

Otimização de Desempenho para Sistemas Lineares Esparsos com Pré-condicionantes

ISADORA BOTASSARI DE SOUZA (GRR20256872)

VICTOR GARCIA RIBEIRO (GRR20203954)

Departamento de Informática - Universidade Federal do Paraná (UFPR)

1. Introdução

O objetivo deste trabalho é aplicar otimizações de código de operações computacionais chave dentro do algoritmo do Método dos Gradientes Conjugados (CG), visando calcular obter maior eficiência no cálculo de soluções de sistemas lineares da forma $Ax=b$. Especificamente, busca-se aprimorar e avaliar o desempenho do programa computacional desenvolvido no 1º Trabalho Prático.

As seguintes operações foram alvo de otimizações:

- Op1: Cálculo da Iteração Central do Método de Gradiente Conjugado com o Pré-condicionador de Jacobi aplicado;
- Op2: Cálculo do resíduo do sistema: $R = b - Ax$.

Esse par de operações apresenta grande custo computacional, tanto por serem operações intensivas de cálculo, quanto por ocasionarem um extenso tráfego de memória. Esse custo se intensifica conforme o tamanho N da entrada cresce, refletindo no tamanho da matriz de coeficientes. Desta forma, essas são seções do código nos quais a otimização é interessante.

2. Otimizações Implementadas

Nesta seção, serão explicadas as otimizações aplicadas ao código na versão v2, descrevendo o motivo de seu impacto positivo no desempenho e os detalhes da implementação. As otimizações focaram em mudanças dos cálculos numéricos e no gerenciamento de memória do programa.

2.1 Representação Matrizes de Coeficientes

A otimização mais significativa na versão v2 foi a alteração da representação das matrizes de coeficientes, que passaram a armazenar apenas as diagonais não-nulas, utilizando a estrutura DiagMat. Sendo que essa estrutura tem: n (tamanho da matriz), k (quantas diagonais estão sendo guardadas), offsets (que dizem a posição de cada diagonal em relação à principal: abaixo é negativo e acima é positivo) e diags (os vetores com os valores de cada diagonal). Assim, em vez de armazenar todos os $n \times n$ elementos, ela guarda só k vetores de tamanho n , economizando muita memória.

Dessa forma, em vez de alocar uma matriz completa de dimensão $n \times n$ (o que exigiria espaço $O(n^2)$), a estrutura aloca apenas k vetores de tamanho n , resultando em um espaço na memória de $O(k \cdot n)$, onde k (com valor fixo de 7) é sempre menor que n . Essa otimização

também reduz significativamente o número de operações aritméticas nas multiplicações matriz-vetor, já que as operações foram adaptadas para iterar apenas sobre as diagonais que realmente possuem dados, evitando operações com elementos nulos e diminuindo acessos desnecessários à memória.

```
// Representação de matrizes k-diagonais: armazenamos apenas as diagonais não-nulas
typedef struct {
    int n;          // dimensão da matriz (n x n)
    int k;          // número de diagonais armazenadas (ímpar)
    int *offsets;   // offsets das diagonais (ex.: -b..0..+b)
    real_t **diags; // cada diags[d] é um vetor de tamanho n com os elementos da diagonal
} DiagMat;
```

Figura 1 - Estrutura DiagMat para matrizes k-diagonais

Tanto a matriz A , originalmente k -diagonal, quanto a matriz $ASP = A \times A^T$ são alocadas nesse formato e como a combinação de diagonais durante a multiplicação pode gerar novos offsets, a ASP passa a ter até $2k - 1$ diagonais, todas devidamente registradas na própria estrutura. Com essa representação, todas as operações mais custosas foram reescritas para iterar exclusivamente sobre as diagonais existentes. O produto matriz-vetor (`prodMatVet`), o cálculo da matriz ASP (`genSimetricaPositiva`) e o cálculo do resíduo $r = b - Ax$ (`calcResiduoSL`) deixaram de percorrer a matriz inteira. O mesmo vale para as matrizes auxiliares do pré-condicionador (D , L , U e M), que também utilizam offsets e diagonais explícitas (principal, superior e inferior) para acelerar os cálculos.

Essa mudança trouxe três benefícios diretos. Primeiro, economiza muita memória, pois em vez de alocar matrizes $n \times n$ com diversos elementos nulos, o código passa a usar apenas $O(n \cdot k)$ posições. Segundo, reduz diretamente o custo computacional de algumas operações, já que operações como Ax deixam de ser $O(n^2)$ e passam a na prática custar $O(n \cdot k)$. E por fim, melhora a localidade de memória, pois os dados passam a estar organizados em vetores contíguos (as diagonais), evitando acessar zeros desnecessários.

2.2 Otimização e Vetorização na Op1

A principal diferença entre a versão 1 e a versão 2 no método de Gradiente Conjugado com Pré-condicionador de Jacobi está na forma como a matriz é representada e em como a função `gradientesConjugadosPrecond` foi otimizada. Na v1, o método trabalhava com matrizes $n \times n$ com elementos nulos (`real_t **A` e `real_t **M`), enquanto na v2 elas passaram a ser armazenadas na estrutura `DiagMat`, composta apenas pelas diagonais não nulas e seus respectivos deslocamentos. Isso fez com que operações como a multiplicação matriz-vetor e o cálculo do resíduo interno da função deixassem de percorrer toda a matriz, reduzindo o custo computacional de $O(n^2)$ para $O(k \cdot n)$. Além disso, na v2 os seguintes cálculos foram otimizados: o código passa a manter localmente o offset e o ponteiro da diagonal atual, permitindo acessar `diag[i]` de forma sequencial, a função que calcula o resíduo usada na função foi reescrita (`calcResidOtim`) para subtrair diretamente Ax de b dentro do próprio loop (evitando leituras e escritas desnecessárias na memória).

