Otimização - Inversão de Matrizes

Stephanie Briere Americo[1], **Talita Halboth Cunha Fernandes**[1]

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

{sba16,thcf16}@inf.ufpr.br

Resumo. *Este trabalho apresenta as otimizações feitas para melhorar o desempenho do programa computacional desenvolvido durante a disciplina de Introdução à Computação Científica para inverter matrizes. Para testes com matrizes de tamanho igual ou superior a 1000x1000 a versão otimizada leva menos de 30% do tempo total de execução da versão original para computar a matriz inversa. As modificações realizadas e suas consequências são descritas aqui de forma bastante expositiva, com a ajuda de gráficos do software* `likwid`.

1. Introdução

Durante a primeira parte do semestre, na disciplina de Introdução à computação Científica, desenvolvemos um programa computacional que, dada uma matriz quadrada A de dimensão n, encontre a matriz inversa de A (inv(A)), tal que A * inv(A) = I, onde I é a matriz identidade.
Na segunda parte do semestre foi então proposta a otimização do programa. Durante as aulas foram apresentadas algumas formas de fazer essa otimização, e consideramos quais modificações no código trariam uma melhora em seu desempenho, principalmente para matrizes muito grandes.
As implementadas são descritas na seção 2. Para saber quais foram efetivamente as diferenças entre a versão otimizada e a versão original utilizamos funções da biblioteca `likwid`. Realizamos testes de desempenho com vários tamanhos de matrizes, e os resultados dos testes estão descritos na seção 3. Para plotar os gráficos apresentados, utilizamos o `gnuplot`, ferramenta software livre desenvolvida especificamente para representação de gráficos. Na última seção apresentamos os aspectos que a equipe considerou mais relevantes/importantes no desenvolvimento do trabalho.

2. Mudanças Realizadas

2.1. Função `pos ()`

Uma das primeiras modificações realizadas no código foi a modificação da função `pos()`, utilizada para calcular a posição `(i,j)` de uma matriz de tamanho `tam`.

```
unsigned int pos(unsigned int lin, unsigned int col, unsigned int tam)
  return (lin*tam + col);
}
```

`pos()` era chamada quase de 30 durante no código, e, em sua maioria, dentro de loops `for`. Definimos então uma macro ao invés de usar a função. Esta modificação foi umas das mais significativas, pois durante a compilação, em todos os lugares onde é chamada a macro, ela é substituída, evitando chamadas desnecessárias de funções.

2.2. Transposição da Matriz

Outra modificação realizada foi utilizar a matriz inversa transposta, para diminuir o cache miss. Da forma que foi estruturado o código, não foi necessário literalmentr transpor a matriz - esta operação é de complexidade $O(n^2)$, se implementada de maneira ingênua, além de ser necessário alocar mais memória, então seu custo muitas vezes supera o ganho. No nosso caso, foi necessário apensar alterar a função `FatoracaoLU ()` e trocar os índices quando outras funções acessam a matriz inversa, como no exemplo:
Versão Original:

```
for (int sl = 0; sl < tam; ++sl) {
        for (int k = 0; k < tam; ++k)
                b[k] = y->dados[pos(k,sl,tam)];
```

Versão Otimizada:

```
for (int sl = 0; sl < tam; ++sl) {
        for (int k = 0; k < tam; ++k)
                b[k] = y->dados[pos(sl,k,tam)];
```

2.3. Alocação de Memória

A memória das matrizes antes era alocada utilizado apenas a função malloc em alguns casos, e a matriz inversa era alocada com calloc. Passamos então a utilizar a função `memalign()` para essa alocação. Além de não percorrer toda a matriz zerando seus elementos, essa função permite a realização de operações AVX de forma mais eficaz. Além disso, todas as as matrizes são alocadas com um tamanho múltiplo de $4*4$, também para caber melhor na cache. O cálculo da posição (lin,col) para uma matriz de tamanho tam considerando o seu tamanho é o sequinte:

```
#define pos(lin, col, tam) (lin*(tam+(4- tam%4))+ col)
```

2.4. Utilização do qualificador de tipo `restrict`

O qualificador de tipo `restrict` serve para dizer ao compilador que, durante todo o tempo de execução do programa, se um ponteiro p é declarado com o qualificador, apenas ele aponta para seu endereço de memória. Não há, por exemplo, nenhum ponteiro q que aponte para [p+50]. Isto ajuda a produzir código mais otimizado.

