Whale: Scaling Deep Learning Model Training to the Trillions

Xianyan Jia, Le Jiang, Ang Wang, Jie Zhang, Xinyuan Li, Wencong Xiao, Langshi chen, Yong Li,
Zhen Zheng, Xiaoyong Liu, Wei Lin
Alibaba Group
{xianyan.xianyanjia,jiangle.jl,wangang.wa,wanglin.zj,lxy268263,wencong.xwc}@alibaba-inc.com
{langshi.cls,jiufeng.ly,james.zz,xiaoyong.liu,weilin.lw}@alibaba-inc.com

ABSTRACT

Scaling up deep neural networks has been proven effective in improving model quality, while it also brings ever-growing training challenges. This paper presents Whale, an automatic and hardware-aware distributed training framework for giant models. Whale generalizes the expression of parallelism with four primitives, which can define various parallel strategies, as well as flexible hybrid strategies including combination and nesting patterns. It allows users to build models at an arbitrary scale by adding a few annotations and automatically transforms the local model to a distributed implementation. Moreover, Whale is hardware-aware and highly efficient even when training on GPUs of mixed types, which meets the growing demand of heterogeneous training in industrial clusters. Whale sets a milestone for training the largest multimodal pretrained model M6. The success of M6 is achieved by Whale's design to decouple algorithm modeling from system implementations, i.e., algorithm developers can focus on model innovation, since it takes only three lines of code to scale the M6 model to trillions of parameters on a cluster of 480 GPUs.

1 INTRODUCTION

Training large-scale deep learning(DL) models is widely adopted in many fields, including computer vision[12, 24], natural language understanding[8, 30, 38, 39], machine translation[14, 21], and so forth. The scale of model parameters increases from millions to trillions, which significantly profits the model quality [8, 19], meanwhile it also brings challenges such as a high computation cost and the complexity of system engineering. Existing parallel strategies to accelerate the training include two broad classes, data parallelism(*DP*) and model parallelism(*MP*)[20]. *DP* parallelizes the training in data dimension with a parameter synchronization at each training step. While *MP* parallelizes the training in model dimension with activation communication among devices as required.

When training large models, applying a single parallel strategy to the whole model can be suboptimal. Considering a large-scale image classification task with 100,000 classes, the model is composed of $ResNet50$[16] for feature extraction and Fully-Connected(*FC*) layers for classification. The parameter size of $ResNet50$ is 90 MB, while the parameter size of *FC* is 782 MB. If we apply *DP* to the whole model, the parameter synchronization of *FC* will become the bottleneck. One solution[20] is to apply *DP* to $ResNet50$ and apply *MP* to *FC*, as illustrated in Figure 4. In this way, the communication overhead can be reduced by 90%, since the *FC* layer is updated locally. This example demonstrates the significant performance benefit from the hybrid parallel strategy, which applies different parallel strategies to different submodules of the model.

Apart from complicated parallel strategies, training giant models requests tremendous computing resources. In industry, scheduling hundreds of homogeneous high-end GPUs usually takes a long queuing time. On the other hand, it is much easier to obtain heterogeneous GPUs (e.g., a mixture of P100[2] and V100[3]) due to the fragmentation of the GPU cluster[18, 41]. Nevertheless, training with heterogeneous GPUs is even more difficult, as we need a consideration on both computing units and memory capacity of GPUs when building the model. For instance, we consider applying *DP* to model training on both P100 and V100. The model computation (i.e., forward and backward) on V100 is faster than that of on P100, because V100 has a higher computing capability than P100. As *DP* requires a parameter synchronization at the end of each step, the model replica on faster GPU must wait for the slower ones, which results in a low utilization of high-end GPUs. In addition, dispensing workload evenly on P100 and V100 leads to a waste of V100's device memory because of its large capacity. Furthermore, due to a dynamic scheduling of GPUs, users are unaware of the hardware specification when building their models, which brings a gap between model development and hardware environment.

In addition, giant model training is severely limited by the system programming expertise. Applying *DP* to distributed model training usually requires only a few lines of code change to the local model. However, the performance improvement of *MP* is usually attained by substantial engineering efforts. Model developers sometimes need to refactor their implementations completely, including subtly program distributed operators, manually handle communication and synchronization, and wisely place neural network model computation among devices. Additionally, implementing hybrid strategies needs to take care of the compatibility among different strategies. Furthermore, the missing runtime hardware information when programming neural network models introduces challenges on an efficient model training. In a word, the challenges of implementing complicated parallel strategies on heterogeneous devices impose new requirements of programming giant models.

Many attempts have been made to support distributed model training from different aspects. Mesh-TensorFlow[36] designs a language for distributed training. However, it requires users to rewrite the whole model and thus brings a migration overhead. DeepSpeed[6] enables the giant model training by combining ZeRO-powered[31] data parallelism with NVIDIA Megatron-LM[38] model parallelism. Whereas, their parallelism implementation is closely coupled with a specific model and cannot be easily adapted to other models. GShard[21] provides several parallel annotation APIs to support different strategies and follows the SPMD (single program multiple data) programming paradigm. However, it cannot support uneven model partitioning or device assignment. From the above

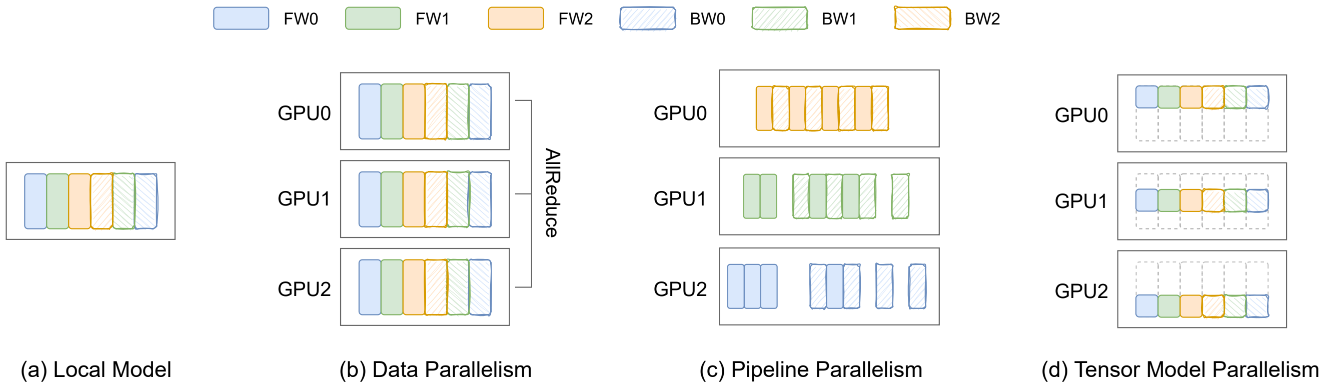


Figure 1: Parallel strategies

discussion, we conclude that supporting flexible and diverse parallel strategies in one general framework remains an important yet challenging topic. Moreover, none of the existing frameworks supports training with heterogeneous GPUs. [6, 21, 36, 38] assume that the training devices are homogeneous and require users to annotate or implement the distributed model statically. The design of existing frameworks cannot handle the dynamic-scheduled and heterogeneous GPUs within a production cluster.

In this paper, we propose Whale, an automatic and hardware-aware deep learning framework for training giant models. Whale preserves the benefits of computation graph, and it introduces a unified abstraction as an enhancement for distributed computation at scale, i.e., allowing flexible combination and nesting of submodules. After a thorough evaluation of existing parallel strategies and model structures, Whale proposes four primitives that can build up all existing parallel strategies. Users only need to annotate the local model with a few primitives to achieve a distributed giant model training. Whale hides the complexity of system implementation details, such as rewriting distributed operations, scheduling parallel executions on multiple devices, and balancing computation workload among heterogeneous devices. Given partial parallel annotations, Whale can automatically infer the undefined behaviors such as device placement within distributed computation graph, tensor partition, bridging submodules with auxiliary operations, and so forth. Furthermore, Whale introduces a hardware-aware load balancing algorithm, which bridges the gap between model development and runtime environment.

We summarize the key contributions of Whale as follows:

(1) By abstracting distributed parallel strategies, Whale defines four parallel primitives. The primitives can be used to express all existing parallel strategies as well as flexible hybrid strategies, and they decouple the model design from low-level parallel implementation.
(2) With a few annotations, Whale automatically converts a local model to a distributed one. Whale can infer the parallelization of each operation smartly.
(3) Whale proposes a hardware-ware load balancing algorithm, which is seamlessly integrated with parallel strategies to accelerate distributed training in heterogeneous GPU clusters.
(4) Whale sets a milestone for training the largest multimodel pretrained model M6[23], which takes only three lines of code to scale the parameters up to one trillion on 480 NVIDIA V100M32 GPUs.

