PPGrad

Filip Kučera

December 15, 2023

1 Introduction

We've designed **PPGrad** - autograd system written in C++ and leveragin both OpenMP and OpenMPI. It implements basic but scalable and fully documented library of Tensors and Tensor operations, which underneath utilize Eigen::Tensor's efficient Tensor calculations and provide automatic differentiation using automatic and fully transparent computational graph building, which in turn allows for computing derivatives with respect to all Tensors throughout the computational chain. The computation of gradients can be selectively turned off for fast inference and turned on for training. *PPGrad* is described more in Sec. 1.1.

We've then built **PPNN** built on top of *PPGrad*, which is a minimalistic Proof-of-Concept Neural Network library, providing some basic building blocks of Neural Networks (*Layers*, *Optimizers*, *Loss functions*, *Trainers*, *Weight Initializers*, *Activation Functions*) and leverage the fact that only the forward calculations need to be programmed and the backward (differentiation of scalar loss function w.r.t. model parameters [i.e., *PPGrad Tensors*]) pass is computed automatically and transparently. *PPNN* library is described more in depth in Sec. 1.2.

We've also used *Google Test* library for testing all the underlying deterministic (i.e., not neural network training for example) calculations, with numerical gradient comparisons. This testing is set up on our GitHub repository to automatically run all the tests whenever something changes to ensure nothing breaks along the course of the development. Part of this *CI/CD* pipeline is Doxygen documentation generation which is automatically published on our GitHub Pages documentation.

1.1 Autograd system in PPGrad

The automatic differentiation in *PPGrad* is done by carefully subclassing the abstract class *TensorBase*, which implements basic interface of every tensor. Each Tensor in *PPGrad* is then either basic instance of *Tensor* class (direct subclass of *TensorBase* class) or a result of some Tensor operation, such as Tensor-Tensor addition, multiplication, Tensor-Scalar divison etc. Each of these operations takes in two arbitrary Tensors which were subclassed from *TensorBase* as ‘shared_ptr‘ for efficient memory management and avoidance of memory copying and produces another subclassed Tensor such as *AddTensor*, *MultTensor* or *DivSTensor*.

Each of these Tensors keeps track of its inputs to be able to later calculate the gradient with respect to its input(s). Only inputs that are marked as requiring gradients are actually differentiated with respect to.

This naturally and transparently builds scalable **Computational Graph** which can then be from any point differentiated and the gradients are accumulated where needed.

The differentiation is then accessed by calling static *backward(std::shared_ptr <PPGrad::TensorBase<Dim, DT>> root))* method which calculates the derivative of ‘root‘ w.r.t. each of its inputs. Each of the inputs then recursively calls ‘_backward()‘ method to compute gradients w.r.t. each of its inputs. This then effectively computes **derivative of root w.r.t. every other node marked as requiring gradient**!

However, the order of calling ‘_backward()‘ is important and all the downstream gradients must be calculated before propagating the gradient upstream. For this reason, we’ve implement efficient **Topological Sort** that creates deterministic Topological Ordering of our computational graph and which can be then used to call ‘_backward()‘ methods in the correct order.

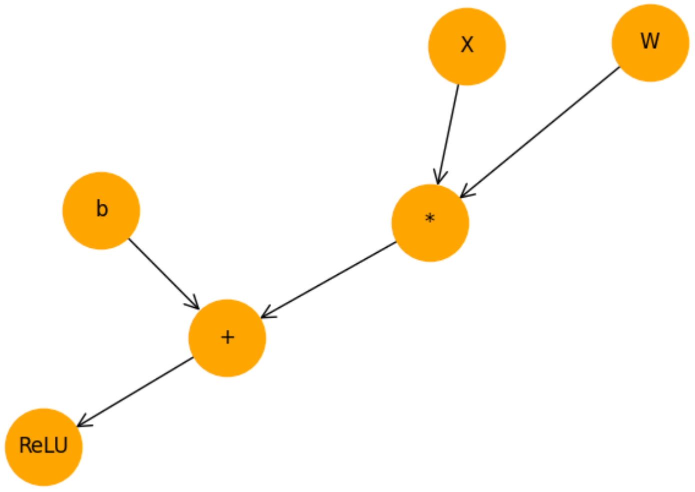

Figure 1: Illustration of a simple Computational Graph for ReLU(W * X + b).

