Relatório sobre otimizações feitas no programa pdeSolver

Guilherme M. Lopes[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)

Abstract. `pdeSolver` *is a software to calculate an approximated discretized solution to a Partial Differential Equation (PDE) using the methods central finite difference and Gauss-Seidel. This article is dedicated to explain the techniques and changes made in the source code of* `pdeSolver` *to increase perfomance.*

Resumo. *O programa* `pdeSolver` *é um software utilizado para calcular uma solução discreta aproximada de uma equação diferencial parcial (PDE) utilizando o método das diferenças finitas centrais e o método de Gauss-Seidel. O presente artigo dedica-se a explicar em detalhes as técnicas utilizadas e modificações feitas no código fonte do programa* `pdeSolver` *para melhora de desempenho.*

1. Introdução

O programa `pdeSolver`, escrito na linguagem C, é um software que consiste em duas principais etapas: a discretização de uma equação diferencial parcial dada utilizando o método das diferenças finitas centrais, e a resolução da matriz pentadiagonal obtida através do método de Gauss-Seidel na função `gaussSeidel()`. A geração da solução da equação passa pelo cálculo do resíduo associado à iteração em questão. Este cálculo é feito na função `calculate_residues()`. O presente artigo dedica-se a enumerar e discorrer sobre as técnicas utilizadas e as modificações feitas nas funções `gaussSeidel()` e `calculate_residues()` a fim de melhorar o desempenho mediantes os seguintes indicadores: tempo de execução (ms), banda de memória (MBytes/s), cache L2 miss ratio (%) e operações de ponto flutuante por intervalo de tempo (MFLOP/s).

2. Detalhes técnicos do computador utilizado para os experimentos

Esta sessão é dedicada a detalhar o computador utilizado em todos os experimentos realizados para coletar os dados presentes neste relatório.

2.1. Processador

A figura 1 foi gerada pelo programa LiKWID e ilustra a topologia da CPU. Lista de informações do processador:

- Nome: Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz;
- Tipo: Intel Coffeelake processor;
- Arquitetura: x86-64;
- Modos de operação: 32-bit, 64-bit;
- Ordem de byte: Little Endian;

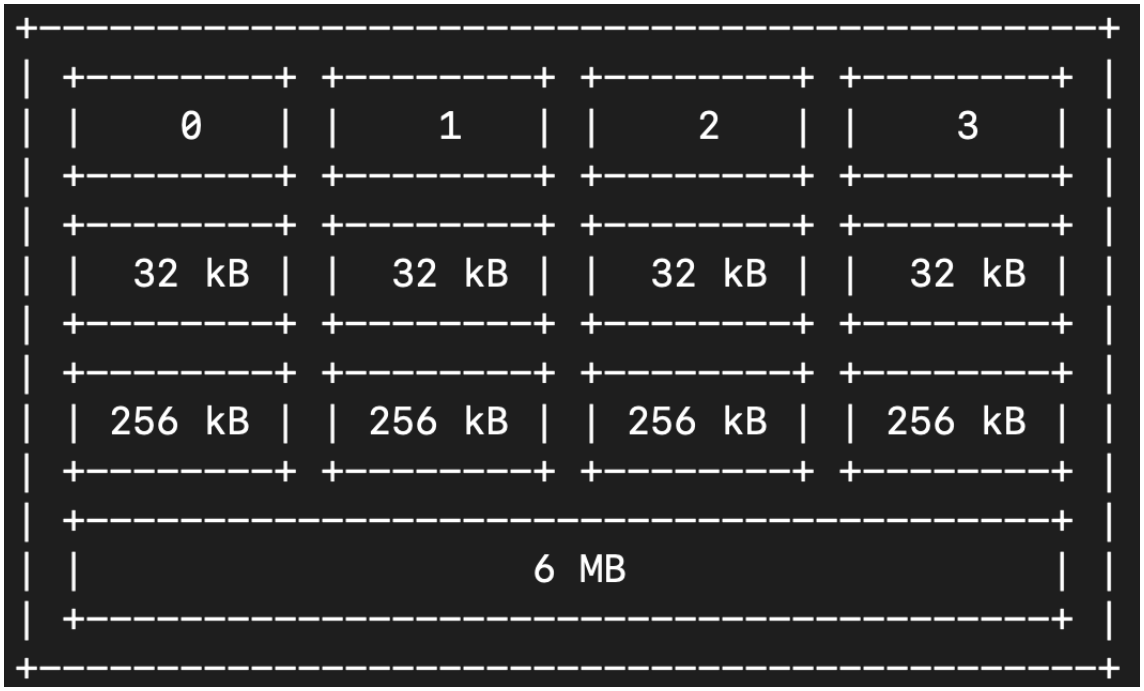

Figura 1. Topologia gráfica do processador gerada pelo Likwid.

- Número de núcleos: 4;
- Threads por núcleo: 1;
- Núcleos por socket: 4;
- Quantidade de sockets: 1;
- Frequência: 3094.585 MHz;
- Frequência máxima: 3800 MHz;
- Frequência mínima: 800 MHz;
- Cache L1d: 32K;
- Cache L1i: 32K;
- Cache L2: 256K; e
- Cache L3: 6144K.

2.2. Memória principal

O computador possui uma placa mãe com quatro slots para dispositivos de memória RAM. Apenas um dos slots está sendo utilizado. Lista de informações sobre dispositivo de memória RAM instalado:

- Largura total: 64 bits;
- Largura dos dados: 64 bits;
- Tamanho: 8192 MB;
- Frequência: 2400 MHz (0,4 ns);
- Tipo: DDR2;
- Detalhes do tipo: síncrono sem buffer (não registrado);
- Fabricante: SMART; e
- Voltagem configurada: 1.2 V.

