UNIVERSIDADE FEDERAL DE PELOTAS

CIÊNCIA DA COMPUTAÇÃO

IMPLEMENTAÇÃO DA ELIMINAÇÃO DE GAUSS

FELIPE LEONARDO KERWALD SANTANA

LUCAS SUPERTI DA SILVA

KALANI SOSA PEREIRA

1. Introdução

O objetivo deste relatório é comparar a implementação do algoritmo de Eliminação de Gauss em três linguagens de programação distintas: C, Rust e Golang. A análise considerará diferenças sintáticas, tipos de dados, gerenciamento de memória, controle de fluxo, além de métricas quantitativas e testes de desempenho em diferentes tamanhos de matrizes.

2. Implementações

2.1 Implementação em C

A implementação original em C utiliza memória completamente estática no seu código dirigente, mas poderia ser usada memória dinâmica por outro utilizador da função que implementa o algoritmo. Assim a implementação consegue escapar de possíveis problemas que poderiam ocorrer devido à má gestão de memória dinâmica no dirigente, a função que faz o trabalho é agnóstica à natureza da memória. Porém, ainda assim, não possui robustos sistemas de segurança de memória, o que faz com que haja uma possibilidade maior de existência de defeitos relacionados à memória comparado a implementações em linguagens mais robustas quanto à memória, como o Rust.

É feito uso de vetores bidimensionais de tamanho fixo. A função requer o uso de um tipo de vetor multidimensional, o que pode ser um problema e dificultar seu uso com memória dinâmica. Uma solução seria utilizar um tipo ponteiro para ponteiro, funcionalmente equivalente nesse contexto, mas ainda menos seguro que o vetor multidimensional, que já não é seguro.

A memória é completamente estática, eliminando a necessidade de manipulação manual de memória dinâmica, que é amplamente utilizada em C. Porém isso é uma característica do dirigente de teste da função. A função poderia ser utilizada com memória dinâmica e depende de uma indicação explícita do tamanho das estruturas (o parâmetro *n*), o tamanho real das estruturas não é verificado nem implicitamente pelas garantias da linguagem nem explicitamente no código. O efeito disso é um grande potencial para mau uso da memória se o parâmetro *n* for incorreto ou tiver sido adulterado. Uma solução possível seria encapsular a matriz e o parâmetro

n em um struct inicializado e destruido por funções próprias para isso, reduzindo a possibilidade de erros do programador.

O programa por si só não aparenta ser defeituoso no que cabe à memória, porém devido às características manuais da responsabilidade sobre a integridade dos dados há grande potencial para defeitos potencialmente comprometedores ao programa se a função for utilizada por outros utilizadores, especialmente se houver a possibilidade de corrupção ou adulteração externa da memória devido a defeitos não diretamente relacionados.

Foram utilizados laços e seletores tradicionais (for e if), fazendo acesso direto às estruturas em seus blocos. Os fors utilizam variáveis declaradas no início do bloco da função e não no escopo dos fors, o que pode gerar problemas e geralmente diminui a robustez.

Em resumo, a implementação em C permite grande controle da execução e memória utilizada pelo programa, o que traz diversos riscos e requer cuidados muitas vezes não necessários em outras linguagens de programação. Por outro lado, a implementação em C não abstrai demasiadamente a execução do programa, aproximando poder absoluto e controle total da execução quando necessário sem a necessidade do uso direto de assembly.

2.2 Implementação em Rust

A implementação do algoritmo de eliminação de Gauss em Rust utiliza conceitos-chave da linguagem, como a gestão eficiente de memória por meio de vetores mutáveis (`Vec<T>`) e a segurança de tipos e propriedades oferecida pelo sistema de *ownership* e *borrowing*. A utilização de `swap` para trocar linhas da matriz foi utilizada no algoritmo, garantindo estabilidade numérica ao realizar o pivotamento.

