The Limitations of Existing Deep Learning Frameworks: Dynamic Scheduling

OneFlow · Follow
15 min read · Sep 10, 2021

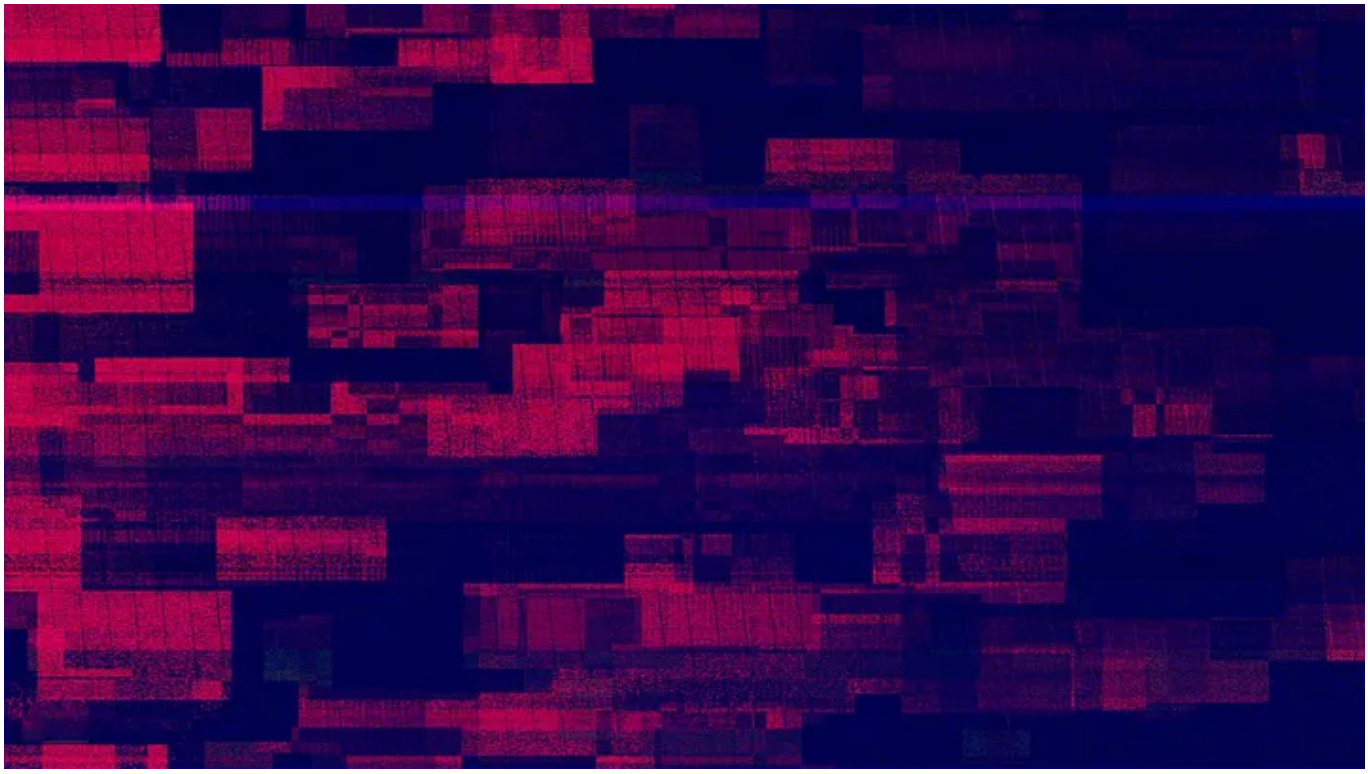
Listen Share

Why redesign a distributed deep learning framework like OneFlow?

An obvious starting point is that the original mainstream deep learning frameworks are inherently deficient. Especially at the abstraction and API levels, they are designed with various shortcomings that greatly inconvenience developers when using them. Although some of the deficiencies are being addressed, there are still some important issues being overlooked.

Therefore, we will launch a series of three articles to discuss in detail the three major shortcomings of the runtime system of the mainstream DL frameworks.