Outra melhoria importante foi a inserção de `restrict` nos ponteiros utilizados nas funções mais críticas como `dot`, `norma` e `prodMatVet`. Isso sinaliza ao compilador que não

existe aliasing entre os vetores, facilitando a geração de código vetorizado (SIMD) quando se compila com `-O3 -march=native -mavx -fopt-info-vec`. Alguns ajustes nos laços, como tirar inicializações de dentro dos loops e guardar ponteiros em variáveis locais, reduziram algumas das checagens desnecessárias e auxiliaram o compilador a otimizar melhor o pipeline.

O pré-condicionador de Jacobi continua conceitualmente igual: na v1 ele usava a diagonal extraída de uma matriz $n \times n$, e na v2 passa a usar diretamente a diagonal armazenada em `DiagMat`. Todo o fluxo do método, o cálculo de $r^t z$, o tamanho do passo, atualização de x e r , atualizar z , cálculo de β e da nova direção de busca permanecem iguais, apenas executado menos cálculos do que na v1.

Assim, a v2 mantém o mesmo algoritmo de Gradiente Conjugado pré-condicionado, mas para de usar uma matriz com muitos elementos nulos e passa a usar uma estrutura com as diagonais não nulas, reduz o número de cálculos feitos dentro do método e habilita vetorização. O resultado é um aumento de throughput e uma redução significativa do tempo por iteração, especialmente nas rotinas internas onde o custo computacional é maior.

Uma observação importante foi que além de reduzir o custo computacional, a v2 produziu valores de resíduo menores quando comparada à v1, o que indica que o método do Gradiente Conjugado está mais preciso em v2 do que em v1. Para testar, criei um script bem simples para rodar v1 e v2 com $n=1000$ e $n=2000$, e depois extrai a norma euclidiana do resíduo usando `grep`. Na figura abaixo, podemos observar os resultados do teste:

```
n=1000 | v1 resid (euclidiana): 0.54350048
n=1000 | v2 resid (euclidiana): 0.0093647599
n=2000 | v1 resid (euclidiana): 2.6789879
n=2000 | v2 resid (euclidiana): 0.02193192
```

Figura 2 - Comparação do valor do resíduo entre v1 e v2

Na nossa visão essa melhoria ocorreu porque a v2 faz bem menos operações desnecessárias. Na v1, algumas funções usadas no método do Gradiente Conjugado (produto matriz-vetor e o cálculo do resíduo no método) percorrem toda a matriz com diversos elementos nulos, fazendo diversas operações com zeros e acumulando erros de arredondamento. Já na v2, apenas as diagonais com elementos são usadas, o que reduz drasticamente o número de operações no método e, conseqüentemente, diminui a propagação de erros numéricos.

2.3 Otimização no Cálculo e Inicialização da Op2

A segunda otimização se concentra em tornar mais eficiente tanto o cálculo do resíduo dentro do método do Gradiente Conjugado quanto o cálculo final da norma euclidiana do resíduo. Na v2, o cálculo dentro do método passou a ser feito numa função própria (`calcResidOtim`), que realiza toda a operação em um único passo: primeiro o vetor de resíduo é preenchido com b , depois cada diagonal não nula da matriz é percorrida. Para cada diagonal, é guardado o deslocamento (`offset`) e o ponteiro para os valores daquela diagonal (`diags`), e para cada linha calcula o índice correspondente $j=i+\text{offset}$. Quando o índice está dentro do intervalo, $A \cdot x$ é subtraído diretamente em `residuo[i]`. Com isso, não é mais necessário montar $A \cdot x$ antes para só depois calcular $r = b - A \cdot x$, assim tudo acontece em um único fluxo na memória, o que reduz leituras, escritas e elimina a necessidade de um vetor temporário.

O cálculo final da norma euclidiana do resíduo também foi reescrito, agora a função `calcResiduoSL`, em vez de construir um vetor completo de $A \cdot x$, percorre apenas as diagonais não nulas e calcula cada $A \cdot x_i$ linha por linha. Em seguida obtém $r_i = b_i - A \cdot x_i$ e já acumula r_i^2 , produzindo a norma euclidiana diretamente. Isso evita a alocação extra de um vetor auxiliar Ax e reduz os acessos à memória, além do custo permanecer em $O(n \cdot k)$.

Essas mudanças melhoram o desempenho sobretudo por reduzir acessos desnecessários sobre dados que não contribuem (como elementos nulos e vetor auxiliar) e por ajudar a organizar o acesso à memória de forma mais favorável. Na prática, isso torna os cálculos desses resíduos mais rápidos, diminuindo operações redundantes/inúteis e reduz o erro de arredondamento acumulado, motivo pelo qual a `v2` tende a produzir resíduos menores do que `v1`.

