# Parallel Matrix Multiplication with OpenMP

#include <iostream>

#include <chrono>

#include <omp.h>

#define SIZE 1000

void parallelMatrixMultiplication(int A[SIZE][SIZE], int B[SIZE][SIZE], int C[SIZE][SIZE]) {

#pragma omp parallel for

for (int i = 0; i < SIZE; i++) {

for (int j = 0; j < SIZE; j++) {

C[i][j] = 0;

for (int k = 0; k < SIZE; k++) {

C[i][j] += A[i][k] \* B[k][j];

}

}

}

}

void serialMatrixMultiplication(int A[SIZE][SIZE], int B[SIZE][SIZE], int C[SIZE][SIZE]) {

for (int i = 0; i < SIZE; i++) {

for (int j = 0; j < SIZE; j++) {

C[i][j] = 0;

for (int k = 0; k < SIZE; k++) {

C[i][j] += A[i][k] \* B[k][j];

}

}

}

}

int main() {

int A[SIZE][SIZE]; // First matrix

int B[SIZE][SIZE]; // Second matrix

int C[SIZE][SIZE]; // Result matrix

// Initialize matrices A and B

for (int i = 0; i < SIZE; i++) {

for (int j = 0; j < SIZE; j++) {

A[i][j] = i + j;

B[i][j] = i - j;

}

}

// Perform matrix multiplication in parallel

auto startParallel = std::chrono::high\_resolution\_clock::now();

parallelMatrixMultiplication(A, B, C);

auto endParallel = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> parallelTime = endParallel - startParallel;

// Perform matrix multiplication serially

auto startSerial = std::chrono::high\_resolution\_clock::now();

serialMatrixMultiplication(A, B, C);

auto endSerial = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> serialTime = endSerial - startSerial;

// Print the result matrix

for (int i = 0; i < SIZE; i++) {

for (int j = 0; j < SIZE; j++) {

std::cout << C[i][j] << " ";

}

std::cout << std::endl;

}

// Print the execution times

std::cout << "Parallel execution time: " << parallelTime.count() << " seconds" << std::endl;

std::cout << "Serial execution time: " << serialTime.count() << " seconds" << std::endl;

return 0;

}

# Write a CUDA Program for Addition of two large vectors

#include <stdio.h>

#include <stdlib.h>

// Kernel function for vector addition

\_\_global\_\_ void vectorAdd(int \*a, int \*b, int \*c, int size) {

int tid = blockIdx.x \* blockDim.x + threadIdx.x;

if (tid < size) {

c[tid] = a[tid] + b[tid];

}

}

int main() {

int size = 1000000; // Size of the vectors

int \*h\_a, \*h\_b, \*h\_c; // Host vectors

int \*d\_a, \*d\_b, \*d\_c; // Device vectors

// Allocate memory for host vectors

h\_a = (int\*)malloc(size \* sizeof(int));

h\_b = (int\*)malloc(size \* sizeof(int));

h\_c = (int\*)malloc(size \* sizeof(int));

// Allocate memory for device vectors

cudaMalloc((void\*\*)&d\_a, size \* sizeof(int));

cudaMalloc((void\*\*)&d\_b, size \* sizeof(int));

cudaMalloc((void\*\*)&d\_c, size \* sizeof(int));

// Initialize host vectors

for (int i = 0; i < size; i++) {

h\_a[i] = i;

h\_b[i] = i;

}

// Copy host vectors to device

cudaMemcpy(d\_a, h\_a, size \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, size \* sizeof(int), cudaMemcpyHostToDevice);

// Launch kernel with 1D grid and block dimensions

int blockSize = 256;

int gridSize = (size + blockSize - 1) / blockSize;

vectorAdd<<<gridSize, blockSize>>>(d\_a, d\_b, d\_c, size);

// Copy result back to host

cudaMemcpy(h\_c, d\_c, size \* sizeof(int), cudaMemcpyDeviceToHost);

// Print some elements of the result

for (int i = 0; i < 10; i++) {

printf("%d + %d = %d\n", h\_a[i], h\_b[i], h\_c[i]);

}

// Free device memory

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

// Free host memory

free(h\_a);

free(h\_b);

free(h\_c);

return 0;

In this program, we first declare the host and device vectors for input (**h\_a**, **h\_b**, **d\_a**, **d\_b**) and the device vector for output (**d\_c**). We allocate memory for the host vectors using **malloc()** and for the device vectors using **cudaMalloc()**.

Next, we initialize the host vectors (**h\_a**, **h\_b**) with some sample data.

We then copy the host vectors to the device using **cudaMemcpy()**.

The vector addition operation is performed on the GPU using the **vectorAdd** kernel function. Each thread is responsible for adding the corresponding elements of the input vectors (**a** and **b**) and storing the result in the output vector **c**.

After the kernel is launched, we copy the result back from the device to the host using **cudaMemcpy()**.

Finally, we print some elements of the result (**h\_c**) to verify the correctness of the addition.

Remember to compile the program with the NVIDIA CUDA compiler (**nvcc**). For example:

nvcc vector\_addition.cu -o vector\_addition

# Implement Boston housing price predictionproblem by Linear regression using Deep Neural network. Use Boston House price predictiondataset.