Whale is now deployed as a production system to serve scalable deep learning training at Alibaba. The industry practice of Whale demonstrates not only the feasibility, but also the great potential, to strike a balance between programming and efficiency in giant deep learning model training. By extending the expressiveness of TensorFlow [7], Whale incorporates various parallelism strategies at scale. The training of M6-10B by using a hybrid parallelism (data parallelism + pipeline[17] parallelism) on 256 NVIDIA V100M32 GPUs achieved 91% scalability. Whale further demonstrates its adaptability in heterogeneous clusters. Evaluations on training various models over heterogeneous GPUs, including Bert-Large[10], Resnet50[16], and GNMT[42], show performance improvements by 1.2x to 1.4x thanks to the hardware-aware load balancing algorithm in Whale.

2 BACKGROUND AND MOTIVATION

In this section, we first recap the background of parallel strategy in deep learning model training. We then present the importance as well as the challenge in utilizing heterogeneous GPU resource. Finally, we discuss opportunities of designing a training framework.

2.1 Parallel Strategies

To scale out the training of deep learning models on multiple devices, two major paradigms, namely data parallelism (*DP*) and model parallelism(*MP*), are proposed. As we have briefly introduced them in Section 1, *DP* and *MP* parallelize the model from data and model dimensions. *MP* can be further divided into layer-wise pipeline parallelism and tensor model parallelism. Figure 1 illustrates three parallel strategies with a simplified model. Noting that the box filled with solid color represents the forward(FW) layer, while the box filled with slash represents the backward(BW) layer.

Data Parallelism speeds up the training by computing on partitioned data in parallel. As shown in Figure 1(b), each worker maintains a model replica with different mini-batches as input. In each training iteration, every model replica performs forward and backward computation independently and produces local gradients.

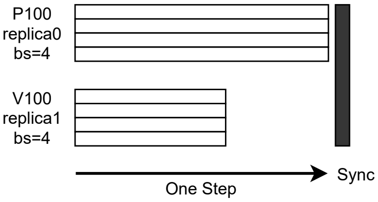


Figure 2: Data parallelism with P100 and V100. Model replica1 in V100 computes faster than replica0. But replica1 must wait until the completion of a global synchronization.

The gradients are then synchronized by a collective communication operation such as AllReduce[35].

Pipeline Parallelism partitions the model into several stages across layers and schedules the execution of stages in an interleaved pipeline. For example, as shown in Figure 1(c), it partitions the model into three stages and places one stage on one GPU. A mini batch is split into four micro batches, and the forward and backward executions of different micro batches are pipelined to mitigate the idle time (pipeline bubble). [13, 17, 27, 46] propose different pipeline algorithms to improve the scheduling efficiency.

Tensor Model Parallelism partitions the computation within a layer into submodules and places them across devices, as shown in Figure 1(d). Unlike pipeline parallelism that applies cross-layer partitioning, tensor model parallelism shards the tensor in parallel and adds communication operations such as AllGather to maintain a mathematical equality. Recent works[14, 21, 38] show the importance of tensor model parallelism in giant model training.

Applying a single parallelism to the whole model is straightforward but can be inefficient in training performance. Given a transformer[39] model with pipeline parallelism as an example, the model cannot be partitioned to an arbitrarily large number of devices, which results in low device utilization. We can improve it by applying hybrid parallelism, such as combining pipeline parallelism with data parallelism. We first find a proper degree of parallelism for *pipeline* and then scale the whole *pipeline* with *DP*. Many efforts adopt hybrid parallelism to scale the giant model training to a trillion parameters. For example, [28] combines data parallelism, pipeline parallelism, and tensor model parallelism to train Megatron-LM[38] model up to one trillion of parameters. [14, 21, 23] focus on large-scale sparse expert models with a high training efficiency by combining data parallelism and tensor model parallelism. In short, hybrid parallel strategies play an important role in large model training.

2.2 Heterogeneous Resource Utilization

Training a giant model is very resource intensive. [28] uses 3072 NVIDIA A100-80GB GPUs to train a GPT3 model with one trillion of parameters. [14] takes 2048 TPUs to train a T5-XXL model. We must resolve the issue of getting enough computing resources to enable the giant model training.

In industry, deep learning tasks run in a shared cluster. As reported in [41], there are many available GPUs in off-peak hours, but with a mixture of GPU types such as NVIDIA-P100 and NVIDIA-V100. Requesting large amounts of homogeneous GPUs takes a long queuing time, which blocks the progress of model experiments. Due to the fragmental usage of GPUs and large amounts of short-running tasks, it is much easier to get heterogeneous GPUs with mixed GPU types.

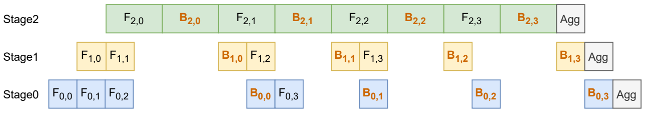
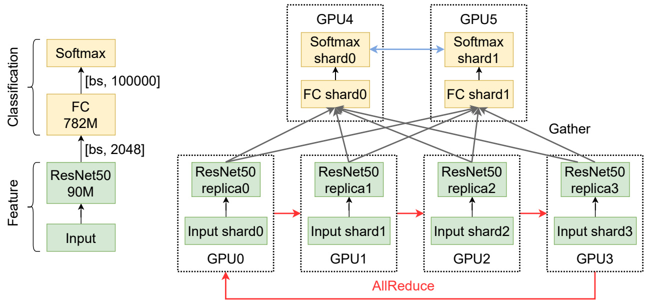


Figure 3: Pipeline parallelism with three stages. $F_{s,m}$ is the forward of stage s, and micro batch m, $B_{s,m}$ is the backward of stage s and micro batch m. The stage2 is two times slower than stage0/stage1. The slow stage2 results in long idle time for stage0/stage1.

However, training with heterogeneous GPUs is more challenging in fully saturating the diverse GPU capabilities, i.e., balancing computation workload among different types of GPU. Current strategies applied in homogeneous cluster no longer works well for heterogeneous training. For example, *DP* requires a global synchronous operation to aggregate the parameters, which means all workers have to wait for the slowest worker to finish the computing. Figure 2 is an illustration of data parallelism trained with both P100 and V100. Noting that replica1 runs faster than replica0, but replica1 has to wait for replica0 to finish one step, which results in low device utilization of V100. In addition, there are execution dependencies among stages in pipeline parallelism. One stage has to wait for the tensor from the previous or next stage. For instance, as shown in Figure 3, the forward of micro-batch m and stage $s(F_{s,m})$ has to wait for the output tensor from $F_{s-1,m}$. The backward of micro-batch m and stage $s(B_{s,m})$ has to wait for the gradient tensor from $B_{s+1,m}$. If one stage is much slower than other stages, it will result in long idle time waiting for the slow stage to complete. In the above example, stage2 is two times slower than stage0 and stage1. Stage0 and stage1 take much time waiting for the gradients from stage2, which results in low utilization of stage0 and stage1. The above analysis shows the limitation of current parallel strategies with mixed types of GPUs, which propose new requirements for the DL training framework in heterogeneous clusters.

2.3 Opportunities in DL Training Framework

Despite the benefits of hybrid parallel strategies as we have discussed in Section 2.1, implementing hybrid strategies requires strong system experience and engineering effort, such as distributed operation rewriting, resolving tensor shape inconsistency when combining different strategies, placing device wisely, balancing computing loads, etc. For example, Figure 4(a) shows an image classification model that consists of feature part and classification part. We apply data parallelism to the feature part with four GPUs and apply tensor model parallelism to the classification part with two GPUs. Figure 4(b) shows the corresponding distributed model. For the feature part, users need to first replicate it onto four GPUs and then insert an AllReduce communicator to synchronize the gradients. For the classification part, users need to implement a distributed *FC* layer, a $Softmax$ layer, and insert a communication



(a) Local Model (b) Distributed Model

Figure 4: Image classification model with hybrid parallel strategies. Apply data parallelism for the feature extraction part(green) with four GPUs, and tensor model parallelism for the classification part(yellow)

operator to make sure the distributed layer is mathematically correct. In addition, since *DP* shards the input batch into four parts, the output of *DP* should be concatenated into one tensor as the input to classification part. Moreover, users should assign the device properly to each part of the model for distributed training. In a more complicated scenario, when training with heterogeneous GPUs, default settings such as using the same batch size for all model replicas in data parallelism, or partitioning model stages evenly in pipeline parallelism, result in low training efficiency.

