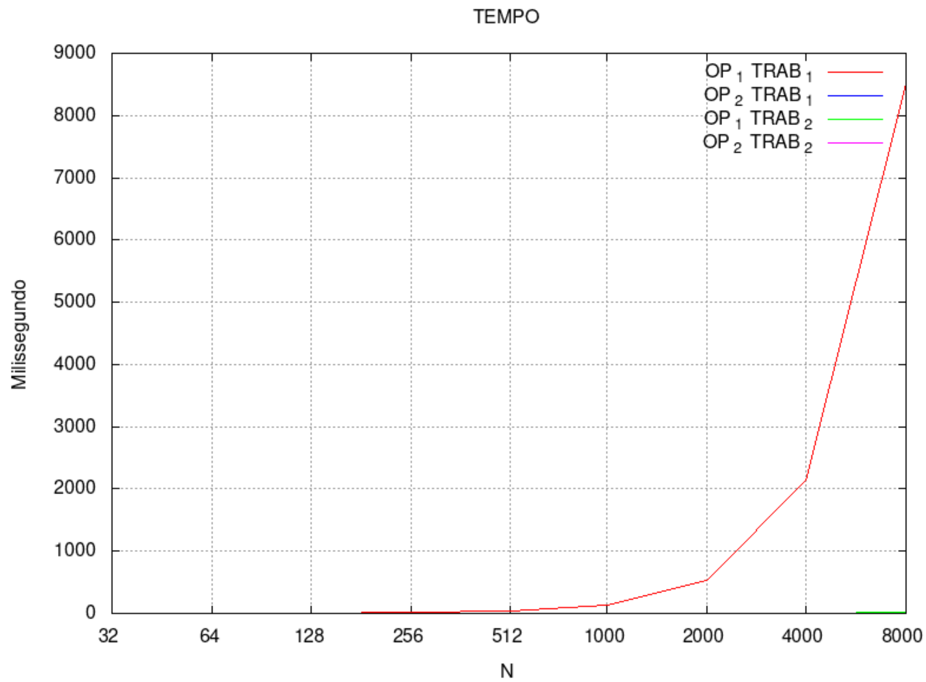

Figure 7: Comparação - Tempo de execução sem *looping unrolling + jam e blocking*

O tempo da operação 1 do trabalho 1 é tão grande que o gráfico ficou fora de escala. O gráfico abaixo é o mesmo porém sem OP1 do trabalho 1.

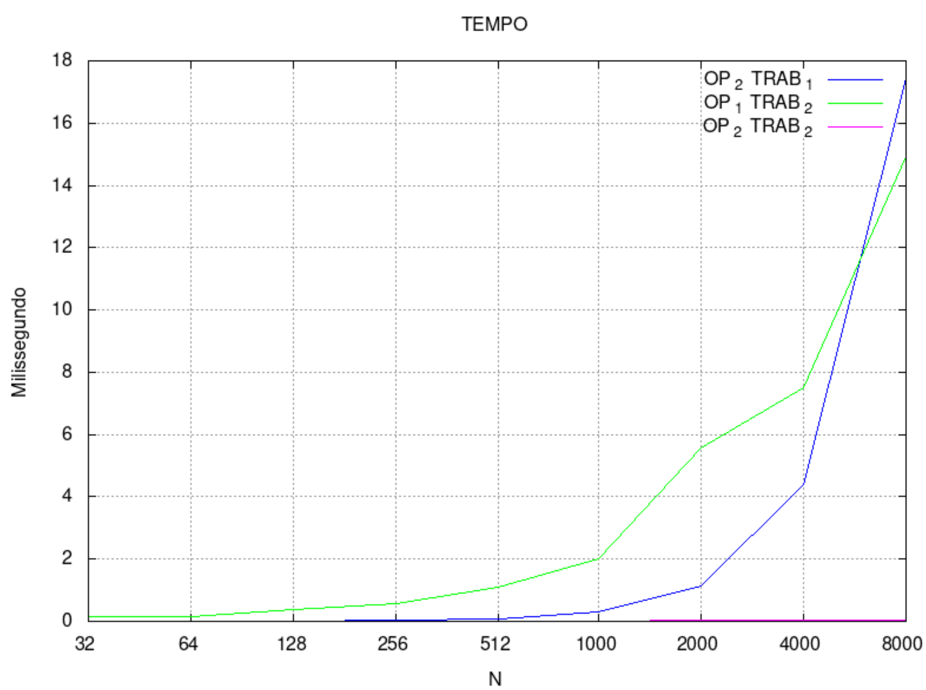

Figure 8: Comparação - Tempo de execução em escala sem *looping unrolling + jam e blocking*

E ficou fora de escala novamente. Apenas 150 interações é muito pouco para poder comparar o tempo da segunda operação do trabalho 2. Nos meus testes quando $N = 8000$, uma única interação levavá 0.1 milisegundos para ser concluida.

3.4.2 Com *looping unrolling + jam e blocking*

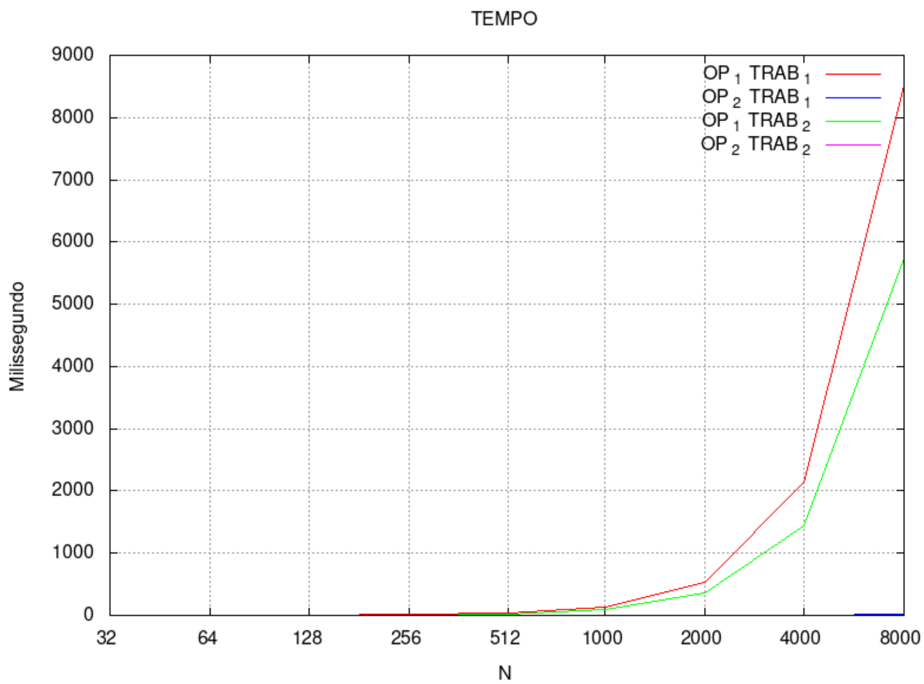

Figure 9: Comparação - Tempo de execução com *looping unrolling + jam e blocking*

Com esse gráfico podemos concluir facilmente que ter *ifs* dentro de 6 *for-loops*, do blocking, não é uma boa ideia.

4 Conclusão

Apenas reorganizando para tirar erros como acessos com strides já melhoraram muito o desempenho do programa.

Ter que otimizar a matriz para não armazenar diagonais nulas me atrapalhou muito na hora de aplicar os métodos que vimos em aula. Talvez haja como fazer, mas não consegui pensar numa solução a tempo.

A minha função de calcular resíduo também demonstrou não ter melhorado nada entre os trabalhos.