ChatGPT

To implement Boston housing price prediction using linear regression with a deep neural network, we can use a neural network with a single linear layer. Here's an example of how you can achieve this using Python and the Keras library:

In this code, we first load the Boston housing dataset using **load\_boston()** from the **sklearn.datasets** module. Then, we split the dataset into training and testing sets using **train\_test\_split()** from **sklearn.model\_selection**. Next, we scale the features using **StandardScaler** from **sklearn.preprocessing**.

import numpy as np

from sklearn.datasets import load\_boston

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

from keras.models import Sequential

from keras.layers import Dense

# Load the Boston housing dataset

boston = load\_boston()

X = boston.data

y = boston.target

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Scale the features

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Build the model

model = Sequential()

model.add(Dense(1, input\_dim=X\_train.shape[1], activation='linear'))

# Compile the model

model.compile(loss='mean\_squared\_error', optimizer='adam')

# Train the model

model.fit(X\_train, y\_train, epochs=100, batch\_size=32, verbose=1)

# Evaluate the model

mse = model.evaluate(X\_test, y\_test, verbose=0)

print('Mean Squared Error:', mse)

# Make predictions

predictions = model.predict(X\_test)

# Print some example predictions

for i in range(10):

print('Predicted:', predictions[i], 'Actual:', y\_test[i])

We then proceed to build the neural network model using **Sequential** from **keras.models**. We add a single **Dense** layer with a linear activation function to perform linear regression. The input dimension of the layer is set to the number of features in the dataset.

After building the model, we compile it using the mean squared error loss function and the Adam optimizer. We then train the model using **fit()** by providing the training data, the number of epochs, and the batch size.

Once the training is complete, we evaluate the model's performance on the testing set using **evaluate()**. The mean squared error (MSE) is printed as the evaluation metric.

Finally, we make predictions on the testing set using **predict()** and print some example predictions along with their corresponding actual values.

Note: Make sure you have the necessary dependencies installed, such as Keras, scikit-learn, and NumPy. You can install them using pip:

pip install keras scikit-learn numpy

Remember to adjust the code as per your specific environment or requirements.

# Binary classification using Deep Neural Networks Example: Classify movie reviews into positive" reviews and "negative" reviews, just based on the text content of the reviews. Use IMDB dataset

ChatGPT

To perform binary classification of movie reviews into positive and negative categories using deep neural networks, we can utilize the IMDB dataset, which consists of movie reviews along with their corresponding sentiment labels. Here's an example of how you can implement this using Python and the Keras library:

import numpy as np

from keras.datasets import imdb

from keras.models import Sequential

from keras.layers import Dense, Embedding, LSTM

from keras.preprocessing import sequence

# Set the maximum number of words to be used as features

max\_features = 5000

# Load the IMDB dataset

(X\_train, y\_train), (X\_test, y\_test) = imdb.load\_data(num\_words=max\_features)

# Limit the review length to a fixed number of words

max\_review\_length = 500

X\_train = sequence.pad\_sequences(X\_train, maxlen=max\_review\_length)

X\_test = sequence.pad\_sequences(X\_test, maxlen=max\_review\_length)

# Build the model

model = Sequential()

model.add(Embedding(max\_features, 32, input\_length=max\_review\_length))

model.add(LSTM(100))

model.add(Dense(1, activation='sigmoid'))

# Compile the model

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model

model.fit(X\_train, y\_train, validation\_data=(X\_test, y\_test), epochs=3, batch\_size=64)

# Evaluate the model

scores = model.evaluate(X\_test, y\_test, verbose=0)

print('Accuracy:', scores[1])

# Make predictions

sample\_review = ['This movie was fantastic!']

sample\_review = imdb.get\_word\_index().values()[:max\_review\_length]

sample\_review = sequence.pad\_sequences([sample\_review], maxlen=max\_review\_length)

prediction = model.predict(sample\_review)[0][0]

if prediction >= 0.5:

print('Positive review')

else:

print('Negative review')

In this code, we first set the maximum number of words to be used as features (**max\_features**) and load the IMDB dataset using **imdb.load\_data()** from **keras.datasets**. The dataset is automatically split into training and testing sets.

Next, we limit the review length to a fixed number of words using **sequence.pad\_sequences()** from **keras.preprocessing**. This ensures that all reviews have the same length for processing.

We then proceed to build the neural network model using **Sequential** from **keras.models**. The model consists of an embedding layer, an LSTM layer, and a dense layer with a sigmoid activation function for binary classification.

After building the model, we compile it using the binary cross-entropy loss function and the Adam optimizer. We also specify the accuracy as a metric to monitor during training.

We train the model using **fit()** by providing the training data, validation data, number of epochs, and batch size.

Once the training is complete, we evaluate the model's accuracy on the testing set using **evaluate()** and print the result.

Finally, we make predictions on a sample review by first preprocessing it with **get\_word\_index()** and **sequence.pad\_sequences()**. We then pass the preprocessed review to the model using **predict()** and interpret the prediction based on the threshold of 0.5.

Note: Make sure you have the necessary dependencies installed, such as Keras, NumPy, and TensorFlow. You can install them using pip:

pip install keras numpy tensorflow

Remember to adjust the code as per your specific environment or requirements.