Relatório Trabalho 2
Otimização do Desempenho para Sistemas Lineares Esparsos

Introdução a Computação Científica
Universidade Federal do Paraná - UFPR
Gustavo Do Prado Silva - GRR20203942
Rafael Gonçalves dos Santos - GRR20211798
Curitiba - PR

I. Introdução

Este presente relatório tem como objetivo discorrer sobre as alterações realizadas com a versão 1 do projeto, aleḿ de comparar o desempenho dos dois programas desenvolvido

II. Alterações

1. Matriz pré-condicionadora

O armazenamento da matriz pré-condicionadora, agora é feito em um vetor unidimensional, que guarda os valores da diagonal principal.

Motivo*:* Como a matriz pré-condicionadora é uma matriz diagonal, os valores nulos estavam sendo armazenados sem necessidade. Além de que operações que envolvem apenas vetores são menos custosas do que aquelas que envolvem matrizes.

2. Inline

Declaração de funções no programa utilizando a palavra chave *inline*

Motivo: Para algumas funções que estão localizadas dentro de laços de repetição, foi utilizada a palavra chave *inline* com objetivo de não quebrar o pipeline durante a execução do programa. Aumentando a velocidade e diminuindo a sobrecarga associada à chamada da função.

3. Restrict

Parâmetros de função que em nenhum momento do programa vão possuir ponteiros apontando para o mesmo lugar receberam a palavra chave *restrict*.

Motivo: Essa palavra acaba “dizendo” para o compilador que durante essa função não ocorrerá *pointer aliasing* entre os ponteiros especificados, permitindo otimizações feitas pelo próprio compilador.

4. **Armazenamento da matriz em vetor**

Ambas as matrizes original e a alterada agora são armazenadas em vetores unidimensionais, que guardam apenas os valores não nulos de cada uma.

Motivo*:* Aloca espaços contíguos na memória, melhorando a localidade e reduzindo a quantidade de operações de alocar e desalocar.

5. **Armazenamento das diagonais não nulas**

MATRIZ ORIGINAL:

Original

1	2	0	0	0
3	4	5	0	0
0	6	7	8	0
0	0	9	10	11
0	0	0	12	13

Transposta

1	3	0	0	0
2	4	6	0	0
0	5	7	9	0
0	0	8	10	12
0	0	0	11	13

Para guardar a matriz original, ao invés de utilizar uma matriz n*n, utilizamos um vetor k*n, que armazena apenas as diagonais da matriz. Consequentemente a transposta será n*k, e ambas seguem a seguinte forma:

Original

0	1	2
3	4	5
6	7	8
9	10	11
12	13	0

Transposta

0	3	6	9	12
1	4	7	10	13
2	5	8	11	0

A matriz diagonal original fica salva na struct do sistema linear, em outra struct MatDiag_t que possui:

- Vetor n*k (A)
- B original (b)
- Vetor com posições de início originais das linhas (jstart)
- Vetor com posições de fim originais das linhas (jend)

Os dois vetores (jstart e jend) servem para identificar em que posição os elementos originalmente estariam, visto que após algumas linhas, os elementos não-nulos começam a aparecer depois da primeira coluna. Isso é útil para o caso de multiplicação com vetores (como no resíduo por exemplo), onde precisamos multiplicar o 1º elemento de cada linha pelo 1º elemento no vetor, mas caso a linha comece com 0s, nós "pulamos" e vamos para os elementos não-nulos, que são multiplicados pelo vetor a partir da posição de jstart.

MATRIZ SIMÉTRICA:

1	2	3	0	0
2	4	5	6	0
3	5	7	8	9
0	6	8	10	11
0	0	9	11	12

1	2	3
4	5	6
7	8	9
10	11	x
12	x	x

(Obs: posições marcadas com x **não existem no vetor**)

Possui k linhas de tamanho k e depois elas vão diminuindo até o tamanho 1.

Guarda apenas da diagonal principal para cima, sem nenhum zero
A matriz simétrica fica salva na struct do sistema linear, em outra struct MatSim_t que possui:

- Vetor de tamanho $n \times k + \frac{k^2-k}{2}$ (A)
- B após ser multiplicado pela transposta (b)
- Vetor com posições de início originais das linhas (jstart)
- Vetor com posições de fim originais das linhas (jend)

As operações que precisaram ser alteradas foram: transposição e multiplicação pela transposta, multiplicação de matriz por vetor (foi adicionada a função multMatSimVet) e o cálculo do resíduo com a matriz diagonal (adicionada a função calcResiduo).