2.5. Remoção da Soma de Kahan

Muitas vezes, ao trabalhar com números de ponto flutuante, um programador deve tomar decisões que afetam a precisão ou a velocidade do programa. Este caso é um deles. Optamos por diminuir ligeiramente a precisão do programa, para conseguir resultados mais rápido. A soma de Kahan dificulta algumas otimizações porque cada valor depende do valor anterior em uma soma. Ao remover essa dependência, abrimos espaço para que o compilador realizasse mais otimizações.

3. Análises

As análises foram realizadas em uma máquina com as seguintes características:

```
CPU name:       Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
CPU type:       Intel Core Haswell processor
CPU stepping:   3
********************************************************
Hardware Thread Topology
********************************************************
Sockets:                1
Cores per socket:       4
Threads per core:       2
--------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1               0               1               0               *
2               0               2               0               *
3               0               3               0               *
4               1               0               0               *
5               1               1               0               *
6               1               2               0               *
7               1               3               0               *
--------------------------------------------------------
Socket 0:               ( 0 4 1 5 2 6 3 7 )
--------------------------------------------------------
********************************************************
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:      2
Cache groups:           ( 0 4 ) ( 1 5 ) ( 2 6 ) ( 3 7 )
--------------------------------------------------------
Level:                  2
Size:                   256 kB
Type:                   Unified cache
Associativity:          8
Number of sets:         512
Cache line size:        64
Cache type:             Non Inclusive
Shared by threads:      2
Cache groups:           ( 0 4 ) ( 1 5 ) ( 2 6 ) ( 3 7 )
--------------------------------------------------------
Level:                  3
Size:                   8 MB
```

```
Type:                   Unified cache
Associativity:          16
Number of sets:         8192
Cache line size:        64
Cache type:             Inclusive
Shared by threads:      8
Cache groups:           ( 0 4 1 5 2 6 3 7 )
--------------------------------------------------------------------------------

**************************************************
NUMA Topology
**************************************************
NUMA domains:           1
--------------------------------------------------------------------------------

Domain:                 0
Processors:             ( 0 4 1 5 2 6 3 7 )
Distances:              10
Free memory:            5791.97 MB
Total memory:           7875.85 MB
--------------------------------------------------------------------------------


**************************************************
Graphical Topology
**************************************************
Socket 0:
+-------------------------------------------------------+
| +---------+ +---------+ +---------+ +---------+ |
| |  0 4    | |  1 5    | |  2 6    | |  3 7    | |
| +---------+ +---------+ +---------+ +---------+ |
| +---------+ +---------+ +---------+ +---------+ |
| |  32 kB  | |  32 kB  | |  32 kB  | |  32 kB  | |
| +---------+ +---------+ +---------+ +---------+ |
| +---------+ +---------+ +---------+ +---------+ |
| | 256 kB  | | 256 kB  | | 256 kB  | | 256 kB  | |
| +---------+ +---------+ +---------+ +---------+ |
| +---------------------------------------------+ |
| |                    8 MB                     | |
```

O eixo y (ordenadas) do gráfico é mostrado em uma escala logarítmica de base 10, enquanto o eixo das abcissas usa escala logarítmica base 2, para facilitar a visualização dos dados.
Op1 nos gráficos é a resolução do sistema linear triangular: LUx=b → Ly=b → Ux=y;
Op2 é o cálculo do resíduo: R = I - A * inv(A);

3.1. Teste de Tempo

Como é possível ver na figura 1, em quase todos os tamanhos de matriz a resolução do sistema linear na versão otimizada é mais rápida que na versão original, e em todos eles o cálculo do resíduo é mais rápido. A diferença na maior parte das vezes é bem significativa. Essa diferença vem principalmente da utilização de uma macro ao invés de uma função