3. Otimizações realizadas

Esta sessão dedica-se a descrever os detalhes das técnicas utilizadas e modificações feitas no código fronte do programa para melhorar o desempenho. A próxima sessão discutirá os impactos da otimizações aqui descritas.

3.1. Otimizações gerais

Nas funções `gaussSeidel()` e `calculate_residues()`, o índice `((*SL)->ny) + 2` (valor utilizado para acessar um vetor) era calculado todas as vezes que o acesso era feito. Foi feita a atribuição do valor `((*SL)->ny) + 2` a uma variável chamada `offset`, de modo que só precisasse ser calculada uma vez, e utilizada em todos os acessos subsequentes.

Foi adicionada a palavra reservada `restricted` nas declarações de ponteiros para qualificação de tipo. A palavra informa ao compilador que o ponteiro em questão é a única maneira de acessar o objeto apontado, e o compilador não precisa de checagens adicionais.

Para acessar um vetor bidimensional armazenado sequencialmente, é necessário um cálculo de índice que anteriormente era feito na função `calculate_index()` passando os valores de `i` e `j` (índices das matriz que deseja-se acessar). O problema desta solução é que toda chamada de função possui um *overhead* associado com a criação de todas as estruturas necessárias para a execução do procedimento. A solução foi utilizar *macros* que são literalmente substituídas pela expressão definida em tempo de compilação.

Alguns processadores conseguem lidar com aritmética de inteiro sem sinal consideravelmente mais rápido do que com sinal. Por isso, todas as declarações de números inteiros onde se tem certeza de que nunca serão negativos (como contadores de laços de repetição) foram marcadas com a palavra reservada `unsigned`.

Por fim, a palavra reservada `registered` também foi utilizada nas declarações de variáveis utilizadas em laços de repetição e outras variáveis com muitos acessos. Esta palavra força estas variáveis a estarem sempre acessíveis ao processador, em um registrador, reduzindo o *overhead*.

3.2. *Loop Unrolling*

A técnica de *loop unrolling* consiste em ”desenrolar”laços de repetição para que sejam feitas mais operações por iteração. Esta técnica exige que um ”resto”, o chamado *loop remainer*, seja processado em seguida. A função `gaussSeidel()`, cujos laços podem ser vistos na figura 4, teve uma redução substancial de tempo de execução após a aplicação da técnica com fator m = 4 (como pode ser visto na figura 5), seguida de um *unroll & jam* (quando misturam-se os laços internos depois do *unroll*). O fator m = 4 foi escolhido porque o programa deixou de ter melhora no desempenho para fatores maiores, provavelmente por conta das particularidades da memória cache do processador em relação aos cálculos feitos no `pdeSolver`.

4. Experimentos realizados

A fim de medir diferentes indicadores de performance do `pdeSolver`, foi utilizado o LiKWID para monitoramento do *hardware* durante a execução dos testes. Os testes foram

```
t_float calculate_residues(t_LS5Diag *SL, t_float *u) {
    t_float *residue = malloc(sizeof(t_float) * ((SL->nx) + 2) * ((SL->ny) + 2));
    t_float absolute_value = 0;

    for(int i = 1; i < ((SL->nx) + 1); i++) {
        for(int j = 1; j < ((SL->ny) + 1); j++) {
            residue[ calculate_index(i, j, (SL->ny) + 2) ] = (SL->b[ calculate_index(i, j, (SL->ny) + 2) ]);
            residue[ calculate_index(i, j, (SL->ny) + 2) ] -= ((SL->away_bottom_diagonal) * u[ calculate_index(i, j - 1, (SL->
            residue[ calculate_index(i, j, (SL->ny) + 2) ] -= ((SL->bottom_diagonal) * u[ calculate_index(i - 1, j, (SL->ny) +
            residue[ calculate_index(i, j, (SL->ny) + 2) ] -= ((SL->away_upper_diagonal) * u[ calculate_index(i, j + 1,( SL->n
            residue[ calculate_index(i, j, (SL->ny) + 2) ] -= ((SL->upper_diagonal) * u[ calculate_index(i + 1, j, (SL->ny) + 
            residue[ calculate_index(i, j, (SL->ny) + 2) ] -= (SL->main_diagonal) * u[ calculate_index(i, j, (SL->ny) + 2) ];

            absolute_value += residue[ calculate_index(i, j, (SL->ny) + 2) ] * residue[ calculate_index(i, j, (SL->ny) + 2) ];
        }
    }
    absolute_value = sqrt(absolute_value);

    return absolute_value;
}
```

Figura 2. Código da função calculate_residues() antes das otimizações.

```
t_float calculate_residues(t_LS5Diag * restrict SL, t_float * restrict u_vector) {
    register unsigned int offset = (SL->ny) + 2;
    t_float *residue = malloc(sizeof(t_float) * ((SL->nx) + 2) * offset);
    t_float absolute_value = 0;
    t_float *u = __builtin_assume_aligned(u_vector, 16);
    t_float *b = __builtin_assume_aligned((SL)->b, 16);
    for (register unsigned int i = 1; i < ((SL->nx) + 1); i++) {
        for (register unsigned int j = 1; j < ((SL->ny) + 1); j++) {
            register unsigned int index = calculate_index(i, j, offset);
            residue[ index ] = (b[ index ])
            - ((SL->away_bottom_diagonal) * u[ calculate_index(i, j - 1, offset) ])
            - ((SL->bottom_diagonal) * u[ calculate_index(i - 1, j, offset) ])
            - ((SL->away_upper_diagonal) * u[ calculate_index(i, j + 1, offset) ])
            - ((SL->upper_diagonal) * u[ calculate_index(i + 1, j, offset) ])
            - (SL->main_diagonal) * u[ index ];

            absolute_value += residue[ index ] * residue[ index ];
        }
    }
    absolute_value = sqrt(absolute_value);

    return absolute_value;
}
```

