Relatório: Inversão de Matrizes - Otimização

Paulo Mateus Luza Alves
Universidade Federal do Paraná
Curitiba, Brasil
pmla20@inf.ufpr.br
GRR20203945

Vinícius Mioto
Universidade Federal do Paraná
Curitiba, Brasil
vm20@inf.ufpr.br
GRR20203931

***Resumo*—Esse relatório tem como objetivo mostrar os resultados obtidos com a otimização de código do trabalho de inversão de matrizes, feito em linguagem C utilizando técnicas aprendidas na disciplina de Introdução à Computação Científica (CI1164).**

***Palavras-chave*—matrizes, otimização, desempenho**

I. Introdução

Nesse trabalho desenvolvemos um experimento que possibilitou a comparação os resultados da otimização de código para o problema de inversão de matrizes utilizando método da fatoração LU.

Na primeira versão (**v1**) encontramos alguns problemas, como realocação de vetores em tempo de execução, acesso *strided* nas matrizes de resíduos e inversa, dentre outros.

Com isso, utilizamos algumas técnicas, como *Unroll & Jam*, transposição de matrizes, além de outros métodos, com o objetivo de otimizar projeto nas operações específicas, fomentando a versão otimizada do algoritimo (**v2**). O código fonte pode ser encontrado no GitHub[1].

II. Análise de Performance

A ferramenta adotada para gerarmos as métricas necessárias para analisarmos e compararmos o desempenho de cada versão do trabalho foi o LIKWID (Like I Knew What I'm Doing) na versão 5.2.2 cujas configurações de uso estão descritas nas próximas seções.

Para avaliar as técnicas de otimização adotadas para a segunda versão do trabalho, selecionamos as seguintes operações:

A. Fatoração LU e Retrosubs. para Cálculo da Inversa (OP1)

Na fatoração LU temos a construção das duas matrizes que serão utilizadas no cálculo da matriz inversa, sendo essas obtidas por meio da Eliminação de Gauss. A matriz L possui os coeficientes de multiplicação do método e a U possui a matriz escalonada resultante.

No cálculo das retrosubstituições temos a solução do seguinte sistema:

$$Ly = b \longrightarrow Ux = y \tag{1}$$

Sendo a solução dessa sequência de operações a matriz inversa resultante.

[1]https://github.com/viniciusmioto/invert_matrix

B. cálculo da Matriz de Resíduos (OP2)

Nesta operação, temos o cálculo da matriz de resíduos obtidos nas resoluções das retrosubstituições. Sendo esses resíduos utilizados para obtermos um resultado mais preciso da solução. Esta matriz é obtida a partir do seguinte cálculo:

$$R = I - A \times A^{-1} \tag{2}$$

Onde,

- R: Matriz de Resíduos;
- I: Matriz Identidade;
- A: Matriz de Entrada;
- A^{-1}: Matriz Inversa Calculada.

III. Arquitetura do Computador

A. Sobre o Computador

Utilizamos o computador pessoal de Giovanni Mioto para executarmos os *scripts* e a validação dos experimentos em ambiente GNU/Linux, especificamente com o sistema operacional Pop!_OS na versão 22.04.

B. Topologia do Processador

Para obter as informações detalhadas da topologia do processador utilizado para os experimentos, executamos o seguinte comando do LIKWID:

```
likwid-topology -g -c
```

Com isso, as especificações do processador são:

- CPU
  - name: AMD Ryzen 5 2600 Six-Core Processor
  - type: AMD K17 (Zen+) architecture
  - stepping: 2
- Hardware Thread Topology
  - sockets: 1
  - cores per socket: 6
  - threads per core: 2
- Cache Topology:
- Cache Groups
  - Level 1 e Level 2:
    (0 6) (1 7) (2 8) (3 9) (4 10) (5 11)
  - Level 3:
    (0 6 1 7 2 8) (3 9 4 10 5 11)

TABELA I
TOPOLOGIA DE CACHE

	Level 1	Level 2	Level 3
Size	32 kB	512 kB	8 MB
Type	Data cache	Unf. cache	Unf. cache
Associativity	8	8	16
Nº of sets	64	1024	8192
Cache line size	64	64	64
Cache type	Non Incl.	Non Incl.	Non Incl.
Shared by thd	2	2	6

IV. SCRIPT DE TESTE E COMPILAÇÃO

Como mencionado anteriormente, os programas principais desse trabalho foram escritos em linguagem C. Somado a isso, criamos três *scripts* em linguagem Python, com o objetivo de automatizar e agilizar alguns processos.

