Beyond Data and Model Parallelism for Deep Neural Networks

MLSys 2019

Proceedings of Machine Learning and Systems 1

Zhihao Jia

Stanford University

1 Introduction

- 第1段提出问题。DNN模型越来越大，训练越来越昂贵，目前标准做法是在分布式异构机群下训练。然而现有框架（如TensorFlow、Caffe2）的并行训练策略还十分简单，并且往往是suboptimal的。

  *As a result, it is now **standard practice** to parallelize DNN training across distributed heterogeneous clusters. Although DNN models and the clusters used to parallelize them are increasingly complex, the strategies used by today's deep learning frameworks to parallelize training remain **simple, and often suboptimal**.*

- 第2段介绍目前最常用的分布式策略——数据并行，并讨论其缺点。数据并行在每一个设备上放置模型的副本，各设备只需要处理训练数据的子集，并在每一个iteration结束后进行参数同步。

  *Data parallelism is **efficient for compute-intensive operators with a few trainable parameters** (e.g., convolution) but **achieves suboptimal parallelization performance for operators with a large number of parameters** (e.g., embedding).*

  数据并行对于只有少量训练参数的计算密集型算子是高效的，但是对于有大量的算子性能表现较差（因为有较多参数需要同步）

- 第3段介绍另一个常用的分布式策略——模型并行，并讨论其缺点。模型并行将模型的不相交子集分配给各个设备，它消除了设备间的数据同步，但需要在算子间进行数据传输。

  *Another common parallelization strategy is model parallelism, which assigns **disjoint subsets** of a neural network each to a **dedicated device**. This approach **eliminates parameter synchronization between devices but requires data transfers between operators**.*

1 Introduction

- 第3段还提到了ColocRL，一个使用强化学习来为模型并行找到高效算子分配的并行策略。但这一策略只探索了算子的维度。（从这里开始铺垫这篇论文将要探索哪些维度）

  ColocRL (Mirhoseini et al., 2017) uses reinforcement learning to learn efficient operator assignments for model parallelism but ***only explores parallelism in the operator dimension****.*

- 第4段介绍了作者之前的工作OptCNN。 OptCNN采用layer-wise parallelism来为有线性计算图的CNN加速， 采用动态规划来联合优化如何并行每个算子，但没有考虑不同算子之间的并行性。此外， OptCNN不适用语言模型、机器翻译、推荐系统等倾向于使用RNN或者其他非线性网络的领域。（先阐述之前工作的不足，从而更好地对比出当前工作的迭代与改进）

  We recently proposed OptCNN (Jia et al., 2018), which uses layer-wise parallelism for parallelizing CNNs with linear computation graphs. OptCNN uses dynamic programming to jointly optimize how to parallelize each operator ***but*** *does not consider parallelism across different operators.* ***Moreover****, OptCNN does not apply to many DNNs used for language modeling, machine translation, and recommendations, which tend to be RNNs or other non-linear networks.*

前4段介绍了课题的研究背景以及现有技术的不足

- 主流的分布式策略太简单，有着各自的不足，不能只用其一 → Beyond Data and Model Parallelism
- 有的工作探索了更高效的分布式策略，但探索的维度少，可优化空间较大 → SOAP搜索空间
- 之前的工作优化空间小、适用范围窄 → For DNNs

1 Introduction

- **第5段介绍了论文解决问题的方法。论文提出了一个SOAP搜索空间，分为4个维度，其中S代表Sample、O代表Operator、A代表Attribute、P代表Parameter，它推广并超越了之前的方法。**

  In this paper, we introduce a comprehensive SOAP (Sample-Operator-Attribute-Parameter) search space of parallelization strategies for DNNs that generalizes and goes beyond previous approaches. The operator dimension describes how different operators in a DNN are parallelized. For a single operator, the sample and parameter dimensions indicate how training samples and model parameters are distributed across devices. Finally, the attribute dimension defines how different attributes within a sample are partitioned (e.g., the height and width dimensions of an image).

搜索维度	功能	级别
Operator	描述DNN中不同的算子是如何并行化的	计算图级别
Sample	描述对于单个算子，其训练样本如何跨设备分布	算子级别
Parameter	描述对于单个算子，其模型参数如何跨设备分布	算子级别
Attribute	定义如何对Sample中的不同属性进行partition，例如可将图片拆分为宽和高两个维度	样本级别

1 Introduction

- 第6段介绍了工程实现。作者构建了一个深度学习引擎FlexFlow，在其中使用了SOAP搜索空间，可以为任意的DNN找到高效的并行策略。目前的方法都只考虑了SOAP的子集

We use SOAP in FlexFlow, a deep learning engine that automatically finds fast parallelization strategies in the SOAP search space for ***arbitrary DNNs****. Existing approaches only consider one or a subset of SOAP dimensions. For example, data parallelism only explores the sample dimension, while OptCNN parallelizes linear CNNs in the sample, attribute and parameter dimensions.* ***FlexFlow considers parallelizing any DNN (linear or non-linear) in all SOAP dimensions and explores a more comprehensive search space that includes existing approaches as special cases. As a result, FlexFlow is able to find parallelization strategies that significantly outperform existing approaches.***

Table 1. The parallelism dimensions used by different approaches.

Parallelization Approach	Parallelism Dimensions	Hybrid Parallelism	Supported DNNs
Data Parallelism	S		partial**
Model Parallelism	O, P		all
Krizhevsky (2014)	S, P		CNNs**
Wu et al. (2016)	S, O		RNNs#
ColocRL	O		partial#
OptCNN	S, A, P	✓	linear%
FlexFlow (this paper)	S, O, A, P	✓	all

** Does not work for DNNs whose entire model cannot fit on a single device.
# Does not work for DNNs with large operators that cannot fit on a single device.
% Only works for DNNs with linear computation graphs.

FlexFlow对任意DNN (线性或非线性)并行SOAP中的所有维度，探索更全面的搜索空间，现有的方法是FlexFlow的特殊情况。因此，FlexFlow能够找到显著优于现有方法的并行化策略。

1 Introduction

- 第7段介绍关键挑战及解决方法。搜索空间更广、更全面所必须面临的关键挑战是如何快速遍历搜索空间，解决办法有两个：
  ① 使用一个增量的执行模拟器来为并行策略进行快速评估
  ② 使用马尔科夫链蒙特卡洛搜索算法（Markov Chain Monte Carlo, MCMC）

  *The **key challenge** FlexFlow must address is how to efficiently explore the SOAP search space, which is much larger than those considered in previous systems and includes more sophisticated parallelization strategies. **To this end, FlexFlow uses two main components**: a fast, incremental execution simulator to evaluate different parallelization strategies, and a Markov Chain Monte Carlo (MCMC) search algorithm that takes advantage of the incremental simulator to rapidly explore the large search space.*

- 第8段介绍FlexFlow的执行模拟器。模拟器准确度高、速度快，比实际profiling快了三个数量级。 FlexFlow的执行模拟器思想继承于OptCNN，测量每个算子在不同configuration下的性能，提供给任务图，从而为模型架构和机群拓扑进行建模。

  *FlexFlow's execution simulator can accurately predict the performance of a parallelization strategy in the SOAP search space for arbitrary DNNs and is **three orders of magnitude faster than profiling real executions**.*

  此外还使用了一个delta模拟算法， 该算法在之前的模拟结果上进行增量更新，这比原始的模拟快了6.9倍

  *In addition, we introduce a delta simulation algorithm that simulates a new strategy **using incremental updates to previous simulations** and further improves performance over naive simulations by up to 6.9×.*

1 Introduction

- 第9段介绍了执行模拟器的性能与评估。作者在两个GPU Cluster上进行了6个DNN模型的测试，对于所有的模拟，误差都在30%以内，达到了高准确率。

  The execution simulator achieves high accuracy for predicting parallelization performance. We evaluate the simulator with six real-world DNNs on two different GPU clusters and show that, for all the measured executions, the relative difference between the real and simulated execution time is less than 30%.

  疑问：为什么不用平均误差？30%以内的误差是否能称为高准确率？

- 最重要的是，对于给定DNN的不同策略，模拟执行时间保持了真实的执行时间顺序。

  Most importantly for the search, we test different strategies for a given DNN and show that their simulated execution time preserves real execution time ordering.

  回答：因为这项工作不在乎，作者只用关心模拟结果与实测结果的趋势是否一致。

之前读这篇论文的时候过分关注准确率，忽视了作者做这个执行模拟器的初衷与目的，总是纠结准确率这么高够不够？为什么作者不使用看上去更漂亮的指标——平均准确率？如果进一步提高准确率的话FlexFlow的性能会不会更高？现在发现舍本逐末了，任何一项工作都应该为它的目的服务。作者的目的是找到最优的分布式策略，模拟准确率是70%、80%还是90%对他来说没有太大区别。

1 Introduction

- 第10段介绍FlexFlow的优化器。使用MCMC搜索算法来探索SOAP空间，根据模拟的前一个候选策略的性能，迭代提出新的候选策略。搜索结束时，优化器返回发现的最佳策略。

  Using the execution simulator as an oracle, the FlexFlow execution optimizer uses a MCMC search algorithm to explore the SOAP search space and iteratively propose candidate strategies based on the simulated performance of previous candidates.

- 第11段介绍FlexFlow的性能与评估。

➢ **评估方法：在AlexNet、ResNet-101、Inception-v3、RNN Text Classification 、RNN Language Modeling和Neural Machine Translation六个真实的DNN上进行了评估。**

➢ **实验结果：①与专家设计策略和数据并行、模型并行相比，FlexFlow提升了至多3.3x的训练吞吐，减少了至多5x的通信开销、达到了显著更好的缩放性；②在相同的硬件配置下，FlexFlow超过了ColocRL发现的策略3.4-3.8x；③由于对大搜索空间的支持，即使是对于线性DNN，FlexFlow也超过了OptCNN。**

We evaluate FlexFlow on a variety of real-world DNNs including AlexNet, ResNet-101, Inception-v3, RNN Text Classification, RNN Language Modeling and Neural Machine Translation. ***Compared to*** *data/model parallelism and strategies manually designed by domain experts (Krizhevsky, 2014; Wu et al., 2016), FlexFlow increases training throughput by up to 3.3×, reduces communication costs by up to 5×, and achieves significantly better scaling.* ***In addition, FlexFlow outperforms*** *the strategies found by ColocRL by 3.4-3.8× on the same hardware configuration evaluated in ColocRL.* ***Finally, FlexFlow also outperforms*** *OptCNN,* ***even on*** *linear DNNs, by supporting a larger search space.*

2 Related Work

- 第1段再次回顾了数据并行和模型并行，内容是Introduction第2段和第3段的精简。

  Data and model parallelism have been widely used by existing deep learning systems to distribute training across devices.