To reduce the programming burden of distributed models and make large-scale model training accessible to more users, a general training framework that provides an easy-to-use parallel programming API and comprehensive parallel strategies is desired. Existing frameworks have made many efforts in supporting diverse parallel strategies[6, 17, 21, 36]. GPipe[17] is a library dedicated to pipeline parallelism. DeepSpeed[33] provides optimized data parallelism and pipeline parallelism. It also includes Megatron-LM[38] as a distributed transformer model example. However, Megatron-LM is not designed as a framework but made targeted inplace modifications on existing transformer implementations. Mesh-TensorFlow[36] designs a new language that allows users to achieve distributed training with tensor model parallelism and data parallelism. However, Mesh-TensorFlow requires users to rewrite the whole model with its new programming paradigm, which results in model migration overhead. GShard[21] allows users to implement distributed models by adding annotations, and it supports the combination of data parallelism and tensor model parallelism. GShard follows SPMD (single program multiple data) paradigm, where the same program is launched on all devices for a parallel execution. Regardless of the simplicity of SPMD, it cannot express uneven assignments of devices for different model parts, as shown in Figure 4.

The above frameworks are built upon the assumption of using homogeneous devices. Gshard[21] distributes one program for all devices, which cannot fully utilize the heterogeneous GPUs with different capacities. In addition, the inplace parallelism implementation[33, 38] cannot adapt to dynamically allocated devices. Due to the dynamic runtime devices, a hardware-aware framework is needed to fill the gap between model programming and runtime environment. To achieve this, we need to consider three factors: (1) GPU characteristics such as computing and memory capacity, (2) model structure, and (3) parallel strategy. Ideally, the computing workload in each device is proportional to its computing capacity. In practice, we should ensure that the workload on each device will not run out of memory. The above hardware-aware load balancing process should be performed automatically at runtime.

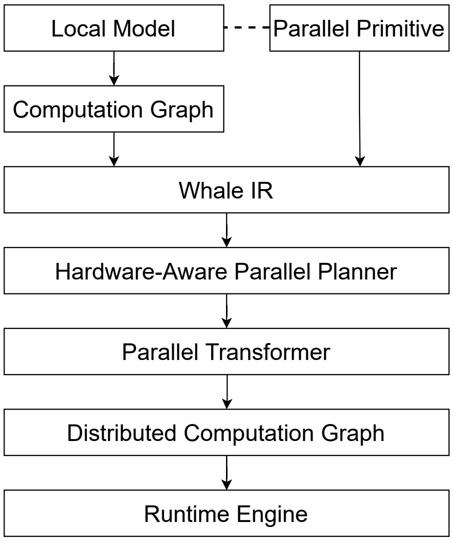


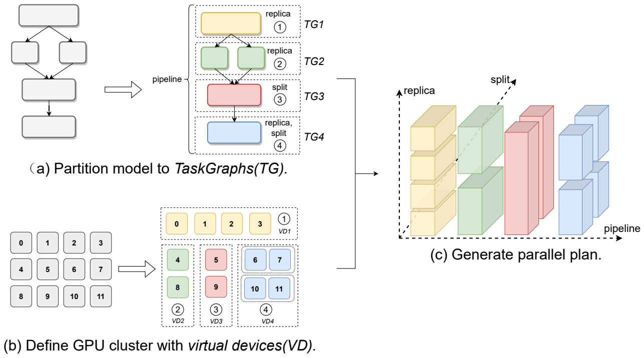
Figure 5: System overview of Whale

3 DESIGN

In this section, we present Whale by first describing the system overview. Then we introduce two key abstractions. After that, we define parallel primitives and explain them with programming examples. Next, we describe the parallel planner that automatically transforms parallel primitives into an execution plan. In the end, we propose a hardware-aware load balance algorithm to speed up the training with heterogeneous GPU clusters.

3.1 Overview

We introduce the overview of Whale by walking through its components as shown in Figure 5. The system takes a local model as input, which is represented as a computation graph. Users can add a few parallel primitive annotations in the local model to suggest the parallel strategy (Section 3.3). The original computation graph is augmented with parallel information, which is represented as Whale IR(intermediate representation). The parallel information includes parallel strategy, operation phase (forward, backward, optimizer update, etc.), model profiling information such as computation and memory cost, hardware information, and so forth. A parallel planner (Section 3.4) generates an efficient parallel execution plan by inferring the strategies for the whole graph. After that, a parallel transformer rewrites the computation graph and generates a distributed computation graph (Section 4). Finally, a runtime engine(e.g., TensorFlow[7] runtime) compiles the distributed



(a) Partition model to *TaskGraphs(TG)*.

(b) Define GPU cluster with *virtual devices(VD)*.

(c) Generate parallel plan.

Figure 6: Whale parallelism workflow.

computation graph to execute. Whale generalizes the distributed training based on computation graph, which can be directly applied to TensorFlow models. In addition, Whale can be easily extended to support other deep learning frameworks such as PyTorch[29], whose model can be transformed into a computation graph.

3.2 Abstraction

We observe that different parallel strategies are implemented in different frameworks. For example, [7, 22] provides data parallelism; [13, 17, 33, 46] implement data parallelism and pipeline parallelism; [21, 36] support data parallelism and tensor model parallelism. However, none of the above frameworks can achieve all the parallel strategies. It is even harder to express a flexible hybrid parallel strategy upon nesting or a combination of strategies. A straightforward way to express hybrid parallel strategy is to apply different parallel strategies to different operations. However, the operation-wise strategy results in a combinatorial explosion, as there are tens of thousands of operations in a model. Instead, Whale modularizes the computation graph into smaller none-overlapped subgraphs, and it applies different parallel strategies to different subgraphs. Formally, we define a modularized subgraph as a *TaskGraph*.

TaskGraph (*TG*) is a subset of the computation graph. It is a basic unit for parallel annotation and execution. Each *TaskGraph* can be annotated with one or more parallel strategies. As the example shown in Figure 6, the original computation graph is partitioned into four *TaskGraph*s.

In addition, placing model operations to physical devices is challenging for complicated hybrid parallel strategies. A user must understand the mapping relationship between the model and physical devices. What's more, it is even more challenging when the model is computed with heterogeneous GPUs. To hide the complexity of physical device mapping from end users, Whale proposes *virtual device* as follows.

Virtual Device (*VD*) is the abstraction of the computing resource for *TaskGraph*. One *virtual device* contains one or more physical devices. One *virtual device* is assigned to one *TaskGraph*. For example, as shown in Figure 6(b), four *virtual devices* are defined.

3.3 Parallel Primitive

End-to-end automatically parallel training is still a hard problem as the system needs to consider both model and runtime hardware configuration. Whale allows users to suggest parallel strategies with a few primitives. In this way, the system can easily incorporate the experience from experts to generate a parallel strategy. There are three goals in designing the parallel primitive: (1) It provides a consistent view for the local model and distributed model. (2) It is lightweight, i.e., users only need to add a few lines in the local model to achieve a distributed model. (3) It should be flexible enough to express all existing parallel strategies, as well as the hybrid of them.

After thoroughly evaluating existing parallel strategies, such as data parallelism, pipeline parallelism[13, 17, 27], and tensor model parallelism[20, 21, 38], Whale proposes four intuitive parallel primitives: *stage*, *replica*, *split*, and *pipeline*. The parallel primitive is a context manager similar to TensorFlow scope[1], where operations defined under parallel primitive are regarded as one *TaskGraph*, with corresponding annotated parallel strategy. The four parallel primitives are defined as follows.

replica annotates *TaskGraph* to be replicated. *replica* can be used to express data parallelism.

split annotates *TaskGraph* to be sharded. *split* can be used to express tensor model parallelism.

stage annotates operations to be grouped as a *TaskGraph*. *stage* is used to partition the computation graph manually.

pipeline annotates *TaskGraph*s with *pipeline* parallelism. Users can manually partition the model into multiple *TaskGraph*s with *stage*. If no *stage* is provided under *pipeline* scope, Whale automatically partitions the operations based on a balanced strategy (introduced in Section 3.5).

