Universidade Federal do Espírito Santo
Programa Institucional de Iniciação Científica
Relatório Final de Pesquisa
Ciências Exatas e da Terra (CNPq)

Estruturas de dados otimizadas para armazenar e manipular matrizes esparsas de grande porte – PIIC/UFES

Edital:	Edital PIIC 2023/2024
Grande Área do Conhecimento (CNPq):	Ciências Exatas e da Terra
Área do Conhecimento (CNPq):	Ciência da Computação
Título do Projeto:	Estrategias Inovadoras na Implementação do Método dos Elementos Finitos utilizando Processamento de Alto Desempenho
Título do Subprojeto:	Estruturas de dados otimizadas para armazenar e manipular matrizes esparsas de grande porte
Professor(a) Orientador(a):	Lucia Catabriga
Estudante:	Miguel Vieira Machado Pim

Resumo

O armazenamento e a manipulação eficiente de matrizes esparsas, resultantes da discretização por elementos finitos, são cruciais para a solução de sistemas lineares em computação científica. Este subprojeto teve como objetivo analisar diferentes estruturas de armazenamento otimizadas, visando melhorar operações essenciais como o produto matriz-vetor e a fatoração incompleta ILU(p). Utilizando a linguagem de programação C e ferramentas de análise de desempenho, como gprof e Valgrind, foi possível implementar o código e realizar uma avaliação detalhada da eficácia de cada estrutura. Os resultados indicaram que a *Lista Encadeada* teve o pior desempenho, principalmente devido à ineficiência no uso da memória cache. Em contraste, a estrutura CSR (*Compressed Sparse Row*) destacou-se como a mais eficiente, apresentando o melhor desempenho em todas as operações avaliadas. Com base nesses resultados, conclui-se que a CSR é a opção de armazenamento mais adequada para a execução do produto matriz-vetor e da fatoração ILU(p) entre as estruturas analisadas.

Palavras-chave: Matrizes esparsas, produto matriz-vetor, fatoração ILU(p), Estruturas de dados

1 Introdução

O armazenamento e manipulação computacional de matrizes esparsas, resultantes da discretização por elementos finitos, é o núcleo de maior importância na solução de sistemas lineares. Portanto, o uso de bibliotecas computacionais que disponibilizem para pesquisadores operações e armazenamento de matrizes esparsas de maneira otimizada é de extrema importância.

Este subprojeto está vinculado ao projeto de pesquisa *Estratégias Inovadoras na Implementação do Método dos Elementos Finitos utilizando Processamento de Alto Desempenho* (EIMEFPAD) em desenvolvimento no Laboratório de Otimização e Modelagem Computacional (LabOtiM[1]). Um dos objetivos deste projeto é investigar estratégias inovadoras de soluções de sistemas lineares e, para isso, é preciso ter bibliotecas que manipulem matrizes

[1] `https://labotim.inf.ufes.br`

esparsas de maneira otimizada, sendo portanto esse o objetivo principal deste subprojeto. Geralmente, as matrizes esparsas são armazenadas de forma otimizada considerando somente os coeficientes não nulos por diferentes estratégias, Williams et al. (2007); Bell & Garland (2009); Davis & Chung (2012); Liu et al. (2013). Geralmente, as operações matriciais mais importantes são o produto matriz-vetor e operações de fatoração de matrizes. Entretanto, em operações, como a fatoração LU incompleta com nível de preenchimento (ILU(p)), a esparsidade das matrizes pode ser alterada - ou seja, posições originalmente nulas deixam de ser nulas durante as operações matriciais. O uso de bibliotecas computacionais que auxiliam o armazenamento e manipulação tem sido uma opção atraente para os pesquisadores desta área.

Os métodos iterativos não-estacionários considerando aceleradores de convergência baseados nos espaços de Krylov são preferencialmente utilizados para a solução de sistemas lineares esparsos e de grande porte Saad (2003), e comumente utilizados nas implementações do método dos Elementos Finitos. Sendo o produto matriz-vetor e a fatoração ILU(p) as principais operações matriciais presente nos algoritmos desses métodos iterativos. O tempo de processamento desses métodos depende do custo de cada iteração que fundamentalmente depende da operação produto matriz-vetor e de operações da fatoração ILU(p).

No decorrer deste subprojeto foi estudado o produto matriz-vetor considerando quatro formas diferentes de armazenamento de matrizes esparsas: COO (*Coordinate Storage*), CSC (*Compressed Sparse Column*), CSR (*Compressed Sparse Row*) e armazenamento por *Lista Encadeada* Davis & Chung (2012). Por fim, foi implementado um método comparativo entre as estruturas a fim de observar quais delas iriam obter um desempenho melhor na execução do produto matriz-vetor. Para o estudo comparativo da fatoração ILU(p) optou-se por comparar apenas os armazenamentos CSR e *Lista Encadeada*, pois as estruturas COO e CSC tem estilos muito parecidos de armazenamento e o próprio produto matriz-vetor demonstrou que os tempos de processamento entre eles são muito próximos.