- 第2段介绍了一些专家手动设计的并行策略，顺便对比了一下强调FlexFlow的强悍。这些策略是借助专家的专业领域知识和经验构建的，例如Krizhevsky对于卷积和池化层使用数据并行、对于密集连接层使用模型并行，从而加速CNN；Wu采用节点间数据并行、节点内模型并行的方式来并行化RNN。尽管这些方式确实相较于数据并行和模型并行有性能提升，但仍然是suboptimal的。而哪怕是以这些专家设计的策略为baseline， FlexFlow依旧有至多2.3x的训练吞吐提升。

  *Expert-designed parallelization strategies manually optimize parallelization for specific DNNs by using experts' domain knowledge and experience. **For example**, ……. **Although** these expert-designed strategies improve performance over data and model parallelism, **they are suboptimal**. We use these expert-designed strategies as baselines in our experiments and show that FlexFlow can further improve training throughput by up to 2.3×.*

- 第3段介绍了一些自动化框架。自动化框架用于在有限的搜索空间中寻找有效的并行策略，作者举了三个例子：①ColocRL使用强化学习来为模型并行找到有效的设备放置策略；②OptCNN使用动态规划来并行化线性CNN，但OptCNN没有探索算子间的并行，不适用于具有非线性计算图的DNN；③Gao等人在tiled domain-specific硬件上利用混合并行化，针对层内和层间的数据通信提出各种数据流优化。

  Automated frameworks have been proposed for finding efficient parallelization strategies in a limited search space. ……③Gao et al. (2017; 2019) exploited hybrid parallelization on tiled domain-specific hardware and proposed various dataflow optimizations for both intra-layer and inter-layer data communication.

2 Related Work

- **第4段将已有方法的并行维度总结为表格1，直观地体现FlexFlow对于已有工作的创新和改进。**

*Table 1 summarizes the parallelism dimensions explored by existing approaches. Data parallelism uses ……, while model parallelism exploits ……. Expert designed strategies exploit …… **but** do not support hybrid parallelism that uses a combination of the sample, attribute, and parameter dimensions to parallelize an operator (see Figure 2). **Compared to these manually designed strategies**, FlexFlow considers more sophisticated, and often more efficient, strategies to parallelize a single operator. **In addition, compared to existing automated frameworks** (e.g., ColocRL and OptCNN), FlexFlow supports more generic DNNs and finds strategies that are up to 3.8× faster by exploring a significantly larger search space.*

Table 1. The parallelism dimensions used by different approaches. S, O, A, and P indicate parallelism in the Sample, Operator, Attribute, and Parameter dimensions. Hybrid parallelism indicates an approach supports parallelizing an operator in a combination of the sample, attribute, and parameter dimensions (see Figure 2).

Parallelization Approach	Parallelism Dimensions	Hybrid Parallelism	Supported DNNs
Data Parallelism	S		partial**
Model Parallelism	O, P		all
Krizhevsky (2014)	S, P		CNNs**
Wu et al. (2016)	S, O		RNNs#
ColocRL	O		partial#
OptCNN	S, A, P	✓	linear%
FlexFlow (this paper)	S, O, A, P	✓	all

** Does not work for DNNs whose entire model cannot fit on a single device.
# Does not work for DNNs with large operators that cannot fit on a single device.
% Only works for DNNs with linear computation graphs.

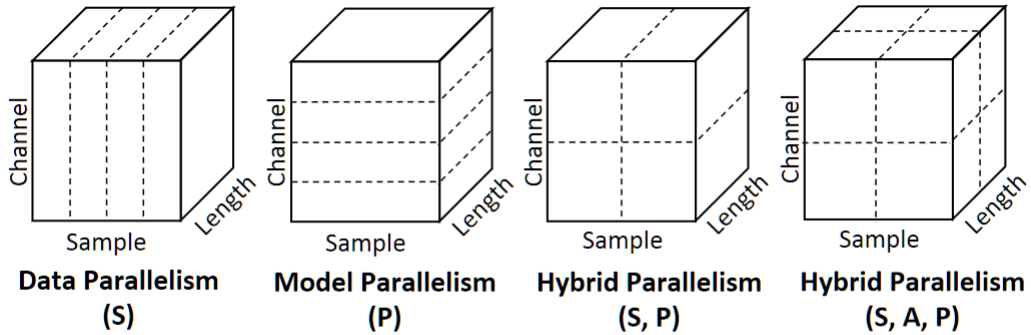


Figure 2. Example parallelization configurations for 1D convolution. Dashed lines show partitioning the tensor.

2 Related Work

- 第5段介绍了基于图的机群调度器。以前的工作提出了机群调度器，通过使用基于图的算法调度机群范围内的任务。Quincy将任务调度映射到流网络，并使用最小代价最大流(MCMF)算法来寻找有效的任务分配；Firmament推广了Quincy ，采用多种MCMF优化算法来减少任务布置延迟。

  Previous work *has proposed cluster schedulers that schedule cluster-wide tasks by using graph-based algorithms.* ***For example****, Quincy (Isard et al., 2009) maps task scheduling to a flow network and uses a min-cost max-flow (MCMF) algorithm to find efficient task placement. Firmament (Gog et al., 2016) generalizes Quincy by employing multiple MCMF optimization algorithms to reduce task placement latencies.*

- 现有的基于图的调度器通过假设一个固定的任务图来优化任务放置，而FlexFlow解决了另一个问题，即联合优化如何通过利用SOAP维度中的并行性将算子划分为任务，以及如何将任务分配给设备。

  Existing graph-based schedulers *optimize task placement by assuming a fixed task graph.* ***However****, FlexFlow solves a different problem that requires jointly optimizing how to partition an operator into tasks by exploiting parallelism in the SOAP dimensions and how to assign tasks to devices.*

3 Overview

- 第1段介绍了FlexFlow的内部表达形式。使用算子图来表示DNN内部的算子和状态，图中每一个结点是一个算子，每一条边是一个tensor。使用设备拓扑图来描述所有可用设备（如图1），图中每一个节点是一个设备（CPU/GPU），每一条边是两个设备间的连接（NVLink、PCI-e、网络等），并且在边上标记连接的带宽和延迟。

  *Similar to existing deep learning frameworks (e.g., TensorFlow and PyTorch), FlexFlow **uses ……to describe ……**. Each node oi ∈ G is ……, and each edge (oi, oj) ∈ G is ……. In addition, FlexFlow **also takes a device topology graph D = (DN , DE) describing** ……. Each node di ∈ DN represents ……, and each edge (di, dj) ∈ DE is ……. The edges **are labeled with** the bandwidth and latency of the connection.*

- 第2段介绍了FlexFlow的工作流程，以算子图和设备拓扑作为输入，自动在SOAP空间中寻找一条高效策略。在搜索空间中，所有策略执行相同的DNN定义的计算量，因此通过设计保持了相同的模型精度。

  FlexFlow takes an operator graph and a device topology as inputs and automatically finds an efficient strategy in the SOAP search space. All strategies in the search space perform the same computation defined by the DNN and therefore maintains the same model accuracy by design.

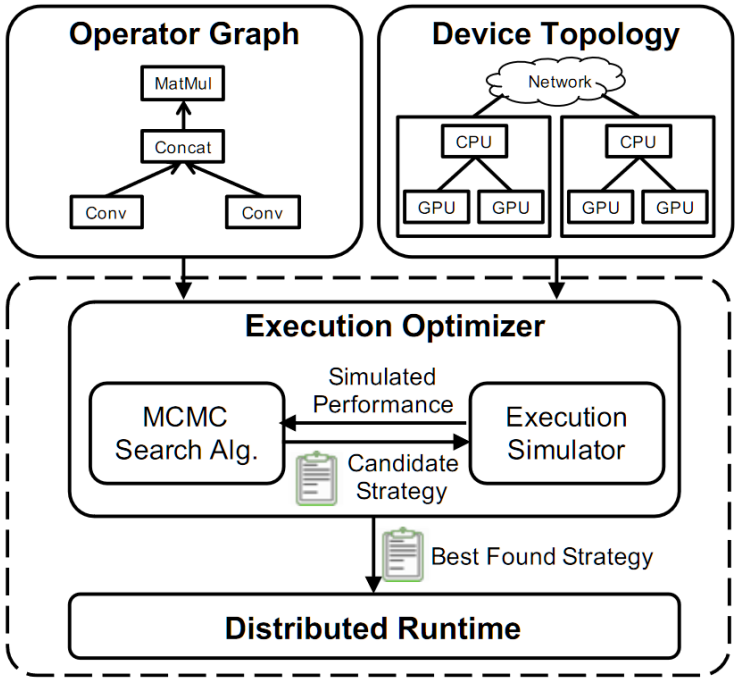


Figure 1. FlexFlow overview.

3 Overview

- 第3段介绍了FlexFlow的主要模块及功能。执行优化器用MCMC算法来探索可能的并行策略空间，并迭代地提出由执行模拟器评估的候选策略；执行模拟器通过delta模拟算法在上一次模拟的基础上进行增量更新；模拟的执行时间指导搜索生成候选策略。当搜索时间预算耗尽时，执行优化器将发现的最佳策略发送给分布式运行时，从而实际地并行化执行。

  The main components of FlexFlow are shown in Figure 1. ***The execution optimizer*** *uses a MCMC search algorithm to explore the space of possible parallelization strategies and iteratively proposes candidate strategies that are evaluated by an execution simulator.* ***The execution simulator*** *uses a delta simulation algorithm that simulates a new strategy using incremental updates to previous simulations.* ***The simulated execution time guides the search in generating future candidates****. When the search time budget is exhausted, the execution optimizer sends the best discovered strategy to a distributed runtime for parallelizing the actual executions.*

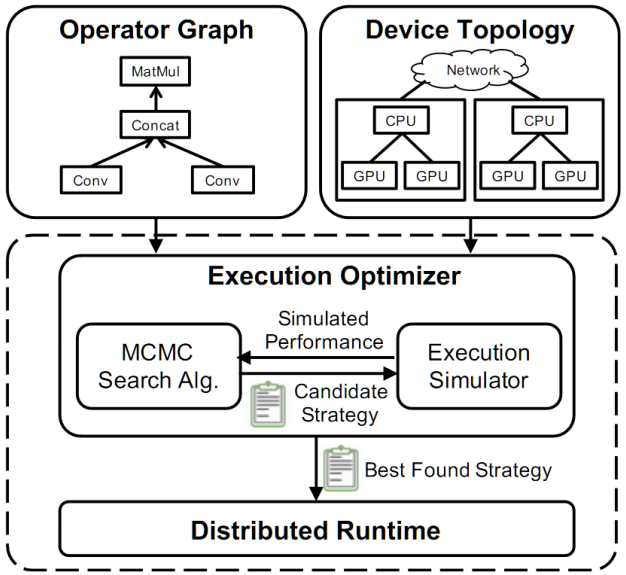


Figure 1. FlexFlow overview.