This article, starting from the key issue of thread pool allocation, introduces the computation graph scheduling mechanism, discusses the defects of the traditional frameworks using dynamic scheduling and the problems of setting the optimal number of threads, as well as explains the more elegant solution implemented by OneFlow.



via Pixabay

Written by Yuan Jinhui; Translated by Dong Wenwen

The Problem of Thread Pool Allocation

In the previous article, "The Limitations of Existing Deep Learning Frameworks: Resource Dependency", we discussed an example that an op blocks the thread where it locates because it cannot request enough memory resources, so the thread cannot schedule and execute another op that would otherwise satisfy the execution conditions.

To overcome this problem, one solution is to create another thread, as needed, to schedule and execute the originally executable op. However, scheduling the computation graph or executing the op may also block the thread it locates for other reasons. This risk of stability can be avoided by making the number of threads greater than the parallelism of the task. But as a system resource, the cost of creating and managing threads is considerable, and it is undoubtedly wasteful to always create threads according to the maximum demand.

This raises a new question, how many threads should be set?

While the problem seems simple, the original framework's implementation strategy was unsatisfactory: either implement a fixed-size thread pool or one that can dynamically change the size, which can lead to vulnerabilities or inefficient operation efficiency.

In this article, we'll dig deeper into the nature of this problem and discuss how to solve it gracefully.

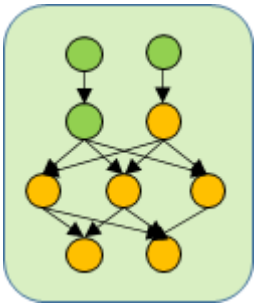


Figure 1: Data flow and directed acyclic graph

As shown in the directed acyclic graph (DAG) above, each node represents an op (either a computation or a data movement operation). The edges between the nodes point from the producer to the consumer, representing the direction of data flow. The scheduler fires the ops to the appropriate hardware resources for execution according to the topological order of the DAG.

For example, in the above diagram, the green nodes indicate that the execution has been completed, while the yellow nodes indicate that the execution has not yet been completed. Obviously, in the next moment, only the yellow node in the second row satisfies the execution condition, and when the execution of this node is completed, the three yellow nodes in the third row satisfy the execution condition.

The advantage of using the data flow model to describe distributed concurrent tasks is that the DAG graph completely and explicitly shows the parallel possibilities of the

program. Using the above graph as an example, the two nodes in the first row can be executed simultaneously if the underlying hardware resources allow. It is also possible for the three nodes in the third row to be executed in parallel when both nodes in the second row are executed. This advantage makes the data flow model widely used in distributed big data processing, functional model, compiler parallel analysis, CPU out-of-order execution and other scenarios.

We define the maximum degree of parallelism of a computational task as the theoretical maximum number of ops executed in parallel. For example, the maximum degree of parallelism in Figure 1 is 3.

The parallelism of the computation graph represents only the theoretically best possibility of parallelism; the actual parallelism depends on the amount of hardware resources available to the system at each moment and the implementation mechanism of the scheduler.

Given the hardware resources available for a computation graph, the scheduler needs to constantly decide which op to start next and which hardware resource to place (map) that op. Mapping from ops to hardware resources in an "optimal" way is the specialty of the scheduler and execution engine.

Pedigree of Dynamic and Static Scheduling

There are two implementation philosophies for mapping ops to specific time slots on specific hardware, one is static scheduling and the other is dynamic scheduling.

Static scheduling means that the mapping relationship is analyzed during the compilation period before running and is executed according to this rule at runtime; dynamic scheduling means that no "preplanning" is done before running, but relies entirely on the online state at runtime to "improvise" and find a feasible mapping solution and implement it within an acceptable time "step by step".

A pedigree is formed from fully dynamic scheduling to fully static scheduling, and the realistic systems usually fall in between, where there may be both a part of the mapping completed at compile-time and a part of the mapping completed at runtime.

The dedicated AI chip can be understood as a completely static mapping, where, for example, convolutional ops and activation layers have dedicated components to perform, and the correspondence between ops and hardware may be completely solidified. Obviously, completely static scheduling may reduce flexibility.