With the above primitives, Whale can express parallel strategies with a few lines of annotation without model intrusion. The scope-style primitives can be used to represent hybrid strategies with both combination and nesting patterns. As shown in Figure 6(a), the computation graph is partitioned into four *TaskGraph*s, as identified by different colors and numbers. *TG1* and *TG2* are annotated with *replica*, *TG3* is annotated with *split*, *TG*4 is annotated with nested *replica* and *split*. All these *TaskGraphs* are annotated with *pipeline* in the outer scope.

Besides parallel strategy representation, we simplify the device placement by introducing *cluster*. ***cluster*** transforms physical devices to logical virtual devices. Each virtual device is annotated with worker and GPU information for a hardware-aware analysis(described in Section 3.5). In Figure 6(b), each row is one node that contains four GPUs. The cluster is split into four virtual devices. *TaskGraph* TG_i is computed with virtual device VD_i, i.e., TG_{1-4} are computed with 4,2,2,4 GPUs respectively.

Next, we use several code samples to illustrate different parallel strategies with Whale primitives.

Example 1: data parallelism

```
with wh.cluster():
  with wh.replica():
    out = Model()
```

[1] TensorFlow scope: https://www.tensorflow.org/api_docs/cc/class/tensorflow/scope

Example 1 shows a simple data parallelism, where models are replicated to all devices.

Example 2: Vanilla Model Parallel

```
with wh.cluster():
  with wh.stage():
    x = ModelPart1(x)
  with wh.stage():
    x = ModelPart2(x)
```

Example 2 shows vanilla model parallelism with two *TaskGraph*s.

Example 3: Hybrid of pipeline parallelism and data parallelism

```
with wh.cluster():
  with wh.replica():
    with wh.pipeline(num_micro_batch=4):
      with wh.stage():
        x = ModelPart1(x)
      with wh.stage():
        out = ModelPart2(x)
```

Example 3 shows a hybrid parallelism that nests pipeline parallelism with data parallelism. In this example, the model is first partitioned into two *TaskGraph*s with *stage*. Then we apply pipeline parallelism with 4 micro-batches. In the end, the whole *pipeline* is replicated to perform the data parallelism in the outer scope.

Example 4: Hybrid of auto pipeline parallelism and data parallelism

```
with wh.cluster():
  with wh.replica():
    with wh.pipeline(num_micro_batch=4):
      out = Model(x)
```

Example 4 shows a hybrid strategy similar to Example 3. The difference lies in the graph partition method. Instead of manually annotating *TaskGraph*s with *stage* API, this example automatically partitions the pipeline stages. The stage number is set to the number of virtual devices.

Example 5: Hybrid of data parallelism and tensor model parallelism

```
with wh.cluster():
  with wh.replica():
    with wh.replica():
      features = ResNet50(inputs)
    with wh.split():
      logits = FC(features)
      predictions = Softmax(logits)
```

Example 5 shows a hybrid strategy that first applies data parallelism and tensor model parallelism to two *TaskGraph*s respectively. It then applies a nested data parallelism for the above combination.

Example 6: Automatic parallelism

```
wh.auto_parallel()
out = Model(x)
```

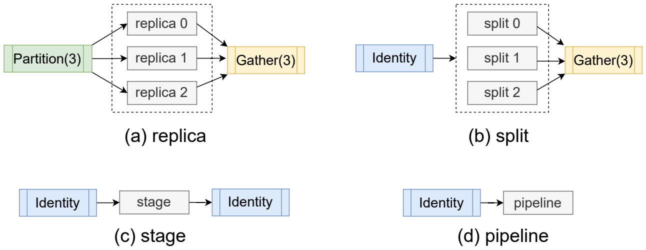


Figure 7: Bridge patterns.

Whale also supports automatic parallelism by adding one line of code, as shown in Example 6. In this mode, Whale explores the parallel strategies automatically without any user prompts.

3.4 Parallel Planner

After getting the local model with parallel annotations, a parallel planner is responsible for producing an efficient parallel plan. The parallel plan abstracts the parallel process as follows: (1) *TaskGraph* partition. Recall that *TaskGraph*s are partitioned either by the primitive *stage* or by the system automatically. When the *TaskGraph*s are partitioned automatically, Whale applies a hardware-aware load balancing algorithm to find an efficient partition method. The detail of the load balancing algorithm is described in Section 3.5. (2) *TaskGraph* device mapping. As we have mentioned in Section 3.2, one *TaskGraph* maps to one virtual device. Whale determines the parallelism degree for each parallel strategy by the number of GPUs in the corresponding virtual device. We assume that a *TaskGraph* is mapped to a virtual device with *n* GPUs. If the *TaskGraph* is annotated with *replica*, it applies data parallelism with *n* *TaskGraph* replicas. If the *TaskGraph* is annotated with *split*, it is split into *n* submodules. Figure 6(c) demonstrates such a parallel plan. The three axes represent the degree of *replica*, *split*, and *pipeline* respectively. The first *TaskGraph* applies data parallelism over four GPUs. The second *TaskGraph* applies data parallelism over two GPUs. The third *TaskGraph* is sharded across two GPUs. The last *TaskGraph* is first sharded into two submodules and then replicated twice. In the end, we apply pipeline parallelism for the four *TaskGraph*s.

In the above example, we apply different parallel strategies to different *TaskGraph*s. However, the input/output tensor number and shape may change due to different parallelism degrees, which results in a mismatch of tensor shapes among *TaskGraph*s. For example, in Figure 6c, the output from the first *TaskGraph* cannot be used directly as an input to the second *TaskGraph*. If the global batch size is 32, the local batch size for each replica in *TaskGraph*1 is 8(four replicas), while the local batch size for *TaskGraph*2 is 16(two replicas). To address the aforementioned mismatch, Whale designs a *bridge layer* to connect *TaskGraph*s with different parallel strategies. We design three types of *bridge layer* as follows: (1) $Partition(n)$ partitions a tensor into *n* parts. (2) $Gather(n)$ gathers *n* tensors and concatenates them into one tensor. (3) *Identity* transfers the tensor as it is.

Since different parallel strategies process their input and output tensors differently, we build a *bridge pattern* for each parallel primitive as shown in Figure 7. (a) *replica*. Assume the replica

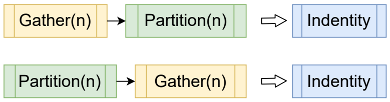


Figure 8: Fused bridge patterns.

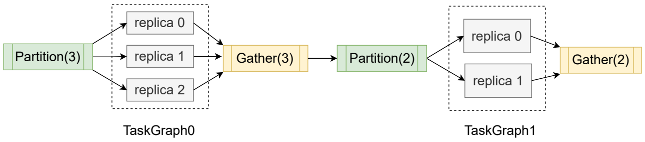


Figure 9: Bridge two *TaskGraphs*, *TaskGraph*0 applies DP with three replicas, *TaskGraph*1 applies DP with two replicas.

number is n, we first partition the input of *TaskGraph* into n parts with $Partition(n)$. Each part is consumed by one model replica. Then the outputs of model replicas are then concatenated with $Gather(n)$. (b) $split$. When the *TaskGraph* is annotated with $split$, the input to the *TaskGraph* is used as it is, while the outputs are concatenated with $Gather(n)$. (c) $stage$. Since we do not change the input and output of $stage$, $stage$ is simply connected by $Identity$. (d) $pipeline$. If a *TaskGraph* is annotated with $pipeline$, we do not change the input of *TaskGraph*. Since we do not allow connecting *TaskGraphs* after $pipeline$, $pipeline$ pattern only has input bridges. Whale automatically inserts the *bridge layers* for each *TaskGraph*. Some bridge layers have opposite functions and result in unnecessary communication. To reduce communication overhead, Whale detects and fuses opposite *bridge patterns*. For example, as shown in Figure 8, $Gather(n)$ is opposite to $Partition(n)$, and they are fused as $Identity$ operation.

Figure 9 shows an example to connect two *TaskGraphs*. *TaskGraph*$_0$ has three model replicas, while *TaskGraph*1 has two replicas. The input of *TaskGraph*0 is partitioned into three parts according to the number of replicas. The outputs of *TaskGraph*0 are gathered as one tensor. Next, the concatenated output tensor is partitioned into two parts, which are consumed by *TaskGraph*1. In the end, the outputs of *TaskGraph*1 are gathered.

3.5 Hardware-aware Load Balance

As we discussed in Section 2.2, requesting a large number of heterogeneous GPUs is easier than homogeneous GPUs in the industrial cluster, while it brings low training efficiency due to unbalanced device workloads. To solve this problem, Whale proposes a Computation-balanced partition algorithm to balance the computing time among *TaskGraphs*.

