UNESP

Ian Rangel Passos, João Vítor de Oliveira Mantilha, Lucas Eiki Okada

Relatório de Linguagem de Montagem

Comparação entre linguagens de alto nível e baixo nível

Rio Claro

2020

1. **Introdução**
2. **Discussão do código**
   1. **Código em Assembly**
   2. **Código em C**
   3. **Programa principal em C**
3. **Teste dos tempos de execução**
4. **Conclusão**
5. **Introdução:**

Neste trabalho, desenvolvemos um programa em C que executa uma mesma função em duas linguagens diferentes (C e Assembly: sintaxes AT&T e Intel), com o objetivo de medir o tempo de execução de cada uma delas.

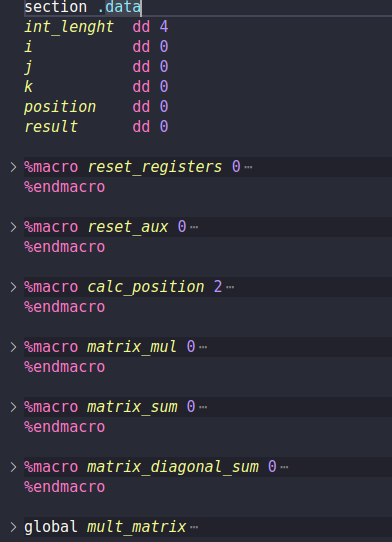
A função recebe três matrizes A, B e C de tamanho L x L, e deve calcular o resultado da expressão (A x C) + B e retornar a soma da diagonal principal da matriz resultante.

1. **Discussão do código:**

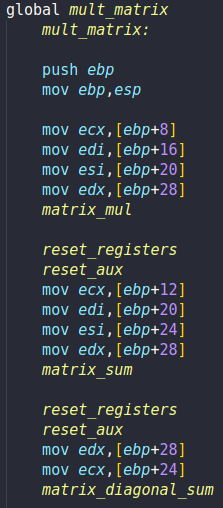
**2.1. Código em Assembly**

O algoritmo foi escrito em duas versões Assembly, uma em NASM e outra em GAS. Como ambas fazem parte da mesma linguagem, as diferenças entre essas duas funções são exclusivamente sintáticas, devido a isso, a análise do código em Assembly será feita apenas em NASM.

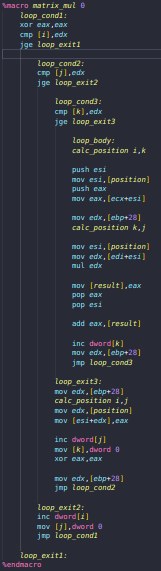
Em NASM, criamos vários macros que executam operações diferentes, e que são chamados a partir da função global mult\_matrix.



Essa, por sua vez, é chamada pelo programa principal em C. Ela coloca o endereço de retorno e os parâmetros passados em uma pilha.

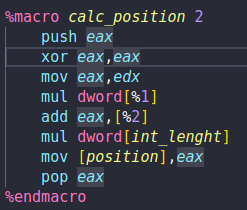


A função matrix\_mul é chamada após os parâmetros necessários para a execução dela serem empilhados. Ela é a responsável pelo cálculo de (A x C).

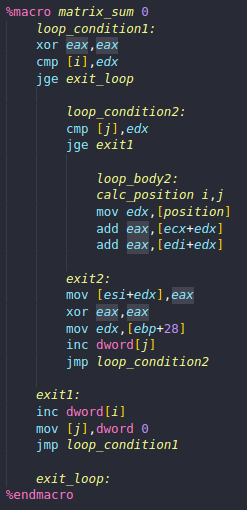


Nela, utilizamos três loops para executar a multiplicação entre matrizes. Para armazenar os valores, usamos uma pilha.

Como a memória só possui uma dimensão, é preciso fazer um macro secundário (calc\_position) para fazer o mapeamento das posições da matriz e encontrar o local onde o elemento desejado está alocado. Neste macro, utilizamos uma fórmula: ((L X I) + J) \*4.

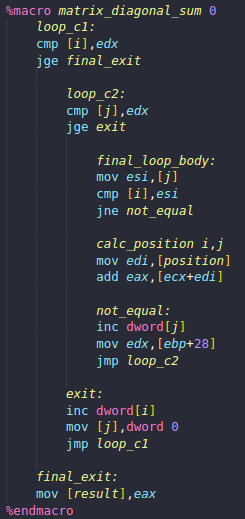


Após a mult\_matrix, a função global chama o macro matrix\_sum, que irá retornar o resultado de (A x C) + B.