Figura 3. Código da função calculate_residues() depois das otimizações.

```
for (int i = 1; i < (((*SL)->nx) + 1); ++i) {
    for (int j = 1; j < (((*SL)->ny)+1); ++j) {

        Uij = (*SL)->b[ calculate_index(i,j,(((*SL)->ny)+2)) ];

        Uij -= (((*SL)->away_bottom_diagonal) * (*u)[ calculate_index(i, j - 1, ((*SL)->ny) + 2) ]);

        Uij -= (((*SL)->bottom_diagonal) * (*u)[ calculate_index(i - 1, j, ((*SL)->ny) + 2) ]);

        Uij -= (((*SL)->away_upper_diagonal) * (*u)[ calculate_index(i, j + 1, ((*SL)->ny) + 2) ]);

        Uij -= (((*SL)->upper_diagonal) * (*u)[ calculate_index(i + 1, j, ((*SL)->ny) + 2) ]);

        Uij /= ((*SL)->main_diagonal);

        (*u)[ calculate_index(i, j, ((*SL)->ny) + 2) ] = Uij;

    }
}
```

Figura 4. Os dois laços de repetição da função gaussSeidel() antes do *loop unrolling*.

executados com 10 iterações do método de Gauss-Seidel, e, para cada indicador, foram executados 19 testes diferentes, cada um com valores de nx = ny diferentes (tamanhos da malha para discretização).

4.1. Teste de tempo de execução

Este teste teve como objetivo medir o tempo de execução, em milissegundos (ms), das funções `gaussSeidel()` e `calculate_residues()`. Neste caso, foi utilizada a função `timestamp()`, e o resultado dos testes (antes e depois das otimizações) encontra-se na figura 6 e 7. Observe que o eixo das ordenadas (tempo) está em escala logarítmica.

Listing 1. Função timestamp()

```
double timestamp(void) {
    struct timeval time_pointer;
    gettimeofday(&time_pointer, NULL);

    return((double)
    (time_pointer.tv_sec * 1000.0 +
    time_pointer.tv_usec / 1000.0));
}
```

4.2. Teste de banda de memória

O objetivo deste teste foi observar o grupo L3 do LiKWID e apresentar o resultado do indicador *"memory bandwidth"* para cada função antes e depois da otimização.

```
for (register unsigned int i = 1; i < (((*SL)->nx) + 1) - ((((*SL)->nx) + 1) % 4); i += 4) {
    for (register unsigned int j = 1; j < (((*SL)->ny) + 1); j++) {
        // (i)
        Uij = b[ calculate_index(i, j, offset) ]
            - (((*SL)->away_bottom_diagonal) * (*u)[ calculate_index(i, j - 1, offset)])
            - (((*SL)->bottom_diagonal) * (*u)[ calculate_index(i - 1, j, offset) ])
            - (((*SL)->away_upper_diagonal) * (*u)[ calculate_index(i, j + 1, offset) ])
            - (((*SL)->upper_diagonal) * (*u)[ calculate_index(i + 1, j, offset) ]);
        Uij /= ((*SL)->main_diagonal);
        (*u)[ calculate_index(i, j, offset) ] = Uij;
        // (i + 1)
        Uij = b[ calculate_index((i + 1), j, offset) ]
              - (((*SL)->away_bottom_diagonal) * (*u)[ calculate_index((i + 1), j - 1, offset)])
              - (((*SL)->bottom_diagonal) * (*u)[ calculate_index(i, j, offset) ])
              - (((*SL)->away_upper_diagonal) * (*u)[ calculate_index((i + 1), j + 1, offset) ])
              - (((*SL)->upper_diagonal) * (*u)[ calculate_index(i + 2, j, offset) ]);
        Uij /= ((*SL)->main_diagonal);
        (*u)[ calculate_index((i + 1), j, offset) ] = Uij;
        // (i + 2)
        Uij = b[ calculate_index((i + 2), j, offset) ]
              - (((*SL)->away_bottom_diagonal) * (*u)[ calculate_index((i + 2), j - 1, offset)])
              - (((*SL)->bottom_diagonal) * (*u)[ calculate_index((i + 1), j, offset) ])
              - (((*SL)->away_upper_diagonal) * (*u)[ calculate_index((i + 2), j + 1, offset) ])
              - (((*SL)->upper_diagonal) * (*u)[ calculate_index(i + 3, j, offset) ]);
        Uij /= ((*SL)->main_diagonal);
        (*u)[ calculate_index((i + 2), j, offset) ] = Uij;
        // (i + 3)
        Uij = b[ calculate_index((i + 3), j, offset) ]
              - (((*SL)->away_bottom_diagonal) * (*u)[ calculate_index((i + 3), j - 1, offset)])
              - (((*SL)->bottom_diagonal) * (*u)[ calculate_index((i + 2), j, offset) ])
              - (((*SL)->away_upper_diagonal) * (*u)[ calculate_index((i + 3), j + 1, offset) ])
              - (((*SL)->upper_diagonal) * (*u)[ calculate_index(i + 4, j, offset) ]);
        Uij /= ((*SL)->main_diagonal);
        (*u)[ calculate_index((i + 3), j, offset) ] = Uij;
    }
}
```

Figura 5. Os dois laços de repetição da função gaussSeidel() depois do *loop unrolling* e de outras otimizações.