2 Objetivos

Este subprojeto teve por objetivo analisar um conjunto de estruturas de dados comumente utilizadas e investigar novas estrutura de dados para armazenar matrizes esparsas resultantes da discretização de equações diferenciais parciais via elementos finitos. Os principais objetivos específicos podem ser sumarizados por:

- Comparar e implementar formas otimizadas de estrutura de dados encontradas na literatura para armazenar e manipular matrizes esparsas.
- Definir estruturas de dados baseadas em *Lista Encadeada* para armazenar e manipular matrizes esparsas.
- Implementar operações matriciais, tais como, produto matriz-vetor e fatoração ILU(p) utilizando diferentes estruturas de dados de armazenamento
- Executar um estudo minucioso da eficácia das estruturas de dados implementadas.

As novas estruturas de dados desenvolvidas, serão inseridas na FEM_CODES[2] para auxiliar os pesquisadores que necessitarem de códigos cada vez mais otimizados e velozes no contexto do métodos dos elementos finitos.

[2] `https://github.com/mod-comp-ufes/FEM_CODES`

3 Armazenamento e Operações de Matrizes Esparsas

Nesta seção apresentamos as operações matriciais produto matriz-vetor e fatoração ILU(p), assim como as cinco estruturas de armazenamento de matrizes esparsas implementados neste subprojeto.

Dada uma matriz esparsa A de ordem N e um vetor v de N, a multiplicação matriz-vetor é definida por $p = Av$, onde p é o vetor resultante do produto. Cada elemento p_i do vetor é calculado por:

$$p_i = \sum_{j=1}^{N} a_{ij} * v_j \tag{1}$$

sendo que a componente p_i deve ser calculada de forma otimizada considerando somente os coeficientes não nulos de A.

A fatoração ILU(p) consiste em achar dois fatores $\tilde{L}$ e $\tilde{U}$, tais que $A \approx \tilde{L}\tilde{U}$, sendo $\tilde{L}$ uma matriz triangular inferior com diagonal unitária e $\tilde{U}$ uma matriz triangular superior. De forma geral os coeficientes da matriz U e L, na etapa $k+1$ do processo de fatoração podem ser representados por:

$$l_{ik} \leftarrow a_{ik} = \frac{a_{k+1,k}}{a_{kk}} \tag{2}$$

$$u_{ij} \leftarrow a_{ij} = a_{ij} - l_{ik} * a_{kj} \tag{3}$$

sendo que as componente de L e U são armazenadas em A e devem ser calculadas considerando somente as posições não nulas de A. Entretanto, o parâmetro p determina o nível de preenchimento permitido, sendo considerado preenchimento até a etapa $p+1$ do processo de fatoração. No caso específico da fatoração ILU(0), ou seja, com $p = 0$, nenhuma posição que originalmente era nula na matriz A será preenchida durante o processo de fatoração. À medida que o nível de preenchimento p aumenta, permite-se o preenchimento de posições que anteriormente eram nulas, aproximando gradualmente o número de elementos não nulos ao máximo que a matriz A pode suportar em termos de preenchimento. Dessa forma, o parâmetro p controla o compromisso entre a precisão da aproximação e o custo computacional da fatoração, visto que maiores valores de p aumentam o preenchimento e, consequentemente, a densidade dos fatores $\tilde{L}$ e $\tilde{U}$ Saad (2003).

Utilizaremos a Figura 1 para exemplificar como essa matriz seria armazenada em cada uma das estruturas de dados implementadas.

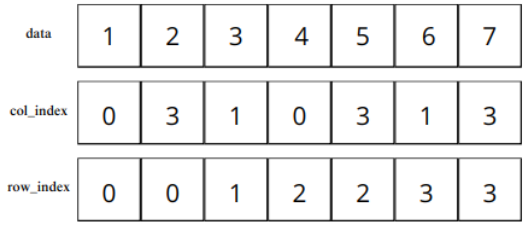
Figura 1: Matriz de Exemplo

1	0	0	2
0	3	0	0
4	0	0	5
0	6	0	7

Fonte: Produção do próprio autor

- COO (*Coordinate Storage*)
  A estrutura COO consiste em armazenar três *arrays*: *data*, *row_index* e *col_index*. O *row_index* armazena o índice da linha, o *col_index* armazena o índice da coluna e o *data* armazena o valor do elemento da matriz. A Figura 2 mostra a matriz da Figura 1 armazenada na estrutura COO.

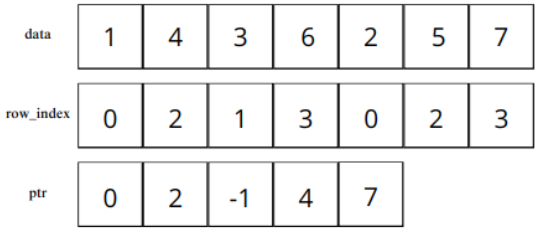
Figura 2: Exemplo do armazenamento COO



Fonte: Produção do próprio autor

- CSC (*Compressed Sparse Column*)
  A estrutura CSC consiste em armazenar três *arrays*: *data*, *row_index* e *ptr*. O *ptr* armazena na posição i, em qual índice de *row_index* e *data* se encontra o primeiro elemento da coluna $i+1$, sendo que para colunas que não contém elementos não nulos armazena-se o valor de -1 em *ptr*. O *row_index* armazena o índice da linha, e o *data* armazena o valor do elemento da matriz. A Figura 3 mostra a matriz da Figura 1 armazenada na estrutura CSC.