4 The SOAP Search Space

- 第1段说明本节介绍将SOAP搜索空间。为了跨设备并行一个DNN算子，需要每个设备计算该算子输出张量的不相交子集。因此，将通过输出张量如何拆分来为算子的并行建模。

  This section introduces the SOAP search space of parallelization strategies for DNNs. To parallelize a DNN operator across devices, we require each device to compute a disjoint subset of the operator's output tensors. ***Therefore****, we model the parallelization of an operator oi by defining how the output tensor of oi is partitioned.*

- 第2段介绍了算子的可并行维度。对于一个算子，将其可并行维度Pi定义为其输出张量中所有可分维度的集合。Pi总是包含一个样本维度，对于Pi中的所有其他维度，如果在该维度上进行划分需要分离模型参数，我们称其为参数维度，否则称其为属性维度。

  For an operator oi, we define its parallelizable dimensions Pi as the set of all divisible dimensions in its output tensor. Pi always includes a sample dimension. For all other dimensions in Pi, we call it a parameter dimension if partitioning over that dimension requires splitting the model parameters and call it an attribute dimension otherwise. Table 2 shows the parallelizable dimensions of some example operators. Finally, we also consider parallelism across different operators in the operator dimension.

Table 2. Parallelizable dimensions for different operators. The *sample* and *channel* dimension index different samples and neurons, respectively. For images, the *length* and the combination of *height* and *width* dimensions specify a position in an image.

Operator	Parallelizable Dimensions		
	(S)ample	(A)ttribute	(P)arameter
1D pooling	sample	length, channel	
1D convolution	sample	length	channel
2D convolution	sample	height, width	channel
Matrix multiplication	sample		channel

4 The SOAP Search Space

- **第3段介绍了算子的并行配置。并行配置定义了算子如何在多设备上并行，图2展示了1D卷积算子在单维度和多维度组合上的并行配置范例。**

A parallelization configuration ci of an operator oi defines how the operator is parallelized across multiple devices. Figure 2 shows some example configurations for parallelizing a 1D convolution operator in a single dimension as well as combinations of multiple dimensions.

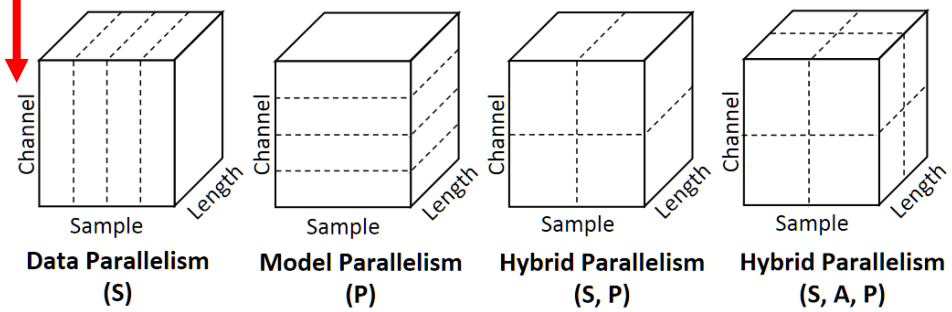


Figure 2. Example parallelization configurations for 1D convolution. Dashed lines show partitioning the tensor.

Table 2. Parallelizable dimensions for different operators. The *sample* and *channel* dimension index different samples and neurons, respectively. For images, the *length* and the combination of *height* and *width* dimensions specify a position in an image.

Operator	Parallelizable Dimensions		
	(S)ample	(A)ttribute	(P)arameter
1D pooling	sample	length, channel	
1D convolution	sample	length	channel
2D convolution	sample	height, width	channel
Matrix multiplication	sample		channel

同样是channel，对于不同算子其所属的维度也是不同的，例如对1D pooling而言算是样本的属性，而对于卷积则是模型参数的一部分（划分依据在第2段）

搜索维度	功能	级别
Operator	描述DNN中不同的算子是如何并行化的	计算图级别
Sample	描述对于单个算子，其训练样本如何跨设备分布	算子级别
Parameter	描述对于单个算子，其模型参数如何跨设备分布	算子级别
Attribute	定义如何对Sample中的不同属性进行partition，例如可将图片拆分为宽和高两个维度	样本级别

4 The SOAP Search Space

- 第4段介绍了可并行维度的具体内容。
  - 对于Pi中的每个可并行维度，ci都包含一个描述该维度并行度的正整数。用正整数描述每个维度的degree
  - |ci|是ci的所有可并行维度的并行度的乘积。 |ci| 是所有degree的乘积，表示总共的分区数
  - 我们在每个维度中使用大小相同的分区，以保证工作负载分布的良好平衡。每个分区大小相等
  - 并行化配置ci将操作符oi划分为|ci|独立任务，表示为ti:1，…， ti:|ci|，同时ci还包括对每个任务ti:k(1≤k≤|ci|)的设备分配。将一个算子拆分为|ci|个独立任务，分配给|ci|个独立设备

给定一个任务的输出张量和它的算子类型，我们可以推断出执行每个任务所必需的输入张量。

For each parallelizable dimension in Pi, ci includes a positive integer that is the degree of parallelism in that dimension. |ci| is the product of the parallelism degrees for all parallelizable dimensions of ci. We use equal size partitions in each dimension to guarantee well-balanced workload distributions. A parallelization configuration ci partitions the operator oi into |ci| independent tasks, denoted as ti:1, ..., ti:|ci|, meanwhile ci also includes the device assignment for each task ti:k (1 ⩽ k ⩽ |ci|). ***Given the output tensor of a task and its operator type, we can infer the necessary input tensors to execute each task.***

4 The SOAP Search Space

- **第5段解释了图三，使用一个case study帮助读者更好地理解并行配置。图3显示了矩阵乘算子(即U = VW)的并行化配置示例。将算子划分为四个独立的任务，分配给不同的GPU设备。**

  Figure 3 shows an example parallelization configuration for a matrix multiplication operator (i.e., U = V W). The operator is partitioned into four independent tasks assigned to different GPU devices. The input and output tensors of the tasks are shown in the figure.

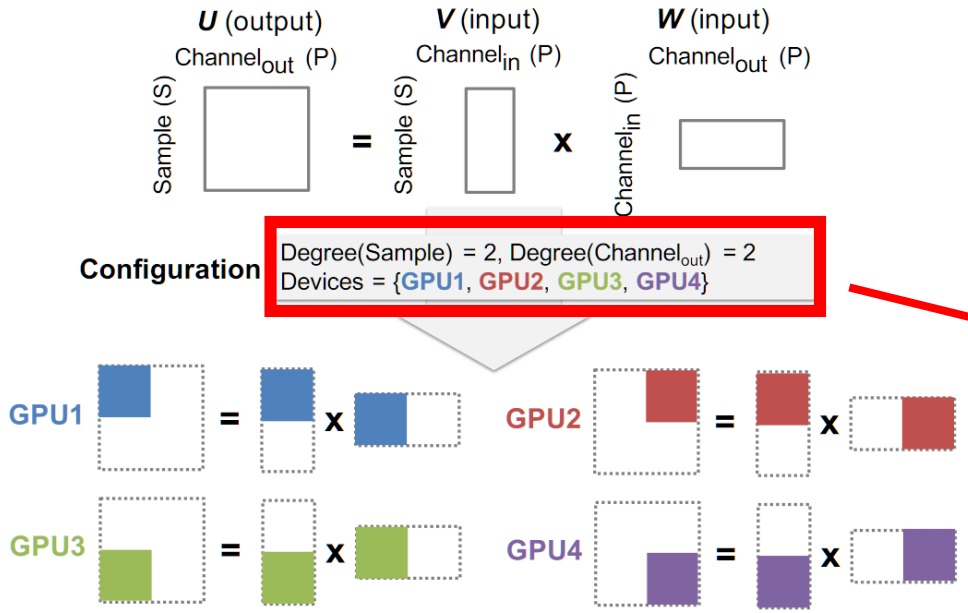


Figure 3. An example parallelization configuration for a matrix multiplication operator.

Table 2. Parallelizable dimensions for different operators. The *sample* and *channel* dimension index different samples and neurons, respectively. For images, the *length* and the combination of *height* and *width* dimensions specify a position in an image.

Operator	Parallelizable Dimensions		
	(S)ample	(A)ttribute	(P)arameter
1D pooling	sample	length, channel	
1D convolution	sample	length	channel
2D convolution	sample	height, width	channel
Matrix multiplication	sample		channel

4 The SOAP Search Space

- **第6段介绍了并行策略。并行策略S描述了一个应用程序的一种可能的并行化。S包含每个算子oi的并行化配置ci，每个oi的配置可以从oi的所有可能配置中独立选择。**

  A parallelization strategy S describes one possible parallelization of an application. S includes a parallelization configuration ci for each operator oi, and each oi's configuration can be chosen independently from among all possible configurations for oi.

  第一段：SOAP搜索空间，通过拆分输出张量来建模——方法论
  第二段：算子的可并行维度，定义为输出张量中所有可分维度的集合——解决方案
  第三段：算子的并行配置，定义了算子如何在多设备上并行——描述形式
  第四段：可并行维度的具体内容——具体内容
  第五段：并行配置的case study，将并行配置和可并行维度结合起来——示例
  第六段：并行策略，所有算子并行配置的集合——将这套步骤组合为更高层次的抽象

5 Execution Simulator

- 第1段介绍本章主要内容。这一章介绍了执行模拟器，它以算子图G、设备拓扑D和并行策略S作为输入，预测G在D上使用策略S的执行时间。

  In this section, we describe the execution simulator, which takes an operator graph G, a device topology D, and a parallelization strategy S as inputs and predicts the execution time to run G on D using strategy S.