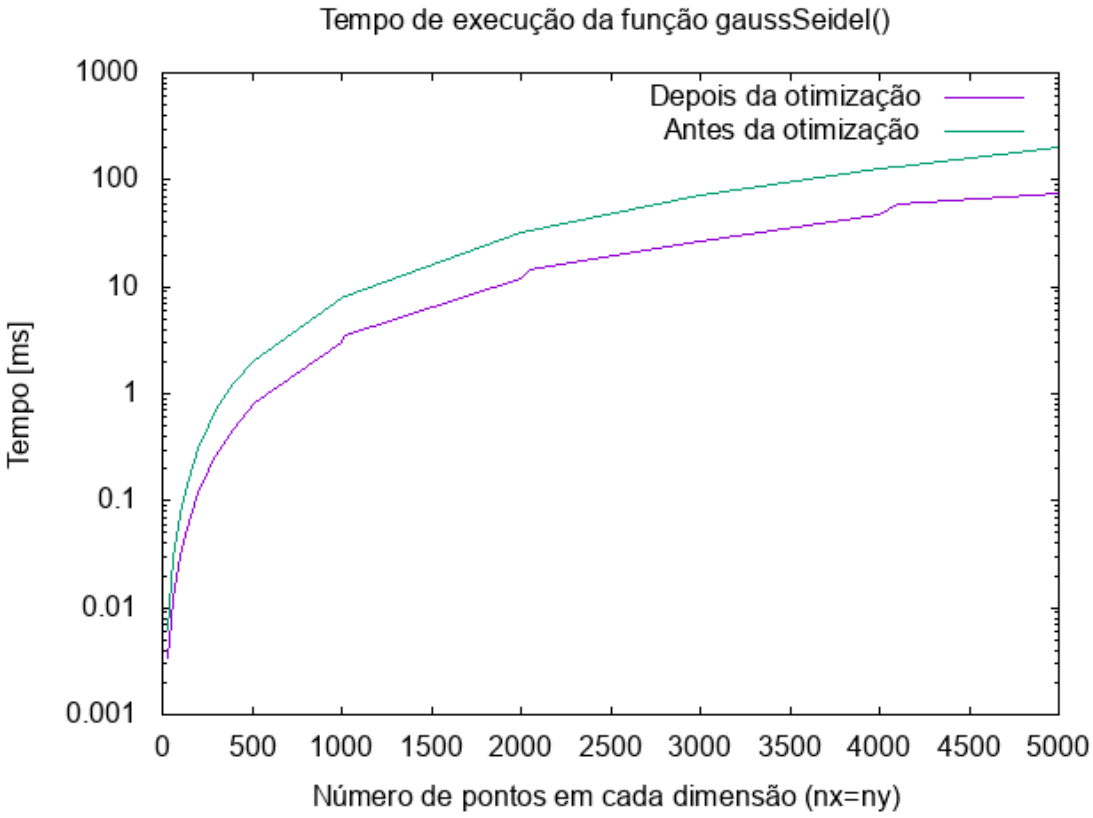

Figura 6. Tempo de execução para a função gaussSeidel().

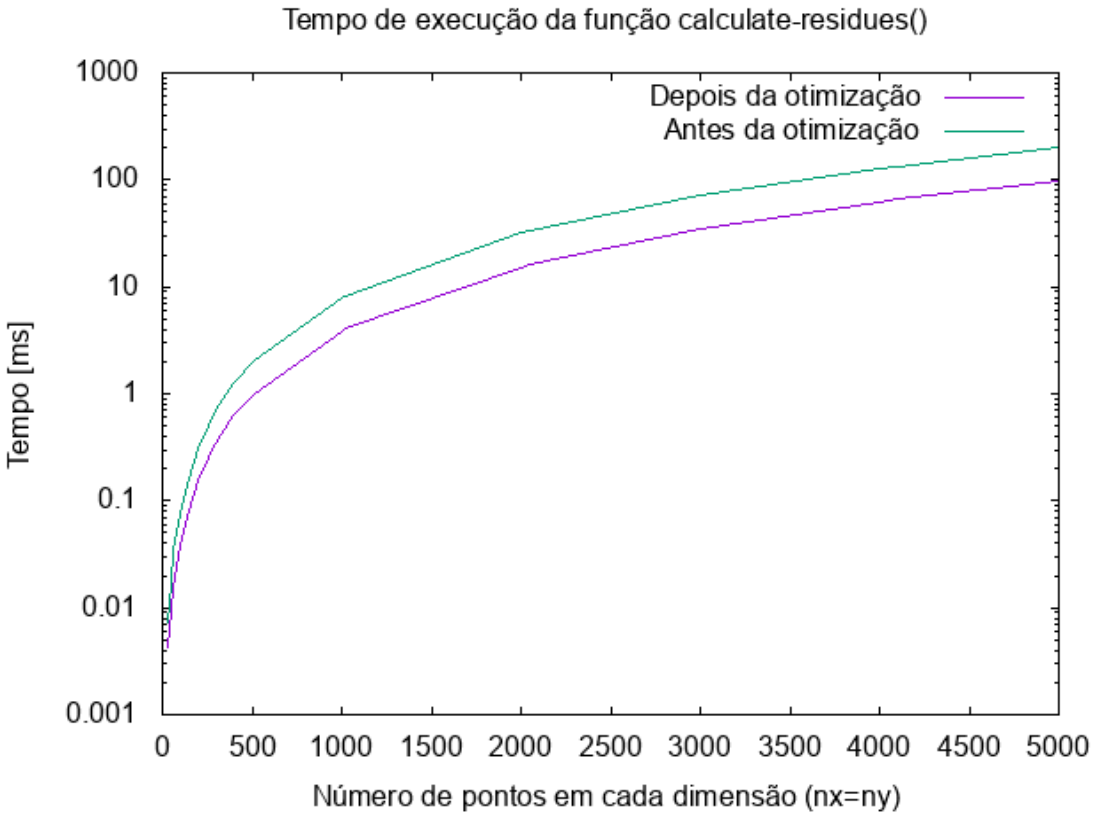

Figura 7. Tempo de execução para a função calculate_residues().