3. Gráficos

Neste tópico vamos apresentar os gráficos correspondentes às métricas de desempenho extraídas com a ferramenta LIKWID para as operações `op1` e `op2`. São analisados o tempo de execução, a banda de memória, o cache miss da L2 e o desempenho em operações aritméticas (MFLOP/s). Cada gráfico foi obtido a partir de testes realizados para diferentes tamanhos de matriz com n variando entre 32, 64, 128, 256, 512, 1000, 2000, 4000, 8000, 9000, 10000 e 20000.

3.1 OP1: Gradientes Conjugados

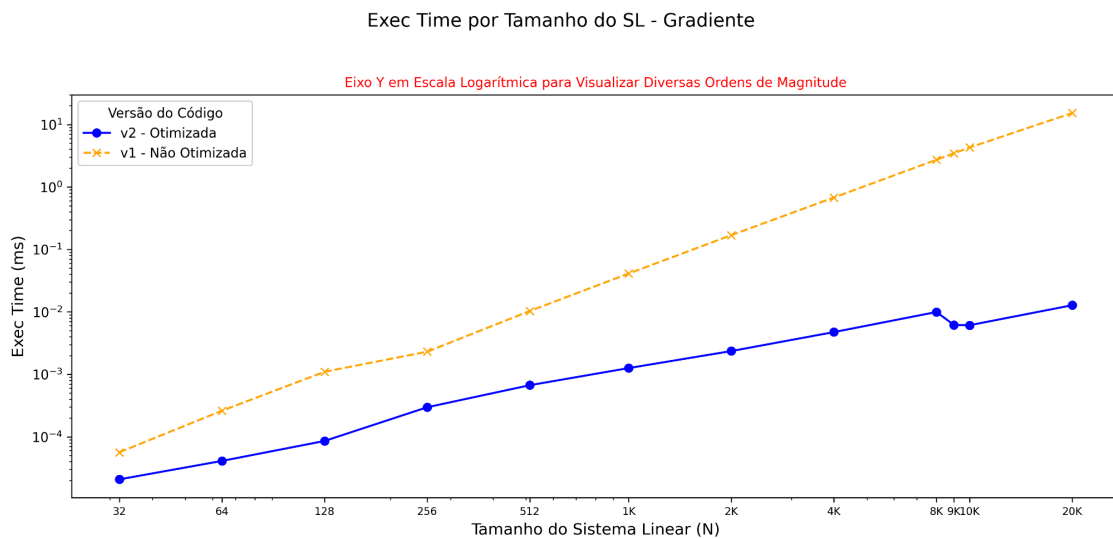


Figura 3 - Tempo de Execução (Método do Gradiente Conjugado)

Um dos principais ganhos obtidos foi o tempo de execução, o qual caiu drasticamente com a aplicação das otimizações. É possível ver que, conforme o N aumenta, a distância entre as curvas de tempo de execução da `v1` e `v2` se distanciam cada vez mais. Esse ganho pode ser atribuído principalmente à ausência de diversas operações necessárias que estavam presentes na versão não otimizada. A `v2`, com a estrutura de dados que representa apenas as diagonais

preenchidas da matriz, eliminou operações feitas sob elementos zerados. Quanto maior o N , mais elementos zerados estão presentes na matriz de coeficientes; assim, a versão otimizada evita o custo associado a realizar esses cálculos desnecessários.

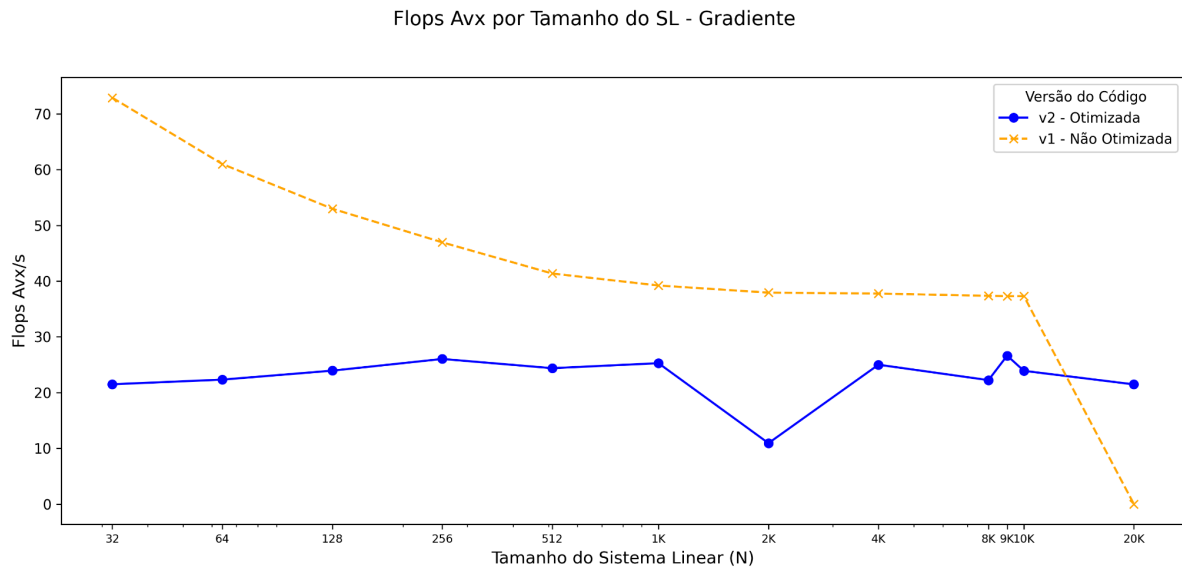


Figura 4 - Operações de Ponto Flutuante de Dupla Precisão por Segundo AVX (Método do Gradiente Conjugado)

A quantidade de MFLOPS/s com instruções AVX, para a versão otimizada, se manteve relativamente estável entre os diferentes valores de N , variando entre 20 e 30. Isso demonstra que o compilador foi capaz de vetorizar algumas seções do cálculo do gradiente, aproveitando as operações SIMD disponíveis.

