Relatório MC970
Lucas Souza Guimarães - 195948

Descrição do sistema:

Estou usando o WSL2: 5.15.167.4-microsoft-standard-WSL2 (Ubuntu 24.04.2 LTS);

Compilador: gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0;

CUDA Version: 12.9

Informações do CPU:

- Model name: 12th Gen Intel(R) Core(TM) i7-12700H
- CPU family: 6
- Model: 154
- Thread(s) per core: 2
- Core(s) per socket: 10
- Socket(s): 1

Informações da GPU:

- Modelo: NVIDIA GeForce RTX 3060
- Driver: **576.40**
- CUDA: **12.9**
- Memória: 6 GB (6144 MiB)

Ferramentas:

python e matplotlib: para a criação dos gráficos;

IA's generativas pra gerar o makefile.

Descrição do Projeto e implementação:

A ideia do projeto é implementar, do zero, um Multi-Layer Perceptron, que é uma pequena rede neural. Me baseei bastante em como funcionam as bibliotecas tensorflow e numpy, do python. Meu processo de implementação consistiu em:

- implementar em python, um MLP, usando tensorflow e numpy.
- implementar em c++, um MLP, mas dessa vez, implementando as funções do tensorflow em Layer.cpp e MultiLayerPerceptron.cpp e as funções do numpy em Toolkit.cpp, Lembrando que apenas implementei funções que eu estava usando no modelo, numpy e tensorflow tem muitas outras funções que vão além do escopo do projeto.

Seguem detalhes importantes da implementação:

Em Layer.cpp:

- operator() → sobrecarga do operador de chamada de função, desse modo, para aplicar o passo foward fazemos Layer(inputs)
- predict() → é uma versão const do operator() (que garante que não vai haver alteração nos parâmetros do layer) , foi necessário por detalhes do funcionamento do c++.

Em MultiLayerPerceptron.cpp:

- foward() → executa o foward de todos os layers consecutivamente
- backward() → utiliza regra da cadeia e gradient descent pra alterar os pesos das camadas
- compile() → apenas inicializa os parâmetros relacionados a fução de erro.
- fit() → divide a entrada em batches e realiza 1 foward e 1 backward repetindo até o número de épocas especificado
- accuracy() → mede a acurácia do modelo
- predict() → passa os dados pelas camadas e retorna oque o modelo prevê

Em Toolkit.cpp:

- matmul() → multiplicação de matriz simples
- transpose() → transpõe uma matriz
- elementwise_operation() → possui várias sobrecargas, mas é uma operação elemento a elemento entre matriz e escalar, ou matriz e matriz etc... detalhes de implementação estão no header.
- log() → realiza log em todos os elementos de uma matriz(acabou não sendo usada)
- funções de custo(mean_squared_error, binary_cross_entropy, categorical_cross_entropy), estão implemetadas por completude do projeto, embora a unica que usemos seja a entropia cruzada categórica pois nosso dataset possui 26 possíveis respostas (as letras do alfabeto).
- mean() → calcula a média da matriz em uma direção (colunas ou linhas)
- std() → calcula o desvio padrão da matriz em uma direção

Tem algumas outras funções implementadas, mas essas são as mais importantes.
Todo o detalhamento com argumentos e tipos de retorno estão nos headers.

O QUE FOI PARALELIZADO?

Paralelizei algumas funções de toolkit.cpp usando OpenMP, o Objetivo inicial era usar Cuda, mas por simplicidade e também pois não consegui atingir um speedup significativo em relação a paralilização com CPU(OpenMP) eu resolvi manter apenas OpenMP.

Foi testado paralelizar todas as funções de toolkit.cpp com parallel for, mas algumas funções, por serem pouco usadas não faziam diferença no speedup final, e outras não compensavam em paralelismo o que gastavam em comunicação, portanto após alguns testes paralelizei apenas matmul() e elementwise_operation().

COMO RODAR?

Foi feito um makefile que alterna a implementação serial e paralela de Toolkit.h, portanto, se quiser testar a versão paralela, use 'make omp' e se quiser testar a versão serial use 'make'.

Isso vai criar um executável e para rodar basta usar './mlp' no terminal.

Estudo de granularidade:

- Detalhes sobre o MLP e o dataset utilizados:

O dataset usado foi o de reconhecimento de letras da UCI, que tem 20.000 casos de teste, nesse caso são usados 16.000 pra treinamento e 4.000 pra a avaliação do modelo.

O modelo treina dividindo o dataset em batches de tamanho configurável, para os testes de desemepenho, os batches dos modelos testados tem tamanho 512, e os learning rates são 0.001

A seguir, estão o tamanho das camadas dos modelos testados, cada número representa a quantidade de neurons que uma camada possui, em ordem.

