Otimização do Desempenho na Inversão de Matrizes Universidade Federal do Paraná

Guilherme Gomes dos Santos[1]

[1]Bacharelado em Ciência da Computação – Universidade Federal do Paraná (UFPR)

ggs12@inf.ufpr.br

Resumo. *O trabalho prático da disciplina Introdução à Computação Científica é dividido em duas partes. Na primeira, foi desenvolvido um algoritmo na linguagem C que calculava a inversa de uma matriz, utilizando fatoração LU com pivotamento parcial e refinamento sucessivo. Na segunda parte do trabalho, efetuou-se melhorias no código com o objetivo de melhorar seu desempenho. Este documento apresenta os resultados obtidos após estas otmizações.*

1. Introdução

Durante a primeira etapa do trabalho prático (**v1**), foi desenvolvido um programa computacional em linguagem C que, dada uma matriz quadrada A de dimensão n, devolve a matriz inversa de A (A^{-1}), tal que $AA^{-1} = I$, onde I é a matriz identidade. O algoritmo desenvolvido utiliza eliminação de Gauss com pivotamento parcial, fatoração LU e refinamento sucessivo.

Para a segunda etapa do trabalho (**v2**), dois trechos do código foram otimizadas de forma a obter uma melhora de desempenho durante a execução do método:

- Na operação de resolução do sistema linear triangular (**op1**): $Ly = b; Ux = y$;
- Na operação de cálculo do resíduo (**op2**): $R = I - AA^{-1}$;

Os testes de desempenho foram executados utilizando a ferramenta Likwid[1], analisando os seguintes aspectos: Tempo médio do cálculo da **op1** e tempo médio do cálculo da **op2**, banda de memória (Memory bandwidth [MBytes/s]) do grupo L3, cache miss (data cache miss ratio) do grupo L2 e operações aritméticas (MFLOP/s) do grupo FLOPS_DP.

A arquitetura do computador utilizado durante os testes de otimização, obtida através da ferramenta likwid-topology, é exibida abaixo:

Tabela 1. Processador

CPU name	Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz
CPU type	Intel Core IvyBridge processor
CPU stepping	9
Sockets	1
Cores per socket	4
Threads per core	2

Tabela 2. Cache Topology

Level	1
Size	32 kB
Associativity:	8
Cache line size:	64
Level	2
Size	256 kB
Associativity:	8
Cache line size:	64
Level	3
Size	6 MB
Associativity:	12
Cache line size:	64

2. Estrutura dos dados

Para a primeira etapa do trabalho todas as matrizes foram armazenadas em vetores unidimensionais, visando alocação de memória contígua. Os vetores contendo as matrizes L e U apresentavam as seguintes estruturas:

L:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ l_{10} & 1 & 0 & 0 & 0 \\ l_{20} & l_{21} & 1 & 0 & 0 \\ l_{30} & l_{31} & l_{32} & 1 & 0 \\ l_{40} & l_{41} & l_{42} & l_{43} & 1 \end{pmatrix}$$

U:

$$\begin{pmatrix} u_{00} & u_{01} & u_{02} & u_{03} & u_{04} \\ 0 & u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & 0 & u_{44} \end{pmatrix}$$

Observa-se que armazenar o conteúdo desta forma é ineficiente, uma vez que todos os zeros das matrizes não são utilizados durante a execução do método.

3. Otimizações

3.1. Estrutura de dados

O primeiro objetivo durante a **v2** foi criar estruturas para armazenar estas matrizes de maneira mais eficiente, evitando o desperdício de memória e acesso não contínuo aos dados. As tabelas 3 e 4 demonstram a maneira encontrada para armazenar as matrizes L e U.