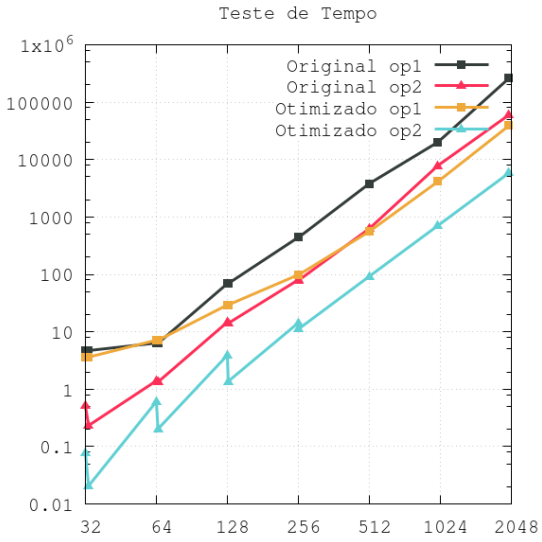

Figura 1. Análise do tempo para a realização de cada operação

para calcular a posição, mas todas modificações interferem nesse teste.

3.2. Largura de Banda

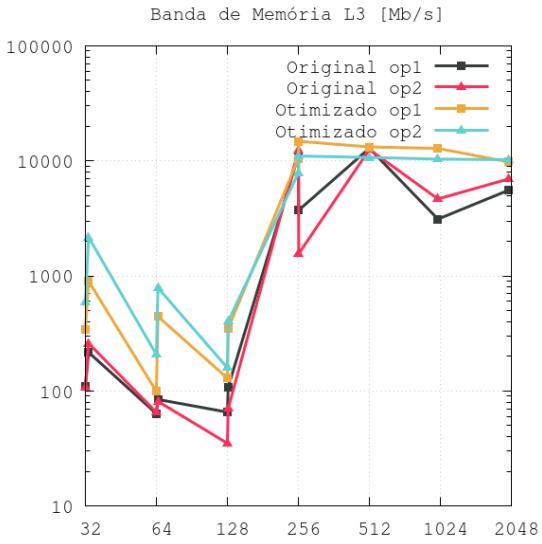

Figura 2. Análise da Banda de Memória da cache L3

O gráfico na figura 2 mostra que a Banda de Memória da versão original, para quase todos os tamanhos, é mais rápida do que a da versão otimizada. Porém, isto pode ser explicado por outro fator: O volume de dados na L3 na versão otimizada é muito menor do que na versão original, como podemos ver na figura 3. Isto significa que a L3 é necessária com menor frequência, pois há mais dados nas caches L2 e L1 - isto é consequência da utilização da matriz transposta. Comparando a variação do comportamento dos dois gráficos conforme o tamanho da matriz varia, notamos que quando a diferença na banda de memória entre as duas versões é maior, o volume de dados em compensação