Figura 3: Exemplo do armazenamento CSC

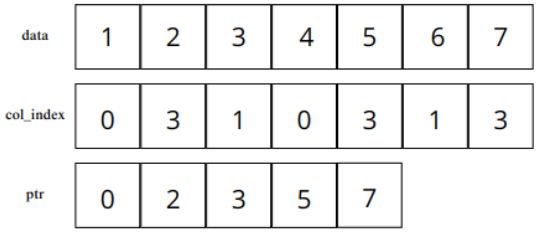


Fonte: Produção do próprio autor

- CSR (*Compressed Sparse Row*)
  A estrutura CSR consiste em armazenar três *arrays*: *data*, *col_index* e *ptr*. O *ptr* armazena na posição i, em qual índice de *col_index* e *data* se encontra o primeiro elemento da linha $i+1$, sendo que para linhas que não contém elementos não nulos armazena-se -1 em *ptr*. O *col_index* armazena o índice da coluna, e o *data* armazena o valor do elemento da matriz. A Figura 4 mostra a matriz da Figura 1 armazenada na estrutura CSR.

- *Lista Encadeada*
  No armazenamento por *Lista Encadeada* cada elemento não nulo da matriz será um nó que armazenará: o valor, índice da linha, índice da coluna, ponteiro para o próximo elemento da linha, ponteiro para o próximo

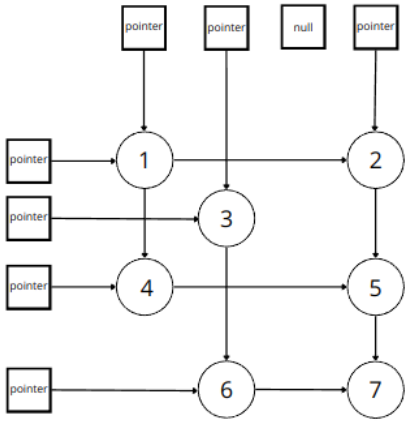
Figura 4: Exemplo do armazenamento CSR



Fonte: Produção do próprio autor

elemento da coluna do elemento. A Figura 5 mostra como a matriz da Figura 1 é armazenada na *Lista Encadeada*.

Figura 5: Exemplo do armazenamento *Lista Encadeada*



Fonte: Produção do próprio autor

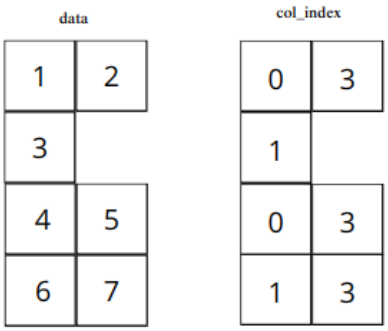
- *SparMAT*
  O armazenamento *SparMAT* consiste em armazenar dois *array* de *arrays*, um para os valores não nulos da matriz, e o outro para os índices da coluna de cada elemento não nulo como mostra a Figura 6.

4 Metodologia

Este subprojeto iniciou-se com a leitura das seguintes referências: Williams et al. (2007); Bell & Garland (2009); Davis & Chung (2012); Liu et al. (2013). Estas referências contém estudos sobre a relação do *hardware* com o produto matriz-vetor, analisando a eficiência da operação em diferentes tipos de processadores e até no uso da partição CPU-GPU. A partir destas referências foi possível iniciar a implementação do código para os diferentes tipos de armazenamento de matrizes esparsas.

A implementação da fatoração ILU(p) em estruturas otimizadas apresenta desafios, principalmente devido ao preenchimento de posições originalmente nulas. Para simplificar a implementação da fatoração ILU(p) com pre-

Figura 6: Exemplo do armazenamento *SparMAT*



Fonte: Produção do próprio autor

enchimento na estrutura CSR, utilizamos duas estruturas auxiliares: *SparMAT* e *SparILU*. A estrutura *SparMAT* é responsável por armazenar as matrizes L e U resultantes da fatoração, assim como a matriz original A, ou seja, a fatoração ILU(p) é realizada sobre a estrutura *SparMAT*, e não diretamente na estrutura CSR. A estrutura *SparILU*, por sua vez, armazena duas instâncias de *SparMAT* correspondentes às matrizes L e U, além de um vetor auxiliar para armazenar a diagonal da matriz e outro vetor que auxilia nas operações durante a fatoração. Para o estudo da fatoração ILU(p) utilizou-se a seguinte referência: Saad (2003).

A linguagem de programação escolhida para a implementação das estruturas de matrizes esparsas foi a Linguagem C. Esta escolha foi motivada pela sua eficiência e familiaridade do discente com a linguagem. Os testes foram conduzidos através de um *script*[3] em Python[4]. Foi escolhido utilizar um *script*, uma vez que facilita a automatização dos testes com as matrizes, já que necessitamos de executar testes com um número elevado de matrizes. A escolha da linguagem Python se deu por ela ser de fácil utilização, devido as suas inúmeras funcionalidades já implementadas em suas bibliotecas. Cada programa foi executado dez vezes. Após as dez execuções, o *script* descarta o menor e o maior tempo, mantendo apenas oito tempos para o cálculo da média, que foi utilizada na análise dos resultados. Para cada uma das execuções o produto matriz-vetor é executado cinco vezes, enquanto que a fatoração ILU(p) é executada apenas uma vez.