Computation-balanced partition algorithm. Assuming that the single-precision FLOP (floating-point operations) of the model is MF, the single-precision FLOPS (FLOP per second) of the GPU is GF, and the GPU utilization is α, the computing time is roughly estimated as follows.

$$t = \alpha * MF/GF$$

To achieve similar computing time per machine, we assume that α is a constant and the model FLOP MF should be in proportional to

Algorithm 1: Peak Shaving and Valley Filling Algorithm

```
Function $PSVF(subgraphs, flop\_ratios, mem\_ratios,$
$GPUs, shift\_func)$
    while $flop\_ratios \neq \emptyset$ & $any(mem\_ratios > 1)$ do
        $p = argmax(mem\_ratios)$
        $flop\_ratios.pop(p)$
        $flop\_indices = sort(flop\_ratios.values())$
        foreach $\_, v \in flop\_indices$ do
            $shift\_func(subgraphs, p, v)$
            $mem_v = profile\_mem(subgraphs[v])$
            if $mem_v > GPUs[v].mem$ then
                $shift\_func(subgraphs, v, p)$
                $flop\_ratios.remove(v)$
            else
                $mem\_ratios[v] = \frac{mem_v}{GPUs[v].mem}$
                $mem\_ratios[p] = \frac{profile\_mem(subgraphs[p])}{GPUs[p].mem}$
                $flop_v = profile\_flop(subgraphs[v])$
                $flop\_ratios[v] = (\frac{flop_v}{GPUs[v].flops}, v)$
                $break$
```

the GPU device FLOPS GF. For data parallelism, we adjust the batch size of each replica according to their GPU FLOPS. For pipeline parallelism, the FLOP of each stage is partitioned proportional to GPU FLOPS.

However, the Computation-balanced partition algorithm may fail if the model runs out of memory. For example, if we train a model with a global batch size of 32 on one Tesla-P100(9.3 teraFLOPS/12 GB) and another Tesla-P40(12 teraFLOPS/24 GB). If we adopt the data parallelism with a computation-balanced partition algorithm, the batch size for the replica on Tesla-P100 is $9.3/(9.3 + 12) * 32 \simeq 14$, and the batch size for the replica on Tesla-P40 is set to 18. Here we assume that adding batch size by 1 increases 1 GB memory. If the replica on Tesla-P40 uses 20GB memory, the replica on Tesla-P100 runs out of memory as it needs 16GB memory. Thus, we must ensure that the partitioned model satisfies the memory constraint. In the above example, after partitioning the DP batch size, we find that there is still 4GB available memory remained on Tesla-P40, while Tesla-P100 requires another 4GB memory. If we move 4 samples from Tesla-P40 to Tesla-P100 in each iteration, we are able to train a global batch size of 32 on the two GPUs. In general, when encountering OOM (out-of-memory) in some GPUs, we can re-partition the model to alleviate memory-overload GPUs by allocating more computing to spare GPUs.

Based on the above discussion, Whale proposes a ***peak shaving and valley filling (PSVF)*** algorithm. The main idea of $PSVF$ is to shift the computing load from the memory-overload device to a memory-free device with the lowest FLOP load. $PSVF$ is an iterative process. In each step, $PSVF$ finds the peak device and valley device. The peak device is the device with the highest memory utilization

mem_ratio.

$$mem\_ratio = \frac{Model\ memory}{GPU\ memory}$$

The valley device is the device with the lowest FLOP utilization ($flop\_ratio$) and spare memory.

$$flop\_ratio = \frac{Model\ flop}{GPU\ FLOPS}$$

$PSVF$ shifts one unit of the workload away from the peak device into a valley device. This process iterates until the model is able to run with sufficient memory. Figure 10b shows an example that shaves the peak in GPU0 to fill the valley in GPU2. The pseudo-code of the $PSVF$ algorithm is shown in Algorithm1. The input *TaskGraph*s can be *TaskGraph* replicas or pipeline stages. $flop\_ratios$ is a map whose key is the *GPU* index, and the value is a tuple of FLOP utilization and *GPU* index. mem_ratios is a list of memory utilization of *TaskGraph* in the corresponding GPU. $shift\_func$ is the shift function that defines how to shave peak and fill the valley in each step. Different parallel strategies may need different $shift\_func$. In each $PSVF$ iteration, when OOM is detected, the peak device index p with maximum memory is found(line 4). Since the valley device is the device with the lowest $flop\_ratio$ and free memory, it is searched from a device with the lowest $flop\_ratio$ (line 6-7). For each peak index p and valley index v, the $shift\_func$ shifts one unit of computing from peak to valley(line 8). If OOM is detected after valley filling, this process would be reverted(line 9-12). Otherwise, the mem_ratio and $flop\_ratio$ for peak and valley are updated(line 14-18). The above process iterates until all devices avoiding OOM.

Next, we show how to combine the FLOP-balanced partition and $PSVF$ policy with data parallelism and pipeline parallelism. In general, Whale first applies FLOP-balanced partition strategy. If OOM is estimated in any device, $PSVF$ policy is applied to shift load from peak device to valley device.

Hardware-aware data parallelism partition. The pseudo-code of hardware-aware data parallelism partition algorithm is shown in Algorithm 2. Given a certain global batch size, Whale first splits the input batch of each replica in proportional to their GPU FLOPS as shown in line 6. The memory utilization for all devices is then profiled[15] as mem_ratios(line 7). Meanwhile, flop utilization is estimated as $flop\_ratios$(line 8). Figure 10a shows an example of DP with four GPUs. The blue line represents the mem_ratios for each GPU, and the orange line represents the $flop\_ratios$. As we can see, after a flop-balanced input batch split, the flop utilization is similar for all GPUs. However, the GPU0 and GPU1 run OOM in this case. Thus, we need to apply $PSVF$ policy(line 9-10) to solve the OOM problem. The shift function for data parallelism is defined as $shift\_batch$(line 11-13), where the batch size of replica i decreases by 1 while the batch size of replica j increases by 1. In this way, we do not change the global batch size. As the example shown in Figure 10, in step1, the peak device is GPU0, while the valley device is GPU2. Thus, we shift one batch from replica0 to replica2. The memory utilization curve changes from the dotted line to the solid line. Moving to step 2, the peak device is still GPU0, while the valley device is GPU3. We continue to shave the peak of GPU0 and shift the batch to GPU3. Finally, in step3, the peak device is GPU1, while the valley device is GPU2. After shifting one batch from GPU1 to GPU2, the memory utilization of all GPUs is within memory constraints.

Algorithm 2: Hardware-aware Data Parallel Partition

```
1  Function DP_Partition (replicas, global_bs, GPUs)
2      $flop\_ratios \leftarrow [], mem\_ratios \leftarrow \{\}$
3      $N \leftarrow len(GPUs)$
4      $total\_flops \leftarrow \sum_{i=0}^{N}$ GPUs[i].flops
5      foreach $i \in [0, N)$ do
6          $replicas[i].bs \leftarrow \frac{global\_bs * GPUs[i].flops}{total\_flops}$
7          $mem\_ratios.add(\frac{profile\_mem(replicas[i])}{GPUs[i].mem})$
8          $flop\_ratios[i] \leftarrow (\frac{profile\_flop(replicas[i])}{GPUs[i].flops}, i)$
9      if $any(mem\_ratios > 1)$ then
10         $PSVF(replicas, flop\_ratios, mem\_ratios, GPUs, shift\_batch)$
11 Function $shift\_batch(replicas, i, j)$
12     $replicas[i].bs \leftarrow replicas[i].bs - 1$
13     $replicas[j].bs \leftarrow replicas[j].bs + 1$
```