Traditional big data systems (e.g. Hadoop) are usually fully dynamically scheduled. The scheduler tracks the progress of the computation graph and the usage status of all hardware resources (CPU, memory, etc.) in the cluster in real-time, and decides online to which node and which CPU core of that node to distribute the ops or tasks to. Thus dynamic scheduling is the most flexible.

Practice in DL frameworks shows that fully dynamic scheduling is not feasible.

The execution time of DL ops is too short, often tens of milliseconds, and the scheduler takes time to solve the optimal mapping scheme, and it is too late to solve the optimal mapping scheme dynamically online. Moreover, solving the scheme requires a lot of contextual information. Even if a centralized scheduler is preferred, it is not practical to put all the pressure on the central scheduler.

It should be said that the existing frameworks are somewhere between dynamic and static, some are statically pre-determined and some are dynamically determined.

However, static scheduling has more advantages: concise runtime system implementation (of course, complexity is shifted to compile-time), high efficiency, and high stability. Static scheduling does not compromise flexibility because the software framework can generate a different mapping scheme at compile time for each different computational task, except that the same task uses a static mapping scheme at runtime.

As we will see, the trouble of setting the optimal number of threads for the thread pool is unique to dynamic scheduling.

Abstraction of Hardware Resources

The DL framework execution engine involves several concepts related to hardware resources. Operators are used for functional descriptions, and their concrete implementations on specific hardware are called kernels. The state of the operator is managed by OS threads of the operating system. Each hardware resource is

abstracted into a task queue. At runtime, the kernel (i.e., the implementation of the op) is dispatched to the task queue. The hardware scheduler takes the kernel out of the queue and executes it on the hardware resource according to the first-in-first-out (FIFO) rule.

Take Nvidia GPGPU as an example, CUDA stream is a kind of task queue. Other hardware resources, including CPU and network, can also be abstracted into task queues, and we uniformly call such queues streams.

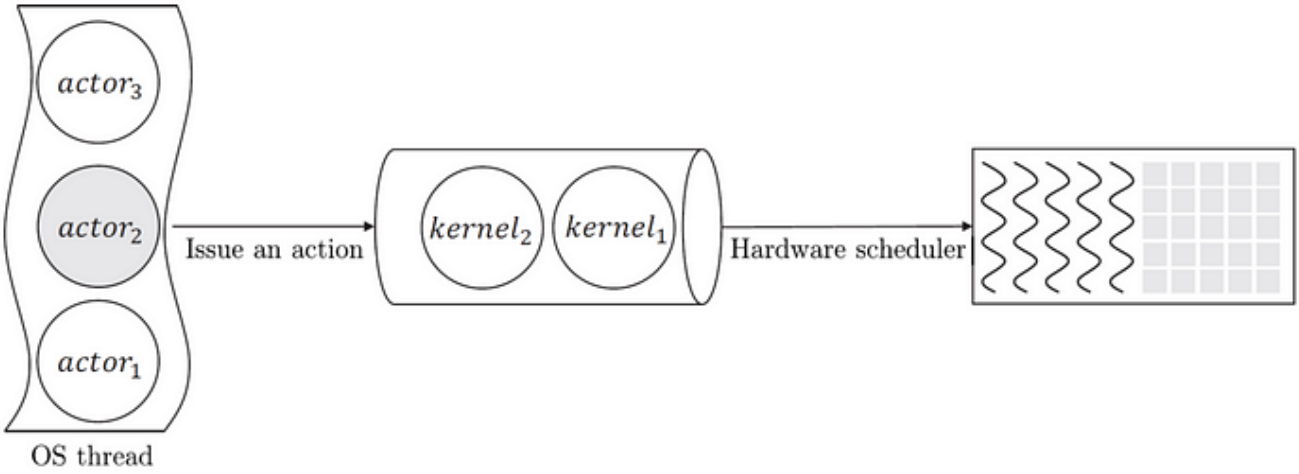


Figure 2: Relationship between ops, task queues and hardware resources

As shown in Figure 2, in OneFlow, each actor manages one op, and one OS thread may be responsible for scheduling and executing multiple ops. The scheduler puts the kernel corresponding to the op that meets the execution conditions into a queue (CUDA stream), and the hardware scheduler executes the kernel in the queue according to the FIFO rules.