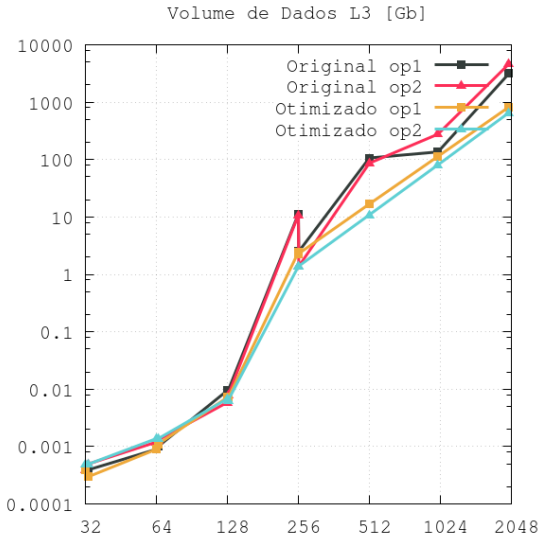

Figura 3. Análise do Volume de Dados da cache L3

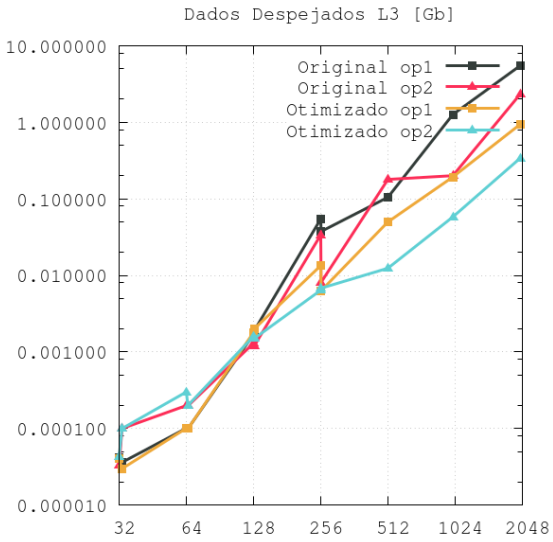

Figura 4. Análise do Volume de Dados Despejados da L3

é mais próximo, e vice-versa. Na figura 4, podemo ver também que o volume de dados despejados da cache L3 é geralmente menor na versão otimizada em comparação à versão original. Isto significa que na versão otimizada os dados são mais bem aproveitados, ao por exemplo utilizar mais comumente a localidade de referência. Isto deve-se tanto à alocação alinhada da memória quanto à utilização da matriz inversa transposta.

3.3. Cache Miss

A análise do gráfico de cache miss na figura 5 permite a inferência de que a escalabilidade da versão otimizada é muito maio do que a da versão original. O cache miss da cache L2 é mais constante na versão otimizada em ambas as operações, enquanto na versão original há muita diferença com o crescimento da matriz. Podemos observar o mesmo comportamento da taxa de requisição da mesma cache no gráfico da figura 6. As requisições

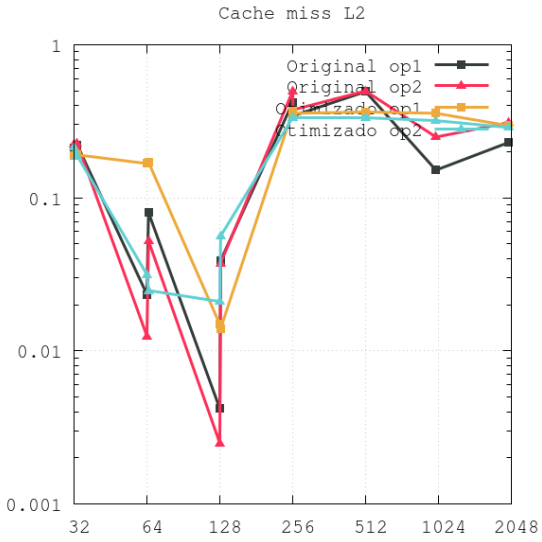

Figura 5. Análise do Cache Miss da cache L2

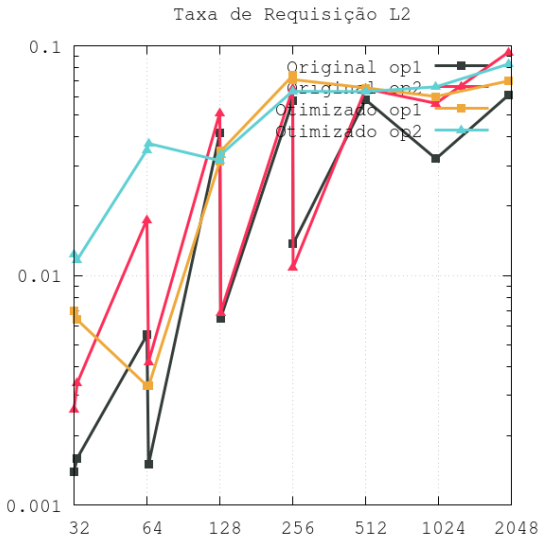

Figura 6. Análise da Taxa de Requisição da cache L2

na cache L2 pelo programa otimizado, principalmente em tamanhos maiores que 128, mantêm-se relativamente estável, enquanto para a versão original essas ocorrem de forma muito mais inconstante.