A mutabilidade das variáveis, como nos vetores `a`, `b` e `x`, permite a modificação durante o cálculo do sistema linear. A necessidade de serem declaradas mutáveis e a robustez do mecanismo de variáveis mutáveis garantem muito mais segurança à implementação;

Rust, sendo uma linguagem compilada e com foco em desempenho, assegura a eficiência da execução, aproveitando a alocação dinâmica de memória e evitando problemas comuns de segurança, como acesso à memória não inicializada. Embora o código não utilize concorrência explicitamente, a linguagem permite a paralelização eficiente para casos de sistemas maiores, o que seria uma adaptação natural para melhorar ainda mais o desempenho.

Em resumo, a implementação é capaz de exemplificar como Rust combina segurança, performance e simplicidade, sendo uma escolha robusta para algoritmos numéricos complexos.

2.3 Implementação em Golang

A implementação em Go apresentada resolve um sistema linear utilizando o método de eliminação de Gauss com pivoteamento parcial de forma direta e eficiente. No programa, o pacote principal define a função main, onde a matriz de coeficientes e o vetor de constantes são inicializados. Em seguida, a função gaussSolver é chamada para executar o algoritmo.

Dentro de gaussSolver, o procedimento inicia com o escalonamento da matriz, buscando em cada coluna o elemento com o maior valor absoluto para ser o pivô, o que garante a estabilidade numérica. Se o maior valor não estiver na posição atual, ocorre a troca de linhas de forma concisa, utilizando a capacidade de atribuição múltipla do Go. Essa etapa é essencial para transformar a matriz em uma forma triangular superior, eliminando os elementos abaixo do pivô e ajustando o vetor de constantes de acordo.

Após o escalonamento, o sistema é resolvido por meio da retro-substituição. O processo começa a partir da última linha, onde cada incógnita é calculada subtraindo os produtos dos elementos já encontrados e, finalmente, dividindo pelo pivô correspondente, desde que este não seja zero. Caso o pivô seja zero, o código identifica que a matriz é singular e interrompe a execução, informando o erro.

O uso de slices para representar a matriz e os vetores demonstra a flexibilidade e eficiência na alocação dinâmica de memória em Go, embora essas estruturas sejam modificadas diretamente durante o processo. Assim, se a preservação dos dados originais for necessária, seria recomendável criar cópias prévias dos slices.

Em resumo, a implementação ilustra de forma clara e direta como aplicar conceitos fundamentais da linguagem Go, como o manejo de slices, estruturas de controle e atribuições múltiplas, para resolver um problema numérico clássico de maneira robusta e prática.

3. Comparação Sintática e Estrutural

3.1 Declaração de Funções

C: A declaração de funções em C é explícita, com o tipo de retorno e parâmetros definidos de forma clara. O uso de void é comum para funções que não retornam valores, e os tipos de entrada e saída são especificados.

Go: A linguagem Go usa a palavra-chave func para declarar funções, com a sintaxe sendo bastante concisa. Go permite o uso de slices, o que facilita o manuseio de matrizes de tamanho variável.

Rust: Em Rust, as funções são declaradas com a palavra-chave fn, sendo uma linguagem fortemente tipada. Rust utiliza o tipo Vec para vetores dinâmicos e oferece maior segurança e controle na manipulação de memória, evitando erros comuns.

3.2 Estrutura de Dados

C: Em C, as matrizes são definidas com um tamanho fixo, sendo necessárias manipulações adicionais para lidar com matrizes dinâmicas. A alocação de memória é feita manualmente, o que pode levar a erros de gerenciamento de memória.

Go: Go utiliza slices, que são uma versão dinâmica dos arrays, permitindo o redimensionamento automático. Isso facilita o trabalho com grandes volumes de dados, mas a linguagem ainda exige um controle manual de memória, apesar de usar garbage collection.

Rust: Rust adota o tipo Vec para arrays dinâmicos, que são redimensionáveis. A grande vantagem de Rust é seu sistema de gerenciamento de memória baseado em "ownership", que assegura a segurança sem a necessidade de garbage collection, evitando vazamentos e corrupção de memória.

3.3 Manipulação de Matrizes

C: Em C, a manipulação de matrizes é feita diretamente por meio de índices, e a alocação de memória é feita manualmente. Esse controle exige cuidados para evitar acesso fora dos limites da memória.