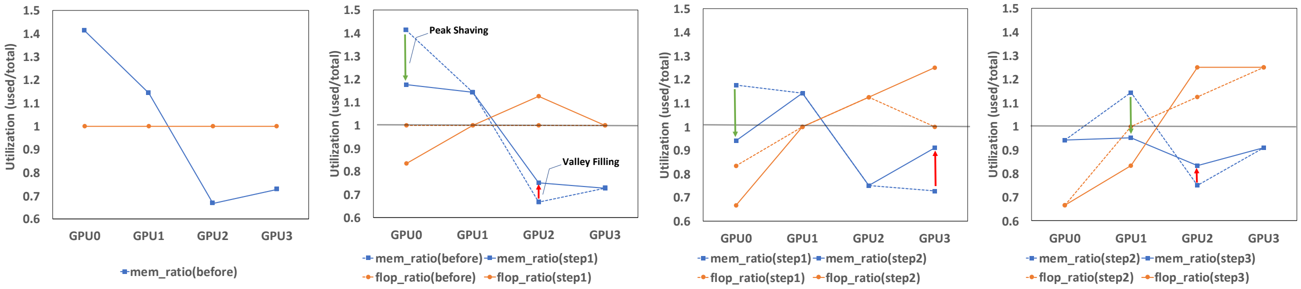
Hardware-aware pipeline parallel partition. In pipeline parallelism, we need to balance the computing time among stages. The pseudo-code of hardware-aware pipeline partition algorithm is shown in Algorithm 3. First, the graph is partitioned into stages, where the FLOP of each stage is proportional to its corresponding GPU FLOPS (line 3-8). Then, Whale find a maximum micro-batch size for each stage (line 9-10). For each stage, mem_ratios and $flop\_ratios$ are profiled. If OOM is detected, we will apply the $PSVF$ policy for the pipeline to adjust the stages. Different from data parallelism that shifts batch size, we shift one operation($shift\_op$) from peak stage to valley stage. In the example shown in Figure 11. Assuming that there are three stages, as we would like to shift one operation from stage0 to stage2, we first shift the last operation from stage0 to stage1, then we shift the last operation from stage1 to stage2. It is worth noting that the $shift\_op$ function does not change the topology order of the computation graph.

4 IMPLEMENTATION

Whale is implemented as a standalone Python library without modification of deep learning framework. Whale currently supports TensorFlow[7] v1.x and can be extended to other DL frameworks whose DL models can be transformed to a computation graph. Whale applies all parallel transformations by editing the computation graph. Next, we will elaborate on the implementation of several key modules.

TaskGraph Clone. *TaskGraph*s are cloned if *replica* or *pipeline* is defined. *replica* copies all operations and tensors defined in the original *TaskGraph*. Different from *replica*, *pipeline* shares variables (trainable parameters), datasets, optimizer-related operations among different micro batches, so we do not clone such elements when cloning *pipeline*.

TaskGraph Partition. Whale automatically partitions a graph into *TaskGraph*s when no *stage* is annotated. The partition number n is set to the number of GPUs of the corresponding virtual device. For *pipeline*, we first sort the operations in a topology order. The



(a) original **(b) step 1** **(c) step 2** **(d) step 3**

Figure 10: Hardware-aware data parallelism: Combine computation-balanced partition with peak shaving and valley filling policy.

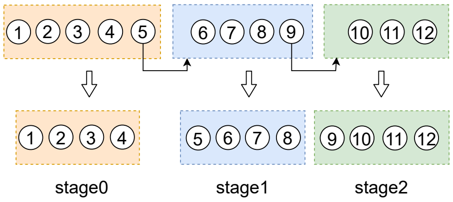


Figure 11: $shift\_op(stages, 0, 2)$ **shift one operation from stage0 to stage2.**

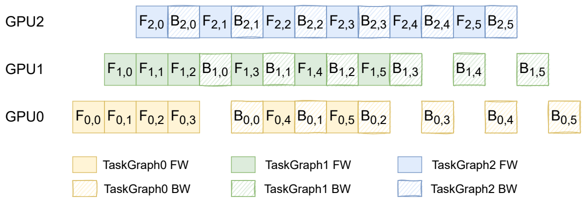

GPU2
F2,0 B2,0 F2,1 B2,1 F2,2 B2,2 F2,3 B2,3 F2,4 B2,4 F2,5 B2,5
GPU1
F1,0 F1,1 F1,2 B1,0 F1,3 B1,1 F1,4 B1,2 F1,5 B1,3 B1,4 B1,5
GPU0
F0,0 F0,1 F0,2 F0,3 B0,0 F0,4 B0,1 F0,5 B0,2 B0,3 B0,4 B0,5
TaskGraph0 FW
TaskGraph1 FW
TaskGraph2 FW
TaskGraph0 BW
TaskGraph1 BW
TaskGraph2 BW

Figure 12: Backward-first pipeline schedule

Algorithm 3: Hardware-aware Pipeline Stage Partition

```
Function Pipeline_Partition(graph, global_bs, GPUs)
    $flop\_ratios \leftarrow \{\}, mem\_ratios \leftarrow [], weights \leftarrow []$
    $ops \leftarrow topo\_sort(graph.ops)$
    $N \leftarrow len(GPUs)$
    $total\_flops \leftarrow \sum_{i=0}^{N}$ GPUs[i].flops
    f foreach $i \in [0, N)$ do
        $weights[i] \leftarrow GPUs[i].flops/total\_flops$
    $stages \leftarrow partition\_stages(ops, weights)$
    $mbs \leftarrow \min_{i=0}^{N} max\_micro\_bs(stages[i], GPUs[i])$
    $mbs \leftarrow \max(mbs, 1)$
    foreach $i \in [0, N)$ do
        $stages[i].bs \leftarrow mbs$
        $mem\_ratios.add(\frac{profile\_mem(stages[i])}{GPUs[i].mem})$
        $flop\_ratios[i] \leftarrow (\frac{profile\_flop(stages[i])}{GPUs[i].flops}, i)$
    if $any(mem\_ratios > 1)$ then
        $PSVF(stages, flop\_ratios, mem\_ratios, GPUs, shift\_op)$
Function $shift\_op(stages, i, j)$
    foreach $k \in [i, j)$ do
        $op = stages[i].operations.popleft()$
        $stages[i + 1].operations.add(op)$
```

sorted operations are then partitioned into n balanced *TaskGraphs* with the algorithm proposed in Section 3.5. If *split* is annotated in a *TaskGraph*, Whale will shard the *TaskGraph* by matching a series of predefined patterns and insert communication operations as needed. Whale can infer split patterns for popular distributed giant models, including MOE[21], Megatron[38], large-scale image classification[20], and so forth.

TaskGraph Schedule. When the *pipeline* parallelism is defined, Whale schedules the *TaskGraph* execution to make the computing devices busy. The *TaskGraph* operations are grouped into four phases: forward, backward, optimizer, and others. By default, Whale adopts a backward-first strategy[13] as shown in Figure 12. $F_{i,j}$ denotes the forward phase of *TaskGraph* i and micro batch j. $B_{i,j}$ denotes the backward phase of *TaskGraph* i and micro batch j. To control the execution order of different phases with different micro batches, Whale first finds the entrance and exit tensors for each *TaskGraph* in forward and backward phases, it then adds control dependencies among them. For example, to make $B_{0,0}$ executes before $F_{0,4}$, we need to make the entrance tensors of $F_{0,4}$ wait for the exit tensors of $B_{0,0}$, and we thus add control edges among $F_{0,4}$ entrance tensors and $B_{0,0}$ exit tensors.

Gradient Aggregation. In synchronous training mode, gradient aggregation is required at every iteration to synchronize the gradients. Whale first runs a local AllReduce[35] operation to aggregate the gradients within one worker, and then it launches a global AllReduce operation for gradients across workers. This process is transparent to users as we replace the original gradients with the All-Reduced gradients in the computation graph, as shown in Figure 13. Whale uses NCCL[5] and ACCL[11] as the communication backends.

Besides, Whale is integrated with optimization technologies such as ZERO[31], recomputation[9], CPU offload[34], automatic mixed precision[26], topology-aware communication, and compilation optimization.

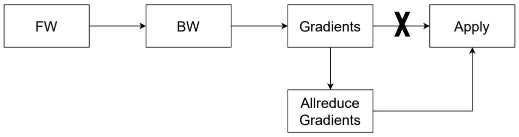


Figure 13: Replace gradients of each replica with All-Reduced gradients.

5 EXPERIMENT

In this section, We first show the effectiveness and efficiency of Whale by training M6-10B[23] model based on a hybrid parallelism. We then demonstrate the training of the largest Chinese multi-modal model M6-1T[45]. In the end, we show the training speedup on heterogeneous GPUs over various models. All the experiments are conducted on a shared cloud GPU cluster. Every cluster node is equipped with a 96-core Intel Xeon Platinum 8163 (Skylake) @2.50GHz with 736GB RAM, running CentOS 7.7. Each node consists of 2/4/8 GPUs, with NVIDIA 32-GB V100 GPUs[3] or NVIDIA 16-GB P100 GPUs[2], powered by NVIDIA driver 418.87, CUDA 10.0, and CUDNN 7. Nodes are connected by 50Gb/s inter-node bandwidth. All the models are implemented by TensorFlow.