The mapping problem from ops to hardware resources that the scheduler has to solve can be decomposed into three subproblems: mapping between ops and threads, mapping between threads and streams, and mapping between streams and hardware resources. Next, we will solve them one by one in the order of hardware resources, streams, threads, and ops.

How Many Streams Should a GPU Create?

Stream is an asynchronous task queue in which tasks are executed in a first-in-first-out (FIFO) order. Based on this fact, some inferences can be drawn.

1. For two hardware resources that can run independently, if they share a stream, the two hardware resources can only execute sequentially, not in parallel. For example, the DMA engine of Nvidia GPGPU (copy host to device, copy device to host, etc.) and the compute core are two different hardware resources. Suppose the transfer and computing share the same stream. In that case, it is impossible to overlap data copy and computation on this GPU, so the different underlying hardware resources should use different streams.

2. For two ops with no dependency and can be executed in parallel, if they are dispatched to the same stream, the two possible parallel ops will only be executed sequentially because they share the same stream. For example, there are 1024 cores on the GPU, if each op only needs to use less than half of the cores, and the two ops are dispatched to different streams, then the two ops can execute in parallel based on different cores, so it seems that the streams cannot be too small. But if each op can use all 1024 cores when executing, then even if they are dispatched to two different streams, the two ops can only be executed sequentially.

3. Suppose two ops with dependencies are dispatched to two different streams. In that case, the producer op in one stream must be executed before the consumer op can be dispatched to the second stream. This means that synchronization (or waiting) is required between different streams, and coordinating multiple streams is usually complicated. If both ops are dispatched to the same stream, then the consumer op does not need to wait for the producer op to finish executing before it can be dispatched to the stream. This is because the FIFO feature of the stream ensures that the second op will start executing only after the first op finishes executing. Using one stream can simplify the complexity of stream management.

To sum up:

- **Different hardware resources should use different streams;**
- **Ops with dependencies are best dispatched to the same stream, and ops without dependencies are best dispatched to different streams;**

- **There are half the pros and cons of creating multiple streams from the same hardware resource, as is creating only one stream from the same hardware resource. So what to do?**

An aggressive strategy is to set the number of compute streams based on the maximum parallelism of the computation graph on that device; a conservative strategy is to create only one compute stream per device, and all the ops on this device occur in sequence in the stream.

For common training tasks, the granularity of op is usually large, and one compute stream per GPU is generally enough. For inference, the batch size is relatively small. If only one stream is created on high-end graphics cards, one op cannot use up the graphics card. It is better to create multiple streams and let multiple ops execute simultaneously. There is another way to solve this problem through the compiler. Rammer (nnfusion at GitHub), presented by MSRA at OSDI 2020, proposes a clever solution to fuse the fine-grained op of inference into larger op to fill up the GPU resources.

Threading Model: How Many Threads Does A Stream Need?

Next, let's study the role of the OS thread (if not otherwise specified, hereinafter referred to as thread). A thread may need to perform two operations, i.e., scheduling and launching.

The scheduling operation is to thread manage (update and check) the dependencies of ops in the computation graph and put the ops whose dependencies have been resolved in the ready set.

The launching operation is to launch the ops in the computation graph. Given that the ops in the graph can be either computation or data movement, the execution of an op may take a while, and the actual operation is usually offloaded to accelerators such as GPGPUs or other worker threads.