Go: Em Go, a manipulação de matrizes também é feita por índices, mas com a vantagem de que os slices são mais flexíveis que arrays fixos. Isso permite que o tamanho das matrizes seja alterado de forma dinâmica, sem a necessidade de reescrever o código.

Rust: Rust usa Vec para trabalhar com matrizes e vetores dinâmicos. Como Go, Rust oferece flexibilidade no gerenciamento de tamanho, mas com a vantagem adicional de garantir a segurança de memória durante o tempo de compilação.

3.4 Estruturas de Controle

C: Em C, o controle de fluxo é feito utilizando loops tradicionais (for, while, do-while) e condicionais (if, else). O uso de return é necessário para retornar resultados de funções.

Go: A estrutura de controle em Go é muito similar à de C, com a diferença de que Go possui uma sintaxe mais concisa e um modelo de concorrência (goroutines) embutido na linguagem.

Rust: A estrutura de controle em Rust é semelhante a C e Go, com suporte a if, else, e loops como for, while. Rust também oferece features como pattern matching, que facilita o controle de fluxo baseado em padrões. O retorno pode ser feito sem a palavra-chave return, a última expressão no corpo da função é retornada caso o controle chegue ao fim do corpo da função.

3.5 Tipagem

C: A tipagem em C é estática e explícita, onde os tipos de dados devem ser definidos no momento da declaração das variáveis. C não tem verificação de tipos no tempo de execução, e todos os tipos precisam ser conhecidos em tempo de compilação.

Go: Go também adota tipagem estática, mas com maior flexibilidade, pois permite inferência de tipos. Isso significa que, em muitos casos, não é necessário declarar explicitamente o tipo de uma variável, e o compilador inferirá o tipo correto.

Rust: Rust tem tipagem estática forte, o que significa que o tipo das variáveis é conhecido em tempo de compilação, mas a linguagem oferece maior segurança, especialmente em operações com memória e concorrência. Além disso, Rust oferece a possibilidade de inferência de tipos, o que pode tornar o código mais limpo e legível.

3.6 Tratamento de Erros

C: O tratamento de erros em C em temo de execução é manual. O programador deve verificar se ocorreu algum erro após cada operação que pode falhar. Funções que falham geralmente retornam valores específicos (como -1 ou NULL) ou atribuem valores mágicos a variáveis globais para indicar falhas. No código, porém, não há nenhum tratamento de erros em tempo de execução. O C, porém, possui alguns mecanismos para verificação de erros em tempo de compilação, geralmente para

evitar problemas devido a especificidades da implementação, um destes mecanismos é o static_assert. Além disso, compiladores muitas vezes fazem análises estáticas e avisam sobre determinados problemas potenciais.

Go: Go introduz uma maneira explícita de lidar com erros, retornando valores de erro juntamente com os resultados das funções. Isso é feito utilizando o valor nil para indicar a ausência de erro ou um erro específico para indicar falha.

Rust: Rust adota um sistema de tipos de erro robusto com Result e Option. Isso permite que o programador trate erros de maneira mais segura e explícita, evitando falhas silenciosas. O compilador exige que os erros sejam tratados em tempo de compilação.

3.7 Exibição dos Resultados

- **C**: A exibição dos resultados é feita utilizando a função printf, onde o programador especifica o formato de saída usando a sintaxe de formatação de entrada e saída do C, que se tornou uma sintaxe também utilizada por inúmeras outras linguagens.
- **Go**: Em Go, o fmt.Printf é utilizado para imprimir resultados formatados de maneira semelhante ao printf do C, mas com uma sintaxe um pouco mais moderna.
- **Rust**: Rust usa a macro println!, que é muito similar ao printf, mas com a vantagem de ser integrada à linguagem como uma macro de tempo de compilação.

4. Comparação de Métricas

Foram analisadas algumas métricas como número de linhas e complexidade do código. Com base nisso, a complexidade dos três algoritmos é O(N^3) e o número de linhas nos 3 algoritmos foram relativamente parecidos.