Para a análise do código, utilizou-se o gprof[5]. O gprof é uma ferramenta que permite ao programador saber em que local no código está sendo gasto mais tempo pelo programa e nos mostra, também, que funções chamaram outras funções durante a execução do programa. Ao executar o comando gprof com a *flag* `-l`, obtém-se um relatório detalhado linha a linha do código, juntamente com um *ranking* das linhas que consumiram mais tempo de processamento. Além disso, foi utilizado o Valgrind[6], uma ferramenta muito utilizada para *dubbuging* de código e identificação de vazamentos de memória.

O comando de compilação para todas as estruturas foi o seguinte: `gcc -o main main.c matriz.c -g -Wall -O3 -pg`. As *flags* mais importantes são a `-O3`,`-g` e `-pg`, as outras não tem um impacto específico para este

[3] Conjunto de instruções em código para que o PC execute determinadas tarefas.
[4] `https://www.python.org/`
[5] Ferramenta para análise dinâmica da execução de programas escritos em linguagem C, Fortran e Pascal. `https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/`
[6] Ferramenta para análise dinâmica de código. `https://valgrind.org/`

subprojeto e portanto, não serão explicadas. A `-O3` ativa uma série de *flags* para a otimização do código, que diminui o tempo de execução do programa e aumenta o tempo de compilação do código. A *flag* `-g` nos permite rodar o programa com o Valgrind para detectar vazamentos de memória e fazer o *debbuging* do código. E a *flag* `-pg` faz com que o programa gere um arquivo `.gmon` que contém informações para o gprof gerar um relatório da execução do programa.

Destacamos que as matrizes utilizadas foram obtidas no repositório *SuiteSparse Matrix Collection* Davis & Hu (2011), disponíveis no link `https://sparse.tamu.edu`. As matrizes consideradas neste relatório incluem: `Hamrle3`, `StocF-1465`, `Geo_1438`, `Long_Coup_dt6`, `Flan_1565`, `F1`, `Fault_639`, `cage14`, `CurlCurl_4`, `FullChip`, `thermal2`, `CurlCurl_3`, `dielFilterV2real`, `dielFilterV3real`, `nd24k`, `raefsky3`, `venkat01`, `power9`, `language`, `majorbasis`, `xenon2` e `PR02R`, oriundas de diferentes aplicações no contexto de simulações computacionais.

5 Resultados e Discussão

Nesta seção apresentamos os experimentos comparando as diferentes estrutura de dados implementadas para o produto matriz-vetor e para a fatoração ILU(p). As principais métricas de comparação são a ordem N e a quantidade de não nulos nnz.

Os testes para o produto matriz-vetor foram divididos considerando as estruturas COO, CSC, CSR e *Lista Encadeada*, em dois casos, pela ordem da matriz (N) e pela quantidade de não nulos da matriz (nnz). O objetivo no primeiro caso foi manter a quantidade de elementos não nulos o mais próxima possível e aumentar a ordem da matriz para observar o impacto causado no tempo de execução. No segundo caso foi adotado o mesmo princípio, mas agora mantendo a ordem da matriz e aumentando a quantidade de não nulos.

As Tabelas 1 e 2 mostram nnz e N das matrizes, respectivamente, para cada caso da multiplicação matriz-vetor em análise. A Figura 7 apresenta o tempo de execução do produto matriz vetor para as matrizes da Tabela 1, ou seja,

Tabela 1: Matrizes utilizadas para os testes por ordem N - produto matriz-vetor.

Matrizes	**nnz**	**N**
`F1`	26.837.113	343.791
`Fault_639`	27.245.944	638.802
`cage14`	27.130.349	1.505.785
`CurlCurl_4`	26.515.867	2.380.515
`FullChip`	26.621.983	2.987.012

Fonte: Produção do próprio autor.

por N, enquanto a Figura 8 mostra os tempos de execução do produto matriz-vetor para as matrizes da Tabela 2. ou seja, por nnz. É importante ressaltar a ausência dos resultados para o armazenamento por *Lista Encadeada*, já que o tempo de processamento foi dezenas de vezes maior que os demais. Este comportamento será discutido em seguida.

