Projeto final MC970

Rian Radeck - 187793
Igor Brito - 171929
Cirilo Morais - 168838

2 de Julho de 2023

Contents

1 Introdução **3**

2 Regressão linear **3**

3 Código **4**
3.1 Elementos da struct Matrix 4
3.2 Utilização no treinamento 4

4 Benchmark **4**
4.1 Implementação Serial 5
4.2 Implementação paralela na CPU 5
4.3 Implementação paralela na GPU 5

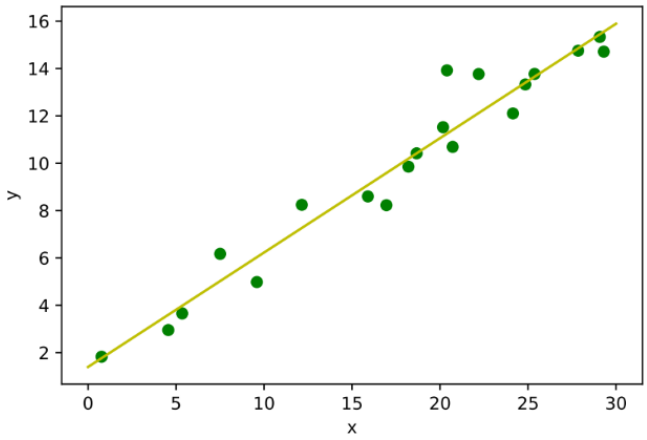
5 Carga sobre os componentes **6**
5.1 CPU - Serial 6
5.2 CPU - Paralelo 6
5.3 GPU 7

1 Introdução

O objetivo principal do projeto é fazer com que seja entendido quão grande é o benefício de paralelizar um modelo de aprendizado de máquina. Para isso nós escolhemos paralelizar modelos de regressão linear que são amplamente utilizados para prever algo a partir de dados iniciais.

2 Regressão linear

O modelo de regressão linear funciona basicamente prevendo uma função linear sobre dados previamente fornecidos. A imagem a seguir é uma reta que minimiza o erro quadrático médio sobre os pontos.



O erro quadrático médio é definido da seguinte maneira:

$$f = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - h(x_i)\right)^2 \tag{1}$$

Onde N é a quantidade de pontos e h é a função linear. Agora nos resta responder como encontrar a função mínima. É simples, devemos seguir a direção do gradiente sobre a função do erro quadrático mínimo. Seja nossa função definida da seguinte maneira:

$$y = w_0x_0 + w_1x_1 + \cdots + w_dx_d \tag{2}$$

onde d é a dimensionalidade do nosso dado (2 na imagem apresentada) e $x_0 = 1$.

Vamos olhar então quanto cada parâmetro w_i da função afeta nossa função de erro. Para isso devemos ver qual a direção do gradiente na dimensão i.

Portanto devemos analisar

$$\nabla f(x_1, x_2, \ldots, x_d) = \begin{bmatrix} \frac{\delta f}{\delta x_1} \\ \frac{\delta f}{\delta x_2} \\ \vdots \\ \frac{\delta f}{\delta x_d} \end{bmatrix} \tag{3}$$

e com base nisso atualizar nossos pesos para ajustar nossa função linear aos dados.

3 Código

A linguagem escolhida foi o C++ e nós decidimos fazer o projeto do zero implementando inclusive nossa própria biblioteca de matrizes. Nós fizemos essa escolha pela flexibilidade e rapidez da linguagem.

3.1 Elementos da struct Matrix

Vamos apresentar o que foi necessário para fazer os cálculos.

- Limites da matriz (row, col, size).
- Vetor que guarda os itens da matriz (*matrix).
- Construtores padrão e de cópia.
- Funções auxiliares de indexação.
- Operador de multiplicação serial (*).
- Operador de multiplicação paralela (%).

3.2 Utilização no treinamento

Foram paralelizadas no código as contas matriciais e o cálculo do gradiente, além da atualização dos pesos.

4 Benchmark

Rodamos os testes numa máquina com as seguintes especificações

- CPU: Ryzen 7 5700G - 8c/16t
- GPU: RTX 3060ti
- Memória: 16GB 2666Mhz

Cada teste foi rodado 5 vezes e pegamos o valor mínimo, para evitar perturbações do SO.

4.1 Implementação Serial

- $N : 20, D : 5 \longrightarrow$ Runtime: $0.02s$
- $N : 100, D : 10 \longrightarrow$ Runtime: $0.14s$
- $N : 100, D : 100 \longrightarrow$ Runtime: $1.11s$
- $N : 1000, D : 10 \longrightarrow$ Runtime: $1.47s$
- $N : 1000, D : 100 \longrightarrow$ Runtime: $11.29s$
- $N : 10000, D : 100 \longrightarrow$ Runtime: $115.86s$
- $N : 10000, D : 1000 \longrightarrow$ Runtime: $(Toolong)s$

4.2 Implementação paralela na CPU

- $N : 20, D : 5 \longrightarrow$ Runtime: $1.27s$
- $N : 100, D : 10 \longrightarrow$ Runtime: $1.30s$
- $N : 100, D : 100 \longrightarrow$ Runtime: $1.83s$
- $N : 1000, D : 10 \longrightarrow$ Runtime: $1.57s$
- $N : 1000, D : 100 \longrightarrow$ Runtime: $4.19s$
- $N : 10000, D : 100 \longrightarrow$ Runtime: $29.97s$
- $N : 10000, D : 1000 \longrightarrow$ Runtime: $4149.96s$