- 第2段介绍了模拟所需的4个假设

① 每个任务的执行时间是可预测的，方差很小，并且与输入张量的内容无关。
② 对于每个带宽为b的连接，传输一个大小为s的张量需要s/b时间(即通信带宽可以得到充分利用)。
③ 每个设备使用先进先出(FIFO)调度策略来处理分配的任务。这是现代设备(如gpu)所使用的策略。
④ 运行时的开销可以忽略不计。当设备的输入张量可用并且设备已经完成之前的任务时，设备就开始处理一个任务。

A1. *The execution time of each task is predictable with low variance and is independent of the contents of input tensors.*
A2. *For each connection (di, dj) between device di and dj with bandwidth b, transferring a tensor of size s from di to dj takes s/b time (i.e., the communication bandwidth can be fully utilized).*
A3. *Each device processes the assigned tasks with a FIFO (first-in-first-out) scheduling policy. This is the policy used by modern devices such as GPUs.*
A4. *The runtime has negligible overhead. A device begins processing a task as soon as its input tensors are available and the device has finished previous tasks.*

5 Execution Simulator

- 第3段介绍模拟器的总体idea。为了模拟执行，借鉴了之前的工作OptCNN 的idea，测量每个配置中每个不同算子的执行时间，并将这些测量结果包含在一个任务图中，任务图包括从算子派生的所有任务和任务之间的依赖关系。该模拟器可以通过在任务图上运行模拟算法来生成执行时间轴。

  To simulate an execution, we borrow the idea from OptCNN (Jia et al., 2018) to measure the execution time of each distinct operator once for each configuration and include these measurements in a task graph, which includes all tasks derived from operators and dependencies between tasks. The simulator can generate an execution timeline by running a simulation algorithm on the task graph.

5 Execution Simulator: 5.1 Task Graph

- 第1段介绍了任务图。任务图对由算子派生独立任务之间的依赖关系进行建模。为了统一抽象，我们将设备之间的每个硬件连接建模为一个只能执行通信任务的通信设备(即数据传输)。值得注意的是，设备和硬件连接被建模为独立的设备，这允许计算(即普通任务)和通信(即通信任务)在可能的情况下重叠。

  A task graph models dependencies between individual tasks derived from operators. ***To unify the abstraction****, we model each hardware connection between devices as a communication device that can only perform communication tasks (i.e., data transfers).* ***Note that*** *devices and hardware connections are modeled as separate devices, which allows computation (i.e., normal tasks) and communication (i.e., communication tasks) to be overlapped if possible.*

5 Execution Simulator: 5.1 Task Graph

- 第2段介绍了构造任务图的步骤。给定算子图G、设备拓扑D和并行化策略S，使用以下步骤构造任务图T = (TN, TE)，其中每个节点T∈TN是一个任务(即普通任务或通信任务)，每个边(ti, tj)∈TE是一个依赖项，任务tj在任务ti完成之前不能启动。值得注意的是，**任务图中的边只是顺序约束，不表示数据流，因为所有数据流都作为通信任务包含在任务图中**。

① 对于每个具有并行化配置ci的算子oi∈G，添加任务ti:1，…， ti:|ci|到TN。

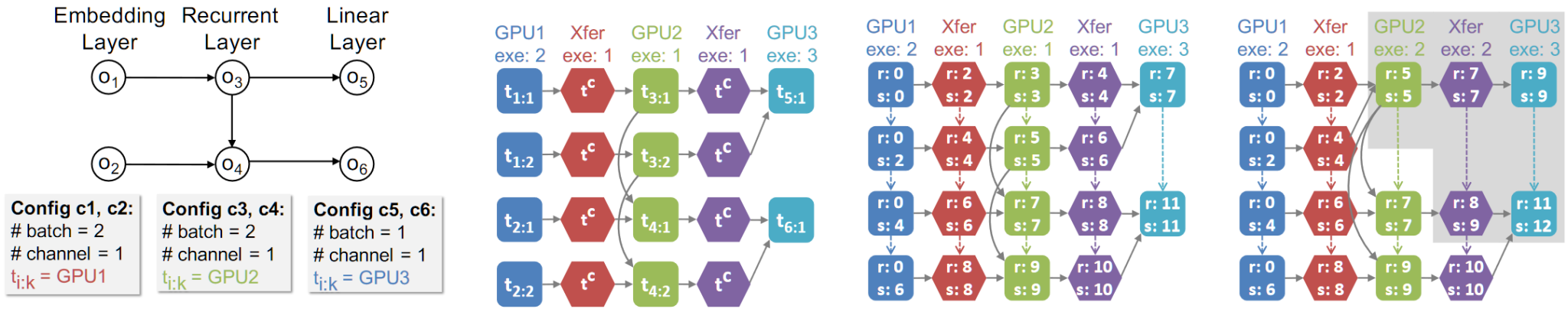
② 对于每个张量(oi, oj)∈G，它是算子oi的输出和oj的输入，我们计算由任务ti:ki(1≤ki≤|ci|)写的输出子张量和由任务tj:kj(1≤kj≤|cj|)读的输入子张量。对于每个拥有共享张量的任务对(ti:ki,tj:kj)，如果两个任务被分配到**同一个设备**上，我们在TE中添加一条边(ti:ki, tj:kj)，表示两个任务之间存在**依赖关系**，不需要通信任务；如果ti:ki和tj:kj被分配到**不同设备**上，我们在TN上添加一个**通信任务**tc和两条边(ti:ki, tc)和(tc, tj:kj)

Given an *operator graph G, a device topology D, and a parallelization strategy S,* ***we use*** *the following steps to construct a task graph T = (TN , TE),* ***where*** *each node t ∈ TN is a task (i.e., a normal task or a communication task) and each edge (ti, tj) ∈ TE is a dependency that task tj cannot start until task ti is completed.* ***Note that*** *the edges in the task graph are simply ordering constraints—the edges do not indicate data flow, as all data flow is included in the task graph as communication tasks.*

5 Execution Simulator: 5.1 Task Graph

- 第3段给了一个并行化策略的case study。图4a给出了一个标准的3层RNN的并行化策略示例，该RNN由嵌入层、循环层和线性层组成。它代表了常用的模型并行，将每一层的算子分配给专用的GPU。图4b显示了相应的任务图。每个正方形和六边形分别表示一个普通任务和一个通信任务，每个有向边表示任务之间的依赖关系。

Figure 4a shows an example parallelization strategy for a standard 3-layer RNN consisting of an embedding layer, a recurrent layer, and a linear layer. It represents commonly used model parallelism that assigns operators in each layer to a dedicated GPU. Figure 4b shows the corresponding task graph. Each square and hexagon indicate a normal and a communication task, respectively, and each directed edge represents a dependency between tasks.



(a) An example parallelization strategy. (b) The corresponding task graph. (c) The task graph after the full simulation algorithm. (d) The task graph after the delta simulation algorithm.

Figure 4. Simulating an example parallelization strategy. The tasks' `exeTime` and `device` are shown on the top of each column. In Figure 4c and 4d, the word "r" and "s" indicate the `readyTime` and `startTime` of each task, respectively, and the dashed edges indicate the `nextTask`.

5 Execution Simulator: 5.1 Task Graph

- 第4段借助表3介绍了任务图中每个任务的属性。对于一个从算子派生的普通任务，它的exeTime是在给定设备上执行任务的时间，通过在设备上多次运行任务并测量平均执行时间来估计(假设A1)。任务的exeTime被缓存，以后所有具有相同操作符类型和输入/输出张量形状的任务将使用缓存的值，而不需要重新运行任务。对于一个通信任务，它的exeTime是在带宽为b的设备之间传输一个张量(大小为s)的时间，估计为s/b (假设A2)

Table 3 lists the properties for each task in the task graph. For a normal task derived from an operator, its exeTime is the time to execute the task on the given device and is estimated by running the task multiple times on the device and measuring the average execution time (assumption A1). A task's exeTime is cached, and all future tasks with the same operator type and input/output tensor shapes will use the cached value without rerunning the task. For a communication task, its exeTime is the time to transfer a tensor (of size s) between devices with bandwidth b and is estimated as s/b (assumption A2).

Table 3. Properties for each task in the task graph.

Property	Description
Properties set in graph construction	
`exeTime`	The elapsed time to execute the task.
`device`	The assigned device of the task.
$\mathcal{I}$(t)	$\{t_{in} \mid (t_{in}, t) \in \mathcal{T}_E\}$
$\mathcal{O}$(t)	$\{t_{out} \mid (t, t_{out}) \in \mathcal{T}_E\}$
Properties set in simulation	
`readyTime`	The time when the task is ready to run.
`startTime`	The time when the task starts to run.
`endTime`	The time when the task is completed.
`preTask`	The previous task performed on `device`.
`nextTask`	The next task performed on `device`.
Internal properties used by the full simulation algorithm	
`state`	Current state of the task, which is one of `NOTREADY`, `READY`, and `COMPLETE`.

5 Execution Simulator: 5.1 Task Graph

- 第5段提到了其他属性。除了exeTime, FlexFlow还在图构造过程中设置设备、I(t)和O(t)(在表3中定义)。表3中的其他属性仍未设置，必须由模拟填充。

  In addition to exeTime, FlexFlow also sets device, I(t), and O(t) (defined in Table 3) during graph construction. Other properties in Table 3 remain unset and must be filled in by the simulation.

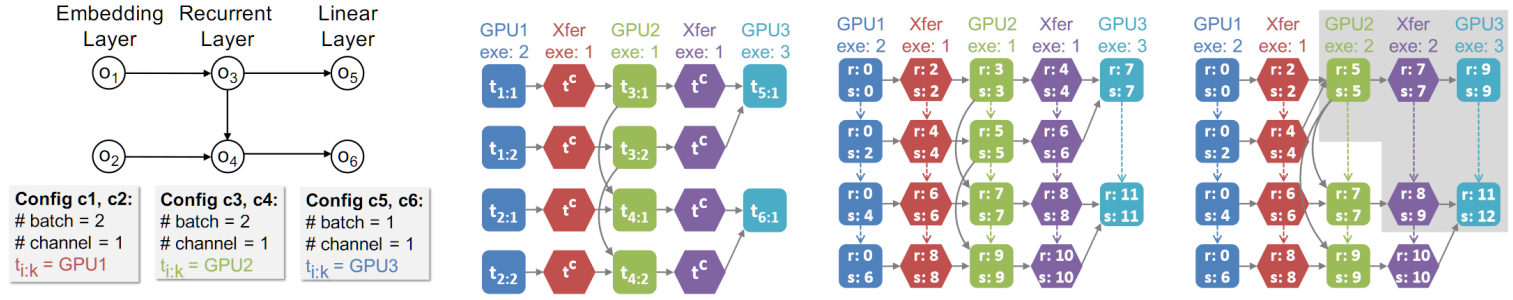
Table 3. Properties for each task in the task graph.