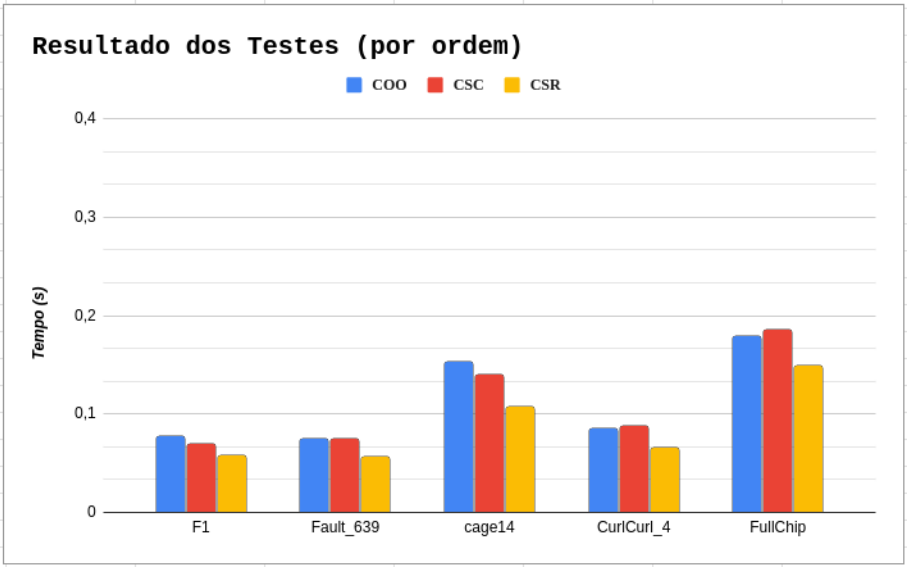
Fica evidente na Figura 7 que para o conjunto de matrizes consideradas – nnz próximos – o produto matriz-vetor considerando o armazenamento CSR obteve o menor tempo computacional para todas as matrizes testadas com

Tabela 2: Matrizes utilizadas para os testes por *nnz* - produto matriz-vetor.

Matrizes	**nnz**	**N**
`Hamrle3`	5.514.242	1.447.360
`StocF-1465`	21.005.389	1.465.137
`Geo_1438`	60.236.322	1.437.960
`Long_Coup_dt6`	84.422.970	1.470.152
`Flan_1565`	114.165.372	1.564.794

Fonte: Produção do próprio autor.

Figura 7: Resultados dos testes por ordem - produto matriz-vetor

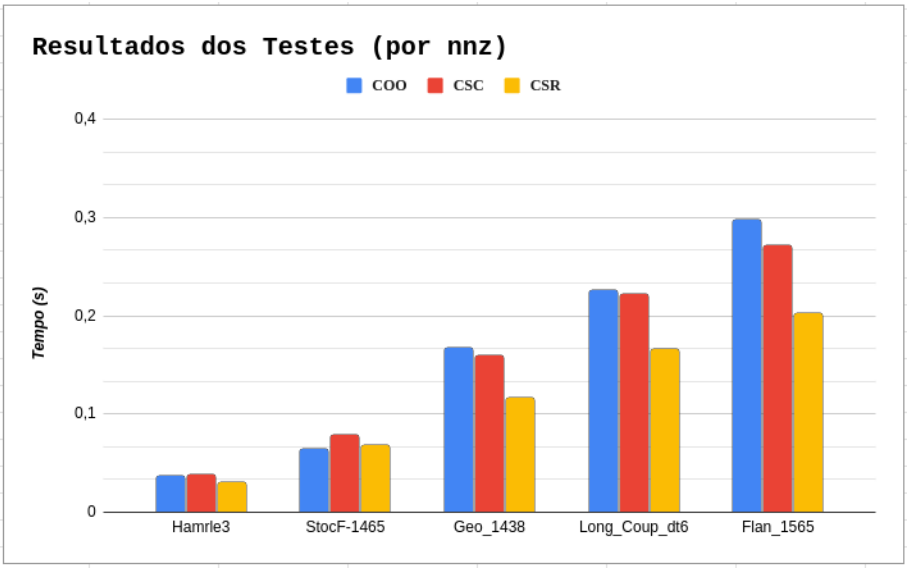


Fonte: Produção do próprio autor

diferentes ordens (*N*). Além disso, podemos observar que a medida que a ordem da matriz cresce não há um aumento significativo do tempo de execução. Portanto, podemos concluir que este método de armazenamento é a melhor escolha para o produto matriz-vetor para o conjunto de matrizes estudadas, levando em consideração matrizes com *nnz* próximos. Já na Figura 8 observamos que a densidade de elementos não nulos (*nnz*) em uma matriz exerce influência significativa no tempo de execução do produto matriz-vetor e mais uma vez a estrutura CSR apresentou os menores tempos de execução para todas as matrizes testadas com ordens próximas.

A análise revela que um aumento na quantidade de *nnz* resulta em um aumento correspondente no tempo de execução, indicando uma relação direta entre essas duas variáveis. Por outro lado, ao aumentar a ordem da matriz, não se observa necessariamente um aumento proporcional no tempo de execução. Este fenômeno é evidenciado pela Figura 7, onde a matriz `cage14` exibe um tempo de execução maior que a `CurlCurl_4`, apesar de possuir uma ordem menor. Essa análise sugere que, embora a ordem da matriz possa impactar o tempo de execução, outros fatores, como a densidade de elementos não nulos, desempenham um papel mais preponderante na determinação

Figura 8: Resultados dos testes por nnz - produto matriz-vetor



Fonte: Produção do próprio autor

do desempenho do produto matriz-vetor.