Linguagem	C	Rust	Golang
Linhas de Código	129	97	105
Número de Funções Declaradas	2	2	2
Número de Funções Utilizadas	8	15	10
Uso de Ponteiros	Sim	Não (ownership)	Não

5. Comparação de Desempenho

5.1 Configuração da máquina utilizada para efetuar os testes:

PROCESSADOR: Ryzen 7 5700x

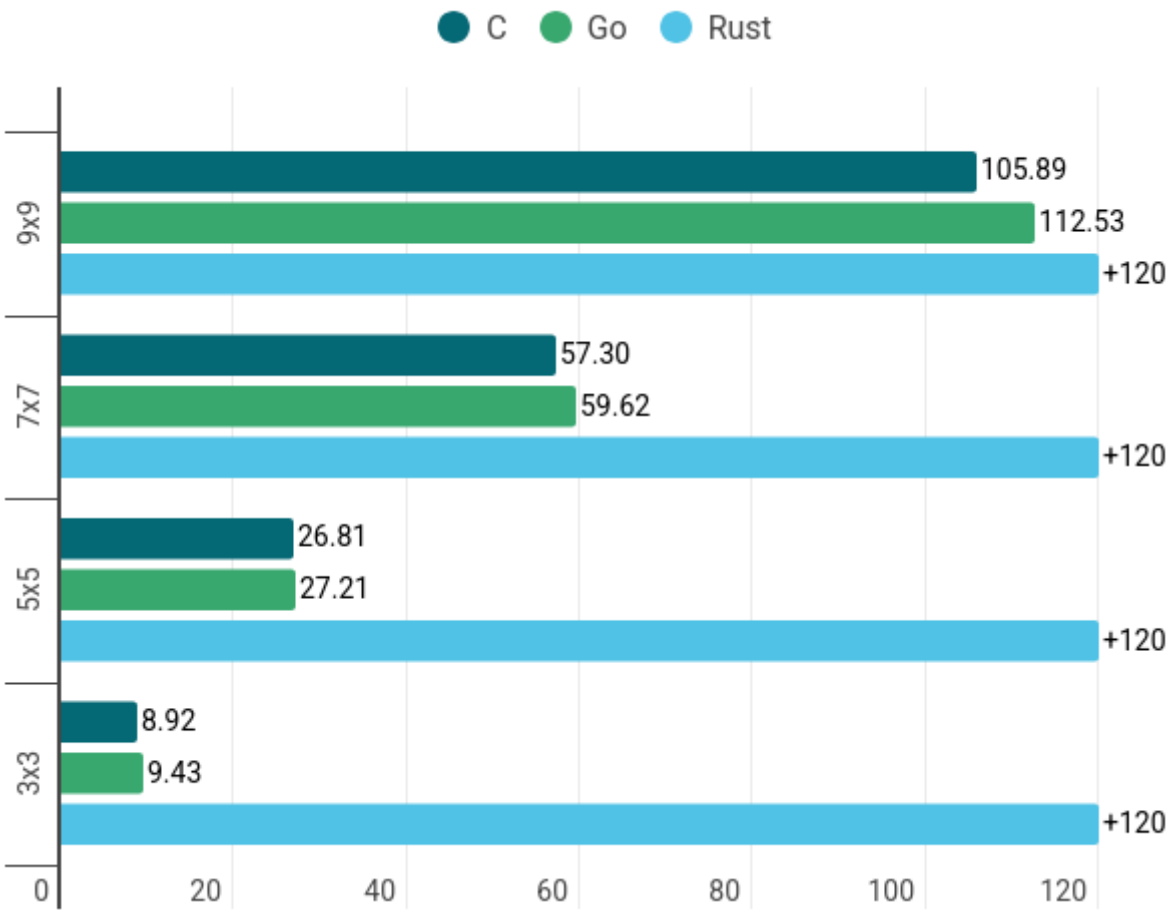
RAM: 32 GB DDR4 3600 MHz

SO: Windows 10 22H2

5.2 Gráficos da comparação de desempenho

O primeiro gráfico mostra os testes feitos sem a utilização de otimizações do compilador, uma observação a ser feita é que o tempo de execução do Rust não otimizado foi muito acima das outras linguagens, por conta disso optamos por utilizar um valor modificado para não atrapalhar a visualização dos outros valores, os valores corretos de Rust estarão logo abaixo do gráfico.