Tabela 3. Matriz L: Vetor melhorado

i	0	1	2	3	4	5	6	7	8	9
L[i]	l_{10}	l_{20}	l_{21}	l_{30}	l_{31}	l_{32}	l_{40}	l_{41}	l_{42}	l_{43}

Tabela 4. Matriz U: Vetor melhorado

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
U[i]	u_{00}	u_{01}	u_{02}	u_{03}	u_{04}	u_{11}	u_{12}	u_{13}	u_{14}	u_{22}	u_{23}	u_{24}	u_{33}	u_{34}	u_{44}

A primeira versão da estrutura de dados, apresentada na**v1** do código, não é eficiente pois além de armazenar dados desnecessários, gera um acesso não linear aos elementos da matriz: O acesso é organizado em linhas de cache, que são carregadas de uma única vez. Assim, ao acessar o elemento l_{10} por exemplo, os elementos seguintes da mesma linha da matriz L são trazidos dentro da linha de cache. O próximo elemento a ser acessado é o l_{20}, que não estará na cache (para um tamanho de linha da matriz suficientemente grande), gerando assim um cache miss.

Assim, ao alinhar os dados nas estruturas descritas os próximos elementos a serem acessados estão sempre na mesma (ou próxima) linha de cache, já que são alocados de maneira contígua. Além disso, consequentemente o tamanho dos vetores é menor: $|L| = (n * (n+1))/2 - n$ e $|U| = (n * (n+1))/2$.

3.2. Código op1

Para otimizar as estruturas de dados das matrizes L e U foi necessário criar um novo método de percorrer os vetores. Isso ocorre pois cada linha da matriz possui um tamanho diferente, gerando a necessidade de cálculos de índices. Além disso, outra dificuldade surge: o pivotamento parcial executa trocas de linhas nas matrizes L e U.

Como a dificuldade para implementar um algoritmo que efetuase estas trocas de linhas em cima dos vetores otimizados seria grande, a decisão foi de manter matrizes auxiliares L_aux e U_aux (vetores de dimensão $n * n$) para se efetuar a decomposição LU. Esta alternativa tira proveito do fato de que a eliminação de Gauss com pivotamento parcial é executada apenas uma vez durante o método.

O código abaixo é executado após a conclusão do pivotamento parcial, no qual se efetua uma cópia dos valores úteis contidos nos vetores auxiliares L_aux e U_aux para os vetores otimizados L e U.

```
// Inicializa o vetor alinhado L com os valores da matrix inferior
for (i = 0; i < N; i++ ) {
    for (j = 0; j < i; j++) {
        L[index(i,j)] = L_aux[i*N+j];
    }
}

// Inicializa o vetor alinhado U com os valores da matrix superior
for (i = 0; i < N; ++i) {
    for (j = i; j < N; ++j) {
```

```
        U[uindex(i,j,N)] = U_aux[i*N+j];
    }
}
```

Observa-se que ainda que adicionando esta cópia de dados ao código obtenha-se um ganho em desempenho, e isso é possível por a eliminação de Gauss é executada uma vez, enquanto os vetores L e U serão utilizados número de iterações $*n$ vezes.

Como cada linha dos vetores L e U possui uma dimensão diferente, foi necessário criar as seguintes macros para que a indexação dos vetores fosse feita de maneira simples:

```
#define size(n) (((n)*(n+1))/2)
#define offset(i,j) (size((i)-1)+(j))
#define index(i,j) ((i)<(j) ? 0 :  (offset((i),(j))))
#define uindex(i,j,n) ((i)>(j) ? 0 : (size(n)-size((n)-(i))+(j)-(
    i)))

//Acessar L[i,j] : L[index(i,j)]
//Acessar U[i,j] : U[uindex(i,j,N)]
```

Como visto na disciplina, se um vetor é alocado de forma que seu primeiro elemento não seja mapeado para o primeiro elemento da linha de cache, sempre que esta linha é acessada há dados que são trazidos e não utilizados. Assim, para melhorar a utilização do tamanho da linha de cache (64 Bytes no computador utilizado) os vetores foram alocados com a função posix_memalign(), garantindo assim que o endereço do espaço de memória alocado seja um múltiplo do parâmetro de alinhamento escolhido.

```
align_L = posix_memalign((void**)&L, 64, lower_size * sizeof(
    double));
align_U = posix_memalign((void**)&U, 64, upper_size * sizeof(
    double));
```

Percebese então que os vetores L e U são alocados em endereços múltiplos de 64 (tamanho da linha de cache), onde lower_size e upper_size são as dimensões dos vetores.