Motivo*:* As matrizes possuíam muitos elementos 0s em memória, principalmente matrizes grandes (1000, 2000, 8000), o que acabava sendo um espaço inútil. Então, guardamos apenas os valores não nulos, utilizando assim menos memória. Por exemplo, para n = 8000 e k = 7, na versão 1 teríamos 64.000.000 posições de memória (8000 x 8000) enquanto na versão 2 temos 56021 posições ($8000 \times 7 + \frac{7\times7-7}{2}$) para a matriz diagonal e mais 56000 da matriz com as diagonais, ou seja, 112.021 no total, que é 0,1750328125% do tamanho original (64.000.000).

6. **Loop Unrolling**

Em algumas funções, nas quais foi perceptível uma melhora no tempo, foram adicionados manualmente Loop Unrolling com fator de desenrolamento igual a 4. Foram adicionadas nas funções:

- multVetVet
- normaL2
- No cálculo da *M_trans* (Matriz Transposta)

Nas outras funções, o desenrolamento não surtia um efeito positivo, então conclui-se que essa pŕatica estava sendo bem executada pelo compilador.

Motivo*:* Essa ação acaba diminuindo o número de repetições do laço, levando a uma diminuição das comparações do loop. Além de que, as operações estão sendo processadas em paralelo, diminuindo o tempo que se gasta para realizar todas elas

