Relatório: Trabalho ICC Parte 2

Guilherme Bastos de Oliveira
GRR20167818
gbo16@inf.ufpr.br

Mariana Carmin
GRR20167282
mc16@inf.ufpr.br

Departamento de Informática - Universidade Federal do Paraná
Novembro de 2017

Resumo

Este relatório irá apresentar e discutir os resultados obtidos na segunda parte do trabalho de Introdução a Computação Científica. O objetivo foi otimizar o código da primeira parte utilizando os conceitos apreendidos em aula. O resultado foi satisfatório em quesito de desempenho avaliando o tempo de execução, porém não foi possível aplicar corretamente todas as melhorias planejadas. Ao longo do texto, uma breve explicação sobre o funcionamento do código será feita, depois uma exposição detalhada das alterações e por fim uma análise dos resultados.

Sumário

Lista de Gráficos

Lista de Códigos

1 Introdução

Este relatório acompanha um pacote de arquivos que contém os códigos analisados e informações sobre o sistema utilizado para os testes, todos os arquivos citados podem ser encontrados neste pacote.

Para realizar a segunda parte do trabalho o código criado durante a primeira parte foi completamente reescrito, para que ficasse mais claro e fácil de ser alterado. Nessa refatoração nenhuma tentativa de otimização foi realizada, o

objetivo foi aplicar as mesmas técnicas da primeira versão implementada, porém de maneira mais clara e modular.

A partir de agora a segunda versão não otimizada será referenciada como *original*. O código com todas as alterações realizadas para esta etapa do trabalho será referenciada como *otimizada*.

Na seção seguinte algumas explicações sobre a estrutura do trabalho e nomenclaturas serão apresentadas.

O principal foco das otimizações aplicadas possui relação com acesso à memória, e cada alteração será explicada na seção 3.

Os resultados obtidos com a análise de desempenho utilizando a *marker API* do LIKWID serão descritos na seção 4.

Por fim na seção 5 serão apresentadas considerações finais.

2 Informações Sobre o Trabalho

Para garantir um compreendimento melhor sobre o que foi realizado é importante definir termos que serão utilizados ao longo do texto e compreender basicamente o funcionamento do programa. Nesta seção serão apresentadas algumas nomenclaturas utilizadas para designar certas estruturas e funções, posteriormente uma sequência das etapas de execução do código será mostrada.

Para facilitar o compreendimento do código em si é possível ler um breve sumário dos arquivos fontes no *readme.md* e gerar uma documentação, instruções para isso estão também no arquivo *readme.md*.

2.1 Termos

- **N:** tamanho do lado de uma matriz quadrada (NxN)
- **matriz entrada:** refere-se ao vetor que armazena a matriz cuja inversa deverá ser calculada
- **matriz inversa:** refere-se ao vetor que armazena a matriz inversa
- **matriz LU:** refere-se ao vetor que armazena as matrizes L e U juntas
- **matriz L:** refere-se ao vetor que armazena apenas a matriz L
- **matriz U:** refere-se ao vetor que armazena apenas a matriz U
- **decompLU:** refere-se à função decompLU()
- **resolveSistema:** refere-se à função resolveSistema()

2.2 Etapas de Execução

1. Leitura de dados
2. Alocação de Memória

3. Decomposição LU (eliminação de Gauss com pivotamento parcial)
4. Primeira solução do sistema para encontrar a inversa
5. Refinamento da solução obtida
   (a) transposição matriz entrada
   (b) cálculo da matriz de resíduo
   (c) resolução do sistema com resíduo no *right hand side*
   (d) acréscimo da solução do sistema a matriz inversa
   (e) todas as iterações realizadas? fim refinamento : volta ao passo (b)
6. Impressão do tempo de execução de cada operação
7. Impressão da matriz inversa obtida

3 Alterações

Antes de iniciar o detalhamento das alterações realizadas é interessante informar que a técnica de *loopblocking* foi testada, porém não gerou resultado positivo no desempenho do código, e por esse motivo foi retirada da versão otimizada final e não será discutida neste trabalho.

A falta de resultado positivo com o uso de *loopblocking* se dá pelo fato de os laços realizarem acessos de dados de maneira desfavorável a tal técnica.