A realização do experimento é dada pela execução dos três arquivos listados abaixo, respeitando a ordem que estão descritos:

- `generate_log_files.py`: responsável por executar as duas versões dos programas em C para os tamanhos de matrizes solicitados na atividade. Como resultado teremos três arquivos de *log* para cada tamanho de matriz e para cada versão;
- `generate_csv_files.py`: realiza a leitura dos arquivos de *log* gerados automaticamente pelo LIKWID e cria arquivos *csv* (planilhas) com os dados relevantes para avaliarmos o experimento em cada operação;
- `generate_graph_files.py`: utiliza os arquivos *csv* para gerar gráficos de linhas. Esses facilitam a visualização de resultados e a comparação entre **v1** e **v2**.

Para isso, devemos entrar na pasta de cada versão e utilizar o comando `make`. Por fim, deve-se entrar na pasta `data` e executar o seguinte comando no terminal:

```
python <nome_do_arquivo.py>
```

A. Requisitos para execução

O compilador utilizado foi o GCC (GNU Compiler Collection) na versão 11.2.0, já para o Python usamos a versão 3.10.4 e os seguintes pacotes adicionais:

- numpy == 1.23.2
- pandas == 1.4.3
- matplotlib == 3.5.3

B. Parâmetros de Compilação

Para compilarmos os algoritmos os seguintes parâmetros foram utilizados:

- -g;
- -Wall;
- -O3;
- -mavx2;
- -march=native.

Também foram necessários indicar os parâmetros do LIKWID para o correto funcionamento da biblioteca.

C. Parâmetros para o LIKWID

Utilizamos grupos específicos do LIKWID, os quais nos auxiliam a extrair as informações relevantes para a avaliação:

- MEM: informações de *bandwidth*;
- CACHE: informações de *Cache Miss Ratio*;
- FLOPS_DP: quantidade de operações em ponto-flutuante.

O *core* 11 era o menos utilizado no momento imediatamente anterior ao início dos experimentos, logo foi escolhido para o parâmetro `-C` do LIKWID, como mostra a Figura 1.

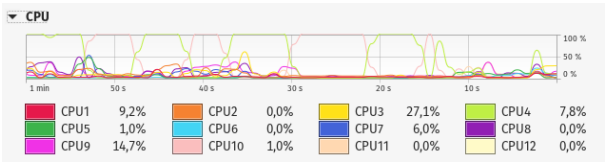

Fig. 1. Utilização da CPU - Monitor do Sistema

O Tempo de Execução das operações foi obtidos pelo retorno do campo *RDTSC Runtime[s]* das chamadas de execução do grupo MEM.

D. Parâmetros para o Algoritmo

Para a execução do algoritmo do experimento, os seguintes parâmetros foram utilizados:

- -i 10: quantidade padrão de iterações a serem executadas;
- -r: tamanhos das matrizes {32, 33, 64, 65, 128, 129, 256, 257, 512, 1000, 2000, 4000, 6000}, cujos índices são pseudo-aleatórios.

V. PROCESSO DE OTIMIZAÇÃO

Nosso processo de otimização foi dado em 2 etapas sendo a primeira as Otimizações Gerais, que foram aplicadas em ambas as operações, e a segunda as Otimizações Especificas de cada operação. Além disso, as condicionais de verificação de *isinf* e *isnan* foram removidas tanto no **v1** quanto no **v2**.

A. Método de Alocação

O método escolhido para alocação de memória de matrizes em ambas as versões foi o Matriz de Linhas Contíguas, o qual permite que a matriz seja alocada de forma contígua na memória como mostra a Figura 2. Portanto, ao realizar acessos temos uma maior probabilidade de buscar as posições da matriz que serão utilizadas para a *cache*, diminuindo a taxa de *cache miss ratio*.