Property	**Description**
Properties set in graph construction	
`exeTime`	The elapsed time to execute the task.
`device`	The assigned device of the task.
$\mathcal{I}$(t)	$\{t_{in} \mid (t_{in}, t) \in \mathcal{T}_E\}$
$\mathcal{O}$(t)	$\{t_{out} \mid (t, t_{out}) \in \mathcal{T}_E\}$
Properties set in simulation	
`readyTime`	The time when the task is ready to run.
`startTime`	The time when the task starts to run.
`endTime`	The time when the task is completed.
`preTask`	The previous task performed on `device`.
`nextTask`	The next task performed on `device`.
Internal properties used by the full simulation algorithm	
`state`	Current state of the task, which is one of `NOTREADY`, `READY`, and `COMPLETE`.

5 Execution Simulator: 5.2 Full Simulation Algorithm

- 第1段介绍完全模拟算法。完全模拟算法作为与delta模拟算法比较的baseline，首先使用5.1节中描述的方法构建任务图，然后使用Dijkstra最短路径算法的变体设置每个任务的属性。当任务就绪时(即所有前驱任务都完成)，任务被放入一个全局优先级队列中，并按照它们的readyTime递增顺序退出队列。因此，当任务t退出队列时，所有具有更早的readyTime的任务都已被调度，我们可以设置任务t的属性，同时保持FIFO调度顺序(假设A3)。图4c显示了示例并行化策略的执行时间表。

We now describe a full simulation algorithm that we use as a baseline for comparisons with our delta simulation algorithm. The full simulation algorithm first builds a task graph using the method described in Section 5.1 and then sets the properties for each task using a variant of Dijkstra's shortest-path algorithm (Cormen et al., 2009). Tasks are enqueued into a global priority queue when ready (i.e., all predecessor tasks are completed) and are dequeued in increasing order by their readyTime. Therefore, when a task t is dequeued, all tasks with an earlier readyTime have been scheduled, and we can set the properties for task t while maintaining the FIFO scheduling order (assumption A3). Figure 4c shows the execution timeline of the example parallelization strategy.



(a) An example parallelization strategy. (b) The corresponding task graph. (c) The task graph after the full simulation algorithm. (d) The task graph after the delta simulation algorithm.

Figure 4. Simulating an example parallelization strategy. The tasks' `exeTime` and `device` are shown on the top of each column. In Figure 4c and 4d, the word "r" and "s" indicate the `readyTime` and `startTime` of each task, respectively, and the dashed edges indicate the `nextTask`.

5 Execution Simulator: 5.3 Delta Simulation Algorithm

- 第1段介绍FlexFlow的搜索算法。FlexFlow使用了一种MCMC搜索算法，该算法通过改变前一种策略中单个算子的并行化配置来提出一种新的并行化策略(参见6.2节)。因此，在一般情况下，大多数执行时间轴不会因不同的模拟策略而改变。基于这一观察结果，我们引入了一种增量（delta）模拟算法，该算法从之前的任务图开始，只重新模拟执行时间轴发生变化的部分所涉及的任务，这种优化大大加快了模拟器的速度，特别是对于大型分布式机器的策略。

  FlexFlow uses a MCMC search algorithm that proposes a new parallelization strategy by changing the parallelization configuration of a single operator in the previous strategy (see Section 6.2). As a result, in the common case, most of the execution timeline does not change from one simulated strategy to the next. Based on this observation, we introduce a delta simulation algorithm that starts from a previous task graph and only re-simulates tasks involved in the portion of the execution timeline that changes, an optimization that dramatically speeds up the simulator, especially for strategies for large distributed machines.

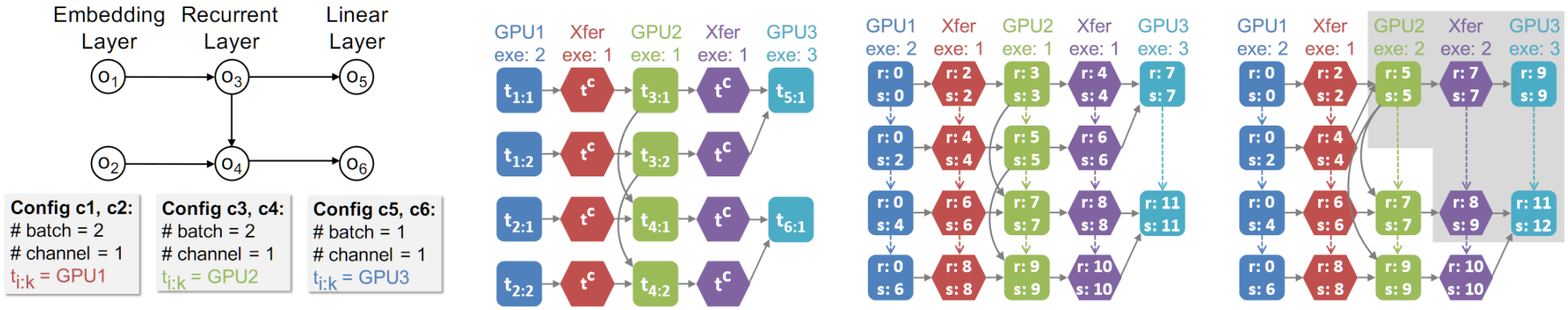
- 第2段介绍增量（delta）模拟算法。为了模拟一种新的策略，增量模拟算法首先从现有的任务图中更新任务和依赖关系，并将所有修改后的任务排队到一个全局优先级队列中。与Bellman-Ford最短路径算法类似，增量模拟算法迭代地将更新的任务出队列，并将更新传播给后续任务。

  To simulate a new strategy, the delta simulation algorithm first updates tasks and dependencies from an existing task graph and enqueues all modified tasks into a global priority queue. Similar to the Bellman-Ford shortest-path algorithm (Cormen et al., 2009), the delta simulation algorithm iteratively dequeues updated tasks and propagates the updates to subsequent tasks.

5 Execution Simulator: 5.3 Delta Simulation Algorithm

- 第3段再次使用图4中的例子作为增量模拟算法的case study。考虑由原策略(图4a)仅将算子o3的并行度降为1(即|c3| = 1)得到的新的并行化策略。图4d显示了新的并行化策略的任务图，该任务图可以由原任务图(图4c)通过更新灰色区域任务的模拟属性生成。

For the example in Figure 4, consider a new parallelization strategy derived from the original strategy (Figure 4a) by only reducing the parallelism of operator o3 to 1 (i.e., |c3| = 1). Figure 4d shows the task graph for the new parallelization strategy, which can be generated from the original task graph (in Figure 4c) by updating the simulation properties of tasks in the grey area.



(a) An example parallelization strategy. (b) The corresponding task graph. (c) The task graph after the full simulation algorithm. (d) The task graph after the delta simulation algorithm.

Figure 4. Simulating an example parallelization strategy. The tasks' `exeTime` and `device` are shown on the top of each column. In Figure 4c and 4d, the word "r" and "s" indicate the `readyTime` and `startTime` of each task, respectively, and the dashed edges indicate the `nextTask`.

6 Execution Optimizer

- 第1段介绍执行优化器的功能。执行优化器以算子图和设备拓扑作为输入，自动找到高效的并行化策略。FlexFlow使用模拟器作为oracle（谕示），将并行化优化问题转化为成本最小化问题，即最小化预期执行时间。

  The execution optimizer takes an operator graph and a device topology as inputs and automatically finds an efficient parallelization strategy. ***Using the simulator as an oracle****, FlexFlow* ***transforms the parallelization optimization problem into a cost minimization problem****,* ***namely*** *minimizing the predicted execution time.*

- 第2段介绍了搜索面临的困难。通过找到最小的makespan来找到最优的并行化策略是NP-hard的。此外，可能策略的数量与算子图中的算子数量呈指数关系(见第4节：S包含每个算子oi的并行化配置ci，每个oi的配置可以从oi的所有可能配置中独立选择)，这使得搜索空间难以穷尽枚举。为了找到低成本策略，FlexFlow使用成本最小化搜索来启发式地探索空间，并返回所发现的最佳策略。

  Finding the optimal parallelization strategy is NP-hard, by an easy reduction from minimum makespan (Lam & Sethi, 1977). In addition, the number of possible strategies is exponential in the number of operators of an operator graph (see Section 4), which makes it intractable to exhaustively enumerate the search space. To find a low-cost strategy, FlexFlow uses a cost minimization search to heuristically explore the space and returns the best strategy discovered.

6 Execution Optimizer: 6.1 MCMC Sampling

- 这一小节介绍了在执行优化器中用于MCMC采样的Metropolis-Hastings算法。该算法保持当前策略S，并随机提出新的策略S*。S*按照公式(1)计算出的概率被接受并成为新的当前策略。
- MCMC倾向于表现为一种贪婪的搜索算法，它随时倾向于向成本更低的方向移动，但也可以避开局部最小值。

This section briefly introduces the Metropolis-Hastings algorithm (Hastings, 1970) we use for MCMC sampling in the execution optimizer. The algorithm maintains a current strategy S and randomly proposes a new strategy S. S* is accepted and becomes the new current strategy with the following probability:*

$$\alpha(\mathcal{S}^*|\mathcal{S}) = \min\left(1, \exp\left(\beta \cdot (cost(\mathcal{S}) - cost(\mathcal{S}^*))\right)\right) \quad (1)$$

MCMC tends to behave as a greedy search algorithm, preferring to move towards lower cost whenever that is readily available, but can also escape local minima.

6 Execution Optimizer: 6.2 Search Algorithm

- 这一小节介绍执行优化器的搜索算法。
  - 优化器生成提议的方法很简单：随机选择当前并行化策略中的一个算子，并将其并行化配置替换为随机配置。
  - 我们使用模拟器预测的执行时间作为方程1中的代价函数，并使用现有策略(如数据并行性、专家设计策略)和随机生成策略作为搜索算法的初始候选策略。
  - 对于每个初始策略，搜索算法迭代提出新的候选策略，直到满足以下两个条件之一：
    (1)当前初始策略的搜索时间预算耗尽；
    (2)搜索过程在一半的搜索时间内不能进一步改进最佳发现策略。

Our method for generating proposals is simple: an operator in the current parallelization strategy is selected at random, and its parallelization configuration is replaced by a random configuration. We use the predicted execution time from the simulator as the cost function in Equation 1 and use existing strategies (e.g., data parallelism, expert-designed strategies) as well as randomly generated strategies as the initial candidates for the search algorithm. For each initial strategy, the search algorithm iteratively proposes new candidates until one of the following two criteria is satisfied: (1) the search time budget for current initial strategy is exhausted; or (2) the search procedure cannot further improve the best discovered strategy for half of the search time.