III. Comparações

I. FLOPS-DP AVX

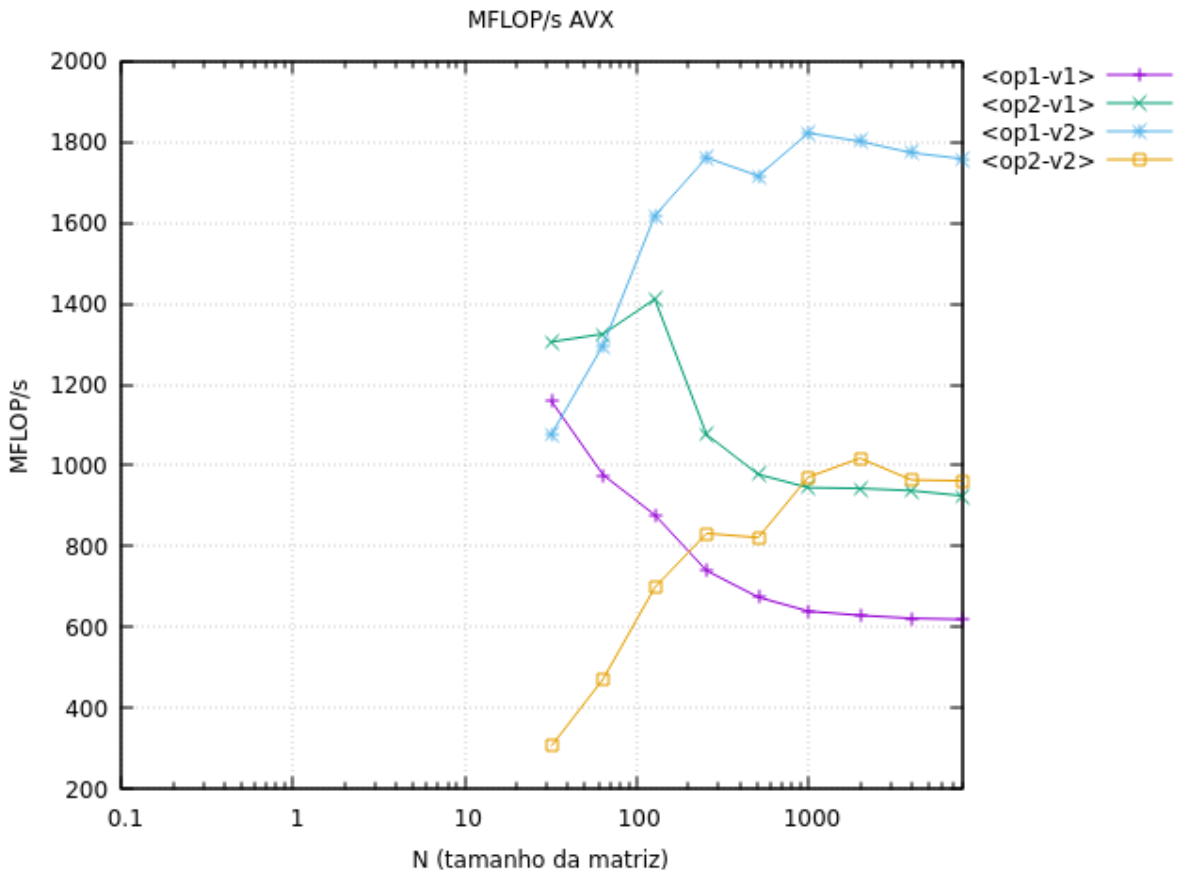

Gráfico com operações de ponto flutuante que usam instruções AVX.

- **op1-v1**: Na primeira versão as funções chamadas conseguem ser vetorizadas (é possível observar ao usar a opção -fopt-vec no gcc), porém não são muitas (apenas 3) por isso mesmo quando as instruções são utilizadas, não são em grandes quantidades.

- **op2-v1:** Ambas as funções de resíduo e da norma conseguem ser vetorizadas, porém também são em pequena quantidade.

- **op1-v2:** Em algumas das operações foi realizado Unroll e Jam de tamanho 4, o que possibilita o melhor uso de instruções AVX, já que estas utilizam 256 bits (4 doubles). Com isso e a mudança na forma do cálculo da multiplicação com a matriz, há mais operações que podem ser vetorizadas, assim utilizando mais essas instruções.

- **op2-v2:** Na nova função de cálculo do resíduo há 2 laços com possibilidade de vetorização, além da norma L2, onde foi realizado Unroll e Jam com tamanho 4, que melhora o uso dessas instruções. Porém, há menos dados, visto que o tamanho da matriz é menor, então o crescimento é maior mas menos operações são necessárias.

