Comparação de Implementações de Eliminação Gaussiana em C, Go e Rust

Miguel Rodrigues Botelho[1], **João Teixeira Reis**[2], **Pedro Zanchin**[3]

[1] Ciência da Computação – Universidade Federal de Pelotas (UFPEL)
R. Gomes Carneiro, 01 - Balsa, Pelotas - RS, 96010-610 - Brazil

mrbotelho@inf.ufpel.edu.br

joao.reis@inf.ufpel.edu.br

pzbaldissera@inf.ufpel.edu.br

1. Introdução

Esta análise tem como objetivo comparar implementações do algoritmo de eliminação gaussiana em três linguagens: C, Go e Rust. Para tanto, foram considerados como métricas: aspectos sintáticos, organização da memória, acesso às variáveis, chamadas de funções, comandos de controle de fluxo, métricas de código (como número de linhas e funções), escalabilidade para diferentes tamanhos de matrizes e diferentes níveis de otimizações entre as linguagens. Além disso, o desempenho das implementações foi avaliado por meio de testes experimentais realizados em ambientes Windows e Linux.

Observa-se que contamos com **duas implementações em C**: uma destinada ao ambiente Windows, que utiliza funções de medição de tempo específicas (`QueryPerformanceCounter` da API do Windows), e outra para ambiente Linux, que utiliza a função `clock_gettime`. As implementações em Go e Rust foram desenvolvidas de forma a explorar as características de cada linguagem.

Além disso, os testes foram organizados em dois grupos de otimizações diferentes, para abranger a verificação dos potenciais de cada linguagem.

- **Compilação Padrão:** As linguagens foram compiladas utilizando suas configurações padrão, sem a aplicação de otimizações específicas. Permitindo mensurar o desempenho natural das implementações, refletindo a performance que um usuário comum obteria com a compilação padrão.
- **Compilação Otimizada:** Para explorar o potencial máximo de desempenho, as implementações em **C** e **Rust** foram compiladas com opções de otimização avançadas. Em C, a flag `-O3` foi utilizada, ativando um conjunto agressivo de otimizações pelo compilador, enquanto em Rust foi utilizada a flag `--release`, que realiza otimizações similares. Em destaque, na linguagem **Go**, não foi aplicada uma flag de otimização, pois o compilador Go já incorpora otimizações durante o processo de compilação, não oferecendo opções que se equiparem às outras otimizações. Ele fornece, na realidade, maneiras de retirar algumas das suas verificações durante a execução, melhorando um pouco seu tempo, mas prejudicando a segurança.

2. Análise Estrutural e Sintática

2.1. Tipos de Dados e Organização de Memória

- **C:** Utiliza *arrays* estáticos para representar a matriz e os vetores. O gerenciamento de memória é manual, possibilitando um controle de baixo nível, mas aumentando o risco de erros, como vazamentos ou corrupção de dados.
- **Go:** Utiliza *slices*, referências a *arrays* dinamicamente alocados, facilitando a manipulação e o gerenciamento da memória, embora com um *overhead* associado ao coletor de lixo.
- **Rust:** Utiliza `Vec<Vec<f32>>` para a matriz e `Vec<f32>` para os vetores. O sistema de *ownership* e as verificações em tempo de compilação garantem segurança na manipulação da memória, proporcionando robustez e eficiência.

2.2. Acesso às Variáveis e Estrutura das Funções

- **C:** O acesso aos elementos é feito diretamente via índices (`A[linha][coluna]`). A organização do código é dividida em funções específicas, como `configurar_parametros()`, `inicializar_dados()` e `gauss()`, porém a modularização é considerada básica.
- **Go:** Utiliza índices para acessar elementos dos *slices* e divide o código em funções com responsabilidades bem definidas, possuindo uma alta modularização.
- **Rust:** Destaca-se pela concisão e uso de recursos como iteradores e *closures* (o uso de `rev()` para iterações reversas). A estrutura do código é um pouco menor, com um número reduzido de funções, uma modularização moderada/alta.

2.3. Controle de Fluxo e Modularização

- Em **C**, o algoritmo utiliza laços tradicionais (loops `for`) com controle explícito dos índices.
- Em **Go**, a divisão das etapas do algoritmo em funções claras melhora a legibilidade e manutenção do código.
- Em **Rust**, o controle de fluxo é otimizado com construções modernas, como iterações reversas com `rev()`, o que resulta em um código mais compacto e de fácil compreensão.