O que chama mais a atenção é a taxa elevada de MFLOPS/s com instruções AVX na versão v1, sem otimização. Isso pode ser explicado pela estrutura de dados e código original: uma vez que se tratava de uma matriz $N \times N$ com elementos contíguos (mesmo que com vários valores nulos), percorrida por loops duplos, o compilador pôde aplicar fortes vetorizações na operação Gradiente em v1. Já em v2, isso ocorreu com menos frequência, devido à struct usada para representar apenas diagonais preenchidas e com tamanho fixo (7).

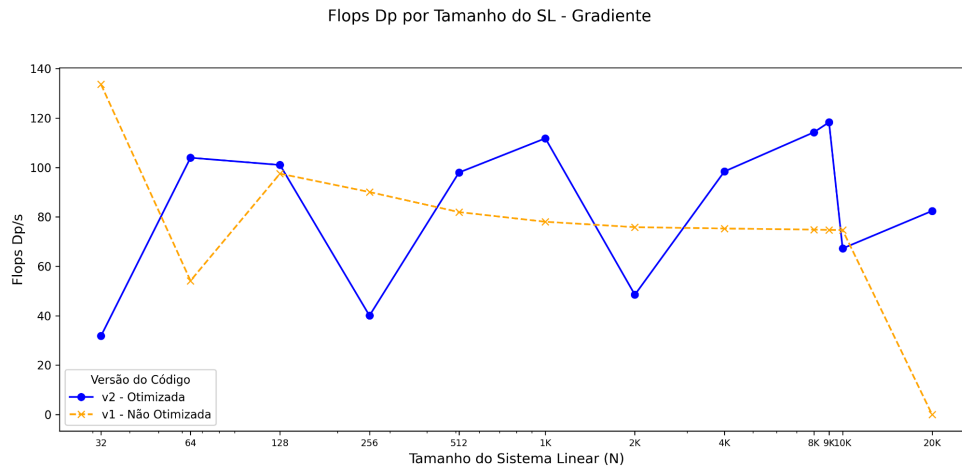


Figura 5 - Operações de Ponto Flutuante de Dupla Precisão por Segundo (Método do Gradiente Conjugado)

A versão otimizada apresenta aumento de MFLOP/s para diversos valores de N, superando a v1 nos casos dos tamanhos 64, 128, 512, 1000, 4000, 8000 e 9000. A taxa maior reflete tanto uma quantidade superior de operações sendo feitas quanto uma redução no tempo de execução do programa. Entretanto, é visível que a métrica apresenta instabilidade; para os valores 256 e 2000, a métrica decaiu consideravelmente. Uma possível explicação para esse fenômeno é a interação com a hierarquia de cache, já que, conforme o tamanho das matrizes cresce, elas deixam de "encaixar" nos níveis mais rápidos (L1 e L2), fazendo com que o desempenho varie.

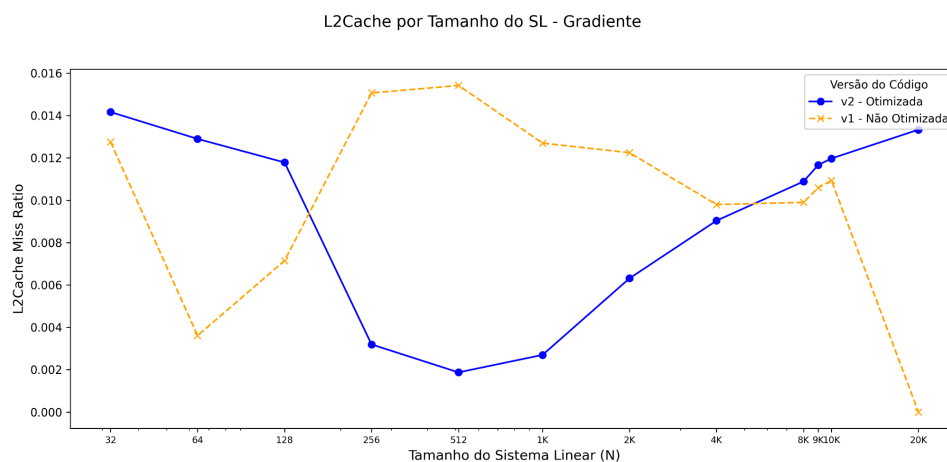


Figura 6 - Uso de L2 Cache (Método do Gradiente Conjugado)

A principal diferença entre os valores de cache miss ratio de v1 e v2 se dá entre $N = 256$ e $N = 2000$, o que condiz com o tamanho da cache. Acima de $N = 4000$, as taxas de cache miss tendem a se aproximar, uma vez que a matriz se torna expressivamente grande e impossibilitando o aproveitamento de localidade de memória.