1. 64 - 26
2. 128 - 64 - 26
3. 512 - 256 - 128 - 64 - 26
4. 1024 - 512 - 256 - 128 - 26
5. 2048 - 1024 - 512 - 256 - 26

Numero de épocas testados

1. 1 época
2. 3 épocas
3. 10 épocas

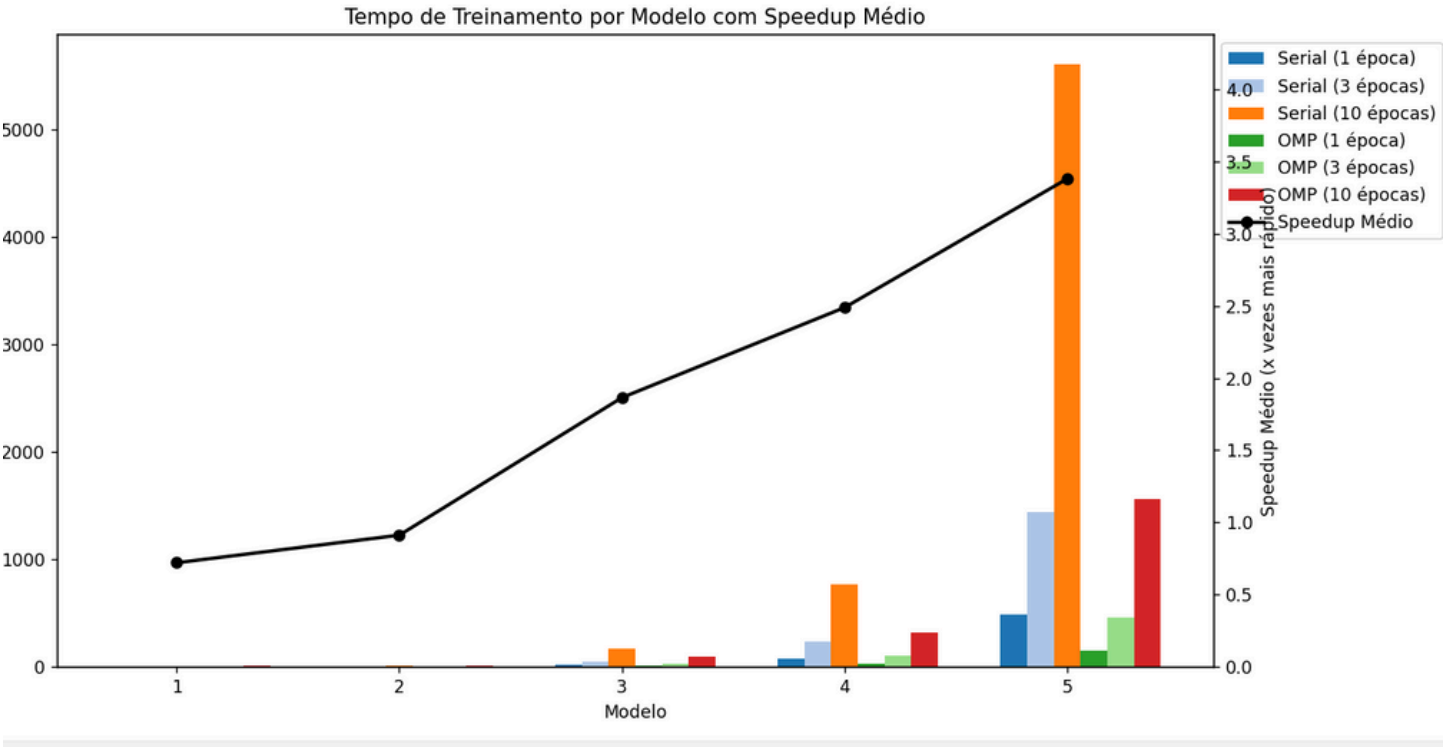
- Tabela e gráfico com os tempos em cada teste:

|O modelo gasta mais tempo durante o seu treinamento, em model.fit(), a operação de multiplicação de matrizes é a que mais consome tempo e dados, visto que é chamada constantemente.

Modelo	1	2	3	4	5
serial (1 época)	0.568 s	1.344 s	17.870 s	77.205 s	485.485 s
serial (3 épocas)	1.770 s	4.061 s	51.525 s	235.219 s	1437.890 s
serial (10 épocas)	5.516 s	13.177 s	168.027 s	769.686 s	5602.943 s
OMP_CPU (1 época)	0.786 s	1.481 s	9.572 s	27.176 s	151.147 s
OMP_CPU (3 épocas)	2.305 s	4.388 s	27.616 s	102.135 s	459.891 s
OMP_CPU (10 épocas)	8.156 s	14.836 s	91.891 s	317.977 s	1564.092 s
Speedup médio	0.721x	0.912x	1.865x	2.489x	3.381x

OBS: o modelo pode conter algumas instabilidades numéricas, eu ajustei o máximo para que não ocorra enquanto está sendo testado, mas caso ocorra, sugiro que reduza o learning rate pela metade na linha mlp.fit() da main (o

quarto parâmetro).



- Análise de granularidade:

Observamos nos dados que o speedup não aumenta de acordo com o número de épocas do treinamento, mas sim com o tamanho das camadas da rede. E com modelos pequenos, com menos camadas e camadas menores, não ocorre speedup, pois as matrizes são muito pequenas para que a multiplicação de matrizes tenha impacto significativo no tempo de treinamento. nesse caso o tempo economizado paralelizando não compensa o tempo de sincronização entre as threads.´

É importante notar que a paralização que foi feita tem a granularidade muito fina, o que foi paralelizado (operações entre matrizes) é praticamente a tarefa de menor tamanho a ser paralelizada nesse projeto. Exemplos de paralelizações que teriam granularidade mais grossa nesse caso são:]

- Paralelismo de modelo, dividiríamos as camadas entre blocos de threads
- Paralelismo de dados, dividiríamos os dados da entrada entre blocos de threads
- Paralelismo pipeline com 1f1b, faríamos um pipeline e alternaríamos os passos foward e backward visando uma bolha menor no pipeline.

Essas técnicas mais avançadas garantiriam uma escalabilidade maior e um maior speedup desse modelo implementado.

Referências

dataset:https://archive.ics.uci.edu/dataset/59/letter+recognition