5.1 Training M6-10B Model

The M6-10B[23] model is a Chinese multimodal model with 10 billion of parameters. The model takes both visual and linguistic inputs. The visual input length is set to 16, and the linguistic input sequence length is set to 512. The vocabulary size is 21128. The model consists of 24 encoder layers and 24 decoder layers. We use Adafactor[37] as the training optimizer. We parallelize the training of M6-10B model with a hybrid parallel strategy, by combining pipeline parallelism and data parallelism. Whale can easily scale a local M6 model to distributed one by only adding 4 lines(highlighted with blue), as shown in Example 7. The number of micro batches of the *pipeline* is set to 35. We apply recomputation[9] to save memory during training. The training performance is evaluated on NVIDIA 32-GB V100 GPUs. Each node contains 8 GPUs. When scaling the computing nodes from 1 to 32, Whale achieved 91% scalability, as shown in Figure 14.

Example 7: Implement M6-10B model

```
import whale as wh
with wh.cluster():
  with wh.replica():
    with wh.pipeline(num_micro_batch=35):
      token_embe = embeddings()
      image_emb = image_embeddings()
      data = tf.concat([token_emb, image_emb],
                       axis=1)
      encoder_mask = get_attn_mask_imagebert()
      for i in range(24):
        data = encoder(data, encoder_mask)
      for i in range(24):
        data = decoder(data)
      logits = predict(data)
```

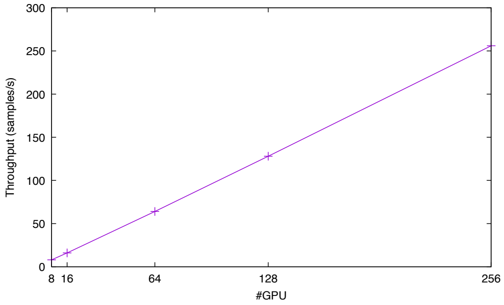


Figure 14: M6-10B with Pipeline and DP Parallelism

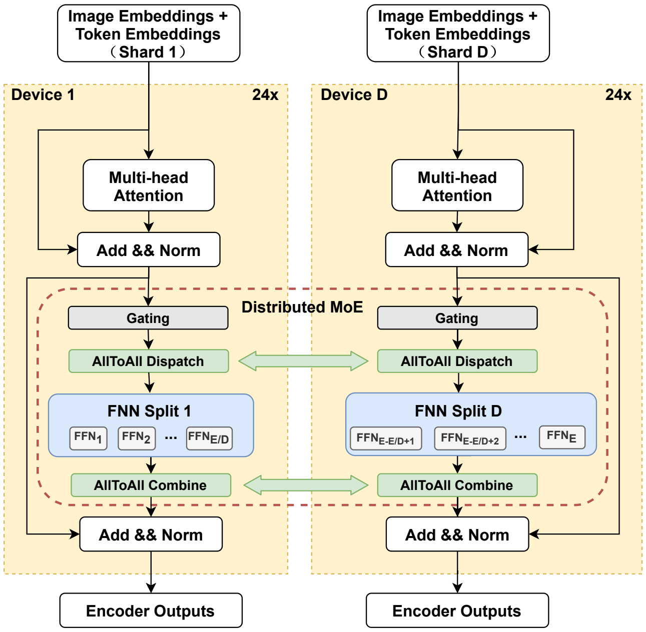


Figure 15: M6-MoE model structure.

5.2 Training M6-MoE Model to Trillions

In this section, we scale the model parameters to 100 billion and 1 trillion. The computation cost of training dense models is in proportional to the model parameters. If we scale the dense 100B model to dense 1T model, to finish training in similar time, we need at least 25,600 NVIDIA V100 GPUs. It is an unreachable number for most users. Instead of scaling M6 model with dense structure, we design M6-MoE model with sparse expert solution[14, 21], as it is regarded as a promising method for model scaling with high training efficiency. We evaluate M6-MoE model with 100B and 1 trillion parameters. Both models are composed of 24 encoder layers. We split the expert layer across multiple devices, and apply data parallelism for the remaining part. The model structure is shown in Figure 15. The model configurations of M6-MoE-100B and M6-MoE-1T are shown in Table 1, we increase the *intermediate_size* and *num_experts* to scale the model parameters from 100B to 1T. More detailed configurations can be found in [45].

Config	**M6-MoE-100B**	**M6-MoE-1T**
hidden_size	1024	1024
num_attention_heads	16	16
intermediate_size	4096	21248
num_experts	512	960

Table 1: Model configuration for M6-MoE-100B and M6-MoE-1T

The sample code of MoE structure is implemented with Whale by adding three lines, as shown in Example 8. In Line 12-13, *split* partitions the computation across devices, and each device has its own unique parameters. Line 2 set the default parallel primitive as *replica*, i.e., we will apply data parallelism for the operations if not explicitly annotated.

Example 8: Implement MoE model

```
 1  import whale as wh
 2  wh.set_default_scope(wh.replica)
 3  atten_outputs = multi_head_attention(inputs)
 4  # Gating and Local Dispatching
 5  gates = softmax(einsum("GSM,ME->GSE",
        atten_outputs, gating_weights))
 6  conbined_weights, dispatch_mask, aux_loss =
        Top2Gating(gates)
 7  dispatch_inputs = einsum("GSEC,GSM->EGCM",
        dispatch_mask, atten_outputs)
 8  # MoE Learning
 9  with wh.split():
10    outputs = MoE(combined_weights, dispatch_inputs)
```

We train M6-MoE-100B with 128 NVIDIA V100 GPUs and train M6-MoE-1T with 480 NVIDIA V100 GPUs. We scaled model parameters by 10 times while only increased GPU number by 3.75 times. Besides the resource saving per parameter, M6-MoE-1T showed significant model quality gain compared to M6-MoE-100B, as shown in Figure 16. We also enable other built-in technologies of Whale to optimize the training process, such as recomputation[9], AMP(auto mixed precision)[1], XLA[4], etc. We are able to train the M6-MoE-100B model with 100 million samples in 1.5 days.

5.3 Training with Heterogeneous GPUs

In this section, we show the benefits of hardware-aware load balancing algorithm by evaluating data parallelism and pipeline parallelism.

For data parallelism, we evaluate three typical models, including ResNet50[16], Bert-Large[10], and GNMT[42]. The experiments are conducted on heterogeneous GPUs that consists of 8 NVIDIA 32GB V100 GPUs and 8 NVIDIA 16GB P100 GPUs. For the baseline, we set the same batch size for all model replicas. We then apply hardware-aware algorithm to each model and get the speedup compared to the baseline performance, as shown in Figure 17. We observe that Whale outperforms the baseline in all three models by a factor from 1.3X to 1.4X. We also measure the GPU Streaming Multiprocessor Activity(SMACT) and report the average metric for each GPU type. As shown in Table 2, hardware-aware policy significantly improved the GPU utilization of V100 by 1.39X to 1.96X for the three models. Meanwhile, we found that the GPU utilization of P100 drops slightly while the overall performance increases, because P100 is not the bottleneck in this case.

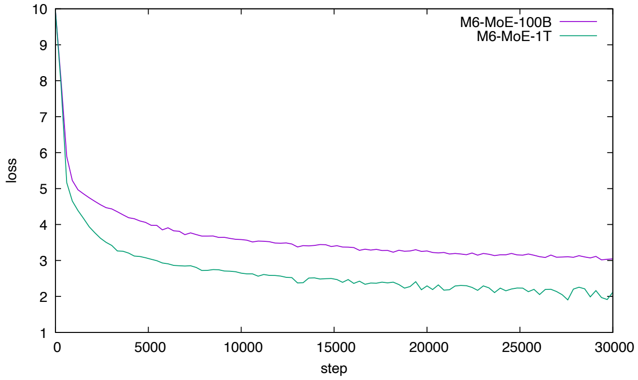


Figure 16: Compare training loss of M6-MoE-100B and M6-MoE-1T.

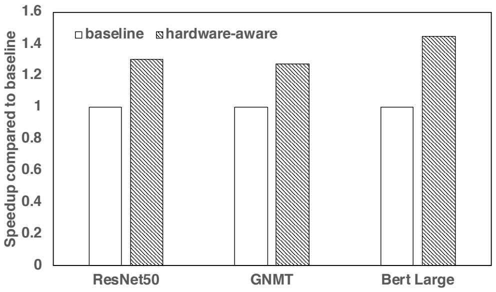


Figure 17: Speedup of hardware-aware training with data parallelism.