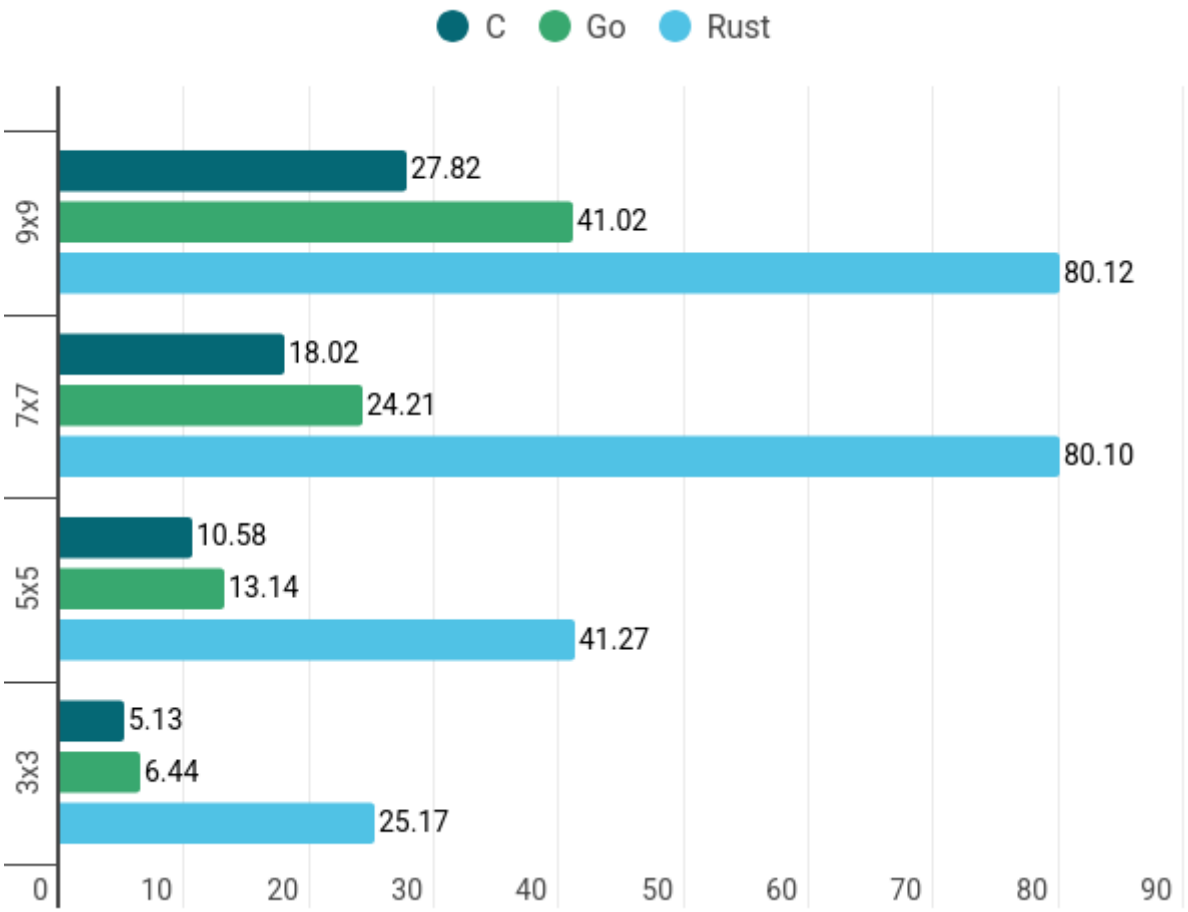
Todos os valores estão em segundos.



