Backpropogation in Neural network using CUDA parallelism

Kartikay Kaul[1],Shreyans Magdum[2], and Umapriya Sankaranainar Renganathan[3]

Abstract— **The Backpropagation Network Simulator developed by Karsten Kutza is a pivotal implementation in time-series forecasting, specifically in predicting the annual count of sunspots. This network model, based on the work of Rumelhart, Hinton, and Williams in the realm of error propagation and learning internal representations, forms the foundation of this simulator. The original implementation in C serves as the basis for this project, which aims to transform the existing sequential code into parallel CUDA C code, leveraging the immense computational power of GPUs.**

This undertaking involves a detailed parallelization process to enhance the simulator's performance, allowing for accelerated training and prediction tasks on GPUs. By harnessing the parallel processing capabilities of CUDA, the objective is to achieve significant speedup in network training, enabling quicker and more efficient forecasting of sunspot activity.

I. INTRODUCTION

The Backpropagation Network Simulator is a pivotal tool in the domain of time-series forecasting, specifically tailored for predicting the annual count of sunspots. Initially developed by Karsten Kutza, this simulator implements the renowned Backpropagation algorithm introduced by Rumelhart, Hinton, and Williams. Their work, outlined in "Learning Internal Representations by Error Propagation" within the Parallel Distributed Processing framework, laid the theoretical groundwork for this simulator's architecture.

The existing implementation of the simulator, written in C, serves as the starting point for this project. The motivation behind this endeavor is to harness the immense parallel processing capabilities offered by GPUs through CUDA programming. By transforming the sequential code into parallel CUDA C code, the objective is to expedite the training phase of the network model.

Sunspot prediction, being a time-series forecasting task, demands substantial computational resources, especially when dealing with large datasets. Leveraging the parallelism inherent in GPUs, this project aims to significantly enhance the simulator's performance. Through parallelization, the goal is to achieve faster convergence during network training, ultimately leading to more accurate and timely predictions of sunspot activity.

The ensuing sections delve into the comprehensive process of converting the existing C code into CUDA C, detailing the strategies employed to parallelize critical computations, optimize memory access, and exploit the GPU architecture for accelerated network training. This transformation aims to unlock the potential for substantial improvements in computational efficiency and forecasting accuracy, marking a significant step forward in time-series prediction using neural networks.

II. IMPLEMENTATION PROCEDURE

A. Variables declared

- **Basic Data Types:** BOOL, INT, and REAL: Alias definitions for int and double data types, respectively. Boolean Values:
- **FALSE and TRUE:** Represent boolean values 0 and 1, respectively.
- **NOT, AND, OR:** Macros for logical operations `!`, `&&`, and `||`. Mathematical Constants and Functions: **MIN**(x, y), **MAX**(x, y): Macros for obtaining the minimum and maximum of two values. **LO**, **HI**: Constants for lower and upper bounds (0.1 and 0.9 respectively). **BIAS**: Constant representing bias (set to 1).
- **Mathematical Operations**: **sqr**x: Macro for squaring a value ($x \times x$). Network Structure Definitions:
- **LAYER and NET**: Structs representing a layer of a neural network and the entire network, respectively. They contain fields such as the number of units, output, error terms, connection weights, and other network parameters. Network Layer and Unit Configuration:
- **NUM_LAYERS, N, M**: Constants defining the number of layers and the number of units in each layer.
- **Sunspot Dataset Configuration**: FIRST_YEAR, NUM_YEARS: Constants representing the first year and the total number of years in the sunspot dataset. Constants defining the range of years for training, testing, and evaluation (TRAIN_LWB, TRAIN_UPB, TEST_LWB, TEST_UPB, EVAL_LWB, EVAL_UPB).
- **Sunspot Dataset Storage**:
- **Sunspot**: Array storing the sunspot data for the specified number of years. **Statistics Variables**:
  Mean: Variable to store the mean value of the sunspot dataset. TrainError, TrainErrorPredictingMean, TestError, TestErrorPredictingMean: Variables to hold various error metrics during training and testing phases.
- **File Handling**: **f**: File pointer for file operations.

These declarations and definitions encompass various aspects of a neural network, including its structure, dataset configuration, mathematical operations, and error metrics. They set the groundwork for implementing a backpropagation network for sunspot prediction.The variales are declared in a separate header file and called in the main and kernel files.

B. Random number Initialization

In the `InitializeRandoms()` function, the `srand(4711)` call initializes the random number generator by seeding it with a constant value, 4711. The `RandomEqualINT()` function generates a random integer within a given range defined by `Low` and `High`. By using `rand() % (High - Low + 1) + Low`, it calculates a random integer by taking the remainder of `rand()` divided by the range size and then adds the lower bound `Low`. Similarly, the `RandomEqualREAL()` function generates a random real number within a defined range by normalizing `rand()` to a value between 0 and 1, then scaling and shifting this value to fit within the specified real number range defined by `Low` and `High`. This method ensures that the generated real numbers are uniformly distributed within the given range.This part of the code is not parallelized as it is random numbers generation.