A Tabela 3 mostra a análise de complexidade feita para o produto matriz-vetor para as quatro estruturas de armazenamento em estudo. Nela podemos ver como a quantidade de *nnz* é mais impactante do que a ordem da matriz para as comparações, acessos a array, acessos a memória e operações.

Tabela 3: Análise de Complexidade do produto matriz-vetor

Armazenamento	**comparações**	**acessos a array**	**acessos a memória**	**operações**
COO	$nnz+N+2$	$5nnz+N$	$6nnz+N+1$	$3nnz+N$
CSC	$nnz+3N+2$	$4nnz+2N$	$4nnz+3N+1$	$3nnz+2N$
CSR	$nnz+3N+2$	$4nnz+2N$	$4nnz+3N+1$	$3nnz+2N$
Lista Encadeada	$nnz+3N+2$	$2nnz+2N$	$5nnz+4N+1$	$4nnz+3N$

Fonte: Produção do próprio autor.

Ao analisarmos o armazenamento por *Lista Encadeada*, observamos na Tabela 4 uma comparação entre o armazenamento COO e *Lista Encadeada*, uma vez que a COO não apresentou resultados tão satisfatórios quanto as outras formas de armazenamento. Os resultados revelam que o produto matriz-vetor utilizando o armazenamento por *Lista Encadeada* é significativamente mais lento. Por exemplo, para a matriz `Flan_1565`, o tempo é cerca de vinte e duas vezes maior do que com o armazenamento COO.

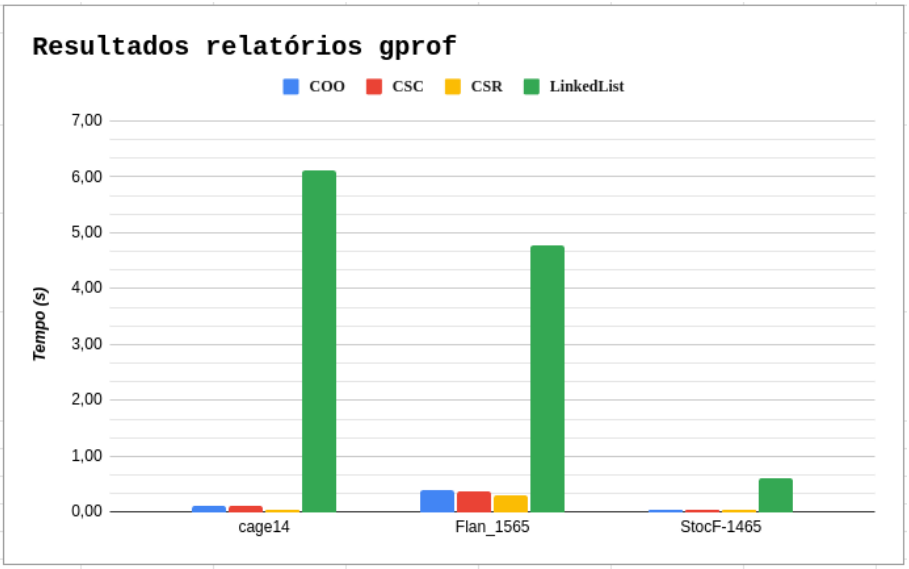
Para investigar a razão dessa disparidade, utilizamos os relatórios gerados pelo gprof. A Figura 9 apresenta o tempo de execução do looping mais interno da função do produto matriz-vetor, especificamente na linha onde ocorrem as operações. Concluímos que o armazenamento por *Lista Encadeada* consome consideravelmente mais

Tabela 4: Tempo de Execução do Produto matriz-vetor - Comparação COO com *Lista Encadeada*

Matrizes	**COO**	**LinkedList**
Hamrle3	0,037	0,262
StocF-1465	0,064	0,632
Geo_1438	0,167	2,856
Long_Coup_dt6	0,226	3,574
Flan_1565	0,297	6,626

Fonte: Produção do próprio autor.

Figura 9: Tempo de execução do looping mais interno da função do produto matriz-vetor



Fonte: Produção do próprio autor

tempo devido ao uso ineficiente de memória. A alocação dinâmica de nós na memória não garante um aproveitamento eficiente da cache do computador, ao contrário dos outros métodos de armazenamento, que utilizam vetores alocados dinamicamente, proporcionando um melhor uso da memória cache.

Os testes para a fatoração ILU(p) também foram organizados em dois cenários, um para a fatoração ILU(p) sem preenchimento e outro para a fatoração ILU(p) com preenchimento. Para cada cenário, foi considerado um conjunto diferente de matrizes, pois a fatoração ILU(p) com preenchimento prevê um tempo maior de processamento, inviabilizando testes com matrizes de ordem muito elevada. Além disso, para cada um desses dois cenários foi considerada a análise de maneira similar ao produto matriz-vetor, por nnz e por ordem N.

As Tabelas 5, 6, 7 e 8 mostram nnz e N, respectivamente, para cada cenário da fatoração ILU(p) em análise. A Figura 10a e a Figura 10b apresentam os tempos de execução da fatoração ILU(p) para as matrizes das Tabelas 5 e 6. Fica evidente que a estrutura de armazanamento CSR obteve o menor tempo de execução para todas as matrizes testadas em ambos os casos de teste. Também é possível observar que o aumento da ordem N não resulta em um

Tabela 5: Matrizes utilizadas para os testes por *nnz* sem preenchimento

Matrizes	**nnz**	**N**
thermal2	8.580.313	1.228.045
CurlCurl_3	13.544.618	1.219.574
dielFilterV2real	48.538.952	1.157.456
dielFilterV3real	89.306.020	1.102.824

Fonte: Produção do próprio autor.