3.4. Operações Aritméticas

A quantidade de operações em ponto flutuante por segundo na versão otimizada é sempre superior à versão original. Considerando a figura 3, sabemos que há mais dados nas caches L2 e L1, e o acesso a esses dispositivos é muito mais rápido. Por isso, o processador passa menos tempo esperando pelos dados e mais tempo realizando cálculos. Além disso, com a utilização do qualificador restrict para os ponteiros, é possível ao compilador realizar mais otimizações em operações com vetores. Essa diferença nas operações é uma das principais responsáveis pela diminuição do tempo de execução na versão otimizada

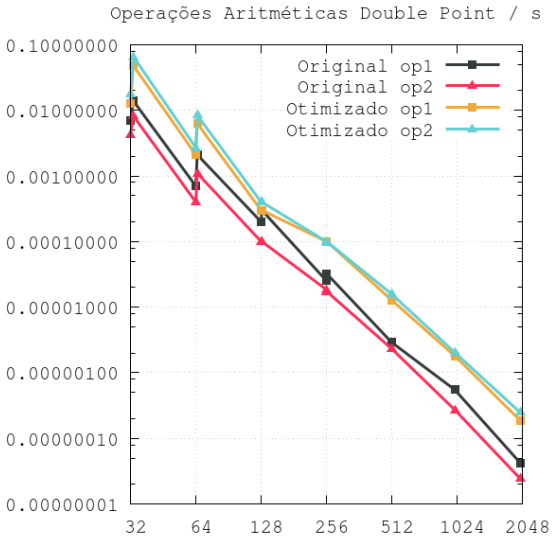

Figura 7. Análise de MFLOPS por segundo

quando comparado à versão original. No entanto, notamos que conforme cresce a matriz, a relação MFLOPS/segundo decresce. Isto acontece porque quanto maior é a matriz, mais dados precisam ser buscados da memória, e o processador fica mais tempo esperando.

4. Dificuldades

Algumas das dificuldades encontradas na otimização foram para não deixar o código incorreto depois das modificações. Além disso, saber quais mudanças fariam mais diferença trouxe alguns obstáculos. A implementação em si também acabou se mostrando um pouco confuso, principalmente na questão da manipulação dos dados. Também foi relativamente desafiadora a utilização e interpretação da saída de testes rodados com `likwid` e plotar os gráficos com as informações obtidas na saída.

5. Conclusão

Durante a realização do trabalho consideramos diversas mudanças que poderiam ser implementadas, mas só efetivamente fizemos algumas delas.
A utilização de uma matriz inversa transposta foi a principal alteração na otimização, pois uma das operações realizadas mais frequentemente é a multiplicação da matriz original pela matriz inversa. Sem a transposta da inversa, a multiplicação gera quase sempre cache misses ao acessá-la por linhas.
Usar uma macro ao invés de uma função para calcular a posição na matriz também foi muito impactante, e escolhemos essa otimização por ser de fácil implementação e evitar cache misses na cache de instruções, e evitar o crescimento da stack do programa ao chamar uma função.
A alocação alinhada da memória e a utilização do qualificador `restrict` foram ambas alterações realizadas com o intuito de facilitar otimizações que o próprio compilador realiza. Alocar as matrizes com tamanhos múltiplos de $4 * 4$ também foi uma mudança realizada neste mesmo sentido, de forma a deixar a utilização das caches mais estável.
Por fim, a soma e Kahan foi removida para realizar menos operações de ponto flutuante e

permitir a vetorização destas operações.
Todas as alterações resultaram em um programa muito mais rápido, apesar de levemente menos preciso, por termos deixado de fazer a soma de Kahan, e com a utilização das caches mais constante. O crescimento do tamanho da matriz passa a ter menos impacto no seu tempo de execução. Além disso, entre diversas tentativas e erros, o grupo aprendeu diversas técnicas diferentes para a otimização de programas computacionais.