3. Métricas de Código

A análise das métricas de código revelou as seguintes características:

Métrica	**C**	**Rust**	**Go**
Linhas de código	75	68	79
Número de funções	4	4	4
Modularização	Básica	Alta	Alta

4. Comparação de Desempenho

Os testes experimentais foram realizados em dois ambientes distintos: Windows e Linux.

4.1. Ambiente Windows

- **CPU:** Intel® Core™ i5-1135G7 (4 núcleos, 2.40 GHz)
- **Memória:** 16,0 GB RAM
- **Sistema Operacional:** Windows 11
- **Compiladores:** GCC (para C, compilado com e sem a flag `-O3`), Go 1.18 e Rust 1.60 (compilado com e sem `--release`)

Sem Otimização: Neste cenário, as linguagens foram executadas utilizando suas configurações padrão (sem a aplicação de otimizações como `-O3` para C ou `--release` para Rust).

Matrizes Médias/Grandes:

Dimensão	C (s/ -O3)	Rust (s/ release)	Go (Padrão)
500	132.014ms	1836.367ms	73.549ms
1000	940.681ms	17153.505ms	577.837ms

Matrizes Pequenas:

Dimensão	C (s/ -O3)	Rust (s/ release)	Go (Padrão)
50	0.186ms	3.339ms	0.753ms
100	1.531ms	26.340ms	1.308ms
150	4.446ms	59.987ms	4.476ms
200	11.866ms	117.958ms	8.309ms

Com Otimização: Para explorar o desempenho máximo, as implementações em C e Rust foram compiladas com otimizações avançadas (`-O3` para C e `--release` para Rust). Na linguagem Go, manteve-se a compilação padrão, uma vez que o compilador Go já aplica algumas otimizações automaticamente, sem oferecer uma flag específica para intensificação das otimizações.

Matrizes Médias/Grandes:

Dimensão	C (c/ -O3)	Rust (c/ release)	Go (Padrão)
500	7.214ms	85.462ms	73.549ms
1000	66.681ms	321.963ms	577.837ms

Matrizes Pequenas:

Dimensão	C (c/ -O3)	Rust (c/ release)	Go (Padrão)
50	0.016ms	0.089ms	0.753ms
100	0.095ms	0.356ms	1.308ms
150	0.306ms	1.352ms	4.476ms
200	0.766ms	2.958ms	8.309ms

Observações:

- Os resultados indicam que a implementação em Rust apresenta melhor escalabilidade, especialmente em matrizes de maior dimensão, beneficiando-se significativamente da compilação com `--release`.
- Para matrizes de dimensão 500, a implementação em Go mostrou desempenho superior, evidenciando sua eficiência em cenários com dados de entrada menores.
- A versão em C, quando compilada sem otimizações específicas (ou seja, sem a flag `-O3`), apresentou desempenho bem inferior, sugerindo a importância das otimizações para explorar totalmente o potencial da linguagem.

4.2. Ambiente Linux

- **CPU:** AMD Ryzen 5 5600G (6 núcleos, 3.9 GHz)
- **Memória:** 16,0 GB RAM
- **Sistema Operacional:** Linux
- **Compiladores:** GCC (para C, compilado com e sem a flag `-O3`), Go 1.18 e Rust 1.60 (compilado com e sem `--release`)

Sem Otimização: Nesta configuração, as implementações foram executadas com as configurações padrão, sem aplicação de otimizações específicas (sem `-O3` para C ou `--release` para Rust).

4.2.1. Matrizes Médias/Grandes

Dimensão	C (s/ -O3)	Rust (s/ release)	Go (Padrão)
500	65.144ms	1275.152ms	51.695ms
1000	503.139ms	10156.395ms	409.013ms

4.2.2. Matrizes Pequenas

Dimensão	C (s/ -O3)	Rust (s/ release)	Go (Padrão)
50	0.090ms	1.373ms	0.054ms
100	0.643ms	10.550ms	0.410ms
150	2.168ms	35.732ms	1.433ms
200	5.066ms	82.645ms	3.391ms

Com Otimização: Para maximizar o desempenho, as implementações em C e Rust foram compiladas com otimizações avançadas (`-O3` para C e `--release` para Rust). Na linguagem Go, manteve-se a compilação padrão.