O código otimizado permitiu que ocorressem operações utilizando AVX, como poderá ser visto melhor na análise de operações de ponto flutuante, mas o feito de utilizar as operações de modo eficiente não foi alcançado.

O teste para isso se baseou em comparar o desempenho do código compilado com as *flags* padrões do trabalho (*-O3 -mavx -march=native*) com o desempenho do mesmo código compilado com as mesmas flags, exceto pela substituição de *-mavx* por *-mno-avx*. O resultado obtido não foi satisfatório para assumir o uso eficiente de AVX.

Todos os arquivos que serão comentados nesta seção podem ser encontrados no diretório raiz do trabalho.

3.1 Alocação de Memória

A alocação de memória na versão original ocorria apenas com a chamada da função **malloc()** da *libc*, essa função não garante alinhamento, por este motivo na versão otimizada foi utilizada a função **posix_memalign()** da *libc*.

3.1.1 Modificações Implementadas

Para chegar no resultado desejado duas principais mudanças foram feitas no código, primeiro foi definida uma macro (ver Código 1) que aloca uma quantidade de memória *t*, e faz o ponteiro *ptr* apontar para o inicio deste espaço.

```
22: #define alocaAlinhado(ptr,t) (posix_memalign((void **)(ptr),32,
    sizeof(double)*(t)))
```

Código 1: macro alocaAlinhado (utils.h)

Com essa definição para auxiliar nesta tarefa, todas as chamadas de **malloc()** foram substituídas pelo usa de *alocaAlinhado*.

3.1.2 Resultados

A motivação inicial para esta modificação era tornar o uso de AVX mais eficiente e permitir acesso de memória mais otimizado, infelizmente como já comentado na introdução os resultados não foram satisfatórios.

Pela falta de resultados claros não foi possível determinar exatamente o efeito que a alteração teve sobre o desempenho. O tempo gasto pela versão com AVX e o tempo gasto pela versão sem AVX é virtualmente idêntico.

3.2 Acesso da Matriz Inversa

A versão original utiliza uma matriz para armazenar a solução obtida pelo programa. Os acessos de memória dessa matriz geralmente ocorrem de maneira transposta, ou seja, existiam mais acessos feitos em colunas → linhas do que linhas → colunas.

Na versão otimizada o acesso foi invertido na maior parte dos casos, ou seja, a matriz é mais acessada por linhas → colunas e como isso ocorre tanto na escrita quanto na leitura os dados estão na ordem correta e são lidos de maneira sequencial da memória.

3.2.1 Modificações Implementadas

Em todos os acessos à matriz inversa os índices i e j foram invertidos, garantindo sequencialidade em quase todos os casos (ver Códigos 2, 3 e 4), exceto na impressão de matriz (ver Código 5). Como a impressão ocorre apenas uma vez durante o código e é composta apenas por leitura de dados o ganho no resto dos acessos compensa.

```
204: aux = inversa[C(i,j)];
```

Código 2: acesso invertido (core.c)

```
224: FOR_IT(N, inversa[C(coluna,IT)] += x[IT])
```

Código 3: acesso invertido (core.c)

```
68: FOR_IT(N, inversa[C(coluna,IT)] = x[IT])
```

Código 4: acesso invertido (invmat.c)

```
26: fprintf(arq, "%.17g ", matriz[C(j,i)]);
28: fprintf(arq, "%.17g\n", matriz[C(N-1,i)]);
```

Código 5: acesso normal (entrada_saida.c)

3.3 Matrizes L e U

o acesso as matrizes LU ocorria de maneira fragmentada na versão original. Isso se deve ao fato dos algoritmos de *forward/backward substitution* não terem sido modificados em sua implementação para realizar um acesso mais eficiente. Na versão otimizada isso foi alterado completamente, permitindo um acesso sequencial tanto de L quanto de U na resolveSistema.

