DSC-204A: Scalable Data Systems, Winter 2024

23. ML Systems - 3

Lecturer: Hao Zhang *Scribe: Sri Koripalli, Krishna Chandu, Rohith Rachala*

1 Static Models vs. Dynamic Models

In static models such as CNNs (Figure 1), we use multiple operators like `Conv 2D` and `Pool` to compose the data flow graph. With a fixed graph, we propagate the data from X to Y, performing prediction, calculating the loss, and propagating gradients through the graph to obtain the results. Despite propagating different data and obtaining different results, the process always involves propagating through the same computation graph without altering the graph.

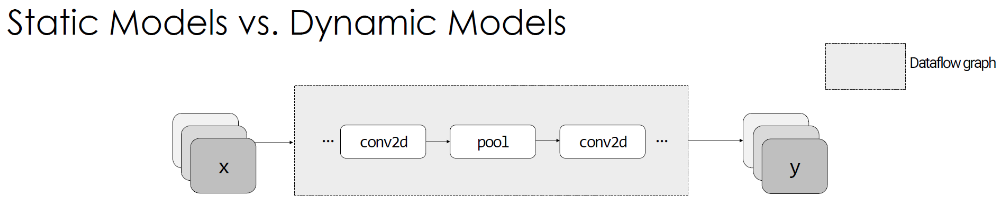

Figure 1: Static Model Example - CNN

However, in many of today's tasks, especially in graph data and natural language data, the data comes with data-specific structures. For example, in the NLP task called syntactic parsing (Figure 2), it comes with a syntax tree expressed in grammar.

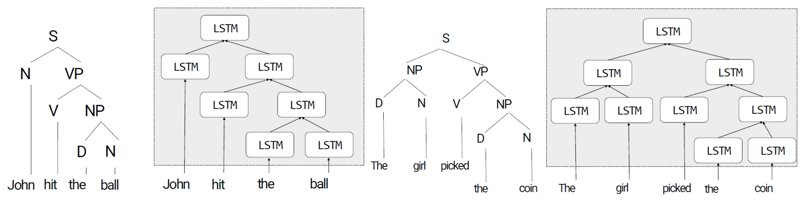

Figure 2: Dynamic Model Example : NLP

We desire the model to encode not only the sentence and the words themselves but also the syntactic structure (Same Figure 2). This requirement leads us to dynamic models. Before transformers became widely adopted, a model called Tree Structure LSTM was prevalently used for many simpler NLP tasks. This Tree Structure model comprises many LSTM units that compose the entire network following the syntactic tree. When a different sentence is fed into the model , we need to compose another graph following its syntactic structure. These dynamic models can be extended to many data types, such as protein structures like AlphaFold and social networks, where the computational data flow graph changes with the input data. This distinction is crucial because, under the assumption that the model is static, the workflow involves defining once and executing many times.

1.1 Static Models Advantages

The advantages of utilizing static models in machine learning workflows are significant, emphasizing efficiency, optimization, and developer convenience:

- The model is defined once and then executed multiple times, which streamlines the processing of data through a consistent computational graph.
- Batching processes become simpler and more efficient, allowing for the exploitation of parallel computing capabilities on GPUs. This enhances the speed and scalability of model training and inference.
- Owing to the static nature of the dataflow work, extensive optimization can be performed at the compilation stage, including techniques such as loop fusion. These optimizations are contingent on the immutability of the graph across different data points.
- The approach benefits developers by reducing complexity and the potential for errors. A static graph model enables easier debugging, versioning, and sharing, thus improving the development workflow.

1.2 Dynamic Dataflow Graphs Using Static Systems

It is conceivable to define a unique graph for each data instance in static systems; however, this approach fundamentally contradicts the intended utility of static models for several reasons. Firstly, articulating the control flow logic within such a framework proves to be exceedingly difficult. The graph would necessitate an extensive use of conditional (if-else) statements to evaluate the progression from one step to the next, an aspect referred to as control flow. Secondly, the pre-runtime expression of this control logic is challenging since the outcomes, which dictate the construction of the current graph based on previous steps, are not predetermined. Lastly, the debugging process for such dynamically constructed graphs within a static system framework becomes notably cumbersome, further complicating the development and maintenance of machine learning models.