3.3. Código op2

Para a otimização da **op2**, na qual é efetuada a multiplicação das matrizes A e A^{-1}, o primeiro passo seria otimizar o acesso ao vetor AI contendo a matriz inversa, já que esse acesso é feito por colunas (collumn major order). Como durante a implementação da **v1** isso já foi levado em consideração, o vetor AI já foi alocado de maneira que o acesso fosse row major order, removendo assim a necessidade de fazer esta otimização na **v2**.

A seguir é apresentado o código para a **op2** desenvolvido na **v1** do trabalho:

```
for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
                soma = 0.0;
                for (k = 0; k < N; k++) {
                        soma += A[i*N+k] * AI[k+N*j];
                }
                A_AI[i*N+j] = soma;
        }
}

```

```
for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
        R[row[i]*N+j] = I[i*N+j] - A_AI[i*N+j];
        }
}
```

O foco para otimizar a **op2** foi manter o código nas especificações necessárias para que o compilador pudesse efetuar e vetorização no loop da operação, possibilitando assim o uso dos registradores AVX. Assim, as estruturas de dados foram alocadas de maneira alinhada:

```
align_A = posix_memalign((void**)&A, 32, N * N * sizeof(double));
align_AI = posix_memalign((void**)&AI, 32, N * N * sizeof(double)
    );
align_R = posix_memalign((void**)&R, 32, N * N * sizeof(double));
```

O alinhamento foi feito em 32 bytes para ser compatível com o uso dos registradores AVX, já que cada registrador armazena 256 bits (4 doubles, 8 bytes cada). Além disso, como a vetorização de loops permite a execução de desvios condicionais na forma de atribuição, foi possível remover a utilização da matriz auxiliar A_AI, economizando assim memória e transferência de dados. O código para a **v2** ficou:

```
// Primeira vers o de otimiza  o
for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
                soma = 0.0;
                for (k = 0; k < N; k++) {
                        soma += A[i*N+k] * AI[j*N+k];
                }
                R[i*N+j] = (i == j) ? 1 - soma: 0 - soma;
        }
}
```

Outra mudança foi deixar de utilizar o vetor de índices row[i], que era utilizado para armazenar as permutações de linhas efetuadas em R. Essa mudança foi necessária para que não ocorressem acessos fora de ordem no vetor R. Assim o vetor R tem suas linhas trocadas durante o pivotamento parcial para que todos os acessos seguintes aos seus dados sejam em ordem.

A utilização da variável auxiliar *soma* dentro do loop da **op2** foi necessária para permitir que o compilador GCC gerasse a vetorização. Caso o vetor R fosse utilizado no lugar da variável, o compilador assumiria uma dependência de dados que impossibilitaria a vetorização.

Com estas mudanças e a utilização das flags -fstrict-aliasing e -ffast-math durante a compilação, o GCC conseguiu gerar código vetorizado para utilização dos registradores AVX, isto foi confirmado utilizando a flag -ftree-vectorizer-verbose, que exibe informações referentes ao processo de vetorização. O compilador vetorizou diversos loops do código, e em cada vetorização o cálculo do unroll e *remainder* foi feito automaticamente. Além de vetorizar o loop apresentado na linha 60 do trecho de código acima, ele acabou vetorizando também os loops da **op1**, que efetuam forward_substitution() e backward_substitution().

Ressalta-se a mudança na definição dessas funções, que na **v2** tornaram-se inline, gerando assim melhora de performance evitando chamadas de funções dentro do loop que é executado n vezes a cada iteração do refinamento.

3.4. Tentativa de loop unroll na op2

Buscando melhorar ainda mais o desempenho da **op2**, foi desenvolvido o seguinte loop unroll exibido no código a seguir.

Com o unroll do laço j, o acesso à matriz A é reaproveitado dentro do laço k. Assim, a cada iteração em j é calculado os valores de R[i,j], R[i,j+1], R[i,j+2] e R[i,j+3], aproveitando o valor de A[i*N+k] que já está em cache. Assim, o acesso à matriz A, que antes acontecia N^2 vezes, é reduzido para $N^2/4$.