3.3.1 Modificações Implementadas

Primeiramente as matrizes L e U foram separadas, cada uma armazenada em um vetor, para que seus dados estivessem em ordem na memória (ver Código 6). Como a estrutura foi alterada o método de acesso também foi alterado, para isso duas definições foram criadas (ver Código 7).

```
28: || alocaAlinhado(L,N*(N-1)/2)
29: || alocaAlinhado(U,N*(N+1)/2)
```

Código 6: alocação de L e U (utils.c)

```
34: #define indiceL(i,j) ((((i)*((i)+1)/2)+(j))-(i))
40: #define indiceU(i,j) (((N-1-(i))*(N-(i))/2)+N-1-(j))
```

Código 7: macros para acesso de L e U (utils.h)

Foi necessário devido a modificação da estrutura de dados uma atualização na maneira de funcionamento de decompLU, além disso, por um motivo que será explicado a seguir foi necessário realizar uma reorganização dos vetores L e U no final da decomposição (ver Código 8).

```
104: for(int i = 0; i < N; i++){
105:     for(int k = i+1; k < N; k++){
106:         Lt[indiceLT(i,k)] = L[indiceL(k,i)];
107:     }
108: }
111: for(int i = N-1; i >= 0; i--){
112:     Ut[indiceUT(i,i)] = U[indiceU(i,i)];
113:     for(int k = i-1; k >= 0; k--){
114:         Ut[indiceUT(i,k)] = U[indiceU(k, i)];
115:     }
116: }
```

Código 8: reordenação de L e U (core.c)

A otimização ocorreu na função resolveSistema, que teve a ordem dos laços alteradas para satisfazer o objetivo da alteração (ver Código 9). Com as definições de acesso de L e U existentes não era possível converter um endereço de 2 índices para de 1 índice corretamente, por esse motivo foram implementados duas novas definições (ver Código 10).

```
140: for(int i = 0; i < N; i++){
141:     yy = y[i];
142:     for(int k = i+1; k < N; k++){
143:         y[k] -= L[indiceLT(i,k)] * yy;
144:     }
```

```
145: }
148: for(int i = N-1; i >= 0; i--){
149:     xx = y[i]/U[indiceUT(i, i)];
150:     for(int k = i-1; k >= 0; k--){
151:         y[k] -= U[indiceUT(i, k)] * xx;
152:     }
153:     x[i] = y[i]/U[indiceUT(i, i)];
154: }
```

Código 9: laço de solução de sistema (core.c)

```
46: #define indiceLT(i,j) (((N-1+N-(i))*i/2)+((j)-1-(i)))
52: #define indiceUT(i,j) ((N+(i))*(N-1-(i))/2+(N-1-(j)))
```

Código 10: macros para acesso modificado de L e U (utils.h)

3.3.2 Custos Adicionais

Como na função decompLU é necessário realizar a cópia da matriz entrada para LU a versão original era mais eficiente neste aspecto, pois tanto na entrada quanto em LU o acesso era sequencial, com a estrutura da versão otimizada as alterações fizeram com que os acessos não ocorressem de forma sequencial devido a separação de LU em L e U.

Ainda na função decompLU é necessário realizar a reorganização dos elementos, o que custou mais memória (2 vetores temporários, um para L, outro para U) e o custo de acesso a memória na reorganização propriamente dita.

3.3.3 Dificuldades

Para implementar as alterações algumas dificuldades foram enfrentadas, vale ressaltar a dificuldade para montar as formulas de acesso LT e UT, a modificação dos laços (tanto a ordem de acesso quanto os limites do laço) e por fim as alterações na decompLU.

A função decompLU demandou cuidado pois para replicar o comportamento da versão original com a estrutura da versão otimizada foram adicionadas condicionais nos laços de repetição, como isso causa perda de desempenho, todos os laços foram segmentados de maneira a evitar o uso de condicionais. A reorganização também foi complicada pois exigiu atenção com os índices que se tornaram demasiadamente confusos.

3.4 Matriz Resíduo

A matriz resíduo desempenha um grande papel na execução do código pois é acessada para escrita durante cada iteração, por isso foi decidido realizar acesso sequencial desta matriz.