Tabela 6: Matrizes utilizadas para os testes por ordem *N* sem preenchimento

Matrizes	**nnz**	**N**
nd24k	28.715.634	72.000
F1	26.837.113	343.791
Fault_639	27.245.944	638.802
CurlCurl_4	26.515.867	2.380.515

Fonte: Produção do próprio autor.

Tabela 7: Matrizes utilizadas para os testes por ordem *N* com preenchimento

Matrizes	**nnz**	**N**
raefsky3	1.488.768	21.200
venkat01	1.717.792	62.424
power9	1.887.730	155.376
language	1.216.334	399.130

Fonte: Produção do próprio autor.

Tabela 8: Matrizes utilizadas para os testes por *nnz* com preenchimento

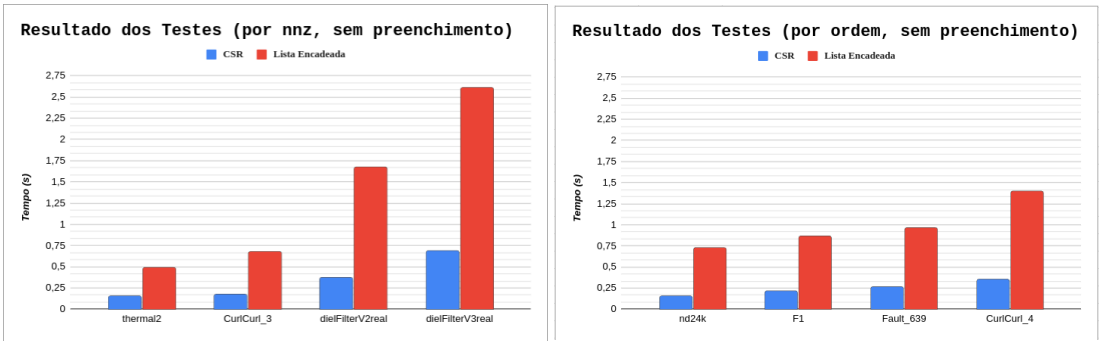
Matrizes	**nnz**	**N**
majorbasis	1.750.416	160.000
xenon2	3.866.688	157.464
PR02R	8.185.136	161.070

Fonte: Produção do próprio autor.

aumento significativo no tempo de execução, mas o aumento de *nnz* da matriz aumenta significativamente o tempo de execução da fatoração ILU(p).

As Tabelas 9 e 10 nos mostram os tempos de execução obtidos para matrizes quando foi considerado os níveis de preenchimento $p = 0, 1, 2$ e 3. É interessante observar que, quando há preenchimento, o que mais impacta no tempo de execução é a quantidade de preenchimento que a matriz irá receber. A Tabela 11 mostra, para cada

Figura 10: Tempos de execução para a fatoração ILU(p) sem preenchimento, (a) por *nnz* e (b) ordem N.



(a) (b)

Fonte: Produção do próprio autor

um dos níveis de preenchimento a quantidade *nnz* adicionada a matriz. Nela podemos observar que as matrizes `power9` e `PR02R` foram as que mais tiveram preenchimento em cada um dos casos, e isso impacta diretamente no tempo de execução das duas.

Tabela 9: Tempo de execução para a fatoração ILU(p) por ordem N com preenchimento

	matriz CSR				**matriz LinkedList**			
Matrizes	**0**	**1**	**2**	**3**	**0**	**1**	**2**	**3**
`raefsky3`	0,079	0,180	0,298	0,451	0,179	0,483	0,985	1,709
`venkat01`	0,057	0,108	0,215	0,356	0,147	0,347	1,0299	3,052
`power9`	0,098	24,895	59,095	122,229	0,317	89,380	124,532	466,382
`language`	0,139	0,231	0,645	1,636	0,200	0,554	1,904	5,946

Tabela 10: Tempo de execução para a fatoração ILU(p) por *nnz* com preenchimento

	matriz CSR				**matriz LinkedList**			
Matrizes	**0**	**1**	**2**	**3**	**0**	**1**	**2**	**3**
`majorbasis`	0,056	0,165	0,532	1,540	0,142	0,685	2,104	12,422
`xenon2`	0,168	0,547	1,211	2,657	0,389	1,374	4,355	34,980
`PR02R`	0,388	1,367	4,316	8,751	1,052	5,257	25,266	865,812

Para investigar a disparidade no tempo de execução entre as estruturas CSR e *Lista Encadeada*, foi conduzida uma análise detalhada utilizando a ferramenta gprof, complementada por uma análise de complexidade do código. A análise com gprof indicou que as linhas de código com maior consumo de tempo estão associadas ao acesso aos nós dinamicamente alocados na memória que a estrutura *Lista Encadeada* utiliza, destacando o impacto negativo dessa abordagem na eficiência da execução. As Tabelas 12 e 13 apresentam as respectivas análises de complexidade para as estruturas CSR e *Lista Encadeada*, evidenciando que a *Lista Encadeada* apresenta uma ordem de complexidade superior à da CSR nas comparações, acessos a memória e operações. A maior complexidade observada na *Lista*