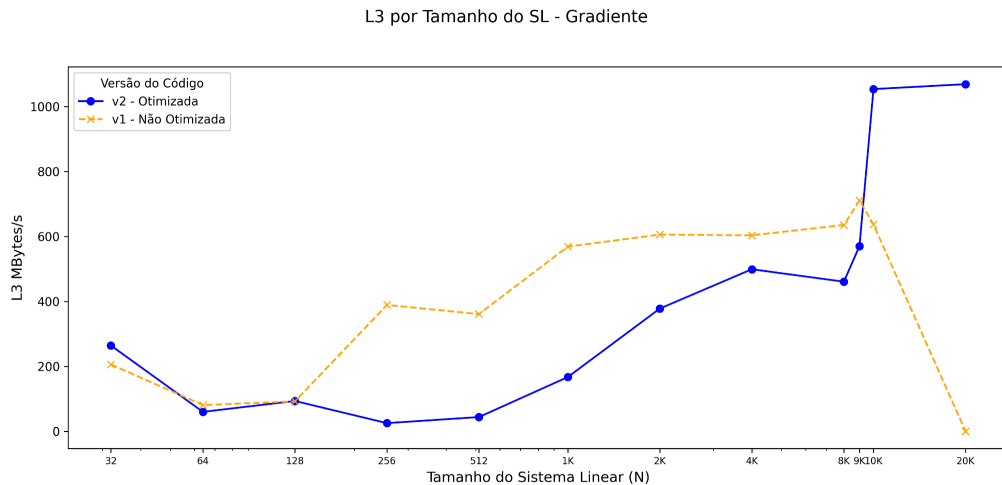


Figura 7 - Largura de Banda da Memória em MBytes/s (Método do Gradiente Conjugado)

Para esta métrica, verifica-se que a versão otimizada v2 mantém um uso baixo da cache L3 em comparação à v1 para toda uma faixa de valores de N (entre 256 e 8000), porém, conforme N cresce, mais largura de banda é requisitada, o que é evidente ao analisar as métricas extraídas em N = 10000 e N = 20000. Ou seja, para valores de N maiores, o código otimizado passa a depender mais do nível L3 para manter seu padrão de acesso, o que é esperado quando os dados deixam de caber nos níveis superiores da cache.

3.2 OP2: Cálculo do Resíduo

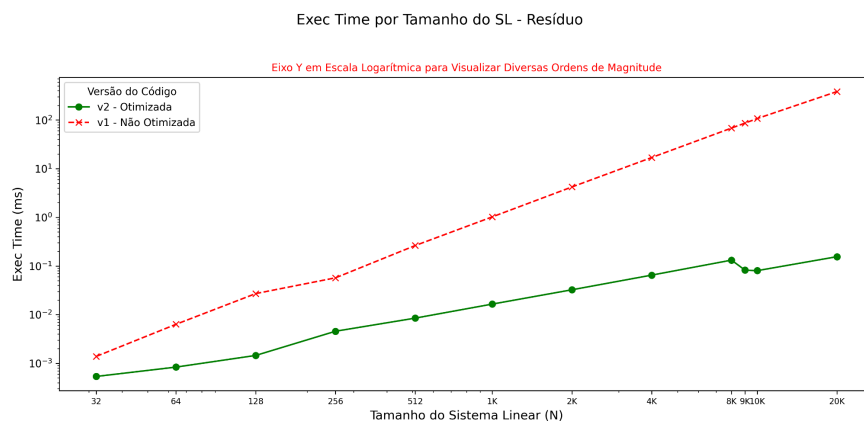


Figura 8 - Tempo de Execução (Cálculo do Resíduo)

A diminuição notável de tempo de execução também ocorreu no cálculo do resíduo, como é possível observar no gráfico. Tal como na operação Gradiente, conforme N cresce, a diferença de tempo de execução entre v1 e v2 também se torna maior, demonstrando um considerável aumento de desempenho.

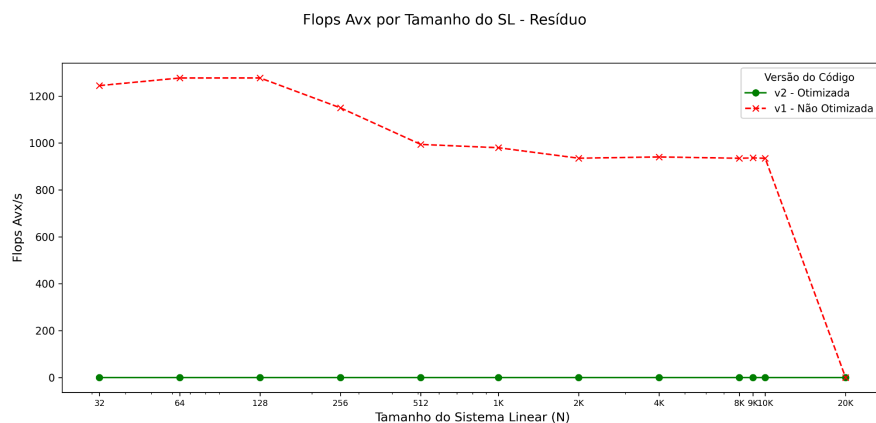


Figura 9 - Operações de Ponto Flutuante de Dupla Precisão por Segundo AVX (Cálculo do Resíduo)

No caso do cálculo do resíduo, as alterações implementadas impossibilitaram o uso de instruções AVX, de forma que, para a v2, os valores permaneceram 0 em todas as execuções. Mais especificamente, a função de cálculo do resíduo em v2 possui dependência de dados em variáveis acumuladoras.