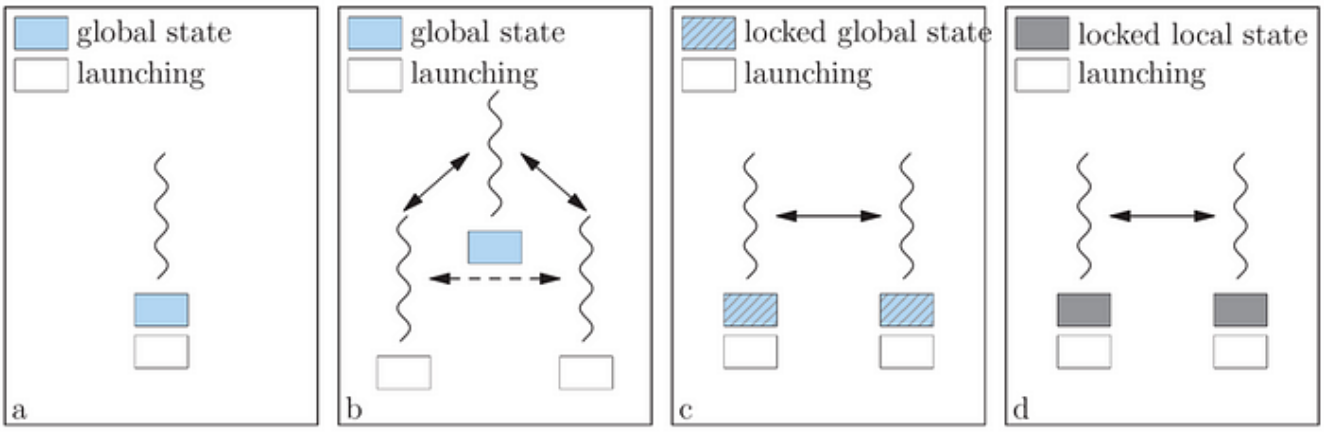


Figure 3: Four threading models

Figure 3 demonstrates four threading models for performing **scheduling** and **launching**.

1. Using a single thread to perform scheduling and launching concurrently. The thread accesses and modifies the state of the calculation graph, but the ops may be dispatched to different devices. It should be noted that when dispatching tasks to the device, the device context needs to be switched accordingly, which will bring a certain amount of overhead.

2. Dividing the operations of scheduling and launching. A specific thread manages the computation graph, and this thread doesn't launch CUDA kernels directly. Instead, launching operations are dispatched to some worker threads, each of which serves a particular device. In this case, the worker thread does not need to pay for the overhead of device context switching. Moreover, the calculation graph is only accessed by the scheduling thread, and there is no need to consider concurrent reading and writing by multiple threads. However, the update of the operator's state must be completed by taking the scheduling thread as an intermediate proxy. For example, as shown in case 2, the producer op and the consumer op are on different worker threads. After the producer op is executed, the scheduling thread will be notified to update the calculation graph. The worker thread where the consumer op locates will not be effected until the computation graph is updated. If the producer op and the consumer op can talk directly, the overhead of going through the scheduling thread as an intermediate proxy can be avoided.

3. To eliminate the proxy overhead in case 2, a thread is allowed to perform scheduling and launching of the ops on a particular device, which avoids device context switching. Each thread can access and modify the state of the calculation graph, and the ops on different threads with upstream and downstream production and consumption relations can directly talk to each other. This method eliminates the extra overhead of taking the scheduling thread between the producer and the consumer ops as an intermediate proxy. However, the global state must be protected with locks to minimize the contention overhead caused by concurrently accessing the shared state from multiple worker threads. However, a coarse-grained lock can severely hurt the overall performance.

4. To minimize the contention overhead in case 3, observing that the states of the ops can be decoupled by op, fine-grained locks can be used (e.g., creating a dedicated lock for protecting the state of each op). However, it is tricky and error-prone while optimizing the program by reducing the scope of a lock in the general concurrent systems.

In short, the choice of thread model needs to consider the factors such as the overhead of device context switching, the overhead of state interaction between the producer and the consumer ops, and the contention overhead caused by concurrently accessing the shared state from multiple worker threads. A good practice is that each thread only serves a fixed device or stream, inlining the launching to the thread of scheduling and minimizing the scope of the lock.

So far, we observe that a thread is best to serve only one stream, and not to introduce intermediary or proxy threads as far as possible. However, the question of how many threads does a stream should create hasn't been solved.

Optimal Number of Threads: How Many Operators Does A Thread Serve?

Now, Let's return to the question of "how many threads should be set up" discussed at the beginning of the article. This question is equivalent to "how many operators do a thread serve", which relates to the mapping relationship between operators and threads. The mapping relationship can be statically planned or dynamically determined at runtime.