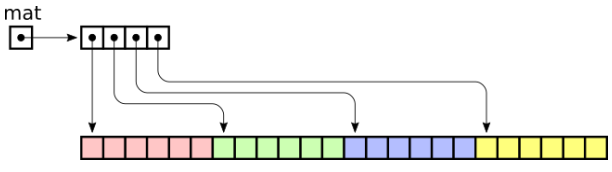

Fig. 2. Vetor de Ponteiros de Linhas Contíguas - Prof. Carlos Maziero

Ademais, no **v2** alteramos a função de alocação da matriz trocando a função `malloc` por `aligned_alloc`, que consiste em uma função que aloca a memória de maneira alinhada.

Como parâmetro de alinhamento, escolhemos o valor **64** que seria o tamanho da linha de cache *Level 1*.

B. Otimizações Gerais

Consideramos Otimizações Gerais aquelas que alteraram o algoritmo de ambas as operações de maneira mais simples, mas que ainda assim carregam significado para o contexto da otimização maior.

1) Adição dos Restricts: dentro das funções utilizadas na OP1 e OP2 adicionamos a palavra-chave *restrict* nos parâmetros que são ponteiros para o mesmo tipo. Dessa forma indicamos ao compilador que não ocorre *Aliasing* e assim, ele pode gerar código otimizado para esta sessão.

2) Preenchimento de Matrizes e Vetores: no quesito de inicialização dos valores alocados em memória, alteramos os laços, tanto para os vetores quanto para as matrizes por *memsets*, que é um método mais eficiente de inicialização de memória.

Na função `generate_identity_matrix` diminuímos o custo de $\mathcal{O}(n^2)$ para $\mathcal{O}(n)$, devido a remoção de um dos *loops*, além de removermos uma condicional que estava dentro do laço mais interno.

3) Padding para o Tamanho da Matriz: visando o problema de *cache thrashing*, o qual pode ocorrer com matrizes de tamanho em potência de 2, optamos por adicionar um *padding* na matriz para estes casos. Para isso, adicionamos a seguinte verificação na função de `alloc_matrix`:

```
log_2 = log10 (n) / log10 (2);
if (ceil (log_2) == floor (log_2))
    internal_n++;
```

Onde *internal_n* é o valor de n que utilizamos para o valor mais interno da matriz. Assim, realizamos a alocação sem o valor em potência de dois, evitando este problema.

C. Otimizações Específicas

Iniciamos o processo de otimização específica buscando aplicar os algoritmos vistos em sala, como *Blockings* e *Unroll & Jam*. Analisando o nosso código, verificamos que a cálculo da matriz de resíduos (OP2) poderia ser modificada com essas duas técnicas, e que poderíamos tentar aplicar o *Unroll & Jam* em outras seções do código que envolviam a OP1.

Sendo assim, começamos a empregar estes métodos e obtivemos sucesso na otimização da OP1, utilizando o tamanho do bloco em **8** para o *Blocking*, pois como a *cache line* tem tamanho 64, ela pode armazenar apenas 8 *doubles*, e com o passo *m* em 4 para o *Unroll & Jam*, visando realizar metade das operações do bloco para cada laço de repetição.

Já na OP2, conseguimos aplicar um *Unroll & Jam* com o mesmo passo na retrosubstituição. Somado a isso, utilizamos mais vezes esse algoritmo em outros lugares do código, como no cálculo do determinante, aplicação da matriz identidade, cálculo da norma e assim por diante.

Porém, quando realizamos os testes para matrizes de tamanho até 6000, verificamos que a OP1 não foi de fato otimizada e estava executando na mesma faixa de tempo que na versão anterior como mostra a Figura 3.