Neste macro, temos dois loops que percorrem as matrizes e somam os elementos em uma mesma posição, e armazena o resultado em uma matriz resultante.

Para finalizar, o último macro a ser chamado é matrix\_diagonal\_sum, que calculará a soma dos valores na diagonal principal da matriz resultante.



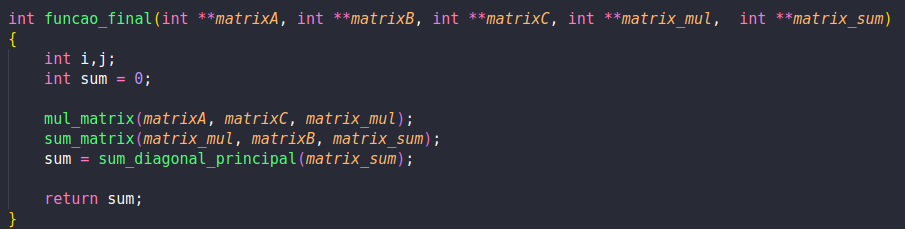
Nesta função, usamos dois loops, que percorrem a matriz e checam se o valor dos índices i e j (linha e coluna, respectivamente) são iguais, e se sim, soma esse valor em um registrador acumulador.

No fim de tudo, global mult\_matrix retorna esta soma para o programa principal em C.

**2.1. Código em C**

No algoritmo em C, as funções seguem a mesma lógica dos macros em Assembly.

Criamos uma função chamada função\_final que chama outras três sub-rotinas: uma que multiplica as matrizes A e C (mul\_matrix), outra que soma a matriz resultante da primeira operação e B (sum\_matrix), e a última que recebe a matriz-retorno da segunda sub-rotina e calcula a soma da sua diagonal principal (sum\_diagonal\_principal).



**2.3. Programa principal em C**

O programa principal em C é o responsável por executar todas as funções descritas acima e mapear o tempo de execução delas.

Para chamar a sub-rotina feita em Assembly, primeiro fizemos a declaração de uma função externa:



E depois, ela pode ser chamada normalmente.



Os métodos usados para medir os tempos de execução dos códigos serão discutidos na próxima seção.

1. **Teste dos tempos de execução**

Com as funções feitas nas diferentes linguagens, é possível medir o tempo de execução de cada uma delas e comparar os resultados.

Para isso, usamos a variável clock\_t da biblioteca time.h, que serve para guardar tempos de processamento.



Agora, para a medição dos tempos, basta usar a função clock(), também da biblioteca time.h. Ela calcula a quantidade de clocks que se passaram desde que o programa foi iniciado. Para calcular o tempo de processamento da função, atribuímos clock() à variável start, chamamos a função desejada, e atribuímos clock() à variável end. Agora, a diferença entre os valores de end e start é equivalente à quantidade de clocks que a função levou para executar.



Para converter este valor para segundos basta dividir por CLOCKS\_PER\_SEC, que armazena a quantidade de clocks por segundo da CPU.



**Resultados:**

Após fazer 10 testes em cada função para diferentes tamanhos de L, obtivemos três médias de tempo para cada caso.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | L=25 | L=50 | L=100 | L=300 | L=500 | L=800 | L=1000 |
| C | 0.0002162 | 0.0012098 | 0.0106987 | 0.1816843 | 0.9002052 | 5.0437007 | 12.8606173 |
| NASM | 0.0021874 | 0.0067409 | 0.0207143 | 0.3564043 | 1.6388757 | 8.0566192 | 18.2043069 |
| GAS | 0.0066165 | 0.0504148 | 0.2955944 | 7.4625527 | 32.6646562 | 158.8369473 | 359.2764413 |

Para melhor visualização dos dados, segue o gráfico:

Os tempos em GAS estão muito acima do esperado, já que está na mesma linguagem do código em NASM, portanto segue outro gráfico comparando os tempos entre C e NASM, somente:

Como é possível ver no gráfico, para valores pequenos de L as diferenças entre as linguagens parecem ser insignificantes, porém, conforme L aumenta, o tempo em NASM fica cada vez maior em relação ao tempo de processamento em C.

1. **Conclusão**

A partir dos dados coletados, podemos concluir que a linguagem C é mais bem equipada para o processamento de cálculos mais complexos e longos em relação à linguagem Assembly. Isso provavelmente acontece devido ao compilador da linguagem C, que é capaz de fazer otimizações no algoritmo, que são mais relevantes na performance do código para cálculos maiores. Além disso, a implementação de códigos em uma linguagem de baixo nível é mais complicada, se tornando mais difícil de obter um código com a melhor eficiência possível.