1.3 Handling Dynamic Dataflow Graphs

In general, there are two flavors to handle dynamic dataflow models. The first one, which significantly contributes to PyTorch's success, is to adopt an imperative approach rather than a declarative one. In this method, the entire graph is not constructed before runtime. Instead, the graph is built line by line: construct a part, execute it, and then construct another part. This approach underlies PyTorch's execution model, which is imperative. The primary reason PyTorch has stood out, especially since 2018, is the machine learning community's increasing use of dynamic neural networks over static ones. TensorFlow, being a framework that relies on compilation and static declaration, struggles with supporting dynamic models effectively. PyTorch addresses this issue by proposing an imperative programming style similar to Python, supporting dynamic neural networks and addressing problems that TensorFlow cannot handle well.

The second flavor involves symbolic representation. Here, users are still asked to define the entire network beforehand, but instead of utilizing a data flow graph, a different form of representation is used. One such representation is called vertex-centric representation. For example, we simply declare the LSTM unit, and then the system handles the orchestration of that unit following the graph's structure, akin to MapReduce.

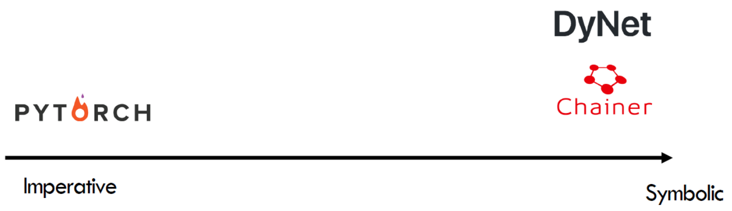

Figure 3: Imperative vs Symbolic

This method has been adopted in Dynet and Chainer. However, this approach has not been very successful for multiple reasons. In the history of machine learning systems, there is still quite a lot of work being done on vertex-centric representation, which is also related to big data processing and called graph processing.

1.4 Questions: Identify whether the below ML tasks follow a static or dynamic Graph

1: CNN Training

A: Static. Because you create a CNN once and continue working on the same graph without changing it.

2: CNN Inference

A: Static. CNNs don't have variables like sequence length. The input is a fixed-size image.

3: GPT3 Training (transformers decoder)

A: Static because every time you make a batch, the batch has a maximum sequence length, and you pad the input if it does not reach the maximum sequence length.

4: GPT3 inference with batch size 1

A: Static. Sequence length and batch size are constant.

5: GPT3 serving (ChatGPT): Model is being served online with arbitrary traffic requests

A: It could be static or dynamic. It depends on the design of your service (serving system). If you opt for a static approach, mimicking training behavior, you always create a batch where you pad all the short sequences, which is inefficient. The latest technique treats the problem as a dynamic graph, batching different requests of varying lengths together, with a constantly changing batch size. This implies that the input has two variable definitions: one for batch size and another for sequence length, requiring the system to manage this dynamism. In most cases, for an efficient implementation, the graph is usually dynamic, necessitating the management of these dynamics.

2 DL Dataflow Graph Optimization

If the user is capable of defining the dataflow graph, then the system can acquire a comprehensive understanding of the entire model. This enables the generation of an optimized version of the graph that achieves the same results in significantly less time. A considerable community is dedicated to these graph optimizations. The logic behind this is that, with the complete graph at hand, numerous transformations can be applied by utilizing the mathematical properties of the subgraphs. For instance, in the left graph of the second illustration (Figure 4), we initially perform $A \times B$ and $A \times C$ to obtain X and Y. However, it is possible to transform the graph by concatenating B and C to form a larger matrix with an extended dimension, then conducting a single matrix multiplication with A and subsequently splitting the result into X and Y. Often, the modified approach runs faster than the original while yielding identical results. Implementing these optimizations can be challenging due to the potential complexity of the graphs. It necessitates the application of graph theory analysis to identify equivalent graphs and determine the more efficient variant.