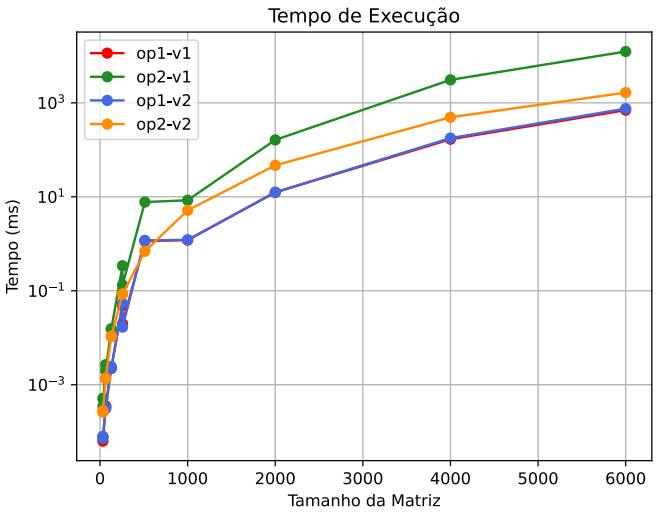

Fig. 3. Tempo de Execução - OP1 e OP2 para cada versão - 1° Otimização.

Por este motivo, começamos a buscar outras maneiras de melhorar esta operação e culminamos na nossa otimização final, onde realizamos mudanças no método de armazenamento das matrizes inversas e de resíduos, além da aplicação de vetorização com *AVX*. Logo, as otimizações finais aplicadas no projeto foram:

1) Modificação do Processo de Pivotamento: no **v1**, nosso armazenamento dos passos de pivotamento se dava por meio de uma *Array Of Structs*, isto é, uma lista de estruturas contendo nelas a linha pivô e a linha anterior que estavam sendo alteradas, e a cada novo pivotamento executado na fatoração LU nos realizávamos um *realoc* deste vetor, como mostra o código abaixo:

```
typedef struct steps {
    int line;
    int pivot;
} steps_t;

typedef struct pivot_steps {
    steps_t *list;
    int qtd;
} pivot_steps_t;
```

Visando este problema dos *realocs*, decidimos alterar o modelo de armazenamento para uma *Struct of Arrays*, removendo a estrutura `steps_t` e substituindo por dois vetores de inteiros para as linhas e pivôs. Com isso, removemos a necessidade de fazer um *malloc* para cada passo novo, além do *realoc* do vetor de passos.

```
typedef struct pivot_steps {
    int *lines;
    int *pivots;
    int qtd;
} pivot_steps_t;
```

2) Alteração do Método de Armazenamento: Notamos que seria de grande benefício armazenar a matriz de resíduos e a matriz inversa calculada de maneira transposta, pois assim, todas as operações que exigiam acesso por colunas dessas

matrizes, agora acessariam por linhas, removendo os *strides* e consequentemente diminuindo a taxa de *Cache Miss Ratio*. Além disso, esse modificação foi essencial para a aplicação de vetorização por *AVX*.

3) Função calc_residue: como comentado anteriormente, removemos o *Blocking* aplicado, e com a modificação de armazenamento da matriz inversa, foi possível aplicar um *Unroll & Jam* nesta função da maneira ideal para a utilização de vetorização por *AVX*.

Utilizamos os registradores `__256d`, que armazenam 4 *doubles*, possibilitando a vetorização de quatro operações por laço de repetição. Também foi necessário adicionar a função `avx_register_sum` que é responsável por realizar a soma dos quatro valores que estão dentro do registrador também utilizando registradores *AVX*. O uso desta ferramenta possibilita executar operações usando registradores específicos do processador, que executam o cálculo muito mais rapidamente que o normal, obtendo assim, um ganho de performance.

4) Função find_max: na primeira versão essa função continha cinco condicionais dentro do laço, o que prejudica demasiadamente o *pipeline* de execução do laço. Por este motivo, modificamos a função e removemos quatro desses cinco *if*'s, restando apenas a verificação de alteração do valor para o pivô atual, diminuindo assim a interferência no *pipeline* de execução.

5) Função retrosubs e inv_retrosubs: a aplicação da técnica de *Unroll & Jam* tornou-se complexa devido à interdependência dos laços das retrosubstituições, onde o *loop* mais interno tem seu valor inicial dependente do valor do iterador do laço mais externo.

Com algumas alterações, além da modificação para armazenamento transposto da matriz inversa, foi possível realizar o *Unroll & Jam* apenas na função *retrosubs*, onde também foi possível aplicar a vetorização por *AVX*.