C. Normalization of the Sunspots values

The CUDA code comprises three distinct kernels aimed at various data processing tasks. The ‘normalizeKernel‘ kernel is designed to normalize an array of sunspot data stored in ‘d_Sunspots‘ on the GPU. It calculates the normalized value for each element based on the provided maximum and minimum values, applying a linear scaling formula. This kernel operates in parallel with each thread handling a specific element in the array, ensuring efficient normalization across the entire dataset. On the other hand, the ‘optimizedReduction‘ kernel implements a parallel reduction algorithm to compute the sum of an input array. It utilizes shared memory to store partial sums during the reduction process, reducing memory access latency and improving performance. This implementation employs a parallel reduction tree approach where threads collaborate to aggregate partial sums into a final result efficiently. Finally, the ‘findMinMax‘ kernel aims to determine the minimum and maximum values within an array in a parallel reduction manner. It employs shared memory to compute local minimum and maximum values per thread and then performs parallel reduction within the thread block to obtain the overall minimum and maximum values for the entire array. The ‘NormalizeSunspots()‘ function processes the sunspot data to normalize it and compute its mean value using CUDA kernels on the GPU. Initially, memory is allocated for storage of minimum and maximum values (‘Min‘ and ‘Max‘). CUDA kernels are utilized for parallel computation of minimum and maximum values within the sunspot dataset stored in ‘d_Sunspots‘. The results are copied back to the CPU for further processing to determine the overall minimum and maximum values. Subsequently, the ‘normalizeKernel‘ CUDA kernel is invoked to normalize the sunspot data stored on the GPU based on the obtained minimum and maximum values. This operation scales the data to fit within the specified range defined by ‘LO‘ and ‘HI‘. Additionally, a parallel reduction algorithm is implemented through the ‘optimizedReduction‘ kernel to calculate the sum of all elements in the sunspot data. This reduction process significantly accelerates the computation by aggregating partial sums in parallel, contributing to the efficient determination of the mean value of the sunspot dataset.

D. Initialization

The `InitializeApplicationCUDA()` function serves as an initializer for the CUDA-accelerated application. It begins by setting network parameters for the given neural network (`Net->Alpha, Net->Eta, Net->Gain`). The sunspot data is then normalized using the `NormalizeSunspotsCUDA()` function, which is implemented for GPU acceleration. Memory is allocated on the GPU for the sunspot array `dev_sunspots`, and the data is transferred from the CPU to the GPU for subsequent processing. The code proceeds to calculate training and testing errors by launching CUDA kernels `computeErrorKernel` on the GPU, targeting specific portions of the sunspot data. The computed errors are copied back to the host, updating global variables `TrainErrorPredictingMean`, `TestErrorPredictingMean`. Finally, device memory is freed for the sunspots array and error storage.

Conversely, the `FinalizeApplicationC` function primarily involves closing an open file f presumably used for file operations during the application execution. This function does not involve any GPU operations but ensures proper closure of any open file streams used in the application, concluding its activities without further GPU-related computations.

E. Neural Network Setup

The `GenerateNetworkCUDA()` function orchestrates the generation of a neural network on the GPU. It allocates memory for the network's layers on the host and then iterates through each layer, allocating memory for its associated Output and Error arrays on the GPU using `cudaMalloc()`. Additionally, it initializes the first element of the Output array to a bias term, transferring this initial value from the host to the device. For layers beyond the input layer, it further allocates memory for Weight, WeightSave, and dWeight arrays using `cudaMalloc()`. Finally, it sets properties such as the InputLayer, OutputLayer, and network parameters (`Alpha`, `Eta`, `Gain`) to configure the generated network on the GPU.

F. Assigning random Weights

The `RandomWeights()` function handles the generation of random weights for the neural network's connections on the GPU. It iterates through each layer of the network (excluding the input layer) and calculates the required number of blocks for CUDA kernel execution based on the dimensions of the weight matrices and the specified number of threads per block. Utilizing the ‘initializeRandomWeights‘ kernel, it initializes random weights for each connection in parallel across the GPU threads. The seed value is fixed for demonstration purposes, ensuring reproducibility, but in practice, a variable seed might be preferable for varied random weight initialization. Finally, the function synchronizes the GPU using `cudaDeviceSynchronize()` and checks for any CUDA errors. We were not able to get the expected output using GenerateNetworkCUDA() and RandomWeights() leading to segmentation error and hence retained the C version.

G. *Forward Propogation*

The code provided utilizes CUDA to accelerate the propagation process within a neural network. The function ‘PropagateLayer‘ handles the data transfer and computation on the GPU. GPU memory is allocated and initialized with the weights, outputs, and intermediate values. The ‘PropagateKernel‘ CUDA kernel performs the actual computation, iterating through the weights matrix rows and calculating the weighted sum of lower layer outputs for each neuron in the upper layer. This is followed by the application of an activation function to compute the final output. The processed results are then copied back from the GPU memory to the host. Finally, the allocated GPU memory is freed to manage resources efficiently.

H. *Executing Code and Links*

To compile: nvcc main.cu
To execute: ./a.out
Output is generated in BPM.txt
The video can be accessed at the following link and is in the READ ME: https://drive.google.com/drive/folders/1p13isx–0krFIwq5ZW3EeZwXsXdbjaa2?usp=share_link

I. *Simulation and Results*

III. Performance

The c code took 8 seconds. The cuda code took a few seconds longer. We believe this is due to the fact that the number of input values for the Sunspots were very small and the cost of host to device data transfers and kernel launches harmed efficiency. The code performance should scale better with input sizes.

A. *Work Split*

TABLE I
Task Breakdown

Umapriya	FindMinMax Parallel CUDA Code
Shreyans	Parallel Reduce and Initializations
Kartik	Propogate Layer Calculations

IV. CONCLUSIONS

The CUDA code additions did not significantly boost performance. We believe this is due to the fact that the number of input values for the Sunspots were very small and the cost of host to device data trasnfers and kernel launches harmed efficiency. The code performance should scale better with input sizes.

url

References

[1] Backpropagation Network Assignment. *University of Washington CSE 599 Course Website.* https://courses.cs.washington.edu/courses/cse599/01wi/admin/Assignments/bpn.html