Nikhil Kanamarla
CS 598 AIE

Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning

This work is important because parallelism strategies are crucial for efficiently training today's large machine learning models, delivering as much as an order of magnitude performance improvement when implemented correctly vs. incorrectly, according to the authors of Alpa. While certain parallelism strategy generation techniques existed when the paper was written, none could produce a comprehensive parallelism strategy to train a given machine learning model on any given hardware system. Thus, experts had to manually produce and implement custom parallelism strategies tailored to their application. Regardless of the type of accelerator(s) used for training, such an approach requires significant manual effort and familiarity with the system. As such, given the importance of an effective parallelism strategy and the complexity of hand-crafting one, there is a strong case for more thoroughly automating the formulation of parallelism strategies. The problem this paper is trying to tackle is automating parallelism for distributed training. Doing so would hasten the machine learning development development cycle and to ensure the strategies are well-suited to the underlying hardware and system architecture, regardless of whether the user is well-acquainted with it. Thus, Alpa proposes a compiler for finding execution plans with an appropriate parallelism strategy for a given combination of model and hardware system.

The main idea of this paper is that Alpa introduces a two-level hierarchical execution plan space for DL models, utilizing inter-operator and intra-operator parallelisms. Alpa's compiler automatically derives efficient parallel execution plans at each level and implements a runtime to orchestrate these plans on distributed devices. It addresses data, operator, and pipeline parallelism in a unified and automated way. The authors of Alpa choose to classify parallelism as intra-operator or inter-operator, instead of using the model, data, and pipeline parallelism that we commonly see. In the context of machine learning models, operators refer to functions like matrix multiplication and activations. As their names imply, inter-operator parallelism strategies are those which divide different operators in a model, while intra-operator parallelism splits individual operators along some dimension (batch or non-batch) across different computational units. The authors use pipeline parallelism as an example of inter-operator parallelism, since it splits the model at the layer granularity. On the other hand, data and model parallelism are examples of intra-operator parallelism since they involve spreading individual operators across multiple devices**.** A crucial observation made by Alpa is that different parallelism strategies incur different levels of communication overhead between devices. Specifically, intra-operator parallelism typically requires a large amount of communication while inter-operator parallelism requires relatively little. This is because splitting the computation of individual operators across several devices requires these devices to communicate significantly with each other to split the operator and merge their results. Meanwhile, inter-operator parallelism only requires communication to transfer the result of a prior operator to the device(s) responsible for the subsequent operators, which causes relatively little data transfer. The authors also note the disparate bandwidth available between devices within a node and devices between nodes – devices within a node are linked with high bandwidth connections while more distant devices have lower bandwidth available to them. They see that this difference in bandwidth can be effectively mapped onto communication difference between intra- and inter-operator parallelism, indicating that we should utilize communication-intensive intra-operator parallelism on devices within the same node and save the communication-light inter-operator parallelism strategies for separate nodes. This allows the overall parallelism optimization problem into two sub-problems, one for intra-operator parallelism and one for inter-operator parallelism. The first part of determining the parallelism strategy involves thinking about how we should distribute the operators amongst available computing resources. Alpa starts by deciding how to form stages – groups of operators – from the list of operators and also how best to assign these stages to form groups of computing devices to handle each stage. Given an overall cluster mesh of N x M devices, Alpa only considers making device meshes of size 1x2^m, (N x s), or (s x M) to avoid wasting resources. To find the best inter-operator parallelism strategy, Alpa first computes the time needed for a given device mesh to compute a given stage using the intra-operator parallelism algorithm discussed below. Then, they apply dynamic programming with some extra performance optimizations to determine the inter-operator execution plan. Given an operator and device mesh – a group of identical computing devices with high communication bandwidth between them, Alpa computes a suitable intra-operator parallelism strategy to evenly distribute the work between devices in the mesh. It considers the various ways the operator can be sharded amongst the devices and solves an integer linear programming problem to determine the best decision plan to minimize the a combination of communication, computation, and resharding overhead – the cost of reorganizing a tensor whose original dimensions are not compatible with the parallelism strategy. After determining the parallelism strategy, Alpa runs a final parallelism orchestration to optimize inter-stage communication between meshes of different sizes, before creating execution instructions for each mesh. The resulting execution plans perform well across a variety of models. In their evaluation, the authors compare Alpa's automatically generated execution plan to manually-optimized Megatron-LM execution plans on GPT3 and find it performs similarly. When running a MoE model, the authors find their consideration of inter-operator parallelism helps Alpa maintain better performance scaling than DeepSpeed. Finally, when applied to a model with a complex, heterogeneous structure like WideResNet where generating an execution plan by hand is infeasible, Alpa significantly outscales the competition with its execution plan.

There are many strengths of this work. Alpa's hierarchical approach to parallelism is a novel contribution that makes existing parallelism strategies much easier to use. Secondly, empirical results show that Alpa matches the training throughput of hand-tuned parallelism strategies. There are some limitations of this work. While Alpa delivers strong empirical results, it does not claim a globally optimal execution plan. This implies that Alpa leaves room for improvement in most cases, although there may not be much. Further, there are certain limitations to the algorithm. In particular, it does not model inter-stage communication because doing so would significantly increase the search space. While the size of most inter-stage communication is currently quite small, if this ever stops being true, we will need to revise Alpa to account for it. Further, Alpa does not attempt to optimize batch size in its inter-operator objective function and also only considers static schedules and computation graphs. The formulation only considers synchronous training.

There are also some future research directions of this work. This work is primarily concerned with parallelism strategies for training, but it seems many of the optimizations discussed in the paper could also be useful for model inference. Even though the device mesh layouts would probably be simpler during inference, it would be interesting to see if Alpa would improve serving throughput. A major contribution of Alpa is the division of a cluster mesh composed of identical devices into possibly differently sized device meshes. Given that there are now many different types/generations of accelerators now in circulation, it might be useful to extend this ML compiler framework to enable scheduling on heterogeneous devices to help companies better reuse their older hardware.