Outrossim, tivemos que adicionar duas novas funções para lidar com as trocas de linhas na matriz de resíduos transporta:

- `apply_transpost_pivot_steps`
- `swap_transpost_line`

Optamos por remover essas chamadas de função (`retrosubs` e `inv_retrosubs`), e inserimos estes algoritmos dentro das funções que os utilizam, sendo essas `calc_inverse_matrix` e `matrix_refinement`.

6) Funções calc_inverse_matrix e matrix_refinement: dentro destas funções, aplicamos um *Unroll & Jam* com passo m de tamanho **4** nos laços de concatenação de resultado temporário, o que também possibilitou a aplicação de vetorização por *AVX*.

7) Função calc_norma: dentro desta função foi possível aplicar um *Unroll & Jam* com passo m de tamanho **4** que possibilitou a aplicação de vetorização por *AVX*.

8) Funções generate_identity_matrix e calc_determinant: visando diminuir o tamanho do *loop* de atribuição dos 1's da matriz identidade e o laço de cálculo do determinante, aplicamos um *Unroll & Jam* com passo m de tamanho **4**. Utilizamos o valor 4 após alguns testes e verificamos que o resultado foi melhor obtido para este tamanho de passo.

VI. Comparação Entre Versões

Ao final das otimizações executamos os *scripts* para que pudéssemos comparar os resultados entre a OP1 e OP2 nas duas versões.

A. Miss Ratio e Memory Bandwitdh

Nestes dois gráficos (ver Figura 4 e Figura 5), podemos notar uma grande queda na taxa de de *Cache Miss Ratio* no **v2**, além da remoção dos "dentes" existentes nos gráficos, provenientes das matrizes de tamanho em potencia de dois. Logo, notamos a efetividade do *padding* juntamente com a transposição das matrizes inversa e de resíduos.

B. Flops e AVX

Nestes dois gráficos (ver Figura 6 e Figura 7), podemos notar um aumento, além de constância, da taxa de *MFLOPS/S* nas duas operações no **v2**, Além do aumento da taxa de *MFLOPS/S* realizadas em registradores *AVX*, os quais não eram utilizados pela OP2 na **v1**. Logo, notamos a efetividade dos *Unroll & Jam* aplicados, além das vetorizações.

C. Tempo

Por fim, como mostrado no gráfico (ver Figura 8), podemos notar que o objetivo final foi alcançado, onde conseguimos diminuir o tempo de execução de ambas as operações, deixando assim o algoritmo mais rápido.

VII. Conclusão

Com a realização do experimento para ambas as versões do projeto, conseguimos obter os dados necessários para que fosse possível compararmos os resultados do código otimizado e da versão antiga.

Notamos que as técnicas mostradas em aula como *Blockings* e *Unroll & Jam* foram extremamente úteis para melhorarmos o projeto em vários pontos, principalmente no acesso à memória, operações por segundo e também no tempo de execução do algoritmo.

Porém tais métodos não foram suficientes para que as diferenças dentre **v1** e **v2** fossem notáveis em ambas as operações. Por isso, buscamos encontrar outras formas de otimizar o código, como a transposição de matriz e a aplicação da vetorização por *AVX*.

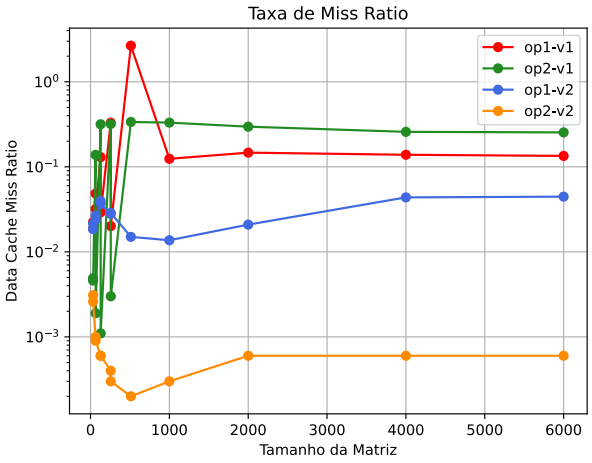

Fig. 4. Taxa de Missa Ratio - OP1 e OP2 para cada versão.