7 FlexFlow Runtime

- 第1段介绍了现有深度学习系统的不足。我们发现，现有的深度学习系统(如TensorFlow、PyTorch、Caffe2和MXNet)仅支持通过数据并行在样本维度上并行化一个算子，而在这些系统中并行化其他维度上的算子或多个SOAP维度的组合是不平凡的。

  We found *that existing deep learning systems (e.g., TensorFlow, PyTorch, Caffe2, and MXNet) only support parallelizing an operator in the sample dimension through data parallelism, and it is* ***non-trivial*** *to parallelize an operator in other dimensions or combinations of several SOAP dimensions in these systems.*

- 第2段介绍了FlexFlow的工程实现基础。为了支持在SOAP搜索空间中使用任何策略并行化DNN模型，我们在Legion中实现了FlexFlow分布式运行时，Legion是一种用于分布式异构架构的高性能并行运行时，并使用cuDNN和cuBLAS作为处理DNN算子的底层库。使用Legion高维分区接口来支持在任何可并行维度组合下并行一个算子。

  To support *parallelizing DNN models using any strategy in the SOAP search space, we implemented the FlexFlow distributed runtime in Legion (Bauer et al., 2012), a high-performance parallel runtime for distributed heterogeneous architectures, and use cuDNN (Chetlur et al., 2014) and cuBLAS (cuBLAS) as the underlying libraries for processing DNN operators. We use the Legion high-dimensional partitioning interface (Treichler et al., 2016) to support parallelizing an operator in any combination of the parallelizable dimensions.*

8 Evaluation

- 第1段总览本节内容，在六个真实的DNN benchmark和两个GPU集群上评估FlexFlow的性能。

  This section evaluates the performance of FlexFlow on six real-world DNN benchmarks and two GPU clusters.

- 第2段借助表4介绍了实验中使用的3个DNN（CNN）。AlexNet、Inception-v3和ResNet-101是三个在ILSVRC比赛中获得最佳精度的。对于AlexNet，迭代训练时间小于从磁盘加载训练数据的时间。我们遵循TensorFlow benchmark中的建议，使用合成数据对AlexNet的性能进行基准测试。对于其他所有的实验，在训练过程中从磁盘加载训练数据。

  *Table 4 summarizes the DNNs used in our experiments. AlexNet, Inception-v3, and ResNet-101 are three CNNs that achieved the best accuracy in the ILSVRC competitions. For AlexNet, the per-iteration training time is smaller than the time to load training data from disk. We follow the suggestions in TensorFlow Benchmarks * and use synthetic data to benchmark the performance of AlexNet. For all other experiments, the training data is loaded from disk in the training procedure.*

Table 4. Details of the DNNs and datasets used in evaluation.

DNN	Description	Dataset	Reported Acc.	Our Acc.
Convolutional Neural Networks (CNNs)				
AlexNet	A 12-layer CNN	Synthetic data	-	-
Inception-v3	A 102-layer CNN with Inception modules (Szegedy et al., 2014)	ImageNet	78.0%[a]	78.0%[a]
ResNet-101	A 101-layer residual CNN with shortcut connections	ImageNet	76.4%[a]	76.5%[a]
Recurrent Neural Networks (RNNs)				
RNNTC	4 recurrent layers followed by a softmax layer	Movie Reviews (Movies)	79.8%	80.3%
RNNLM	2 recurrent layers followed by a softmax layer	Penn Treebank (Marcus et al.)	78.4[b]	76.1[b]
NMT	4 recurrent layers followed by an attention and a softmax layer	WMT English-German (WMT)	19.67[c]	19.85[c]

[a] top-1 accuracy for single crop on the validation dataset (higher is better).
[b] word-level test perplexities on the Peen Treebank dataset (lower is better).
[c] BLEU scores (Papineni et al., 2002) on the test dataset (higher is better).

8 Evaluation

- 第3段介绍了另外3个DNN（RNN）。RNNTC、RNNLM和NMT分别是用于文本分类、语言建模和神经机器翻译的序列到序列的RNN模型。为了提高模型的准确性，我们还在最后一个解码器LSTM层上使用了一个attention层。图13说明了NMT模型的结构。对于三个RNN模型设置每个循环层的展开步骤数为40。

RNNTC, RNNLM and NMT are sequence-to-sequence RNN models for text classification, language modeling, and neural machine translation, respectively. To improve model accuracy, we also use an attention layer on top of the last decoder LSTM layer (Bahdanau et al., 2014). Figure 13 illustrates the structure of the NMT model. For all three RNN models, we set the number of unrolling steps for each recurrent layer to 40.

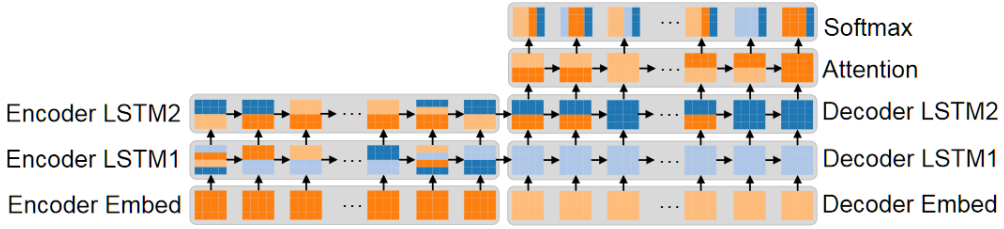


Figure 13. The best discovered strategy for parallelizing NMT on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each grey box denotes a layer, whose operators share the same network parameters. Each GPU is denoted by a color.

Table 4. Details of the DNNs and datasets used in evaluation.

DNN	Description	Dataset	Reported Acc.	Our Acc.
Convolutional Neural Networks (CNNs)				
AlexNet	A 12-layer CNN	Synthetic data	-	-
Inception-v3	A 102-layer CNN with Inception modules (Szegedy et al., 2014)	ImageNet	78.0%[a]	78.0%[a]
ResNet-101	A 101-layer residual CNN with shortcut connections	ImageNet	76.4%[a]	76.5%[a]
Recurrent Neural Networks (RNNs)				
RNNTC	4 recurrent layers followed by a softmax layer	Movie Reviews (Movies)	79.8%	80.3%
RNNLM	2 recurrent layers followed by a softmax layer	Penn Treebank (Marcus et al.)	78.4[b]	76.1[b]
NMT	4 recurrent layers followed by an attention and a softmax layer	WMT English-German (WMT)	19.67[c]	19.85[c]

[a] top-1 accuracy for single crop on the validation dataset (higher is better).
[b] word-level test perplexities on the Peen Treebank dataset (lower is better).
[c] BLEU scores (Papineni et al., 2002) on the test dataset (higher is better).

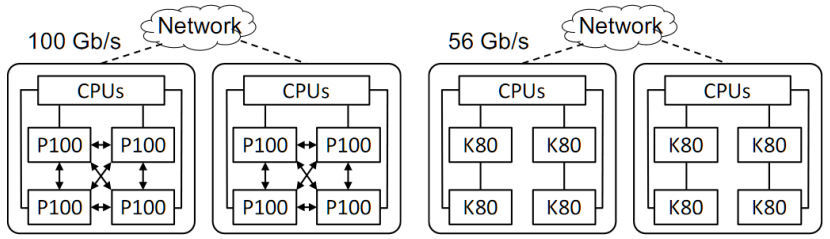
8 Evaluation

- 第4段介绍了训练的参数设置。跟着已有的工作构造算子图和设置超参数，在所有的DNN benchmark中使用同步训练，每个GPU的batch size为64，除了AlexNet，因为它的模型小得多，每个GPU的batch size为256。

  We follow prior work to construct operator graphs and set hyperparameters. We use synchronous training and a per-GPU batch size of 64 for all DNN benchmarks, except for AlexNet, which has a much smaller model and uses a per-GPU batch size of 256.

- 第5段详细地介绍了实验的硬件环境。

  ***To evaluate** the performance of FlexFlow with different device topologies, **we performed the experiments on two GPU clusters**, as shown in Figure 5. **The first cluster contains** 4 compute nodes, each of which is equipped with two Intel 10-core E5-2600 CPUs, 256GB main memory, and four NVIDIA Tesla P100 GPUs. GPUs on the same node are connected by NVLink, and nodes are connected over 100GB/s EDR Infiniband. **The second cluster consists of** 16 nodes, each of which is equipped with two Intel 10-core E52680 GPUs, 256GB main memory, and four NVIDIA Tesla K80 GPUs. Adjacent GPUs are connected by a separate PCI-e switch, and all GPUs are connected to CPUs through a shared PCI-e switch. Compute nodes in the cluster are connected over 56 GB/s EDR Infiniband.*



(a) The P100 Cluster (4 nodes). (b) The K80 Cluster (16 nodes).

Figure 5. Architectures of the GPU clusters used in the experiments. An arrow line indicates a NVLink connection. A solid line is a PCI-e connection. Dashed lines are Infiniband connections across different nodes.

8 Evaluation

- 第6段介绍了执行优化器的耗时。一般将30分钟设置为执行优化器的时间预算，并使用数据并行和随机生成的策略作为搜索的初始候选。如表5所示，在大多数情况下，搜索会在几分钟内结束。

  Unless otherwise stated, we set 30 minutes as the time budget for the execution optimizer and use data parallelism and a randomly generated strategy as the initial candidates for the search. As shown in Table 5, the search terminates in a few minutes in most cases.

Table 5. The end-to-end search time with different simulation algorithms (seconds).