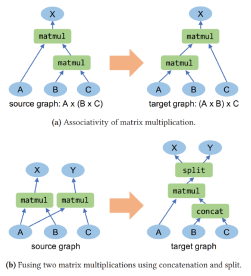
X
matmul
matmul
A
B
C
source graph: A x (B x C)
target graph: (A x B) x C
(a) Associativity of matrix multiplication.
X
Y
split
concat
source graph
target graph
(b) Fusing two matrix multiplications using concatenation and split.

Figure 4: DL Graph Optimization

3 DL Graph Compilation

In deep learning, the purpose of graph compilation is essentially to enable the deployment of a graph representation model, developed in Python by the user, across a wide array of devices with varying architectures (Figure 5). The goal is to avoid the necessity of manually coding the same graph for each unique device. Often, the same machine learning model is deployed on laptops, data centers, smartphones, Raspberry Pis, and Apple Silicon devices. The idea is for the user to develop the model for a single device, and then have a system that compiles this model to run efficiently on different hardware platforms. This process is known as deep learning graph compilation.

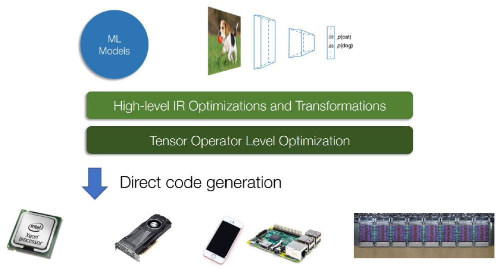
ML Models
p(cat)
p(dog)
High-level IR Optimizations and Transformations
Tensor Operator Level Optimization
Direct code generation

Figure 5: Graph Compilation for multiple Devices

This area of research is vibrant and attracts daily attention within the community, becoming a popular field among major Silicon Valley companies such as Apple and Google. A leading project in this domain is the TVM (Tensor Virtual Machine), which Google heavily invests in to facilitate model deployment across its mobile devices. The core technique behind this involves defining your data flow graph and providing it to the system. The compiler then intercepts the layers within the graph, captures the representation, and generates the necessary code from this representation for each specified hardware platform.

4 Deep Learning Parallelism

4.1 Recap

The math equation for any DL model. What are the problems with parallelising this equation? There are things that we can use if we want to parallelise somethings. 1. Parallelisze the computing, 2. Distribute the memory. This has already been done for big data processing, so lets see if we can apply the same for machine learning. In case of computing, the majority of machine learning computing is basically the Delta function (Put big delta symbol here) and F(.). The forward and backward flows. We need to parallalize this function across many GPUs. So basically we need to parralazie the two functions. In case of memory, we need to store two things, Data (D) and Theta (Put theta here) parameters. These parameters could grow pretty large (350GB for GPT3.) We need to find a way to smartly partition the data across different devices.

The third component that needs to be taken care while parallalising the tasks is the communication. With the help of communication, we are synchronising the parameters after computing each part of the model on different devices. Sometimes the model is cut into different layers, where each layer generates lot of intermediate results (called Activations). These activations should also be communicated to between different nodes.

There are usually two view of ML Parallelisms

4.2 Two views of ML Parallelisms

1. Classical View: Data & Model Parallelism: This view corresponds to the big data processing. Any task can perform data parallelism. It is just partitioning the data. Model parallelism is nothing but task parallelism.

2. New View: Inter-op and Intra-op parallelism: They same set of techniques but different categorization.

4.3 Data and Model Parallelism

In classically view, we are categorizing parallelism into data and model parallelesims. In data parallelsim, we partition D into different devices and each worker works on subset of D. Parameter server is an example of data paralleleism. We also replicate all the computations on different workers where each worker holds a copy of parameter data. All the workers will be synchronised at the end of paramater synchronisation. All the functions and computations graphs will also be replicated in each worker. This kind of parallelism works for any kind of models.