4.3 Implementação paralela na GPU

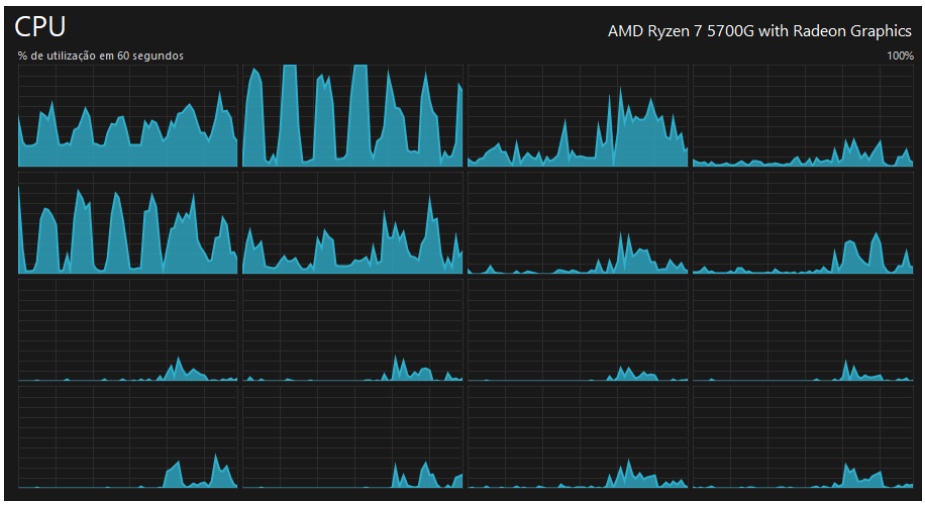
- $N : 20, D : 5 \longrightarrow$ Runtime: $8.23s$
- $N : 100, D : 10 \longrightarrow$ Runtime: $7.57s$
- $N : 100, D : 100 \longrightarrow$ Runtime: $8.90s$
- $N : 1000, D : 10 \longrightarrow$ Runtime: $8.61s$
- $N : 1000, D : 100 \longrightarrow$ Runtime: $10.41s$
- $N : 10000, D : 100 \longrightarrow$ Runtime: $39.33s$
- $N : 10000, D : 1000 \longrightarrow$ Runtime: $2181.24s$

Observe que aqui nós temos um grande overhead para cópia de dados para o Device, porém se isso estiver alinhado com um grande datacenter, o treinamento fica muito mais rápido como observado no teste em que o $N = 10^5$.

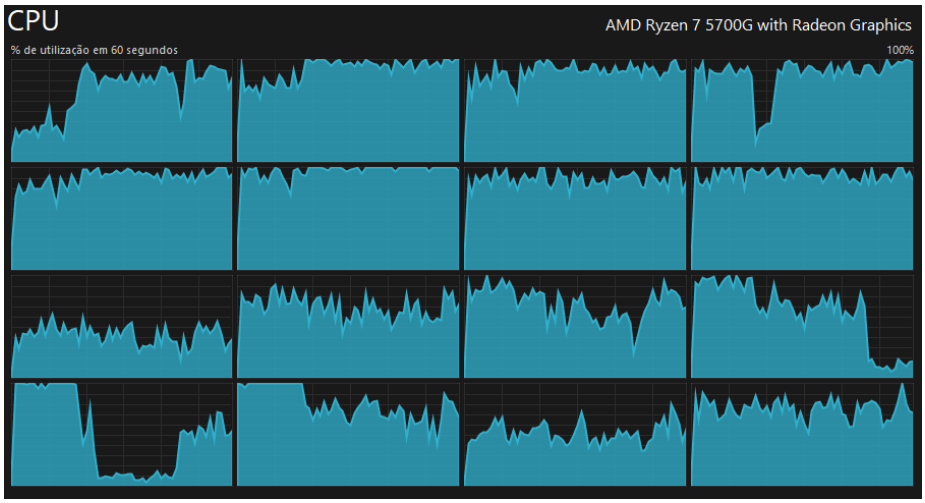
Esse efeito é notado inclusive na implementação paralela da CPU, onde os menores casos rodam mais lentamente quando comparados a implementação seria.

5 Carga sobre os componentes

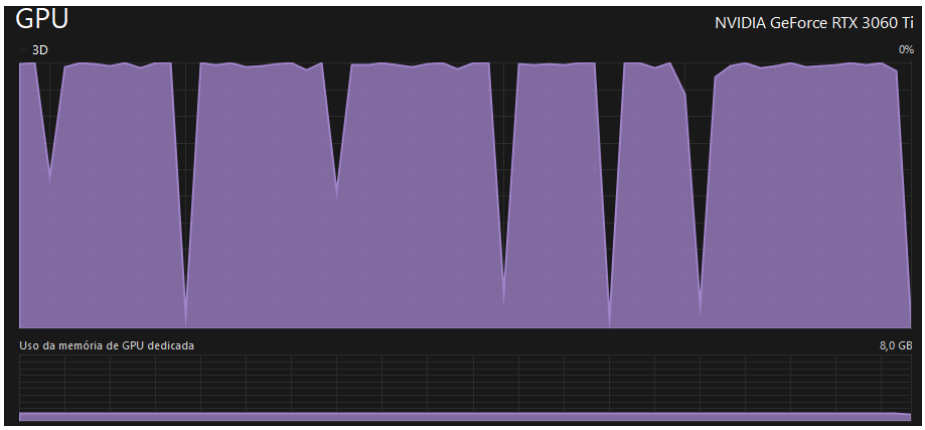
5.1 CPU - Serial



5.2 CPU - Paralelo



5.3 GPU


GPU
NVIDIA GeForce RTX 3060 Ti
3D
0%
Uso da memória de GPU dedicada
8,0 GB