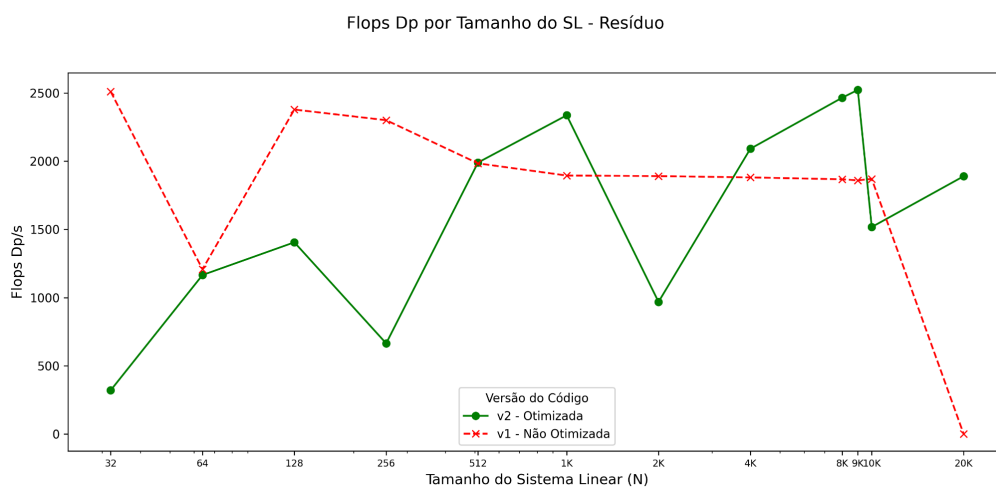


Figura 10 - Operações de Ponto Flutuante de Dupla Precisão por Segundo (Cálculo do Resíduo)

Apesar das quantidades inconstantes de MFLOP/s, percebe-se que os picos e vales são muito semelhantes ao gráfico de FLOPS_DP da operação Gradiente. A explicação das razões para os picos e vales presentes neste gráfico é a mesma da operação Gradiente.

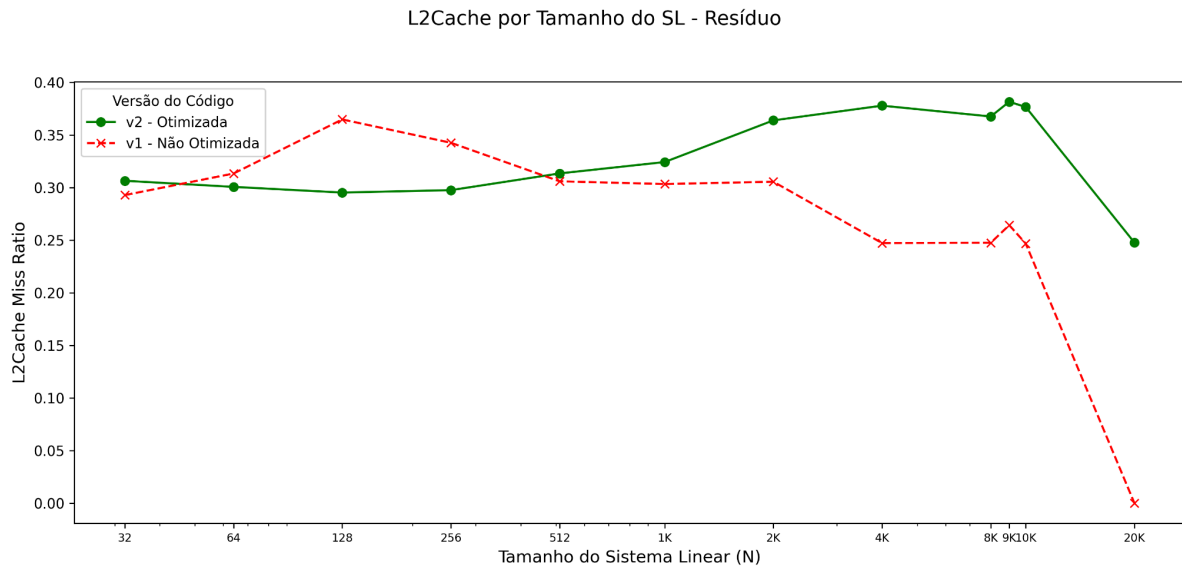


Figura 11 - Uso de L2 Cache (Cálculo do Resíduo)

Nessa métrica, exceto para $N=64$, $N=128$ e $N=256$, não é possível dizer que houve ganhos; em geral, a versão não otimizada apresenta um cache miss ratio menor. Uma explicação plausível para esse ocorrido é a mudança na estrutura de dados: enquanto na v1 foram utilizadas matrizes com alocação contígua, às quais tiram grande proveito da localidade espacial, a v2 usa uma struct que não possui o mesmo grau de aproveitamento no acesso de memória. A struct presente em v2 é composta por diversos vetores diagonais que podem não estar alocados de forma contígua na memória, porém, armazena apenas os elementos não nulos. Foi decidido que a perda nesta métrica era aceitável em prol dos ganhos em outras métricas (por exemplo, Exec_time).