De fato esta operação trouxe uma redução no número de caches misses: para n = 2000, o cache miss ratio era em média 0.3364, reduzindo para 0.2185 após o unroll. Porém ao comparar resultados da otimização com e sem o loop unroll, constatou-se que embora o unroll proporcionasse a redução de cache miss, a utilização da banda de memória diminuiu drasticamente. Além disso, o tempo de execução médio da **op2** permaneceu igual ao da **v1**, ou seja, nenhuma melhoria prática no desempenho.

```
//Segunda versao de otimizacao: tentativa de unroll
for (i = 0; i < N; i++) {            //laco i
        for (j = 0; j < size; ++j) {     //laco j
                soma_v[0] = 0.0;
                soma_v[1] = 0.0;
                soma_v[2] = 0.0;
                soma_v[3] = 0.0;
                for (k = 0; k < N; k++) {    //laco k
                        soma_v[0] += A[i*N+k] * AI[j*N+k];
                        soma_v[1] += A[i*N+k] * AI[(j+1)*N+k];
                        soma_v[2] += A[i*N+k] * AI[(j+2)*N+k];
                        soma_v[3] += A[i*N+k] * AI[(j+3)*N+k];
                }
                R[i*N+j] = (i == j) ? 1 - soma_v[0]: 0 - soma_v
                    [0];
                R[i*N+j+1] = (i == j+1) ? 1 - soma_v[1]: 0 -
                    soma_v[1];
                R[i*N+j+2] = (i == j+2) ? 1 - soma_v[2]: 0 -
                    soma_v[2];
                R[i*N+j+3] = (i == j+3) ? 1 - soma_v[3]: 0 -
                    soma_v[3];
        }
}
//Remainder
for (i = 0; i < N; i++) {
        for (j = size; j < N; j++) {
                soma = 0.0;
                for (k = 0; k < N; k++) {
                        soma += A[i*N+k] * AI[j*N+k];
                }
                R[i*N+j] = (i == j) ? 1 - soma: 0 - soma;
        }
}
```

Por isso esta tentativa na utilização do loop unroll foi descartada. Após exaustivos testes, chegou-se a conclusão de que a primeira versão da otimização é melhor: Mesmo com o aumento de cache miss, o tempo médio de execução da **op1** cai pela metade, a utilização da banda de memória quase dobra.

4. Resultados

4.1. Teste de tempo

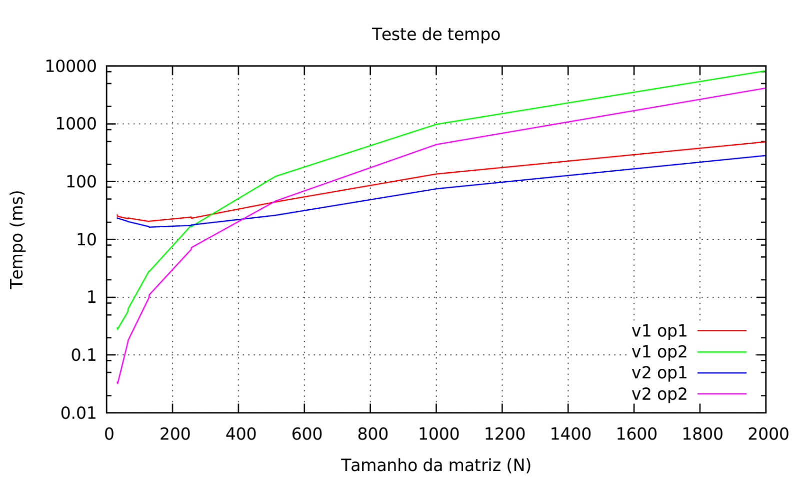

Figura 1. Tempo médio de execução op1 e op2

O gráfico acima apresenta as médias de tempo (em ms) para execução das operações **op1** e **op2** em ambas as versões do código. Percebe-se uma grande redução na média de tempo após $n = 256$. Essa redução ocorre principalmente pela utilização de instruções SIMD. Como os registradores AVX passaram a ser utilizados na **v2**, é esperado que o tempo necessário para executar as operações seja reduzido, uma vez que com AVX as operações são efetuadas em 4 doubles a cada instrução (SIMD). A diferença nos resultados é bastante aparente para matrizes com $n = 2000$ por exemplo, onde a média de tempo na **v1 op2** era de 8285.02ms e foi reduzida para 4165.51ms na **v2 op2**.