Model parallelism is slightly different. In this case we partition the compute (Theta x Delta). This is much more complex than data parallelism because the computation function is defined for data flow graph and that graph differs from model to model. There is no single unit here because each model is so different. We need to look into the graph and see how you can partition the graph. This is the core systems technique that made GPT3 possible. Since the model is too big, you can't replicate it. Once data is partitioned, handling the data is also complex because the data needs to flow through the computaion graph. Even gradient updation is harder with model parallelism.

4.4 Recap of parameter server

We partition the data and each worker gets a partition. All the workers hold the replicated model (parameters, functions, computational graph) and generate a copy of gradients (synchronised gradients) either through synchronous consistensy model or through relaxed consistency model. There are few represented systems called Poseidon for deep learning, GeePS, BytePS (Production grade). While first two are developed by Prof. Hao, BytePS is developed by ByteDance for their recomendation system training. It is usally hard to train neural networks on paramter server because it's too flexible.

4.5 Data Parallalesim with AllReduce

Instead of parameter server, we can use all reduce for communication of gradients. Each worker computes it's gradients and use all reduce for updating the gradients. The disadvantage of all reduce is that we can not implement relaxed consistency. It can be done with synchronisation. One more disadvantage of Allreduce is that it is very expensive when bandwidth is very limited which is not case in recent years due to Nvidia NVLink. But the main advantage is It is highly optimised (by HPC community). Since 2018 Allreduce was widely used compared to prarameter server. Few examples of systems that use Allreduce for data parallelism are Horovod and Torch.DDP.

5 New View of ML Parallelism

With the increasing complexity of ML models, the need for parallelization of the operations involved is greater than ever. The classical view of Model parallelism is heavily dependent on the structure of the model and the parallel partition of operations differs for each model and creates ambiguity. A newer approach to model parallelism is designed around the computational graphs of models and offers a systematic approach to handling large ML models.

Partitioning computation graphs is a crucial strategy in distributing the computational workload of machine learning models across multiple devices or nodes. This process directly impacts the efficiency of model training and inference, especially for large-scale neural networks. We explore two primary partitioning strategies, Inter-op and Intra-op parallelism, and discuss the trade-offs involved.

Inter-op Parallelism assigns different operations or stages of the computational graph to different devices. It maximizes parallel execution but requires careful synchronization and load balancing. Intra-op Parallelism divides the computation of a single operation among multiple devices, enhancing the computation speed for large, divisible operations.

5.1 Inter-op Parallelism

Inter-op Parallelism focuses on splitting the computational graph of a model into separate stages and assigning each stage to a separate device. This approach is suited for models with distinct computational stages that can be parallelized to reduce overall execution time. Considering a simple example of a Multi-Layer Perceptron, Inter-op parallelism would be denoted by the splits shown in Figure 1.

- **Pros:** Requires less communication between the devices. Since each device can perform computations without communication from other devices, peer-to-peer communication is sufficient for an effective inter-op parallelism.

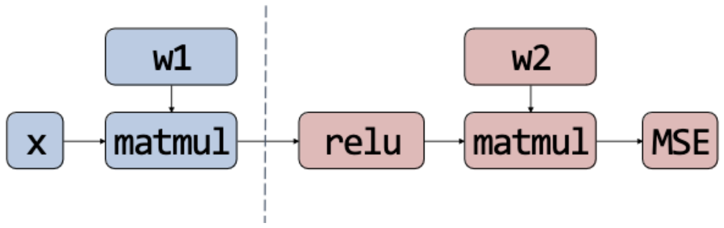

Figure 6: Inter-op Parallelism. Red and Blue indicate separate devices

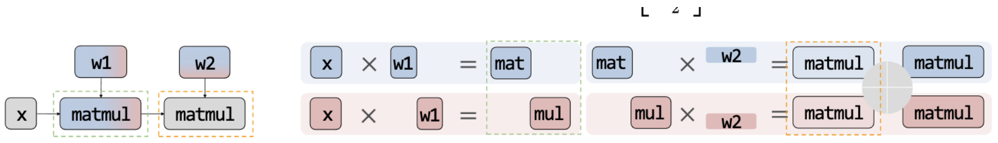

Figure 7: Intra-op operation of Y = X.W1.W2. The last operation is an all-reduce operation to compute the final result

- **Cons:** Given the sequential nature of the split of operations, there can be a lot of idle times in the devices while waiting for devices in the earlier parts of the computations. This demands efficient splitting of loads and scheduling for ideal usage of the devices and maximizing the speedup. We will discuss more about this in a later section.