3.4.1 Modificações Implementadas

O laço de cálculo de resíduo foi alterado (ver Código 11) para garantir um acesso linear (mas ainda assim "redundante"devido ao algoritmo de multiplicação utilizado) da matriz resíduo.

```
201: for (i=0;i<N; i++) {
202:     residuo[C(i,i)] += 1;
203:     for (j=0; j<N; j++) {
204:         aux = inversa[C(i,j)];
205:         for (k=0; k<N; k++) {
206:             residuo[C(i,k)] -= (entrada[C(j, k)] * aux);
207:         }
208:     }
209: }
```

Código 11: laço do cálculo do resíduo (core.c)

Esta alteração do laço fez com que o acesso à matriz de entrada fosse realizado de forma não sequencial, com a matriz armazenada de maneira transposta o acesso ocorreria de maneira ideal, por isso é realizada antes do processo de refinamento a transposição da matriz entrada (ver Código 12).

```
184: for(i = 0; i < N-1; i++){
185:     for(j = i+1; j < N; j++){
186:         double swap = entrada[C(i,j)];
187:         entrada[C(i,j)] = entrada[C(j,i)];
188:         entrada[C(j,i)] = swap;
189:     }
190: }
```

Código 12: transposição da matriz entrada (core.c)

3.4.2 Custos Adicionais

O custo de transpor a matriz entrada foi adicionado, pois esta operação não era necessária anteriormente e não substitui nenhuma outra, portanto resultou diretamente apenas em custo adicionado.

Durante o cálculo de resíduo a matriz inversa é acessada de forma fragmentada temporalmente, ou seja, para cada acesso de um elemento da inversa, são acessados N elementos do resíduo e da matriz entrada antes do acesso ao próximo elemento da inversa.

3.4.3 Dificuldades

As modificações no laço se mostraram menos triviais do que o esperado, e a transposição da matriz entrada exigiu dedicação para ocorrer de forma correta devido aos limites não usuais dos laços utilizados na operação.

4 Análise de Desempenho

Nesta seção serão apresentados gráficos que exibem os dados coletados com o auxilio do pacote de ferramentas LIKWID, chamadas da função *timestamp()* e o comando ***time***.

Os gráficos possuem um padrão em sua exibição, o eixo das abscissas representa o tamanho de um lado da matriz quadrada (N), já as ordenadas são relativas ao valor coletado em escala logarítmica.

4.1 Obtenção de Dados

Para obter os dados necessários para a análise foram utilizadas algumas técnicas diferentes, variando entre os dados desejados.

Isso exigiu várias execuções diferentes do mesmo programa, com pequenas variações nos parâmetros e tratamentos dos dados de saída, para isso o arquivo *bash.sh* foi criado e executado. A organização dos diretórios e arquivos durante o desenvolvimento foi alterada para a entrega, assim o arquivo *bash.sh*, se executado, não irá apresentar o funcionamento esperado.

O ambiente de teste utilizado foi uma máquina rodando o sistema operacional Ubuntu versão 16.04.1, operando sobre um processador Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz, é possível observar informações adicionas sobre o processador no arquivos *topologia*.

4.1.1 Tempo

Com a média da diferença entre uma chamada da função no fim de uma parte do código com uma chamada no começo desta parte de código foram obtidos os dados que compõe o Gráfico 2.

Foi feita também a medição do tempo total de execução de cada versão com o uso do comando ***time*** do *bash*, é possível ver os dados no Gráfico 1.

4.1.2 Cache L2 - Miss Ratio

Para coletar o *miss ratio* da memória cache L2 foi usada a API de marcação do LIKWID. O Gráfico 5 contém o que foi coletado.

4.1.3 Operações de Ponto Flutuante

Com a API de marcação do LIKWID foram analisadas a quantidade de MFLOP/s em duas modalidades, operações DP gerais e operações DP AVX, os Gráficos 3 e 4 mostram o resultado respectivamente.

Como não foi possível realizar operações AVX na versão original, o valor de MFLOP/s é 0 para todas as execuções e portanto não é exibido no Gráfico 4.

4.1.4 Cache L3 - Bandwith

A medição da *bandwidth* da memória cache L3 foi obtida com o auxilio da API de marcação do LIKWID. No Gráfico 6 é possível observar os dados obtidos.

4.2 Análise dos Dados Obtidos

Abordando primeiramente o Gráfico 1, é possível perceber que com matrizes de tamanhos pequenos (N até 65), o desempenho em questão de tempo é melhor obtido com a versão original, e após certo tamanho de N a velocidade da versão otimizada é mais rápida, em 2 vezes na média, com N entre 128 e 2000.