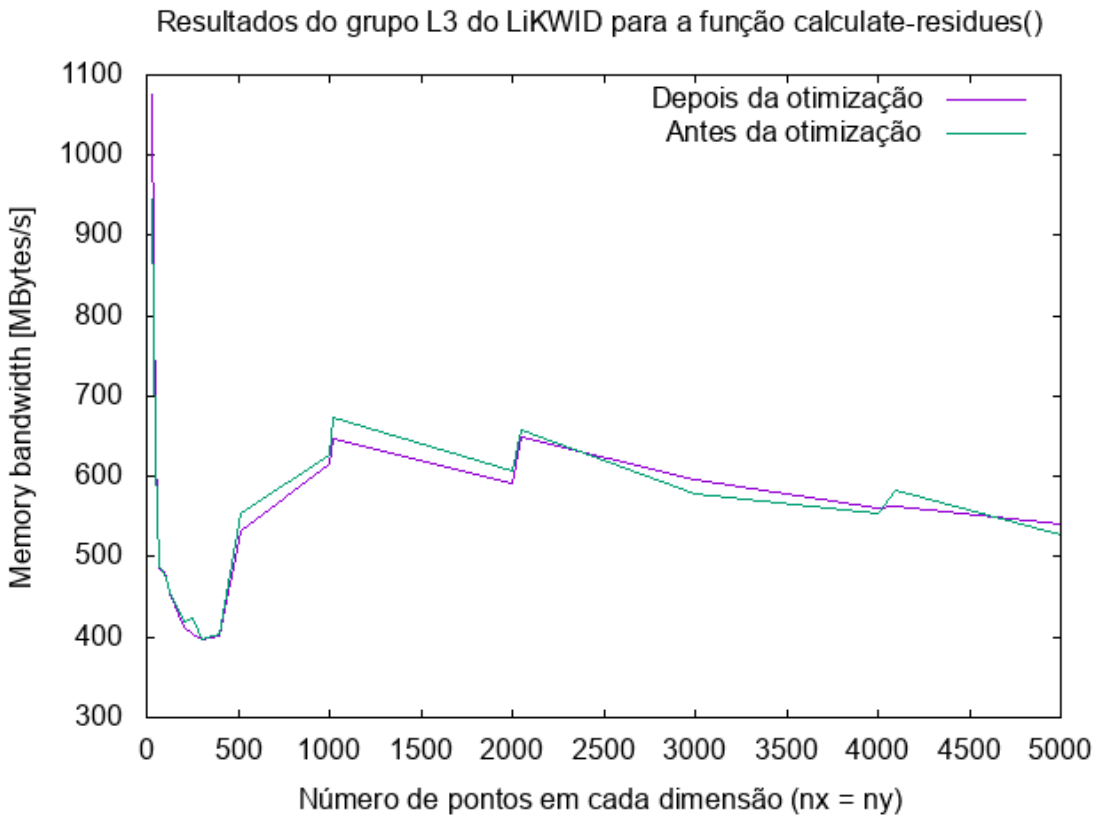

Figura 8. Largura de banda de memória para a função que calcula o resíduo da iteração.

4.2.1. Banda de memória para as funções que implementam o método de Gauss-Seidel e o cáculo do resíduo da iteração

Como podemos observar nas figuras 9 e 8, a largura de banda de memória da cache L3 se manteve muito próxima nas duas versões do programa, indicando que as otimizações não tiverem efeito drástico sobre estes indicadores em específico.

4.3. Teste de taxa de *miss* da cache L2

O objetivo deste teste é medir o indicador *miss ratio* (taxa de vezes em que o processador não encontra o dado que procura na memória cache, gerando *overhead*).

4.3.1. Taxa de *miss* na cache L2 para a função que implementa o método de Gauss-Seidel

Como podemos observar na figura 10, a taxa de *miss* depois da otimização começou mais baixa do que antes, entretanto, se manteve entre 20% e 28% durante todo o período, enquanto que o programa antes da otimização teve o comportamento esperado: bastante *miss* no início da execução (”aquecimento”da cache) e menos no final. Um provável motivo para esse comportamento estranho seria o *loop unrolling*, que, apesar de aumentar muito o número de operações em ponto flutuante por segundo, acaba interferindo no compor-