SMACT	Baseline		Hardware-aware	
	P100	V100	P100	V100
ResNet50	0.68	0.56	0.62	0.87
GNMT	0.63	0.48	0.56	0.94
Bert Large	0.71	0.57	0.62	0.79

Table 2: GPU utilization for data parallelism when training with heterogeneous GPUs

SMACT	Baseline		Hardware-aware	
	P100	V100	P100	V100
Bert Large	0.68	0.63	0.71	0.77
T5	0.7	0.58	0.88	0.83

Table 3: GPU utilization for pipeline parallelism when training with heterogeneous GPUs

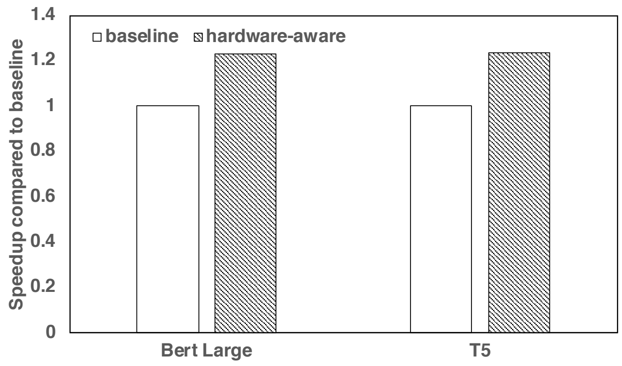


Figure 18: Speedup of hardware-aware training with pipeline parallelism.

For pipeline parallelism, we evaluate two models, including Bert-Large and T5-Large[44]. The training is performed on heterogeneous GPUs that consist 4 NVIDIA 32GB V100 GPUs and 4 NVIDIA 16GB P100 GPUs. Both Bert-Large and T5-Large are partitioned into 4 stages. We further apply data parallelism to the whole *pipeline*. Different pipeline stages have different memory consumption, as each stage must keep the forward activation of N micro batches[13]. Due to the dynamic feature of GPU allocation, users do not know the GPU type when programming the model. To ensure that the distributed model would not run OOM, when setting the baseline, we assume that GPUs with lower memory capacity are allocated to earlier stages. In the above setting, we use the pipeline with the best performance as the baseline. We conducted training with hardware-aware policy and got about 20% speedup on both models, as shown in Figure 18. When checking the GPU utilization, as shown in Table 3, the hardware-aware load balancing strategy improved the GPU utilization of V100 by around 40%.

6 RELATED WORK

DL Training Framework. To facilitate deep learning model training, TensorFlow[7] and PyTorch[29] provide well-supported data parallelism and vanilla model parallelism by explicitly assigning operations to specific devices. Mesh-TensorFlow[36] designs a language to rewrite the model to achieve distributed training. Tofu[40] requires developers to specify what an operator computes using a description language called TDL. DeepSpeed[6] requires users to rewrite the model into a sequential structure when applying pipeline parallelism. In addition, DeepSpeed couples the implementation of tensor model parallelism with model programming, which is not a general API. In comparison, Whale provides more comprehensive and general parallelism primitives without rewriting models. GShard[21, 43] uses parallel annotations similar to Whale, and it infers the parallelism of the remained tensors. GShard adopts SPMD paradigm, where the same partitioned models run on multiple devices. In comparison, Whale supports both SPMD and MPMD paradigms, which can be used to express more complicated scenarios, such as uneven device assignments. FlexFlow[25] adopts layer-wise parallelism search to discover a model parallel training strategy. Each layer can be configured with data or tensor model parallelism. Whale is more general, as it supports both automatic policy exploration and converting a user-annotated graph. Moreover, Whale is hardware aware and thus has more opportunities to discover highly efficient parallel strategy automatically.

Memory Optimization. Zero[31] presents a sharded data parallelism, which partitions the optimizer states, trainable variables, and gradients across workers. Zero-Offload[34] enables large model training by offloading data and computation to CPU. Zero-Infinity[32] further offloads the model to non-volatile memory. Recomputation[9] trades computation for memory saving from activations by recomputing tensors from checkpoints. The above memory optimization techniques save memory to support larger model training. However, they are orthogonal to parallel strategies and have been adopted in Whale for supporting giant model training.

Asynchronous Training. PipeMare[46] applies asynchronous pipeline parallelism to improve the training throughput without performing a pipeline flush at each step. Zero-Offload[34] proposes asynchronous parameter update to overlap offloading time. Asynchronous training obtains speedup by relaxing the synchronous update requirement, while it cannot guarantee the model convergence. So far, Whale focuses on synchronous training as it is the common case in current practice. We leave the integration of Whale with those asynchronous training techniques to the future work.

7 CONCLUSION

We present Whale, an automatic and hardware-aware distributed training framework. Whale generalizes the expression of parallelism with four primitives, which can define various parallel strategies as well as hybrid strategies. It allows users to implement models at an arbitrary scale by adding a few annotations. Whale automatically converts a local model to a distributed one by inferring the parallel transformation for each operation smartly. Moreover, Whale is hardware aware and highly efficient even when training on GPUs of mixed types, which meets the growing demand of heterogeneous training in industrial clusters. Whale has set a milestone for training the largest multimodal pretrained model with trillion of parameters.

REFERENCES

[1] [n.d.]. Automatic Mixed Precision for Deep Learning. https://developer.nvidia.com/automatic-mixed-precision.

[2] [n.d.]. NVIDIA TESLA P100. https://www.nvidia.com/en-us/data-center/tesla-p100/.

[3] [n.d.]. NVIDIA V100 TENSOR CORE GPU. https://www.nvidia.com/en-us/data-center/v100/.

[4] [n.d.]. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla.

[5] 2019. NCCL. https://developer.nvidia.com/nccl.

[6] 2020. DeepSpeed. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[8] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[9] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[11] Jianbo Dong, Zheng Cao, Tao Zhang, Jianxi Ye, Shaochuang Wang, Fei Feng, Li Zhao, Xiaoyong Liu, Liuyihan Song, Liwei Peng, et al. 2020. Eflops: Algorithm and system co-design for a high performance distributed training platform. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 610–622.

[12] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[13] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2020. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. arXiv:2007.01045 [cs.DC]

[14] William Fedus, Barret Zoph, and Noam Shazeer. 2021. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. arXiv:2101.03961 [cs.LG]

[15] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. 2020. Estimating gpu memory consumption of deep learning models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1342–1352.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2018. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965* (2018).

[18] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 947–960. https://www.usenix.org/conference/atc19/presentation/jeon

[19] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[20] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[21] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).

[22] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[23] Junyang Lin, Rui Men, An Yang, Chang Zhou, Ming Ding, Yichang Zhang, Peng Wang, Ang Wang, Le Jiang, Xianyan Jia, Jie Zhang, Jianwei Zhang, Xu Zou, Zhikang Li, Xiaodong Deng, Jie Liu, Jinbao Xue, Huiling Zhou, Jianxin Ma, Jin Yu, Yong Li, Wei Lin, Jingren Zhou, Jie Tang, and Hongxia Yang. 2021. M6: A Chinese Multimodal Pretrainer. arXiv:2103.00823 [cs.CL]

[24] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. *arXiv preprint arXiv:2103.14030* (2021).

[25] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 553–564.

[26] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740* (2017).

[27] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[28] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient Large-Scale Language Model Training on GPU Clusters. *arXiv preprint arXiv:2104.04473* (2021).

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).

[30] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[31] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[32] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. *arXiv preprint arXiv:2104.07857* (2021).

[33] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[34] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. Zero-offload: Democratizing billion-scale model training. *arXiv preprint arXiv:2101.06840* (2021).

[35] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[36] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*. 10414–10423.

[37] Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*. PMLR, 4596–4604.

[38] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[40] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[41] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association.

[42] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[43] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. 2021. GSPMD: General and Scalable Parallelization for ML Computation Graphs. *arXiv preprint arXiv:2105.04663* (2021).

[44] Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. 2020. mT5: A massively multilingual pre-trained text-to-text transformer. *arXiv preprint arXiv:2010.11934* (2020).

[45] An Yang, Junyang Lin, Rui Men, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Jiamang Wang, Yong Li, et al. 2021. Exploring Sparse Expert Models and Beyond. *arXiv preprint arXiv:2105.15082* (2021).

[46] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. 2021. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems* 3 (2021).