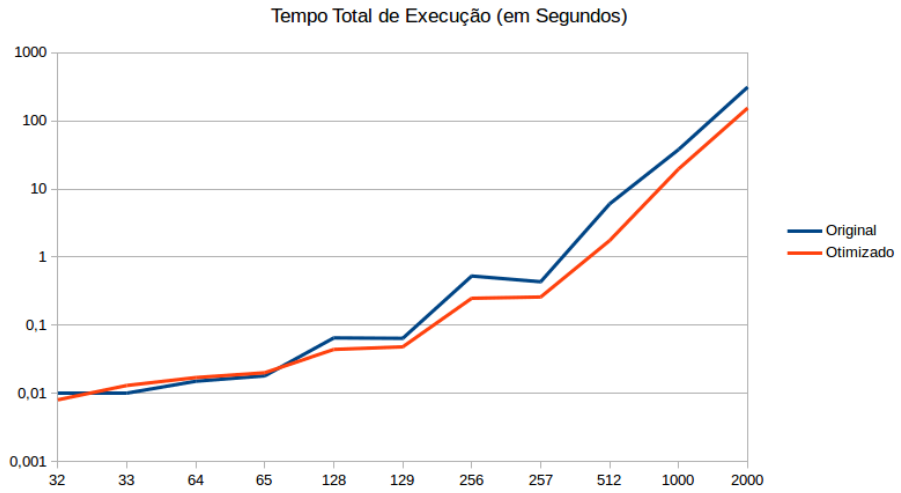

Gráfico 1: tempo de execução

Isso é num primeiro momento contra intuitivo. Com uma investigação sobre o Gráfico 2, é possível perceber com maior precisão onde ocorre essa perda de desempenho na versão otimizada.

A operação 1 possui um comportamento próximo nas duas versões, quando N é pequeno, então a diferença do desempenho não deve ser atribuída as modificações realizadas neste trecho de código.

A operação 2 por sua vez apresenta um diferença para N até 33, porém para N maiores, a versão otimizado apresenta um desempenho maior, então é evidente que isso teve influência na diferença de desempenho, mas não explica o resultado para N igual a 64 e N igual a 65.

Continuando a análise agora pela quantidade de operações ponto flutuante executadas por segundo, mostrada no Gráfico 3, é concebível que, o número mais baixo de Mflop/s durante a operação 1 na versão otimizada em relação a versão original para N menor que 64, esteja relacionado ao desempenho obtido

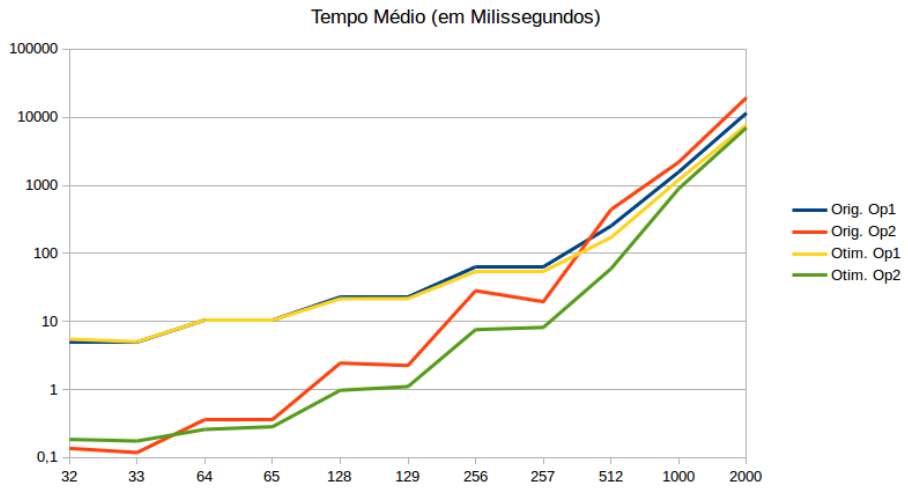

Gráfico 2: tempo médio

nestas execuções. Porém ainda não é possível determinar se é o um efeito ou a causa do problema.