5.2 Intra-op Parallelism

Intra-op Parallelism focuses on splitting individual operations across multiple devices. It is particularly useful for operations that can be naturally divided into smaller tasks that run in parallel. For the same example of a Multi-Layer Perceptron, Figure-2 shows two strategies in intra-op parallelism that could be implemented.

In the first example, we can see that the operations are row partitioned with W1, and W2 replicated among both devices. Although intermediate results can be calculated parallelly in this manner, for computing the final result, a final All-Reduce operation has to be done. This requires sophisticated collective communication among all the devices.

We can discuss the second example in more detail to understand how Intra-op splits operations among different devices for faster and more efficient processing. For further simpllicity we can remove the relu operator, and focus on the simple operation of

$$Y = X \cdot W_1 \cdot W_2$$

To perform this operation in a Intra-op parallel manner, we can column split the W1 matrix, and row split the W2 matrix. By replicating X among both devices, the matrix multiplication gives us intermediate results. Performing a all-reduce operation on these intermediate results gives us the required output of the Matrix multiplication.

Trade-offs:

- **Pros:** Enhances the speed of computationally intensive operations by leveraging the parallel processing power of multiple devices.

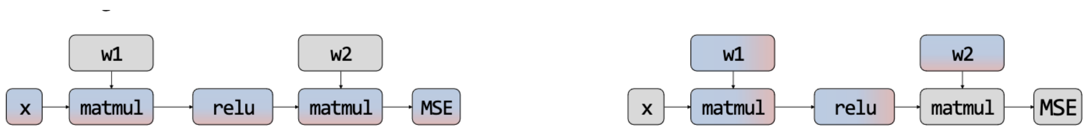

Figure 8: Intra-op Parallelism. The Red-Blue split in the operations indicates the intra-op split of the different operators. The Grey blocks indicate the replicated parts that are used by both devices.

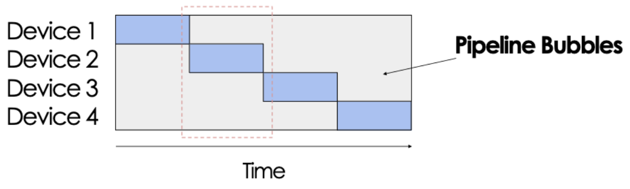

Figure 9: The sequential nature of the inter-op parallelism causes pipeline bubbles resulting in inefficiency

- **Cons:** Requires collective communication mechanisms like all-reduce, which can be more complex and costly than point-to-point communications. It also introduces challenges in aggregating partial results efficiently.

6 Pipeline bubbles in Inter-op parallelism

In this section, we discuss the flow data in the case of inter-op parallelism and how to make it more efficient. Considering the example of a simple neural network divided and split into 4 different stages, each assigned to a separate device for parallel computation, it is evident that the sequential nature of the flow creates a communication dependency of later stages. Stage 2 cannot begin without Stage 1 communicating its result. Figure 3 shows a visualization in a Gannt chart showing the inefficiency of this approach. By pipelining the inputs we can solve the inefficiency problem. Splitting the data into batches and processing the next batch right away helps in the pipelining process and in reducing the bubble, which can be seen in Figure 4.

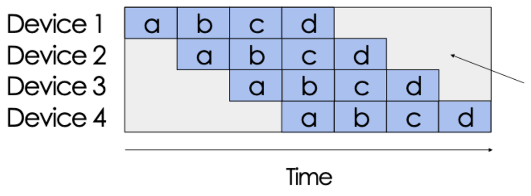

Figure 10: By pipelining the process the bubble inefficiencies can be reduced in the case of Inter-op parallelism