II. FLOPS-DP

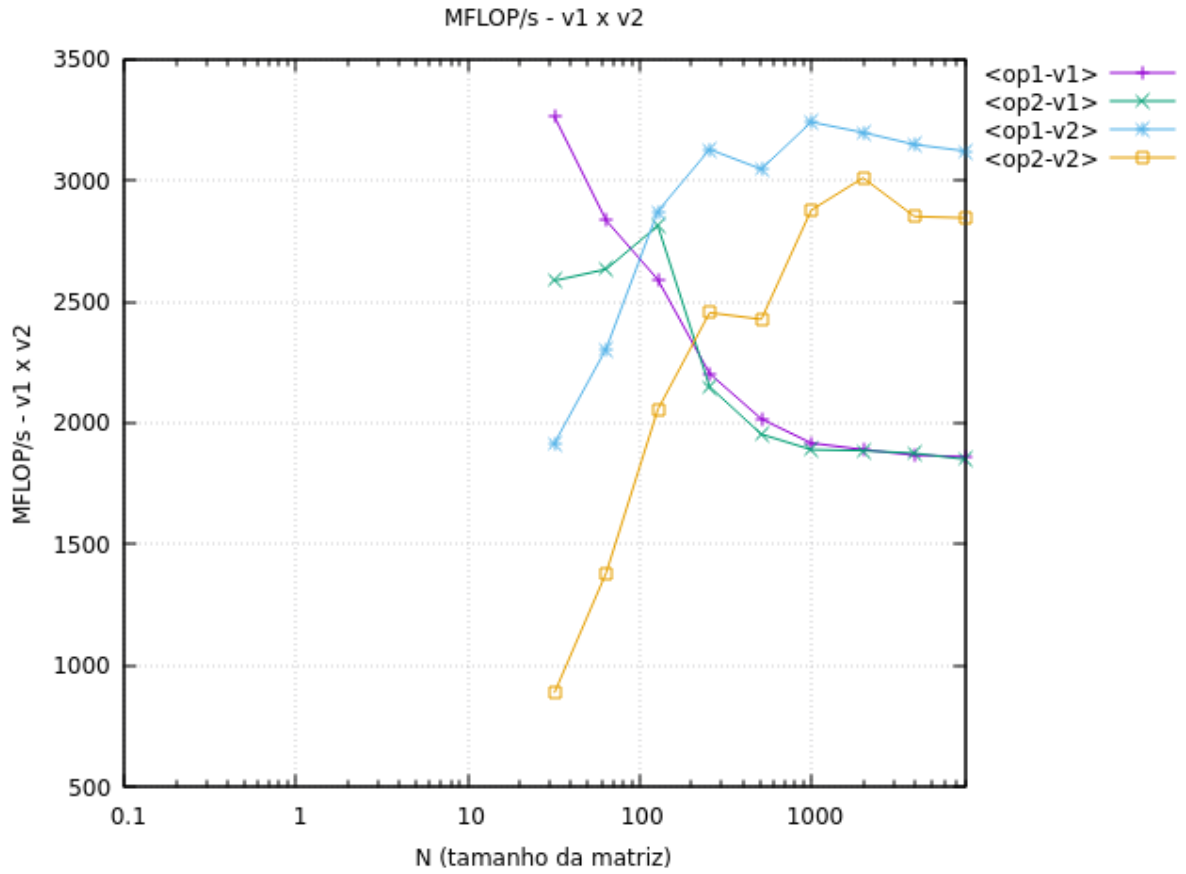

Gráfico com operações de ponto flutuante.

- **op1-v1**: Na versão original, devido a grande quantidade de acesso à memória e o grande tamanho das matrizes, é possível realizar poucas operações por segundo, já que é necessário esperar o valor ser carregado da memória. Quanto maior a matriz, mais os valores têm que ser carregados, visto que a matriz ou vetor não cabem na cache, e mais o Pipeline de instruções tem de ser parado para aguardar a memória.

- **op2-v1:** Acontece o mesmo que no anterior, quanto maiores o vetor de soluções e a matriz, mais é necessário esperar o valor ser carregado da memória, diminuindo a quantidade de operações por tempo.

- **op1-v2:** Devido ao Unroll e Jam em algumas operações, o preenchimento do Pipeline de instruções é melhor, e devido ao fato da matriz simétrica ser menor, são necessários menos acessos à memória principal, consequentemente o tempo de espera ao carregar valores é menor e mais instruções de operações numéricas podem ser realizadas por ciclo de instrução.

- **op2-v2:** Há menos valores na matriz diagonal, sendo então necessários menos acessos à memória e também há Unroll e Jam no cálculo da norma L2, o que preenche mais o

Pipeline de instruções. Porém, a quantidade de dados é menor, o que faz o crescimento ser maior mas com menos operações que a primeira versão.