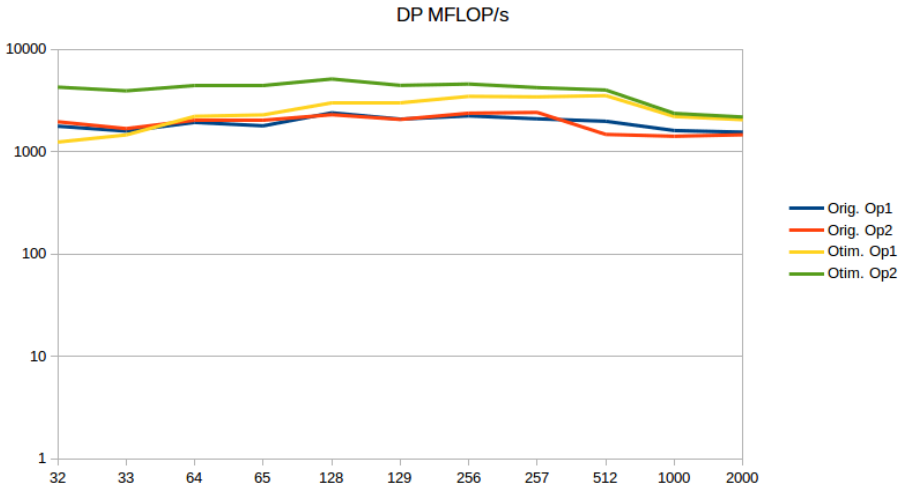

Gráfico 3: operações ponto flutuante

Como já explicado, a versão original não produz instruções AVX, portanto a análise do Gráfico 4 não trará informações para explicar a discrepância do desempenho.

Os dados de *miss ratio* da memória cache L2 e de *bandwidth* da cache L3,

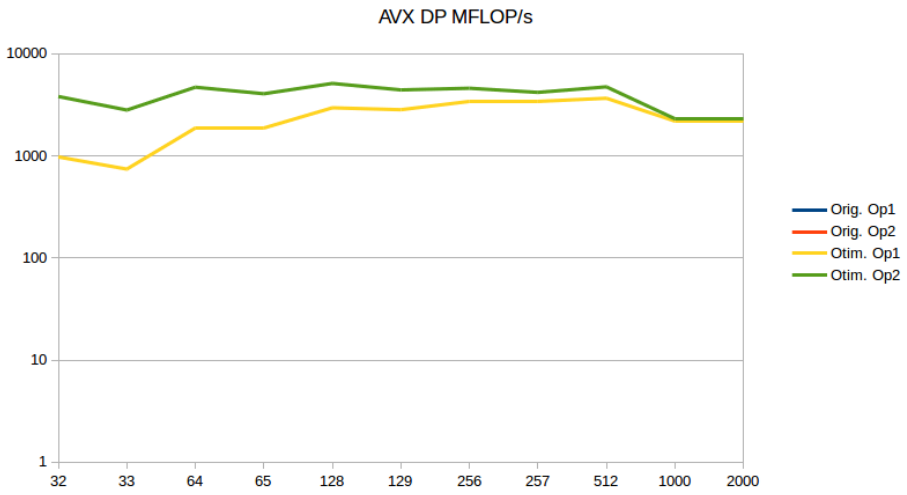

Gráfico 4: operações AVX

que podem ser visualizados nos Gráficos 5 e 6 respectivamente, apresentam um padrão de comportamento não óbvio.