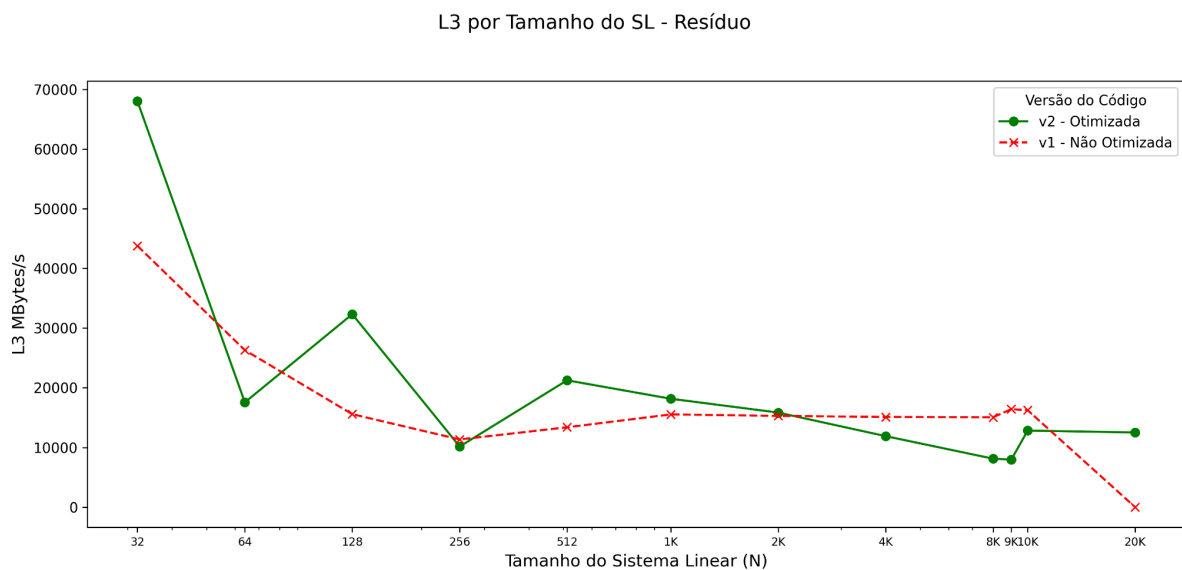


Figura 12 - Largura de Banda da Memória em MBytes/s (Cálculo do Resíduo)

Para esta métrica, nota-se que a v2 usa mais largura de banda em tamanhos pequenos (como $N=32$ e $N=128$), pois a nova implementação do cálculo do resíduo faz uma varredura mais densa e sequencial, gerando acessos concentrados à L3. Porém, conforme N cresce, esse acesso cai e se aproxima da v1, ficando até abaixo em alguns casos. Uma possível razão para isso é que, em v2, não são feitas leituras redundantes e elementos nulos não são acessados.

4. Estimativa de Desempenho

Neste tópico apresentamos as estimativas teóricas de desempenho da versão otimizada (v2), considerando uma máquina com 8 GB de RAM e matrizes esparsas com 7 diagonais ($k = 7$).

1) Qual o valor máximo estimado da ordem N do sistema linear que você seria capaz de resolver ?

Para estimar a memória usada pela versão v2 do programa, partimos do fato de que cada número real ocupa 8 bytes e de que as matrizes armazenadas em formato diagonal têm um número fixo de diagonais: a matriz A possui 7 e sua versão pré-condicionada, ASP , possui 13 ($2 \cdot k - 1$). A v2 mantém as matrizes diagonais A , ASP , D , L , U e M , que somam 48 diagonais ao todo. Como cada diagonal tem tamanho n e cada elemento tem 8 bytes, o custo dominante é: $48 \cdot n \cdot 8 \text{ bytes} \approx 384 \cdot n \text{ bytes}$. Além disso, o programa precisa armazenar três vetores (b , bsp e x), que juntos consomem: $3 \cdot n \cdot 8 \text{ bytes} \approx 24 \cdot n \text{ bytes}$. Assim, a memória total necessária fica em torno de $408 \cdot n \text{ bytes}$. Com esse modelo, podemos calcular o tamanho máximo de n que caberia na memória. Considerando que temos 8 GB (8589934592 bytes), o maior tamanho possível para o problema é de aproximadamente: $n \approx 8589934592 / 408$ que daria algo em torno de $n \approx 20$ milhões, porém isso seria inviável em questão de tempo.

2) Qual o ganho estimado de tempo para uma iteração do método de Gradiente Conjugado com Pré-condicionador de Jacobi (em função de N) ?

Para entender o ganho de tempo da v2 em relação à v1 em uma iteração do método de Gradiente Conjugado pré-condicionado, basta comparar o custo que cada versão precisa ter em cada iteração. Na v1, que usa uma matriz $n \times n$, o custo mais significativo por iteração é o produto matriz-vetor $A \cdot \text{search_direction}$, que exige algo proporcional a n^2 operações. Já na v2, que usa uma matriz com apenas k diagonais (no nosso caso $k = 7$), o custo do mesmo produto cai para algo proporcional a $7 \cdot n$. Como as demais operações do método custam em torno de $O(n)$ em ambas as versões, o que realmente faz diferença é esse produto matriz-vetor.