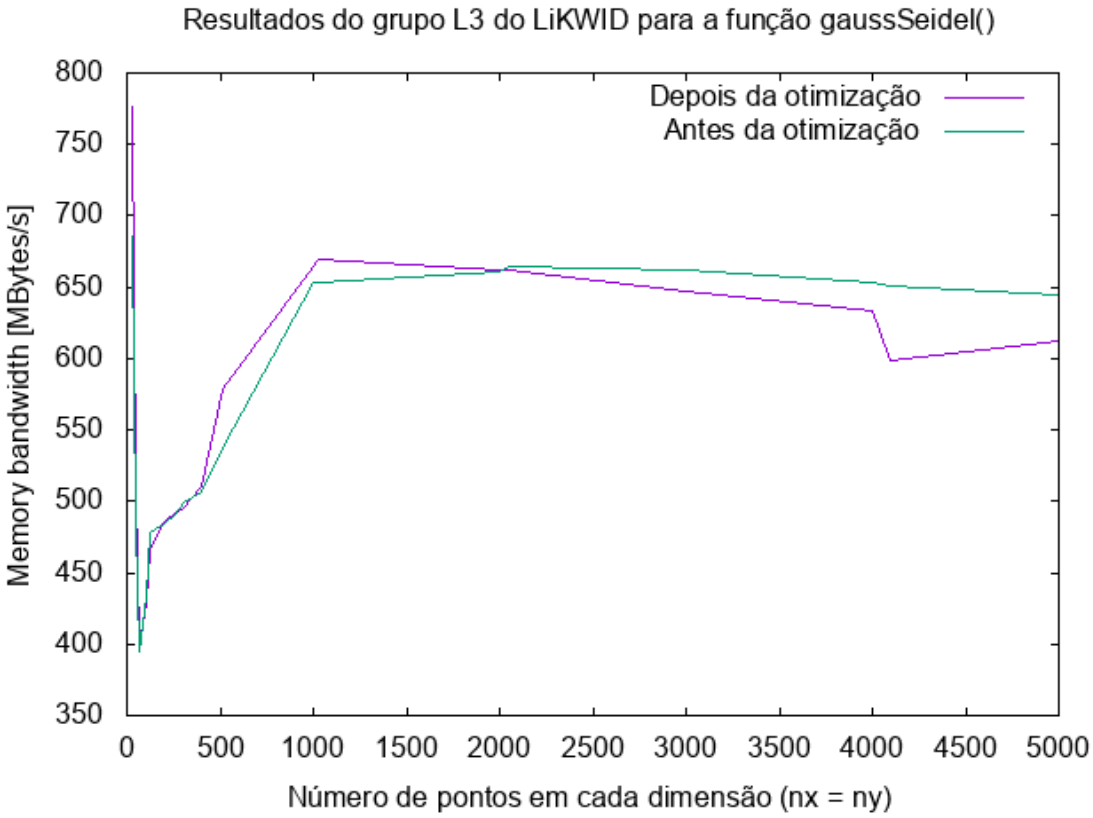

Figura 9. Largura de banda de memória para a função que implementa o método de Gauss-Seidel.

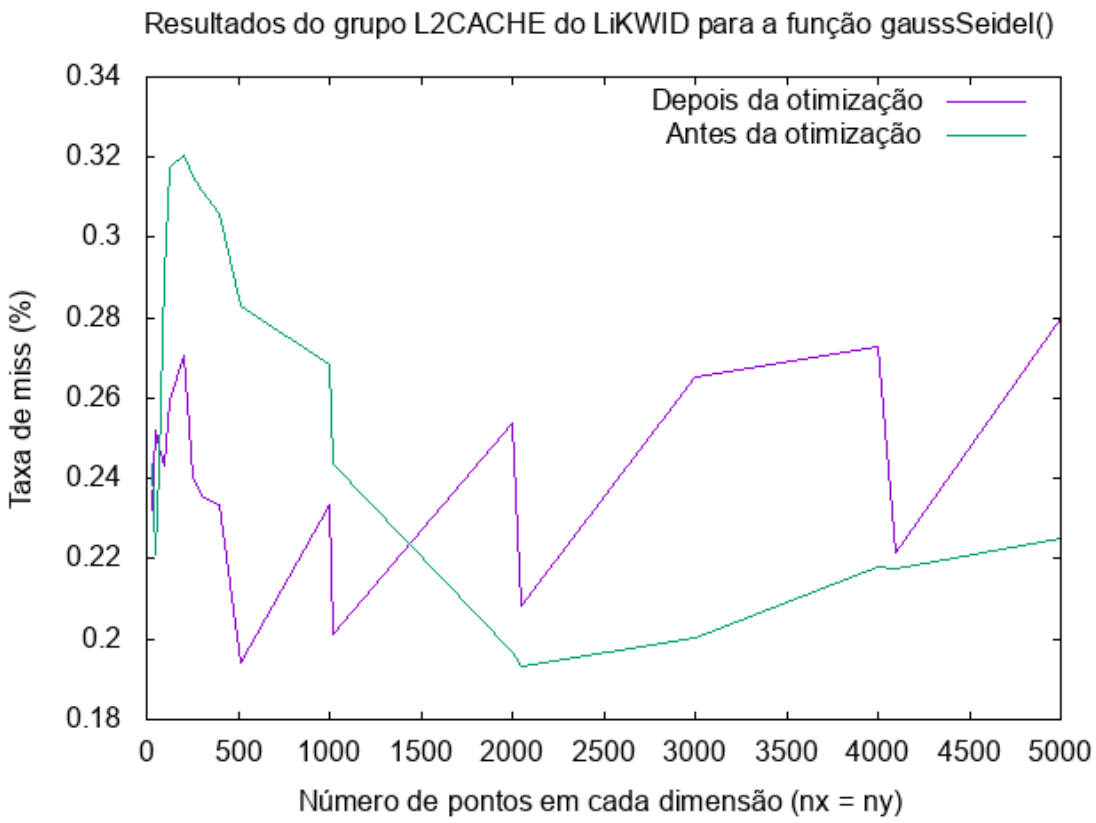

Figura 10. "Miss ratio"na memória cache L2 para a função gaussSeidel().

tamento da memória cache, já que possui um acesso incomum às variáveis. Entretanto, apesar da alta taxa de *miss*, o *trade off* mostra-se positivo, afinal, o ganho em MFLOP/s foi muito grande e influenciou muito mais no tempo de execução do programa.