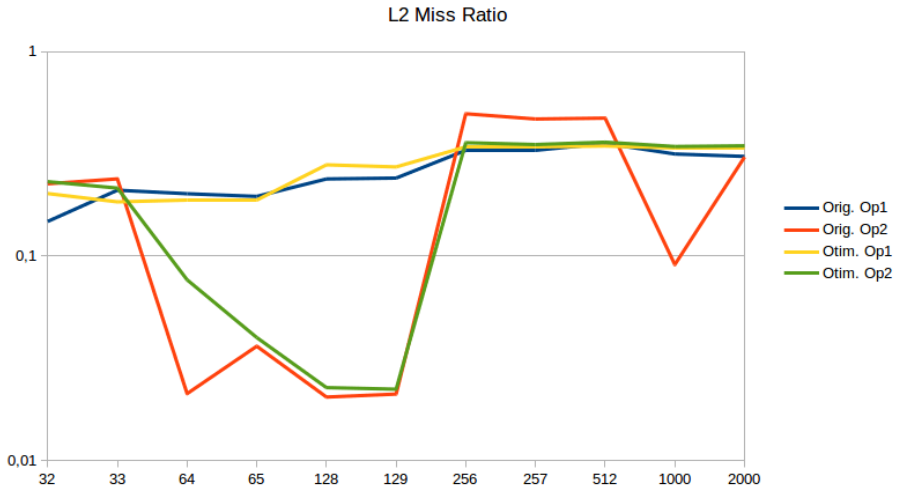

Gráfico 5: miss ratio da L2

É possível assumir que uma possível causa da perda de desempenho é o aumento de *cache misses* para a operação 2 na versão otimizada com N até 65.

Para a diferença de desempenho, as informações sobre *bandwidth* da cache L3 não possuem um resultado que indique um efeito direto no tempo de execução

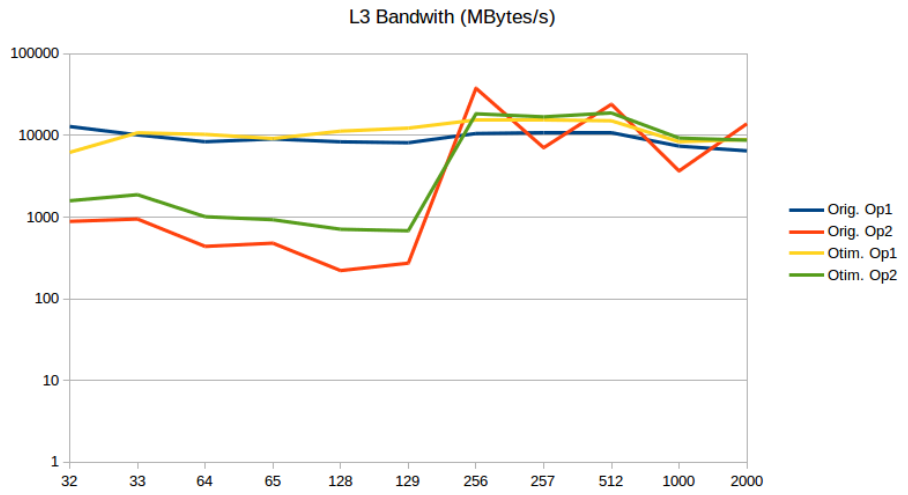

Gráfico 6: bandwidth da L3

para N pequeno.

4.3 Possíveis Conclusões

É importante ressaltar o fato de que os dados utilizados são apenas de um sequência de execução do programa para cada conjunto de dados obtidos, portanto é possível que pequenas interferências como variações da configuração dinâmica do consumo da CPU e distribuição pelo escalonador de processos ocorram. Para dados mais acurados seria necessário executar o programa múltiplas vezes com uma mesma entrada, e calcular a média da saída, mas o tempo necessário para isso foge do escopo do projeto.

Com o esclarecimento sobre a acurácia dos dados, uma possível explicação para o comportamento inesperado da versão otimizado, com N até 65, seriam as pequenas interferências, uma vez que o tempo de execução total pode ter um aumento de acordo com o impacto das interferências. Isso não seria muito aparente para N maiores pois o impacto das interferências não aumentam proporcionalmente com o tempo de execução de um programa.

As potenciais interferências não fornecem uma resposta totalmente satisfatória e nem todas as alterações do código foram medidas individualmente, então é possível que alguma parte externa da op1 e op2 tenha causado o tempo maior para alguns casos de execução da versão otimizada. As alterações expostas na Seção 3 que não atuam somente nas operações 1 e 2 são as explicadas nas subseções 3.1, 3.2 e 3.3.

A alocação de memória dificilmente teve influência no consumo de tempo de modo significativo, uma vez que ainda ocorre na mesma quantidade e a função *posix_memalign()* não tem um custo significativamente superior a função

malloc(). Portanto a alteração da subseção 3.1 não deve é responsável pela divergência de desempenho.