III. DATA CACHE MISS RATIO

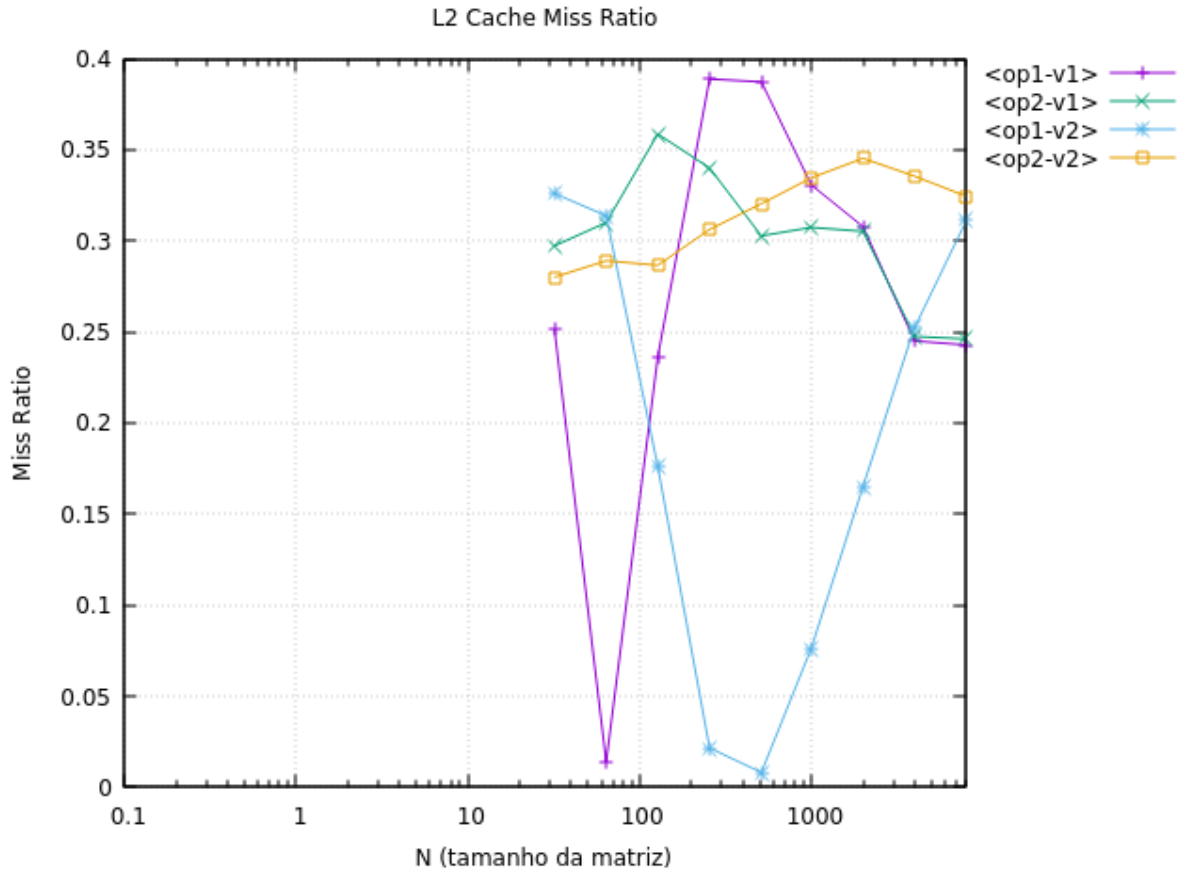

Gráfico de cache miss ratio.

- **op1-v1:** A falta de alinhamento na memória pode causar uma utilização de cache não muito eficiente. Entretanto, como as funções possuem um menor número de acesso a diferentes itens na memória, os blocos trazidos da matriz acabam sendo mais bem utilizados à medida de n.

- **op2-v1:** Situação muito parelha com a op2, onde o cálculo desse resíduo inicializa mais variáveis e acessa diferentes partes da estrutura do sistema linear.

- **op1-v2:** No início o alinhamento das memórias, junto com a estrutura de dados mais enxuta ajuda a ter menos cache miss ao acessar a memória. Entretanto nessa segunda versão existem funções que estão iterando, cujo algoritmo está dependente de mais informações diferentes. Funções que chamam outras funções que consequentemente dependem de outras variáveis acabam disputando esse espaço na cache, aumentando o miss rate à medida que n aumenta.

- **op2-v2:** Na mesma linha de raciocínio da op1**,** ocorre esse aumento natural em relação a v1, pois o cálculo do resíduo está mais complexo e depende de mais informações diferentes**,** como o fato de o vetor de soluções ser acessado com um deslocamento devido a maneira como a matriz foi reorganizada, não sendo acessado sequencialmente (do início ao fim) a cada iteração.