Num.	AlexNet			ResNet			Inception			RNNTC			RNNLM			NMT		
GPUs	Full	Delta	Speedup	Full	Delta	Speedup	Full	Delta	Speedup	Full	Delta	Speedup	Full	Delta	Speedup	Full	Delta	Speedup
4	0.11	0.04	**2.9**×	1.4	0.4	**3.2**×	14	4.1	**3.4**×	16	7.5	**2.2**×	21	9.2	**2.3**×	40	16	**2.5**×
8	0.40	0.13	**3.0**×	4.5	1.4	**3.2**×	66	17	**3.9**×	91	39	**2.3**×	76	31	**2.5**×	178	65	**2.7**×
16	1.4	0.48	**2.9**×	22	7.3	**3.1**×	388	77	**5.0**×	404	170	**2.4**×	327	121	**2.7**×	998	328	**3.0**×
32	5.3	1.8	**3.0**×	107	33	**3.2**×	1746	298	**5.9**×	1358	516	**2.6**×	1102	342	**3.2**×	2698	701	**3.8**×
64	18	5.9	**3.0**×	515	158	**3.3**×	8817	1278	**6.9**×	4404	1489	**3.0**×	3406	969	**3.6**×	8982	2190	**4.1**×

8 Evaluation: 8.1 Parallelization Performance

8.1.1 Per-iteration Performance

- 实验8.1为了凸显出FlexFlow的并行性能
- 实验8.1.1以一个iteration为单位进行性能分析

- 第1段介绍了实验的框架方面的性能对比方法。将FlexFlow的每次迭代训练性能与以下baseline进行比较。数据并行是现有深度学习系统中常用的算法。为了控制实现的差异，在TensorFlow r1.7、PyTorch v0.3和我们的实现中运行了数据并行实验，并比较了性能数据。与TensorFlow和PyTorch相比，FlexFlow在所有6个DNN基准测试中都达到了相同或更好的性能数据，因此我们在实验中报告了使用FlexFlow实现的数据并行性能。

  We compare the per-iteration training performance of FlexFlow with the following baselines. Data parallelism is commonly used in existing deep learning systems. ***To control for implementation differences****, we ran data parallelism experiments in TensorFlow r1.7, PyTorch v0.3, and our implementation and compared the performance numbers. Compared to TensorFlow and PyTorch, FlexFlow achieves the same or better performance numbers on all six DNN benchmarks, and* ***therefore*** *we report the data parallelism performance achieved by FlexFlow in the experiments.*

8 Evaluation: 8.1 Parallelization Performance

8.1.1 Per-iteration Performance

- 第2段介绍了实验的并行策略方面的性能对比方法。专家设计的策略基于领域知识和经验来优化并行：对于CNN，Krizhevsky使用数据并行来并行化卷积层和池化层，并对密集连接层切换到模型并行性；对于RNN，Wu使用数据并行在每个计算节点上复制整个算子图，并使用模型并行将相同深度的运算符分配到各个节点中相同的GPU上。这些专家设计的策略被用作实验的baseline。模型并行本身只揭露了有限的并行性，我们将模型并行作为专家设计策略的一部分进行比较。

  Expert-designed strategies optimize parallelization based on domain experts' knowledge and experience. For CNNs, (Krizhevsky, 2014) uses data parallelism for parallelizing convolutional and pooling layers and switches to model parallelism for densely-connected layers. For RNNs, (Wu et al., 2016) uses data parallelism that replicates the entire operator graph on each compute node and uses model parallelism that assign operators with the same depth to the same GPU on each node. These expert-designed strategies are used as a baseline in our experiments. Model parallelism only exposes limited parallelism by itself, and we compare against model parallelism as a part of these expert-designed strategies.

8 Evaluation: 8.1 Parallelization Performance

8.1.1 Per-iteration Performance

- 第3段展示了FlexFlow在6个DNN benchmark上的每个iteration训练的性能（下页图6）。对于ResNet-101，FlexFlow发现了类似于数据并行的策略(除了对最后一个全连接层使用单个节点上的模型并行)，因此获得了类似的并行性能。对于其他DNN benchmark，FlexFlow找到了比baseline更有效的策略，实现了1.3-3.3x的加速。需要注意的是，FlexFlow执行的算子与数据并行和专家设计的策略相同，性能的提高是通过使用更快的并行策略实现的。我们发现与数据并行和专家设计的策略相比，FlexFlow发现的并行策略有两个优势。

  Figure 6 shows the per-iteration training performance on all six DNN benchmarks. ***For ResNet-101****, FlexFlow finds strategies similar to data parallelism (except using model parallelism on a single node for the last fully-connected layer) and therefore achieves similar parallelization performance.* ***For other DNN benchmarks****, FlexFlow finds more efficient strategies than the baselines and achieves 1.3-3.3× speedup.* ***Note that*** *FlexFlow performs the same operators as data parallelism and expert-designed strategies, and the performance improvement is achieved by using faster parallelization strategies. We found that the parallelization strategies discovered by FlexFlow have two advantages over data parallelism and expert-designed strategies.*

8 Evaluation: 8.1 Parallelization Performance

8.1.1 Per-iteration Performance

- 对于ResNet-101，所发现的最优策略与数据并行几乎相同，加速不明显
- 对于其他DNN，都获得了1.3-3.3x的加速
- FlexFlow相比baseline有两个优势：
  - 降低了整体的通信成本
  - 减少了整体的任务计算时间

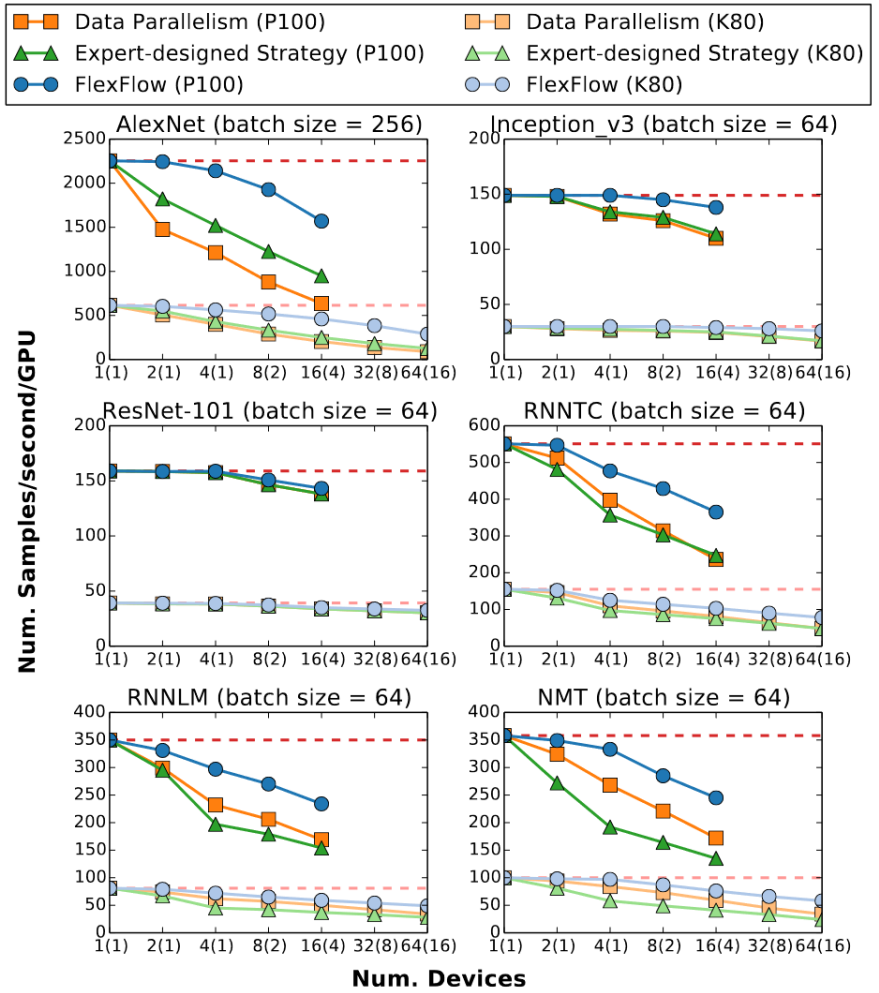


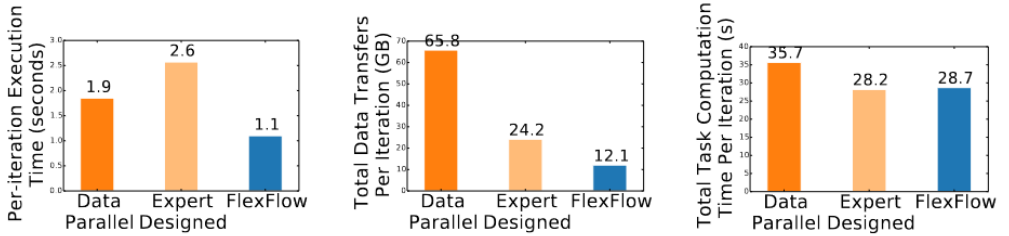
Figure 6. Per-iteration training performance on six DNNs. Numbers in parenthesis are the number of compute nodes used in the experiments. The dash lines show the ideal training throughput.

8 Evaluation: 8.1 Parallelization Performance

8.1.1 Per-iteration Performance

- **降低通信开销：FlexFlow分布式运行时支持重叠数据传输，并通过计算来隐藏通信开销。然而，随着设备数量的增加，通信开销会增加，但用于隐藏通信的计算时间保持不变。因此，降低整体通信成本有利于大规模分布式培训。从图7b可以看出，在64个K80 GPU(16个节点)上并行化NMT模型，与其他并行化方法相比，FlexFlow每次迭代的数据传输减少了2-5.5×。**

  ***However**, as we scale the number of devices, the communication overheads increase, but the computation time used to hide communication remains constant. **Therefore**, reducing overall communication costs is beneficial for large-scale distributed training. Figure 7b shows that, to parallelize the NMT model on 64 K80 GPUs (16 nodes), FlexFlow reduces the per-iteration data transfers by 2-5.5× compared to other parallelization approaches.*



(a) Per-iteration execution time. (b) Overall data transfers per iteration. (c) Overall task run time per iteration.

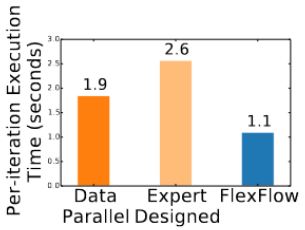
Figure 7. Parallelization performance for NMT on 64 K80 GPUs (16 nodes). FlexFlow reduces per-iteration execution time by 1.7-2.4× and data transfers by 2-5.5× compared to other approaches. FlexFlow achieves similar overall task computation time as expert-designed strategy, which is 20% fewer than data parallelism.