Com isso, o ganho teórico por iteração pode ser visto como: enquanto a v1 possui custo de n^2 , a v2 tem custo de $7 \cdot n$. Dividindo esses custos, o ganho de velocidade por iteração fica aproximadamente $n/7$ vezes. Isso significa que o ganho cresce linearmente com o tamanho do problema: para $n = 1.000$, o ganho seria da ordem de 142 vezes, para $n = 2.000$, por volta de 285 vezes e assim por diante.

3) Qual a estimativa para a quantidade de operações em ponto flutuante executadas em cada iteração do método de Gradiente Conjugado com Pré-condicionador de Jacobi (em função de N) ?

Para estimar quantas operações em ponto flutuante são executadas em cada iteração do Gradiente Conjugado com pré-condicionador de Jacobi na v2, vamos olhar as partes principais do algoritmo. O maior custo do método, como dito na questão anterior, vem do produto matriz-vetor $A \cdot \text{search_direction}$. Como a matriz A é k -diagonal, então em cada uma das n linhas existem no máximo k elementos não nulos. Para cada um dos elementos teremos 1 multiplicação ($A[i, j] \cdot \text{search_direction}[j]$) e 1 soma ($A_search_direction += \dots$) e como há aproximadamente k elementos não nulos por linha e N linhas, o total é aproximadamente: $2 \cdot k \cdot N$ FLOPs.

Além disso, a cada iteração o método atualiza o x ($x = x + \text{step_size} \cdot \text{search_direction} \approx 2 \cdot N$ FLOPs), atualiza o resíduo ($r = r - \text{step_size} \cdot A_search_direction : \approx 2 \cdot N$ FLOPs), aplica o pré-condicionador de Jacobi ($z = M^{-1} \cdot r \approx N$ FLOPs) e atualiza a direção de busca ($\text{search_direction} = z + \beta \cdot \text{search_direction} : \approx 2 \cdot N$ FLOPs). Dessa forma, teremos aproximadamente mais $11 \cdot N$ FLOPs por operação. Somando tudo obtemos um custo aproximado por iteração de $(2k + 11) \cdot N$ FLOPs e, como no nosso caso $k = 7$, chegamos a uma estimativa de $25 \cdot N$ FLOPs por iteração.

4) Compare suas estimativas acima com os resultados obtidos nos testes abaixo.

Comparando os nossos resultados com as perguntas 2 e 3, observamos primeiramente que o ganho de tempo para uma iteração do método de Gradiente Conjugado se mostrou próximo da estimativa de $n/7$ vezes para valores menores de n , como em $n = 32$, onde o ganho foi de 2,72 vezes. Já para valores maiores de n , como $n = 10.000$, o ganho foi menor, atingindo 1195,85 vezes (enquanto o ganho estimado por $n/7$ deveria ser de aproximadamente 2857 vezes).

Por outro lado, a quantidade de operações em ponto flutuante por iteração foi um pouco mais difícil de saber, a estimativa feita foi de $25 \cdot N$ FLOPs mas na realidade com $n = 32$ observamos um valor menor, isso porque, parte das diagonais cortam nas bordas reduzindo o número real de operações feitas.

5. Detalhamento da Arquitetura do Processador (LIKWID-topology)

Nesta seção apresentamos as principais características da arquitetura do processador utilizado nos testes, que foram extraídas por meio do comando `likwid-topology -g -c`. Os testes deste trabalho foram executados na máquina do DINF com identificador h20 que possui um processador Intel Core i5-7500 da família Intel Coffeelake com 4 núcleos.

```
vrg20@h20:~$ likwid-topology -g -c
-----
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
-----
```

Figura 13 - Informações do Processador

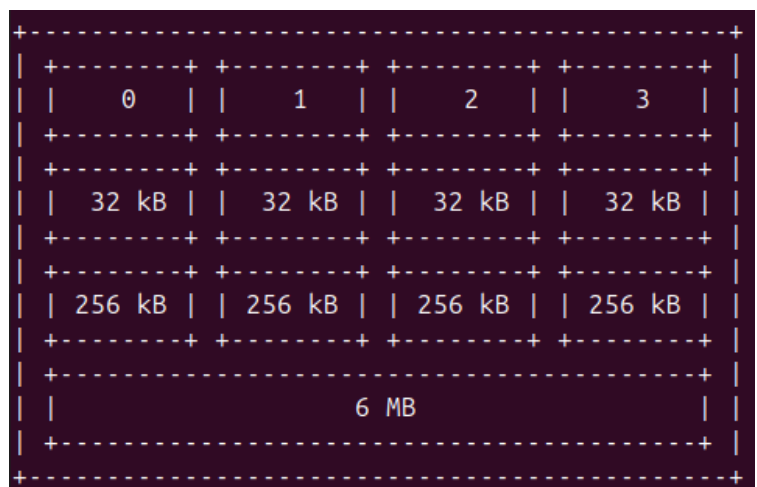


Figura 14 - Hierarquia de Cache