As alterações causadas pela mudança de acesso a inversa que não estão incluídas nas operações 1 e 2 são o acesso para montar a primeira solução calculada e a impressão da matriz, mostradas nos Códigos 4 e 5 respectivamente. A montagem da solução ocorria na versão original com um acesso não sequencial de memória, a versão otimizada gera um acesso sequencial, portanto isso resulta em uma melhoria no tempo, não sendo o motivo de maior gasto de tempo. Já a impressão da matriz, realiza na versão otimizada, acesso não sequencial, porém como só é chamada uma vez em cada execução e o acesso é somente de leitura o impacto não seria suficiente para gerar o gasto observado.

Com a separação de LU em L e U ocorreram várias adições durante o processo de decomposição (na função decompLU), as alterações que poderiam causar maior impacto seriam a reorganização de L e U ao final da decomposição (mostrada no Código 8) e as alterações realizadas no processo de pivotamento, que fazem acessos não sequenciais a L e a U.

É muito provável que as alterações em decompLU tenham causado o aumento no tempo, uma vez que com matrizes pequenas (que caibam inteira, ou quase inteiramente na cache) a única implicação das alterações são mais operações executadas e acessos de memória, porém com matrizes grandes o custo de realizar este procedimento é amortizado pelo desempenho ganho em cada execução da função resolveSistema.

A operação 1 otimizada (exibida no Código 9) na maior parte de sua execução realiza acesso de uma informação, trabalha sobre ela, e nunca volta a utilizar a mesma (até a próxima chamada), portanto o *miss ratio* sobre a L2 é quase constante (como visto no Gráfico 5).

A mesma relação do motivo do *ratio* constante com o acesso realizado na operação 1 otimizada por ser feito com a *bandwidth* da L3 que ocorre durante essa operação (conforme Gráfico 6).

O comportamento da operação 2 otimizada em relação ao *miss ratio* da L2 (Gráfico 5) pode ser explicado pelo fato de que com matrizes pequenas (N até 34) ocorrem muitos misses pois não existe ainda nenhum dado na memória e quando todo o conteúdo acessado está carregado, os dados não são mais utilizados. Com matrizes médias (N entre 64 e 129) ao passo em que a matriz aumenta, o acesso repetido da mesma informação é realizado, com isso se aproveitando da cache e diminuindo os *misses*, até um ponto onde N é grande demais (N a partir de 256) para caber completamente na memória cache e volta a gerar um grande número de cache *misses*.

É possível imaginar que algo parecido ocorre com a operação 2 otimizada se tratando da *bandwidth* da L3, uma vez que a quantidade de memória transitada é diretamente afetada pela quantidade de memória acessada que não se encontra na cache. Até certo tamanho de matriz (N igual a 129) grande parte dos dados operados cabem na cache, com matrizes maiores falta espaço para dados na cache e mais memória é solicitada e assim ocorre uma maior movimentação de dados.

Para maiores conclusões seriam necessários novos experimentos, como a

medição da quantidade total de memória acessada e o número de operações de ponto flutuante realizadas. A coleta desses dados foge do escopo deste trabalho e por isso não foi realizada.

5 Considerações finais

Para concluir o trabalho foi necessário um entendimento profundo do código produzido e dos conceitos básicos de otimização. A refatoração do código foi de grande serventia, promovendo uma maior clareza sobre como o processo realizado ocorre e uma flexibilidade maior para aplicação de alterações. Foi necessário também lidar com a frustração de não atingir todos os objetivos buscados.

O resultado obtido não foi o planejado, mas quando considerado a dificuldade em planejar e implementar as modificações se torna satisfatório. Seria possível com mais tempo e pesquisa modificar ainda mais a solução encontrada, alcançando assim um desempenho ainda maior.

A análise não trouxe certezas sobre os motivos dos resultados, mas as teses derivadas da análise são o suficiente para compreender melhor como as alterações impactaram nos resultados do trabalho. Com mais testes, considerando um espaço amostral maior (aumentado variação de N e realizando execuções repetidas) e coletando informações adicionais seria possível analisar novos dados em busca de uma compreensão ainda maior sobre as alterações.