Tabela 11: Quantidade de preenchimento nas matrizes de acordo com o nível de preenchimento

Matrizes	**1**	**2**	**3**
`raefsky3`	1.393.112	2.055.256	3.050.392
`venkat01`	1.377.924	2.312.644	3.581.076
`power9`	17.087.116	35.554.934	70.274.455
`language`	1.851.132	5.610.477	10.541.286
`majorbasis`	3.446.484	8.955.624	17.089.392
`xenon2`	5.533.344	10.253.322	19.056.726
`PR02R`	11.420.671	24.723.165	41.487.040

Encadeada deve-se à forma como os valores são inseridos na matriz durante o preenchimento. Na *Lista Encadeada*, a inserção de um valor exige percorrer os nós da linha correspondente, resultando em uma complexidade de O(N). Em contraste, na estrutura CSR, o processo de inserção é realizado em tempo constante, pois já existe um vetor pré-alocado com o tamanho da linha, o que permite acessar diretamente a posição desejada e adicionar o valor de forma eficiente. A estrutura CSR apresenta uma complexidade ligeiramente maior apenas no caso de acessos aos *arrays*, no entanto, conforme apontado pela análise, esse fator tem um impacto relativamente baixo no desempenho geral. Com base nas duas análises, bem como nos resultados dos testes, conclui-se que a estrutura CSR é a opção mais eficiente para a operação de fatoração ILU(p).

Tabela 12: Análise de Complexidade para a fatoração ILU(p) usando a estrutura de armazenamento CSR

Matrizes	**Complexidade**
comparações	$(9/2)nnz.N + (25/2)nnz + 20N + 4$
acessos a array	$10nnz.N + (79/2)nnz + 27N - 1$
acessos a memória	$(7/2)nnz.N + (27/2)nnz + 25N - 1$
operações	$(11/2)nnz.N + (49/2)nnz + 37N + 5$

Fonte: Produção do próprio autor.

Tabela 13: Análise de Complexidade para a fatoração ILU(p) usando a estrutura de armazenamento *Lista Encadeada*

Matrizes	**Complexidade**
comparações	$nnz.N^2 + 4nnz.N + (23/2)nnz + 18N + 8$
acessos a array	$5nnz.N + (31/2)nnz + 17N - 3$
acessos a memória	$nnz.N^2 + 11nnz.N + (43/2)nnz + 33N - 4$
operações	$nnz.N^2 + (33/2)nnz.N + 39nnz + 54N + 12$

Fonte: Produção do próprio autor.

6 Conclusões

Com base nos resultados obtidos, conclui-se que a estrutura de armazenamento por *Lista Encadeada* não é adequada para operações como o produto matriz-vetor e a fatoração ILU(p), devido à sua baixa eficiência, conforme demonstrado pelos resultados significativamente inferiores em comparação com outras estruturas analisadas. Embora as estruturas COO e CSC sejam alternativas viáveis, suas características de armazenamento são similares às da CSR, que se destacou como a mais eficiente em todas as operações avaliadas. Dessa forma, a CSR se apresenta como a melhor escolha para a execução do produto matriz-vetor e da fatoração ILU(p) neste estudo.

Dado o fraco desempenho da *Lista Encadeada*, optou-se por não incorporá-la ao FEM_CODES, pois sua inclusão não traria avanços significativos ao projeto.

As principais dificuldades enfrentadas ao longo do estudo incluíram a seleção de matrizes adequadas para os testes de fatoração ILU(p), uma vez que várias matrizes não permitiram a realização dessa operação. Além disso, a indisponibilidade temporária do repositório *Suite Sparse Matrix* comprometeu o *download* das matrizes necessárias, mas o problema foi resolvido posteriormente.

Este subprojeto contribuiu para a criação de bibliotecas otimizadas, fornecendo estruturas de armazenamento eficientes para matrizes esparsas e permitindo a execução de operações matriciais de maneira mais eficaz. Essas contribuições impulsionam o desenvolvimento acadêmico e científico, ao disponibilizar ferramentas de alto desempenho para a comunidade de pesquisadores.

Referências

Bell, N. & Garland, M. (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, New York, NY, USA. Association for Computing Machinery.

Davis, J. D. & Chung, E. S. (2012). Spmv: A memory-bound application on the gpu stuck between a rock and a hard place.

Davis, T. A. & Hu, Y. (2011). The university of florida sparse matrix collection. ACM Transactions on Mathematical Software, 38(1):Article 1.

Liu, X., Smelyanskiy, M., Chow, E., & Dubey, P. (2013). Efficient sparse matrix-vector multiplication on x86-based many-core processors. In Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, page 273–282, New York, NY, USA. Association for Computing Machinery.

Saad, Y. (2003). Iterative methods for sparse linear systems, volume 82. siam.

Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., & Demmel, J. (2007). Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, pages 1–12.