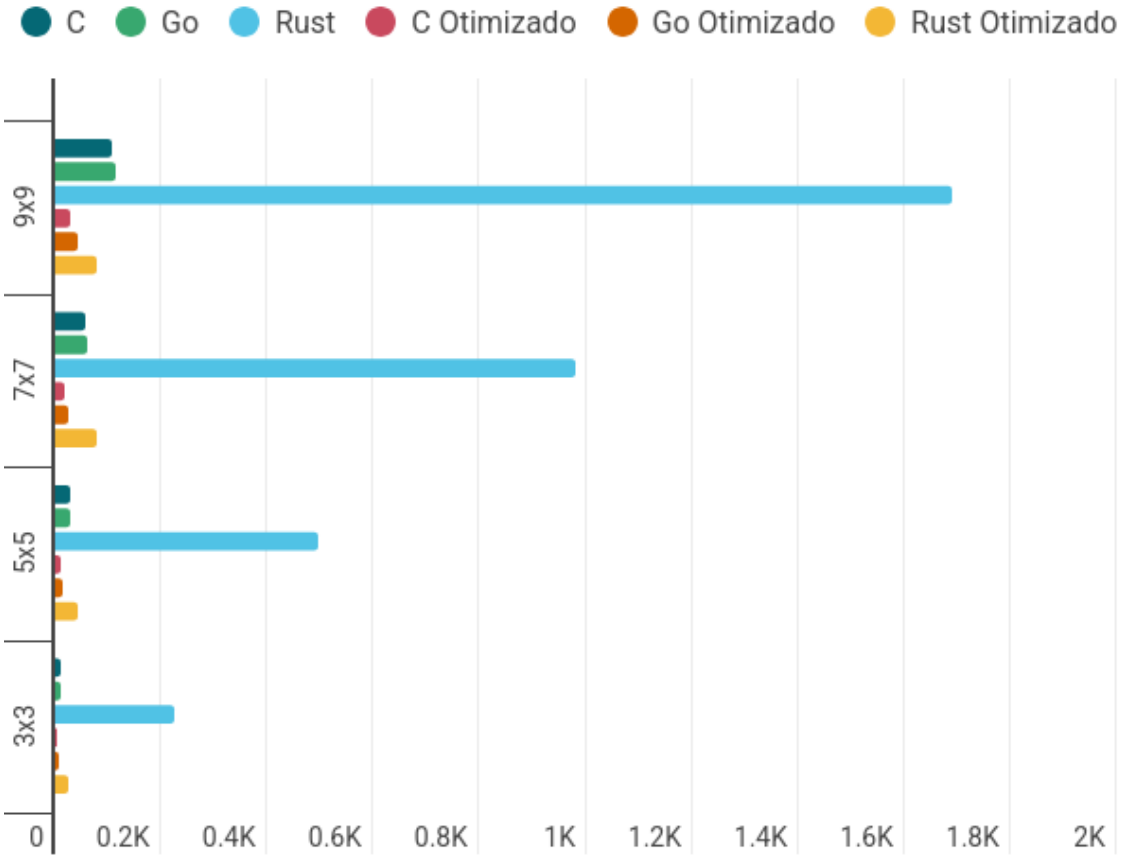
Valores completos de Rust sem otimizações:

- 3x3: 223.80
- 5x5: 493.90
- 7x7: 981.67
- 9x9: 1689.84

O próximo gráfico mostra os resultados dos testes utilizando a otimização.


C
Go
Rust
9x9
27.82
41.02
80.12
7x7
18.02
24.21
80.10
5x5
10.58
13.14
41.27
3x3
5.13
6.44
25.17
0
10
20
30
40
50
60
70
80
90

Por último, apenas para fins de noção, temos um gráfico comparando todos os valores.



6. Conclusão

A implementação em C apresentou o melhor desempenho devido à ausência de overhead de segurança e gerenciamento de memória manual. Rust trouxe segurança de memória e integridade nos dados, enquanto Golang facilitou a implementação com coleta de lixo automática, mas teve um pequeno impacto na performance. A escolha entre essas linguagens depende do equilíbrio entre eficiência e segurança desejado para a aplicação.