4.2.3. Matrizes Médias/Grandes

Dimensão	C (c/ -O3)	Rust (c/ release)	Go (Padrão)
500	6.488ms	29.774ms	51.695ms
1000	43.169ms	237.422ms	409.013ms

4.2.4. Matrizes Pequenas

Dimensão	C (c/ -O3)	Rust (c/ release)	Go (Padrão)
50	0.009ms	0.032ms	0.054ms
100	0.049ms	0.243ms	0.410ms
150	0.192ms	0.822ms	1.433ms
200	0.437ms	1.937ms	3.391ms

Observações:

- Em ambos os cenários (com e sem otimização), a implementação em Go apresentou desempenho competitivo, especialmente para matrizes pequenas, mantendo tempos de execução baixos.
- Sem otimizações, C demonstrou desempenho intermediário, enquanto Rust mostrou tempos de execução elevados, indicando maior sensibilidade à falta de otimizações.
- Com otimizações, C e Rust melhoraram significativamente seus desempenhos, com C se destacando em matrizes pequenas e médias/grandes, seguido de Rust; Go, por não contar com uma flag específica, manteve os mesmos tempos.
- Em ambas máquinas (com ou sem a flag), podemos notar em C, que pelo seu uso de *arrays* estáticos, com o crescente tamanho da matriz, em específico o intervalo [500 - 1000], podemos notar o ponto prejudicial de utilizar esse tipo de armazenamento, resultando em um aumento relativo no tempo de execução. Testes adicionais com matrizes ainda maiores comprovam essa lógica.

4.3. Conclusão

A comparação entre as implementações do algoritmo de eliminação gaussiana em C, Go e Rust evidencia que, embora as três linguagens compartilhem uma estrutura similar, cada uma contando com quatro funções principais, elas se diferenciam significativamente na abordagem de gerenciamento de memória e na organização interna do código. Em C, o uso de *arrays* estáticos, prejudicial para matrizes maiores, e o gerenciamento manual conferem um controle de baixo nível que, quando aliado a otimizações agressivas (como a flag `-O3`), possibilita tempos de execução excepcionais, sobretudo em matrizes pequenas e médias. Contudo, essa performance elevada vem acompanhada de uma modularização considerada básica, o que pode dificultar a manutenção e aumentar a propensão a erros.

Por outro lado, tanto Go quanto Rust apresentam uma modularização elevada, resultando em códigos mais organizados e de fácil manutenção. Nos testes realizados em ambientes distintos, um com hardware modesto (Windows com i5) e outro com maior poder de processamento (Linux com AMD Ryzen 5), observou-se que, sem otimizações, a implementação em Go se destaca em cenários com matrizes de menor dimensão, mantendo tempos de execução consistentes, se saindo melhor entre as linguagens no geral, enquanto a versão em Rust inicialmente apresenta tempos mais elevados. Entretanto, quando Rust é compilado com a flag `--release`, há uma melhora notável, evidenciando uma escalabilidade superior para matrizes maiores e demonstrando seu potencial em aplicações que demandam segurança na manipulação de memória.

Ademais, os testes realizados em ambiente Linux revelaram uma vantagem significativa em termos de desempenho em comparação ao Windows, evidenciando que o um

hardware mais potente e a uma gestão mais eficiente dos processos, proporcionou tempos de execução inferiores para as mesmas implementações, principalmente em C e Rust.

Ao aprofundar a comparação entre Rust e Go, nota-se que Rust, com seu rigoroso sistema de *ownership* e verificações em tempo de compilação, oferece uma manipulação de memória mais segura e eficiente, refletindo-se em uma performance otimizada em operações intensivas. Em contrapartida, Go privilegia a simplicidade e a agilidade no desenvolvimento, proporcionando uma implementação rápida e de fácil manutenção, embora o *overhead* do coletor de lixo possa limitar seu desempenho em situações de alta demanda computacional.

Assim, a escolha entre essas linguagens dependerá dos requisitos específicos do projeto: Enquanto C e Rust se revelam ideais para cenários em que o máximo controle e performance são essenciais, Go se mostra vantajosa para projetos que priorizam a clareza, a modularidade e a velocidade de desenvolvimento. Os resultados sugerem que, para aplicações que exigem segurança e escalabilidade sem abrir mão da eficiência, Rust se destaca como uma opção promissora, enquanto Go continua sendo uma excelente escolha para ambientes onde a agilidade de desenvolvimento e a manutenção facilitada são fatores decisivos.