1.2 PPNN Neural Network Library

For demonstrating the power of **PPGrad** we’ve implemented tiny framework for building arbitrarily large Neural Networks using scalable modular building blocks, such as the **Dense** layer, **ReLU** activation, **MSE** & **NLL** losses and **SGD** & **Adam** optimizers to name a few.

We've then implemented **TrainerDP** which utilizes both **Data Parallelism** using *OpenMPI* and *OpenMP*. Before start of each training, we synchronize the models using *Broadcast* from the root node. We also *Scatter* the data evenly among the nodes. After that, we train on each node independently and only every N steps we synchronize the gradients using *Allreduce* operation and update the weights. This results in all the nodes training exactly the same models, yet mostly (depending on the parameter N) independently. The N can be relatively large, even in the order of thousands as shown in [1].

OpenMP was leveraged both to parallelize inference on each sample in a batch, which is completely independent of each other, as well as parallelizing the backward computation, which needs critical section only at the leaf nodes (the network's parameters) but is otherwise completely independent.

2 Analysis

2.1 Training Time vs Number of Cores (Single Node)

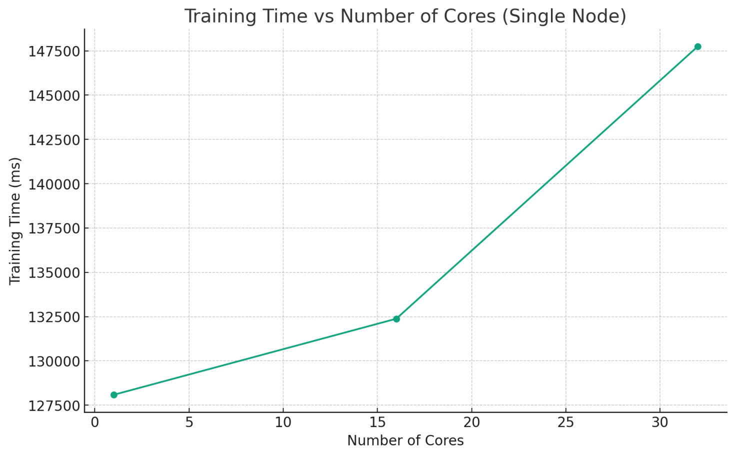

Figure 2: Dependency of training time on the number of cores on a single node.

As we can see on Fig. 2, our choice of benchmark went kind of against us. We deem it mainly due to high variance of measured time, false sharing and the relatively little batch size used for this benchmark which doesn't thus fully exploit parallelism at hand.

2.2 Training Time vs Number of Nodes (16 Cores Each)

On the other hand, scaling with the number of nodes (as seen in Fig. 3 is almost perfectly linear! This is largely due to relatively little communication necessary

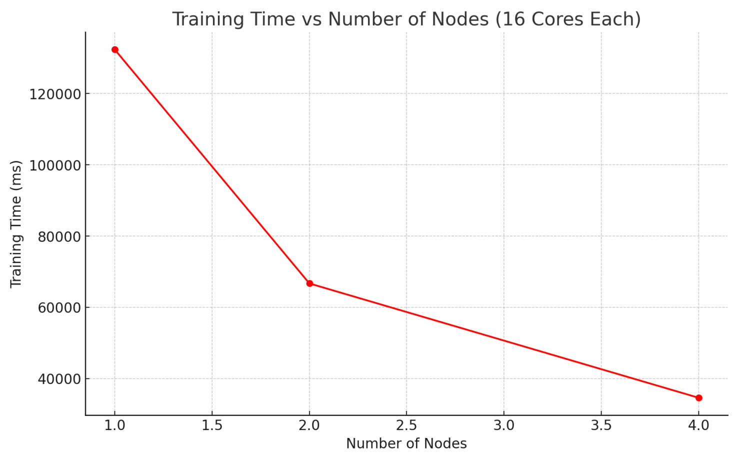

Figure 3: Dependency of training time on the number of nodes, each with 16 cores.

and thus synchronization between the nodes, which allows fully exploiting the parallel nature of Distributed Training.

3 Conclusion

Apart from the wrongly chosen benchmark (and lack of time & mental strength to create another one) which showed actually worse performance with the growing number of CPU cores, we've built rather intricate and mainly very self-enlightening framework which provided a lot of invaluable hands-on experience and fun along the way. Apart from anything else, it helped to make one student tiny step better engineer (hopefully).

References

[1] Arthur Douillard et al. *DiLoCo: Distributed Low-Communication Training of Language Models*. 2023. arXiv: `2311.08105 [cs.LG]`.