IV. BANDA DE MEMÓRIA

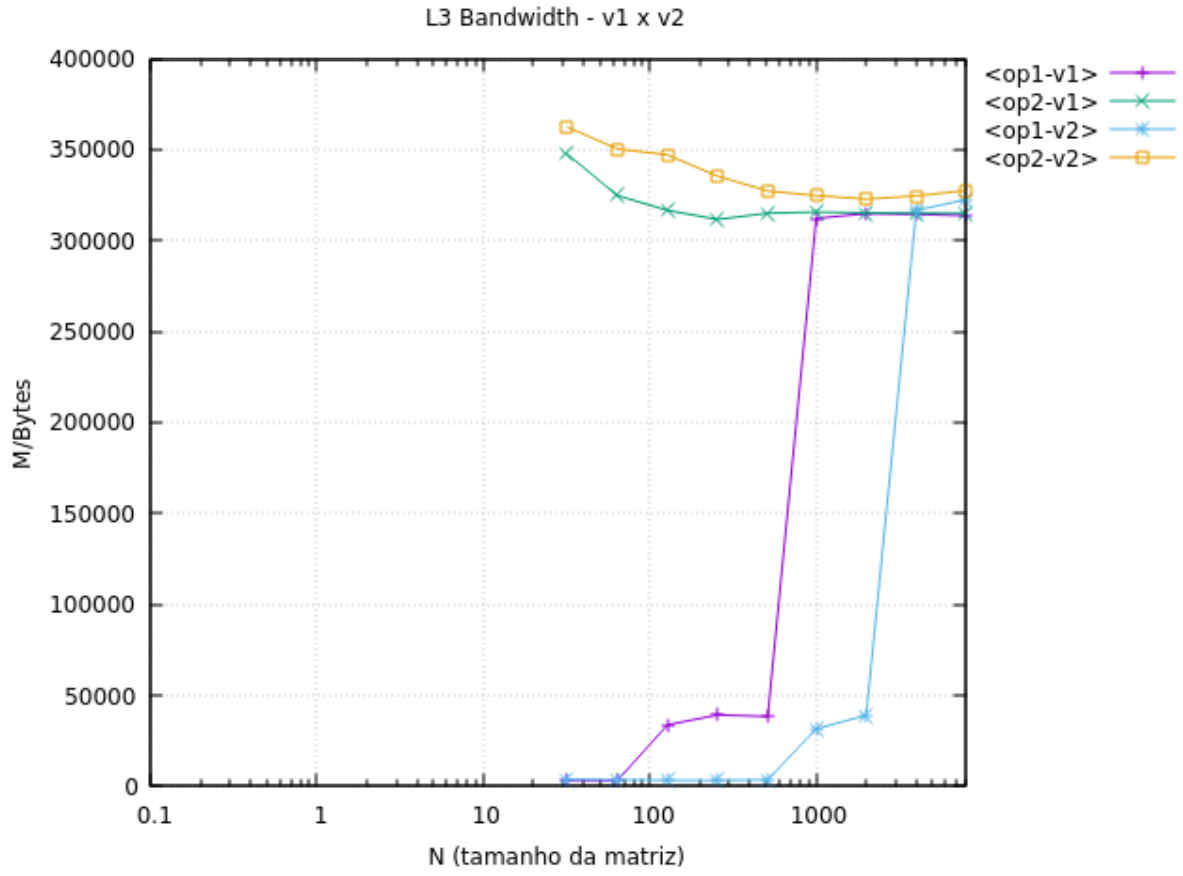

Gráfico de banda de memória.

- **op1-v1:** A quantidade de dados transferidos do processador para a cache é muito parecida se comparada com a v2. Porém, a v1 começa a mexer com um número maior de elementos antes, necessitando um uso da cache L3 antes.

- **op2-v1:** Situação muito parecida com a v2. Entretanto, por não possuir nenhum tipo de desenrolamento no laço e otimização mais avançada, a banda de memória acaba sendo inferior.

- **op1-v2:** A quantidade de dados transferidos do processador para a cache é muito parecida se comparada com a v1. Entretanto, a v2 começa a mexer com um volume maior de informações mais tarde, pois a nova estrutura de dados diminui a necessidade.

- **op2-v2:** Acaba mexendo apenas com operações vetoriais, conseguindo ter uma alta banda de memória. O algoritmo de v2, além de possuir um Unroll no cálculo da norma, ele tem também otimizações que fazem ele possuir uma taxa maior.