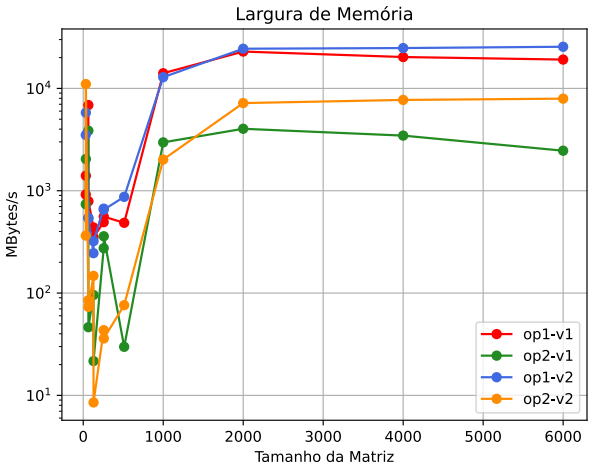

Fig. 5. Largura de Memória - OP1 e OP2 para cada versão.

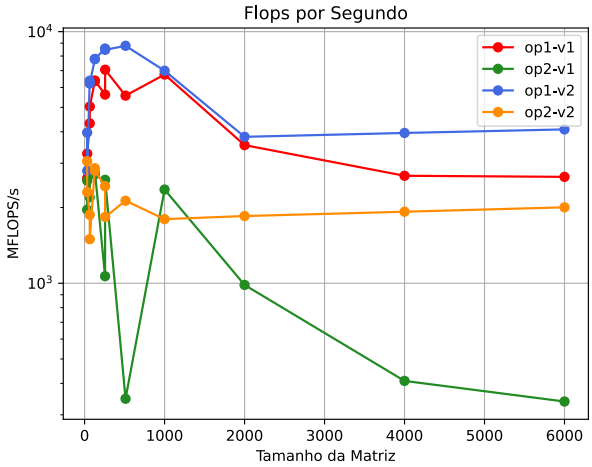

Fig. 6. Flops por Segunda - OP1 e OP2 para cada versão.

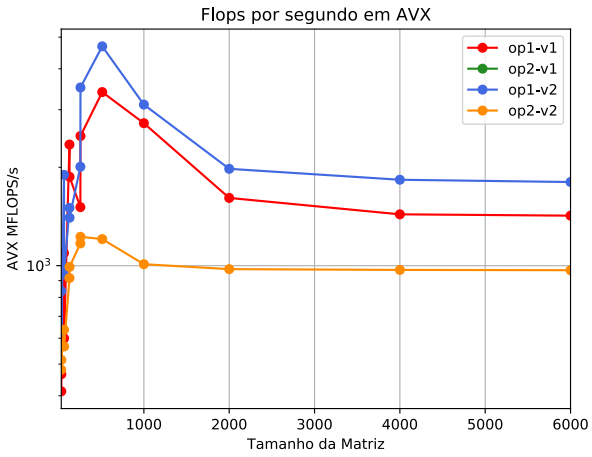

Fig. 7. FLOPS/s em AVX- OP1 e OP2 para cada versão.

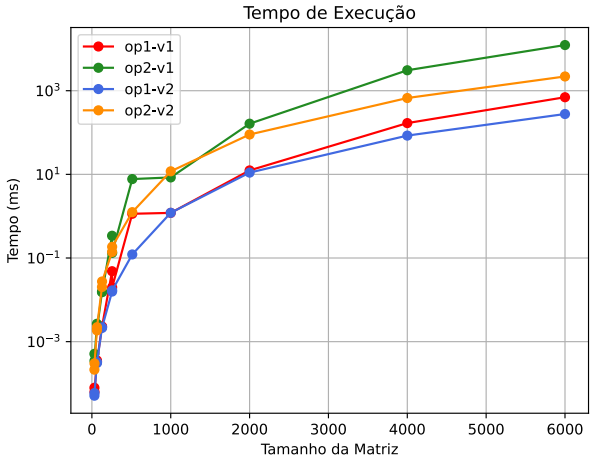

Fig. 8. Tempo de Execução - OP1 e OP2 para cada versão.