4.3.2. Taxa de *miss* na cache L2 para a função que calcula o resíduo da iteração

Diferente da função `gaussSeidel()`, a função que calcula o resíduo da iteração obteve, nos dois casos, o comportamento esperado da memória cache: uma alta taxa de miss no início da execução, seguida de uma baixa expressiva e um aumento tímido ao longo do tempo, conforme o problema cresce. A função não obteve melhoria de desempenho com o *loop unrolling*, portanto, manteve o padrão de acesso das variáveis antes e depois das otimizações, demonstrando apenas uma pequena redução na taxa de *miss*, que não podemos garantir que seja de fato devido às otimzações.

4.4. Teste de número de operações em ponto flutuante por intervalo de tempo

O objetivo deste teste é medir o indicador "operações em ponto flutuante por intervalo de tempo". Neste caso, foi medido em MFLOP/s, que são "milhões de operações por segundo".

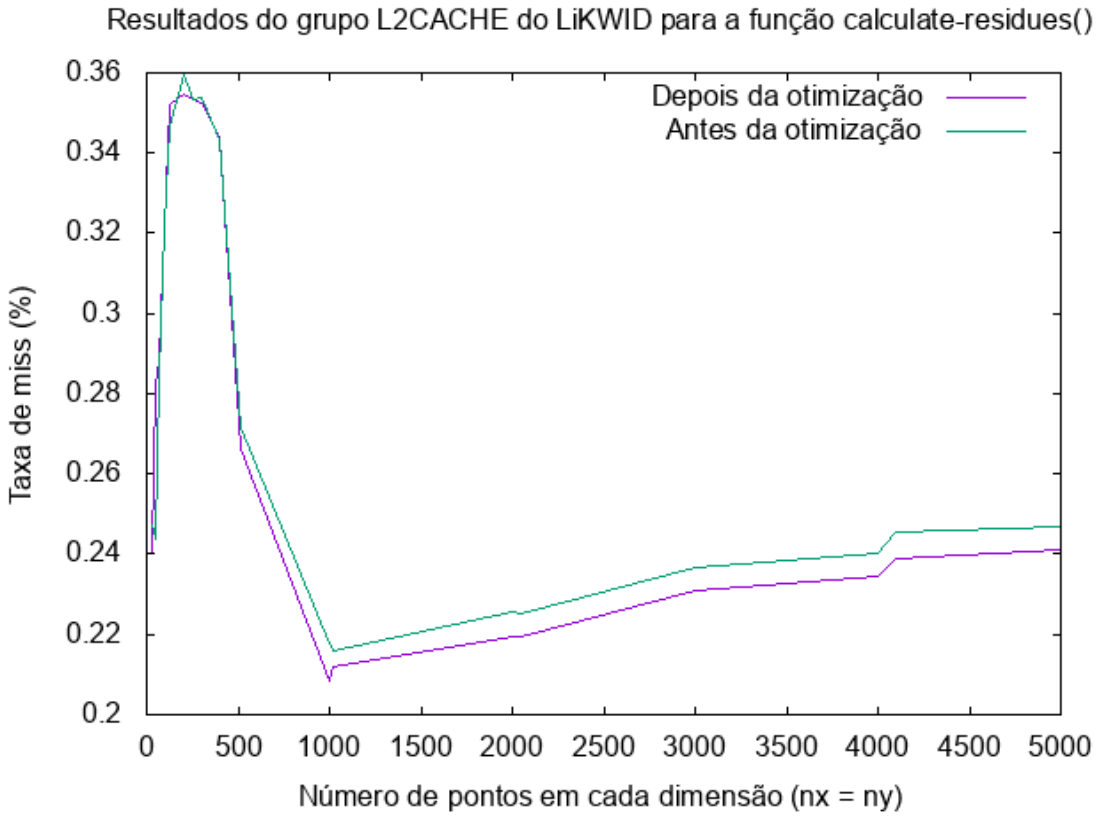

Figura 11. ”Miss ratio”na memória cache L2 para a função calculate_residues().