Obviously, making the number of threads greater than or equal to the parallelism of the calculation graph and dispatching the operators to these threads in a round-robin way, in that way, even if one operator blocks one thread, another operator will not be affected.

This is equivalent to a kind of "saturation rescue": at any period, each thread only serves one operator, and only after one operator has been served can another operator be served. However, the number of operators in deep learning training is usually huge, thus a great number of threads should be created.

If the number of threads is less than the parallelism of the calculation graph, and there is no guarantee that the operator will not block, the risk of deadlock cannot be avoided.

However, setting the number of threads to the maximum demand may be a waste. If there is only one stream, the operators will be executed in sequence, no matter how much parallelism the calculation graph has. Setting up multiple threads is to improve stability, and will not improve efficiency. In other words, the extra threads are redundant (the creation and management overhead of kernel threads, kernel context switching overhead, and contention overhead caused by concurrently accessing cannot be ignored).

Making the number of threads equal to the number of streams is the best choice if not taking the stability into consideration. But is there a way to make the number of threads equal to the number of streams without affecting stability?

The key to the question is whether to allow the operation of scheduling and launching to block the thread where it is located. If it is ensured that scheduling and launching will not block the thread, then it is safe to make one thread serve multiple operators.

Asynchronous Execution, State Saving and Recovery

In the original DL frameworks, many ops are offloaded to the gas pedal for asynchronous execution, and the ops execute on the CPU can be delegated to another worker thread for simulated asynchronous execution.

Each op in the calculation graph may have multiple dependencies, and each event may only affect one dependency. The op cannot executes if the remaining dependencies have not been resolved. If the scheduler makes this op wait for the completion of other dependencies and blocks the current thread, the security problem of multiple ops sharing a thread cannot be solved.

Therefore, a completely asynchronous computation graph scheduling mechanism must be implemented: whenever a dependency condition of an operator is resolved, the scheduler checks whether the operator is ready. If not, save the current state of the operator and release thread resources to handle other operators; when the remaining dependency conditions are resolved, restore the state of the operator and continue to check whether it is ready, if it is, the operator can be launched asynchronously.

The operator will not occupy the thread for too long regardless of whether it is ready.

Readers familiar with the concept of user thread, such as goroutine in Golang and coroutine, may be familiar with the above mechanism, and their principles are very close. These principles are widely used in high-concurrency network service frameworks, such as Baidu's open-source framework bRPC and Tencent's open-source framework Flare.

But unfortunately, before OneFlow, no deep learning framework was aware of this problem and adopted a similar mechanism in the execution engine.

OneFlow achieves complete asynchronous scheduling and launching through the actor mechanism so that one thread can serve multiple operators. TensorFlow also noticed this problem, and in the new version of runtime they are developing, asynchrony is treated as a first-class citizen.

Summary

In order to make the mapping method from the op to the underlying hardware resources ensure efficiency, safety, and stability, we propose a static scheduling scheme to complete the mapping and binding from the device, stream, thread to the op level by level.

The streams and threads are configured according to the actual available hardware resources, and the ops executed on the same stream are bound to the same thread. The number of threads is a fixed value derived independent of the parallelism of the computation graph, there are no redundant threads, and the runtime decision mechanism is minimal.

The key to the feasibility of this static mapping is to make both computation graph scheduling and op execution fully asynchronous, particularly to support the preservation and recovery of op state contexts.

The original framework mainly uses dynamic scheduling, which is theoretically very flexible and does not require such an in-depth analysis of resource usage as static scheduling in advance. However, the original framework either suffers from stability defects or cannot fully utilize hardware resources, and the system implementation is very complex to overcome these problems.

Related articles:

1. ***The Limitations of Existing Deep Learning Frameworks: Resource Dependency***
2. ***The Limitations of Existing Deep Learning Frameworks: Data Movement***

Welcome to visit OneFlow on ***GitHub*** *and follow us on* ***Twitter*** *and* ***LinkedIn****.*

Also, welcome to join our ***Discord group*** to discuss and ask OneFlow related questions, and connect with OneFlow contributors and users all around the world.

Oneflow Deep Machine Learning AI TensorFlow