4.2. Operações Ariméticas

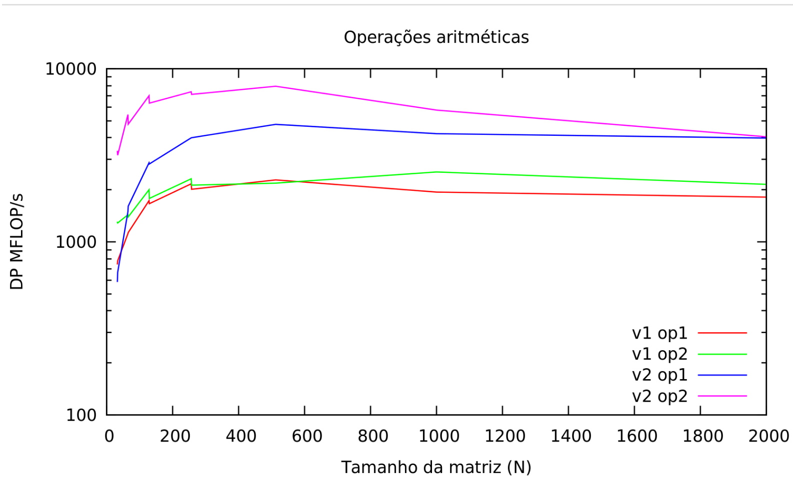

Figura 2. Operações em ponto flutuante

As alterações feitas no código somadas às flags corretas possibilitaram ao compilador vetorizar os principais loops contidos nas operações **op1** e **op2**. Ao executar o likwid-perfctr monitorando grupo FLOPS_DP, observou-se que aproximadamente 90% das operações em ponto flutuante são executadas com AVX. Por consequência, há um grande aumento na quantidade de MFLOPS/s. Destaca-se no gráfico a linha rosa, que representa **v2 op2**, para $n = 512$ as operações aumentaram de 2185.0851 MFLOPS/s para 7927.6770 MFLOPS/s, das quais 7844.5062 são AVX DP MFLOP/s.

4.3. L3 - Banda de Memória

img/L37.png

Figura 3. L3 - 7 bandas

img/L327.png

Figura 4. L3 - 27 bandas

O teste de banda de memória, grupo L3, nos disponibiliza os valores da taxa de transferência de dados entre a memória e o processador, em MBytes/s. Como nossa aplicação é para cálculos científicos, esta troca de dados entre memória e processador é muito intensa, e quanto maior a taxa, mais dados estão sendo transmitidos, consequentemente o desempenho da aplicação se torna maior. Ao analisarmos os gráficos obtidos, vemos que houve uma melhora nesta taxa na 2ª versão da aplicação, e um dos motivos para isso é a paralelização das operações que as instruções AVX possibilitam.

4.4. Tempo de Execução

img/TIME7.png

Figura 5. Tempo - 7 bandas

img/TIME27.png

Figura 6. Tempo - 27 bandas

Um dos critérios mais significativos para comprovar a otimização do algoritmo é a melhora no tempo de execução, mesmo não sendo uma diferença gritante, ficou claro que além do método todo, o tempo para executar as multiplicação foram reduzidos. Notando que na 1ª versão, ao aumentar a quantidade de bandas de 7 para 27 o tempo aumentou muito mais que na 2ª versão, consequentemente o aumento das bandas já não afeta mais tanto o desempenho do algoritmo.

5. Considerações Finais e Conclusão

A primeira etapa do trabalho consistia em implementar o método do *Gradiente Conjugado*. Por conta da complexidade dos testes, fizemos otimizações já na primeira versão, isso inclui o método de armazenamento que adotamos atualmente, a parte difícil é a indexação para multiplicar por um vetor. Na primeira versão foi usado *desvios condicionais* e na segunda foi usado *loops* separados. A otimização se torna mais clara nos momentos em que o tempo de execução é decaído, sendo assim, durante o processo de aperfeiçoamento, este tempo foi um dos fatores no qual mais nos baseamos. Grande parte dos nossos estudos e dedicação nesta segunda etapa do trabalho foi a otimização da multiplicação de matriz por vetor, visto que a função ficou bem extensa devido ao número de laços, porém através do *loop unrolling* e da implementação das instruções AVX, obtivemos um resultado bem satisfatório.