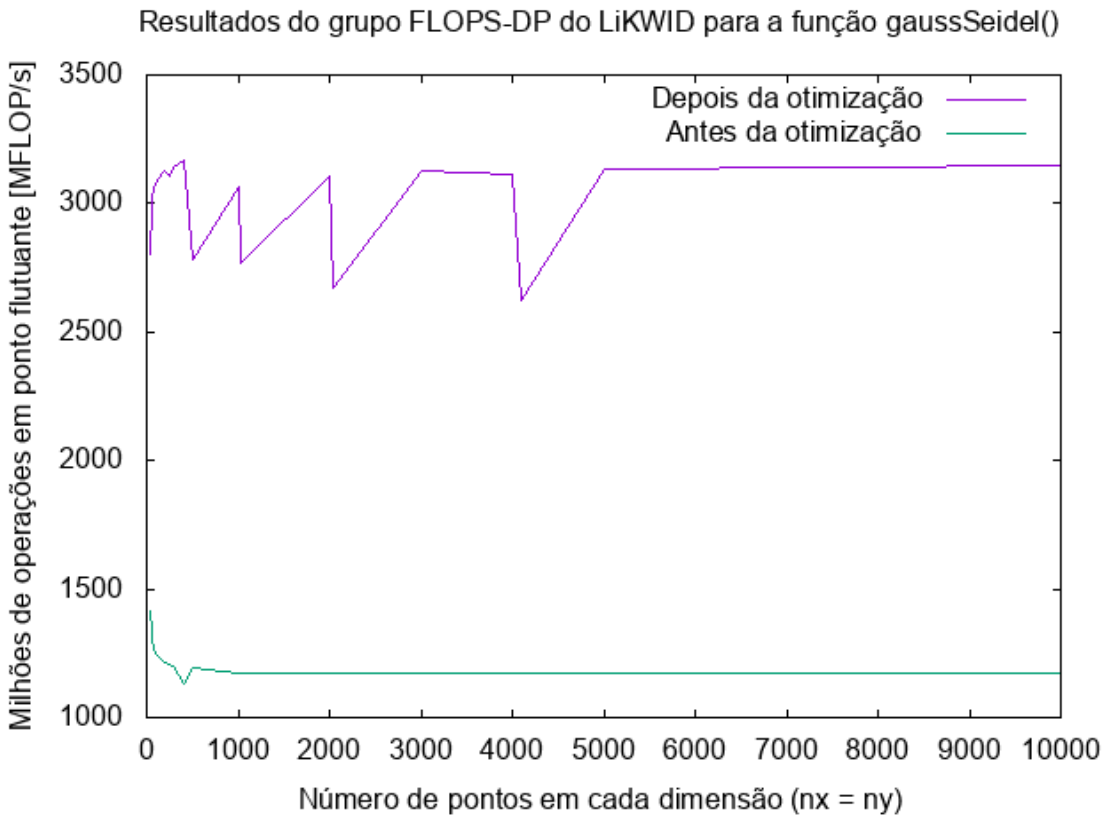

Figura 12. MFLOP/s para a função gaussSeidel().

4.4.1. Número de operações de ponto flutuante por intervalo de tempo para a função que implementa o método de Gauss-Seidel

Observando o gráfico da figura 12, nota-se substancial crescimento no indicador após as otimizacões. Isso dá-se por conta da série de técnicas anteriormente citadas, que foram aplicadas à função `gaussSeidel()`, especialmente o *loop unrolling*. Desta forma, a função e seus dois laços de repetição passam a executar quase quatro vezes mais operações em ponto flutuante por iteração, o que explica a alta no indicador.

4.4.2. Número de operações de ponto flutuante por intervalo de tempo para a função que implementa o cálculo do resíduo da iteração

Semelhante à função `gaussSeidel()`, a função `calculate_residues()` obteve substancial crescimento no indicador de pontos flutuantes, entretanto, sem a técnica *loop unrolling*. Aqui, o aumento significativo deve-se à drástica redução na quantidade de chamadas de função (por conta da substituição da função `calculate_index()` por uma macro) e nas atribuições (pois, depois de otimizada, a função passa a executar apenas uma atribuição dentro dos laços, como demonstrado nas figuras 2 e 3), além das outras otimizações anteriormente citadas.

Referências

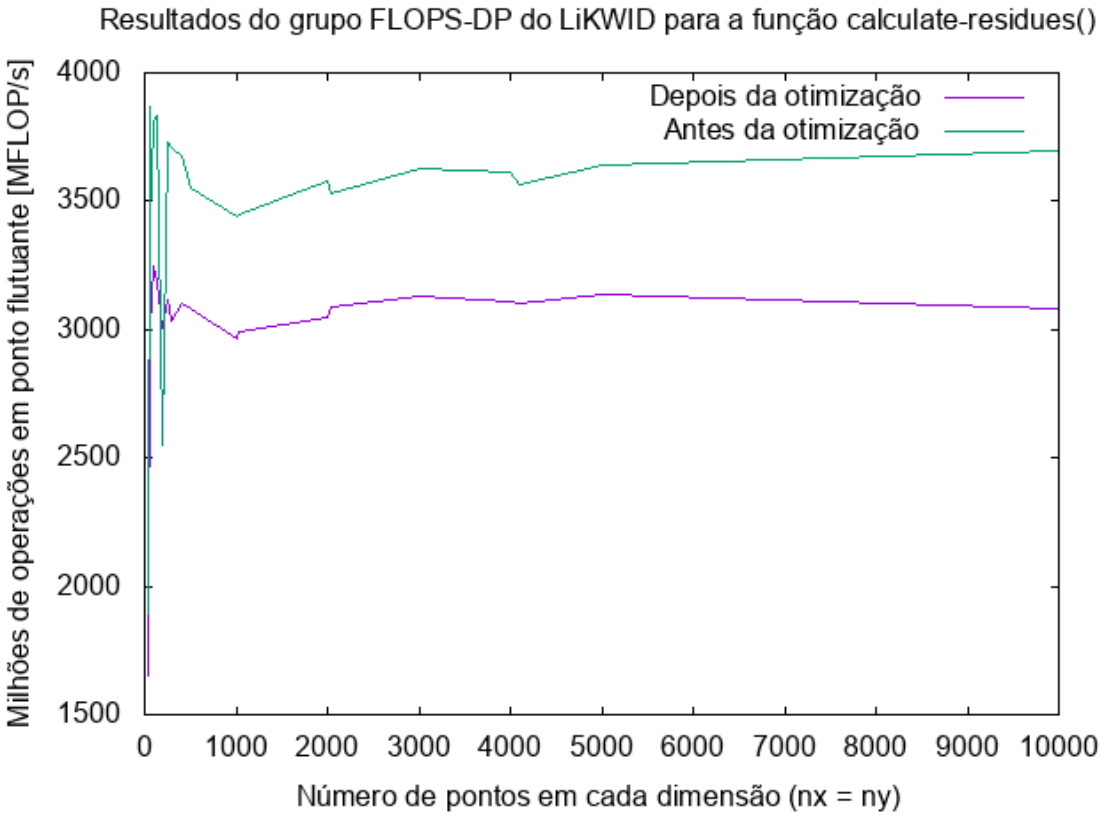

Figura 13. MFLOP/s para a função calculate_residues().