V. TESTE DE TEMPO

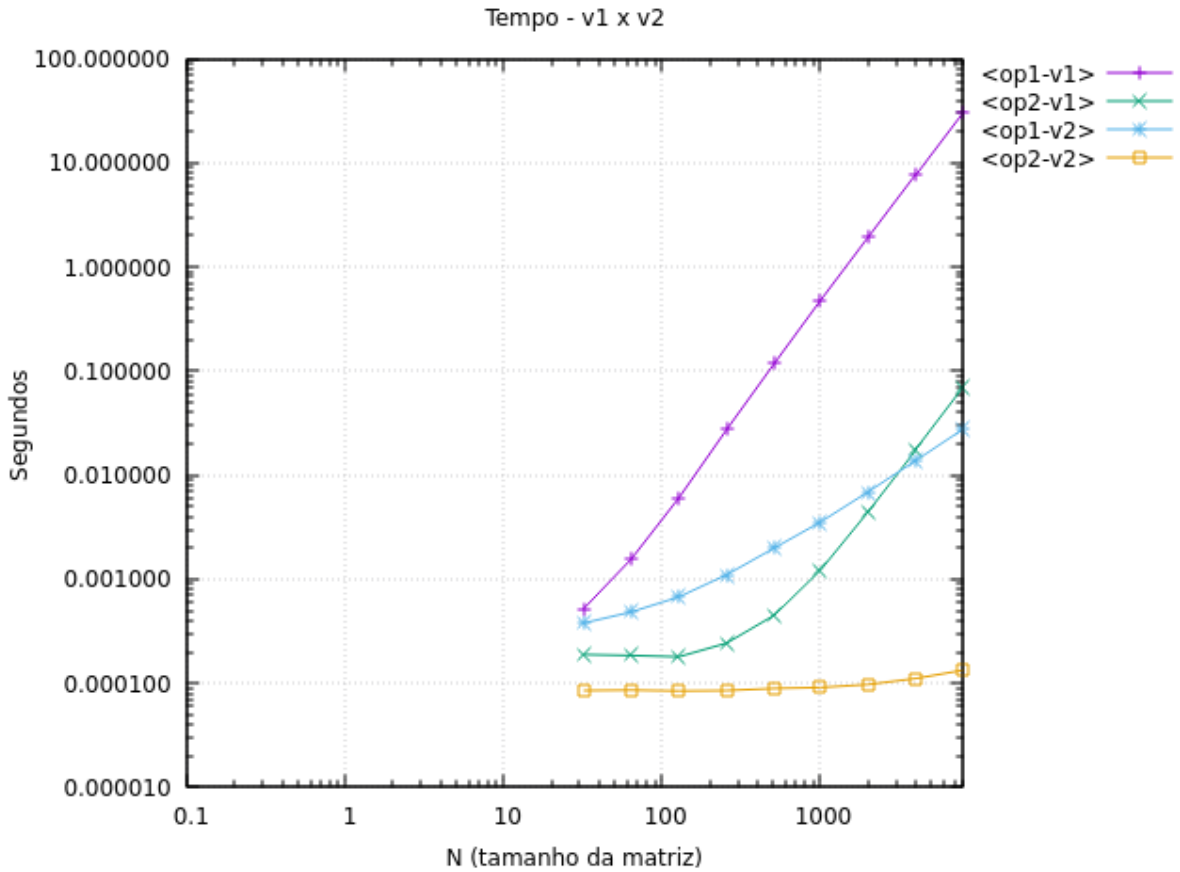

Gráfico de teste de tempo.

- **op1-v1:** O tempo é muito alto, pois as matrizes são armazenadas por completo, isso traz o problema onde operações envolvendo matrizes acabam tendo natureza cúbica, além de vários cálculos serem realizados com valores nulos.

- **op2-v1:** O tempo se difere bastante da segunda versão, pois, como em op1, para calcular o resíduo é necessário realizar operações matriciais, que aumentam quadraticamente a medida em que n cresce.

- **op1-v2:** O tempo é muito menor, devido ao novo modo de armazenamento da matriz principal, que apenas realiza operações utilizando os valores das diagonais não-nulas.

- **op2-v2:** O algoritmo para a resolução desse problema foi alterado para calcular o resíduo em um sistema tridiagonal, para se adaptar à nova estrutura de dados. Por utilizar apenas operações entre vetores o tempo acaba sendo bem menor e o cálculo mais eficiente.

VI. Registradores AVX

As operações AVX, como mostra o gráfico acima, são bem mais utilizadas nessa nova versão do que a anterior. As flags de compilação que vetorizam as operações e/ou fazem o unroll em tempo de compilação auxiliam esses registradores a realizarem instruções de maneira paralela. Além disso, quando tinha-se um resultado positivo, foi utilizado a técnica de Unroll e Jam para melhorar o cálculo das operações. Usando sempre o fator de desenrolamento igual a 4, que era o mais eficiente, levando em conta que estamos trabalhando com 256 bits (4 doubles). Ao compilar com a flag -fopt-vec, podemos ver que mais funções/operações são vetorizadas na segunda versão, fazendo a proporção e progressão do uso ser maior, mas a taxa de dados é menor devido aos laços serem mais curtos e trabalharem com menos dados.

VII. Arquitetura e topologia da máquina:

```
--------------------------------------------------------------------------------
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: 3402.5 MB
Total memory: 7840.09 MB
