Neural Network in CUDA

Introduction

Neural Network in CUDA is a didactic project intended to serve as a gentle introduction to GPU programming, particularly in the context of deep neural networks. It abides by a modular design that's common in libraries such as PyTorch and enables the seamless addition or extension of features.

CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model created by NVIDIA that allows developers to use NVIDIA GPUs for general-purpose processing. In itself, CUDA is a platform and not limited to one exclusive programming language - that is, it can be integrated with multiple languages, including but not limited to CUDA C, CUDA Fortran, etc. This project is implemented in CUDA C++, which will be synonymous with CUDA throughout this report.

Supposing a system has the Nvidia CUDA Compiler (NVCC) installed, compiling a `.cu` program is done through running `nvcc filename.cu dependencies.cu dependencies.cpp`. Here, the dependencies refer to external files utilized by `filename.cu`. For instance, assuming `test/linear.cu` is to be compiled, the dependencies would be `CPU/linear.cpp`, `GPU/linear.cu`, and `utils/utils.cpp`. Thus, the command would be `nvcc test/linear.cu CPU/linear.cpp GPU/linear.cu utils/utils.cpp`.

The following is a succinct overview of CUDA concepts pertaining to this project.

Kernels

Kernels are functions that are executed in parallel by multiple threads on the GPU and are marked using `__global__`.

Threads

Threads are the smallest unit of execution in CUDA, and thousands or even millions of them can be run simultaneously to execute a kernel.

Blocks

Threads are grouped into blocks, with a maximum of 512 or 1024 threads per block depending on the GPU architecture.

Grids

Finally, blocks are organized into a grid. The combination of grids, blocks, and threads forms the basic hierarchy of execution in CUDA.

Thread Synchronization

Within a block, threads can communicate with one another and synchronize using shared memory. Synchronization between blocks is limited but can be achieved using barriers or atomic operations.

Memory Hierarchy

CUDA provides different memory types with varying access speeds and scopes, including global memory, shared memory, constant memory, texture memory, and registers. Efficient memory management is the cornerstone of well-optimized CUDA programs.

Unified Memory

Unified Memory in CUDA refers to a memory management system that allows for easily sharing memory between CPU(s) and GPU(s) in a CUDA-enabled system by automatically handling data movement between processors.

Modules

The building block of this neural network is the `Module` class, containing three fundamental methods: `forward`, `backward`, and `update`. Every layer inherits this class and overrides whichever methods are necessary. Specifically, `forward` transforms the input data by applying the correct operation on it; `backward` receives the upstream gradients (i.e., the derivative of the loss with respect to the output of this layer) and calculates the downstream gradients (i.e., the derivative of the loss with respect to the input of this layer); and `update` updates the parameters of parameterized modules, e.g., linear or convolutional layers. The Automatic Differentiation section elaborates on `backward`.

Linear Layer

This is a fully-connected network and thus employs linear layers extensively. A linear layer matrix-multiplies the input by a learnable matrix, called the weights, and elementwise adds a vector, called the bias, to the result. One kernel might perform the matrix-multiplication and another the elementwise-addition, but they can be fused together for efficiency. The matrix-multiplication procedure in this project is naive and slow since it does not utilize techniques such as cache-aware loops and tiling.

ReLU

Multiple consecutive linear layers degenerate into a single linear layer and must therefore be separated by non-linearities to be useful. The rectified linear unit (ReLU) is a popular non-linear function that behaves like the identity function for positive inputs and evaluates to 0 otherwise. Its simplicity means ReLU is very easy to calculate and has a straightforward derivative: 1 for positive inputs and 0 otherwise.

Sequential

A recurring pattern in neural networks is a cascade of modules placed back to back, e.g., linear -> ReLU -> linear -> ReLU -> …. The `Sequential` container facilitates implementing this sort of pattern by automatically managing consecutive layers. Its `forward` method chains together the series of modules passed to its constructor, and its `update` method backpropagates the gradients and updates the weights & biases of linear layers.

MSE

The mean squared error (MSE) loss, which measures the squared distance between predicted values and targets, is the criterion the model is optimized for. During training, the loss itself need not be computed since it's only the *gradient* of MSE that is necessary to update the network. Hence, to speed things up, the MSE module's `forward` method simply stores the predictions and target values and does not actually determine the loss. Upon the completion of training, it is important to view the loss to gauge the model's capabilities, and an alternative forward method, named `_forward`, is provided to actually calculate the loss when desired.

Training

The training function accepts a sequential model, an input, a target, and a couple of hyperparameters, and performs gradient descent to update the model.

Automatic Differentiation

Calculating gradients is done through a process known as automatic differentiation (autodiff, or AD), a technique that is able to differentiate arbitrarily complex functions (such as neural network) by defining a set of rules for differentiating fundamental primitives and repeatedly invoking the chain rule. For a neural layer F, with input X and parameters P, that involves defining the output Y = F(X, P), the partial derivative of the output with respect to X, and the partial derivative of the output with respect to P. Moreover, it is assumed the gradient of the

output is also provided. Subsequently, the gradients of X and P are computed by using the chain rule: dL / dX = dL / dY * dY / dX and dL / dP = dL / dY * dY / dP (this illustration uses scalars but generalizes to the multivariable case).

Memory Management

Proper memory management is essential to preclude memory leaks and ensure the safety of the program. In CUDA, cudaMallocManaged and cudaFree correspond to C++'s new and delete functionalities, respectively. Each pass maintains a set of intermediate values necessary to conduct automatic differentiation, which need to be freed once an output has been returned.

Tests

Every module has its own test to ensure matching results between the neural net's GPU implementation and its CPU one (the latter was tested against PyTorch).

End-to-End Example

Lastly, to better understand how these components can be weaved together to train a small deep neural network, an example is provided in `test/train.cu` uses synthetic data found in `data/x.csv` and `data/y.csv`. It benchmarks the GPU and CPU code and also compares their final scores.