





Requisitos

Sistema Operacional: Linux Debian-flavor 

Ferramentas: linguagem de programação¹ preferida e Valgrind 

Objetivo

- Implementar um algoritmo que multiplica 2 matrizes que permita a demonstração do efeito positivo ou negativo da memória CACHE com relação a forma de acesso aos dados de um programa.

Tarefas

- Implementar o algoritmo clássico para multiplicação entre duas matrizes $M1_{l1 \times c1}$ e $M2_{l2 \times c2}$
 - Condição de multiplicação: $c1 == l2$
 - $Mr_{l1 \times c2} = M1_{l1 \times c1} * M2_{l2 \times c2}$
 - Requisitos:
 - Alocação dinâmica das matrizes utilizadas
 - Tipo de dado Double
 - Preenchimento de M1 e M2 com números aleatórios
 - Sugestão (em C) de assinatura de função/método:


```
void Mu1M1M2(double **M1, double **M2, double **MR, int l1, int c2)
```
- Validar a implementação do algoritmo clássico
 - Use matrizes pequenas e valide manualmente
 - Use algum programa externo de sua preferência
 - Para tal, será necessário exportar e importar as matrizes M1, M2 e Mr
 - recomendação: arquivo em formato .csv
 - Após validação, remover todos os "prints" em tela desnecessários, E/S consome muito tempo e não é objetivo das análises de desempenho
- Implementar uma função de multiplicação de matrizes melhorada
 - $Mr_{l1 \times c2} = M1_{l1 \times c1} * M2^T_{c2 \times l2}$

```
void Mu1M1M2T(double **M1, double **M2T, double **MR, int l1, int c2)
```
 - É necessário implementar função/método de transposição, sugestão:


```
void transpostaM2(double **M2, double **M2T, int l2, int c2)
```
- Validar implementação melhorada
 - Utilize as mesmas técnicas de validação do algoritmo clássico
- Funcionamento do programa principal:
 - Executa apenas uma multiplicação
 - Tamanho das matrizes é definido via linha de comando de execução

- Método de multiplicação também é definido via linha de comando de execução

```
./mulmatriz.x l1 c1 l2 c2 o|t
```

Onde:

- li: Número de linhas de M_i
- ci: Número de colunas de M_i
- o|t: Método clássico ou M2 transposta

- Após implementações e validações, é hora dos benchmarks de tempo.

- Métricas de tempo requeridas:


- funções **MulM1M2**, **MulMM2T** e **traspostaM2**
- Alocação não é requerido, mas pode ser um extra para enriquecer o relatório

- Em C, existe a biblioteca **time.h**

```
#include <time.h>
...
float tempo = 0.0;
clock_t inicio, fim;
...
inicio = clock();
// aqui está o código que deseja mensurar tempo
fim = clock();
// cálculo para apresentação em segundos
tempo = (float) (((fim - inicio) + 0.0) / CLOCKS_PER_SEC);
```

- Mensurar:

- Execuções para matrizes quadradas de tamanhos $l1 = l2 = i$

- $200 \leq i \leq 2.000$; com passos $i += 200$ (10 variações) 
- Para cada variação, reexecutar 10 vezes, tomar tempo e calcular a média
 - NÃO utilize iterações internas ao programa, isso vicia resultados!
 - Toda reexecução deve ser uma NOVA execução!
- Linux: utilize algum script que execute o programa principal

- Comparar:

- Média de cada variação de execução entre
 - MulM1M2 e MulM1M2T
 - MulM1M2 e (MulM1M2T+transpostaM2)

- Relatório
 - Calcular o speedup entre as soluções MulM1M2 e (MulM1M2T+transpostaM2)

$$\text{speedup} = \frac{\text{Desempenho após melhoria}}{\text{Desempenho antes da melhoria}}$$
 - speedup indica um índice de melhora com relação ao tempo original
 - exemplo: código X melhorado levou 42% do tempo do código sem melhoria.
 - Apresentar os valores de CACHE HIT e CACHE MISS do último nível de cache da sua máquina utilizando a ferramenta valgrind apresentado durante a aula prática.
 - Linha de comando:


```
valgrind --tool=cachegrind ./<nome.x> <args>
```

ATENÇÃO: execução será lenta, muito lenta! Evite usar com matrizes grandes.

Recomendação: matrizes quadradas de **< 1500** linhas/colunas

Entrega

- pacote .zip contendo os arquivos fontes + slides/relatório
 - nome do arquivo: [OAC Prática 03] <nome1 nome2>
- via TEAMS
- Apresentação entre 5 até 10m durante aula prática.