8 Evaluation: 8.1 Parallelization Performance

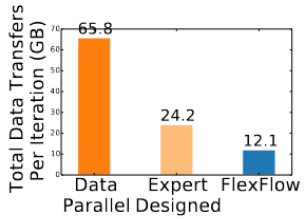
8.1.1 Per-iteration Performance

- **减少计算时间：**数据并行总是将批处理维度中的运算符并行化。然而，正如Jia所报道的，通过不同维度并行化一个运算符会导致不同的任务计算时间。
- **例子：**对于NMT模型中的矩阵乘法运算符，在通道维上并行化比在批维上并行化减少了38%的运算时间。
- **结果对比：**图7c显示，与NMT模型的数据并行相比，FlexFlow减少了20%的总体任务计算时间。
- **好像没有明显优势？：**专家设计的策略比FlexFlow的总任务计算时间略长。
- **从其他地方找找优势：**但专家设计的策略是通过在每个节点上使用模型并行来实现的，这将会禁用算子内的任何并行性，并导致不平衡的工作负载。
- **还是有明显的优势！：**因此，专家设计策略的执行性能甚至比数据并行更差(参见图7a)。
- **总结并且夸一下自己的工作：**FlexFlow减少了任务计算时间，实现了算子的并行性，并保持了负载均衡。

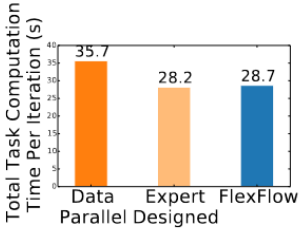
Data parallelism always para
*2018), **parallelizing an ope***
***time**.*
***However**, this is achieved b*
each operator and results in
even worse execution perfo
computation time while ena



(a) Per-iteration execution time.



(b) Overall data transfers per iteration.



(c) Overall task run time per iteration.

Figure 7. Parallelization performance for NMT on 64 K80 GPUs (16 nodes). FlexFlow reduces per-iteration execution time by 1.7-2.4× and data transfers by 2-5.5× compared to other approaches. FlexFlow achieves similar overall task computation time as expert-designed strategy, which is 20% fewer than data parallelism.

*er, as reported in (Jia et al., **different task computation***

bles any parallelism within
igned strategy achieves
***ow** reduces the task*
g load balance.

8 Evaluation: 8.1 Parallelization Performance

8.1.2 End-to-end Performance

- 实验8.1.2对比了端到端训练的性能——也即以一次训练作为对比单位

- 第1段说明了FlexFlow的性能没有牺牲准确性。FlexFlow对DNN模型执行与其他深度学习系统相同的计算，因此实现了相同的模型精度。表4验证了FlexFlow在实验中使用的DNN benchmark上达到了state-of-the-art的准确性。

FlexFlow performs the same computation as other deep learning systems for a DNN model and ***therefore*** *achieves the same model accuracy. Table 4 verifies that FlexFlow achieves the state-of-the-art accuracies on the DNN benchmarks used in the experiments.*

Table 4. Details of the DNNs and datasets used in evaluation.

DNN	Description	Dataset	Reported Acc.	Our Acc.
Convolutional Neural Networks (CNNs)				
AlexNet	A 12-layer CNN	Synthetic data	-	-
Inception-v3	A 102-layer CNN with Inception modules (Szegedy et al., 2014)	ImageNet	78.0%[a]	78.0%[a]
ResNet-101	A 101-layer residual CNN with shortcut connections	ImageNet	76.4%[a]	76.5%[a]
Recurrent Neural Networks (RNNs)				
RNNTC	4 recurrent layers followed by a softmax layer	Movie Reviews (Movies)	79.8%	80.3%
RNNLM	2 recurrent layers followed by a softmax layer	Penn Treebank (Marcus et al.)	78.4[b]	76.1[b]
NMT	4 recurrent layers followed by an attention and a softmax layer	WMT English-German (WMT)	19.67[c]	19.85[c]

[a] top-1 accuracy for single crop on the validation dataset (higher is better).
[b] word-level test perplexities on the Peen Treebank dataset (lower is better).
[c] BLEU scores (Papineni et al., 2002) on the test dataset (higher is better).

8 Evaluation: 8.1 Parallelization Performance

8.1.2 End-to-end Performance

- 第2段对比了FlexFlow与TensorFlow在Inception-v3上的端到端训练性能。我们在ImageNet数据集上训练Inception-v3，直到模型在验证集上达到72%的single-crop top-1精度。两种框架的训练过程都使用学习率为0.045、权重衰减为0.0001的随机梯度下降(SGD)。图8展示了两个系统的训练曲线，显示FlexFlow与TensorFlow相比减少了38%的训练时间。

  In this experiment, ***we compare the end-to-end training performance between FlexFlow and TensorFlow on Inceptionv3****. We train Inception-v3 on the ImageNet dataset until the model reaches the single-crop top-1 accuracy of 72% on the validation set. The training processes in both frameworks use stochastic gradient decent (SGD) with a learning rate of 0.045 and a weight decay of 0.0001. Figure 8 illustrates the training curves of the two systems and show that FlexFlow* ***reduces the training time by 38%*** *compared to TensorFlow.*

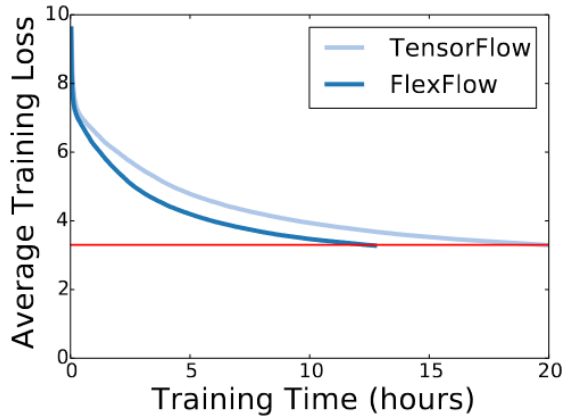


Figure 8. Training curves of Inception-v3 in different systems. The model is trained on 16 P100 GPUs (4 nodes).

8 Evaluation: 8.1 Parallelization Performance

8.1.3 Automated Frameworks

- 第1段介绍8.1.3实验的目的：与两种自动框架进行比较，它们可以在有限的搜索空间中找到并行化策略。

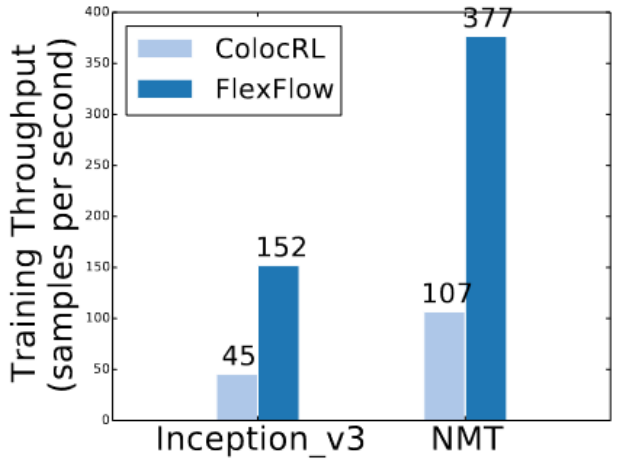
  *We **compare against** two automated frameworks that find parallelization strategies in a limited search space.*

- 第2-4段与ColocRL进行对比。
- 第2段介绍实验方法。ColocRL使用强化学习来学习设备放置的模型并行性，我们不知道ColocRL的任何公开的实现，所以我们按照论文中报道的Inception-v3和NMT的设备放置，并在相同机器上进行实验。

  *We are **not aware of any publicly available implementation of ColocRL**, so we compare against the learned device placement for Inception-v3 and NMT, as reported in the paper, and performed the experiments on the same machine.*

- 第3段介绍性能对比结果。图9a（右图）比较了在单个节点的4个K80 GPU上，FlexFlow和ColocRL发现的策略的训练吞吐量。FlexFlow发现的并行化策略相比ColocRL实现了3.4-3.8×的加速。性能的提高归功于FlexFlow探索了更大的搜索空间。

  The parallelization strategies found by FlexFlow achieve 3.4 - 3.8× speedup compared to ColocRL. We **attribute the performance improvement to** the larger search space explored by FlexFlow.



(a) ColocRL

8 Evaluation: 8.1 Parallelization Performance

8.1.3 Automated Frameworks

- 第2-4段与ColocRL进行对比。
- 第4段介绍FlexFlow除了训练性能以外的两个优势：
  ① 查找速度快：ColocRL需要在硬件环境中执行每种策略来获得激励信号，需要12-27个小时才能找到最佳位置，而FlexFlow在14-40秒内就能找到这些执行的高效并行策略
  ② 需要资源少：ColocRL使用多达160个计算节点(每个节点上有4个gpu)来及时查找位置，而FlexFlow使用单个计算节点来运行执行优化器。

Besides *improving training performance, FlexFlow has* ***two additional advantages over ColocRL****. First, ColocRL requires executing each strategy in the hardware environment to get reward signals and takes 12-27 hours to find the best placement, while FlexFlow finds efficient parallelization strategies for these executions in 14-40 seconds. Second, ColocRL uses up to 160 compute nodes (with 4 GPUs on each node) to find the placement in time, while FlexFlow uses a single compute node to run the execution optimizer.*

8 Evaluation: 8.1 Parallelization Performance

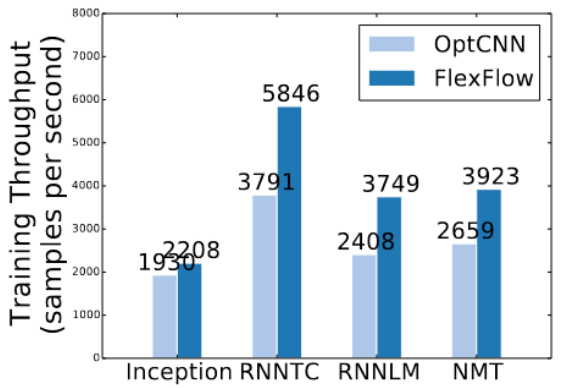
8.1.3 Automated Frameworks

- 第5-6段与OptCNN进行对比。
- 第5段介绍实验方法。为了评估OptCNN在非线性RNN上的性能，我们显式地将所有共享相同参数的循环节点融合到一个单一的算子。

  OptCNN (Jia et al., 2018) uses dynamic programming to parallelize linear DNNs. To evaluate OptCNN's performance on non-linear RNNs, we explicitly fuse all recurrent nodes sharing the same parameters to a single operator.

- 第6段介绍实验结果。比较FlexFlow和OptCNN在16 P100 GPU上不同DNN的性能。FlexFlow和OptCNN对使用线性算子图的AlexNet和ResNet发现了相同的并行化策略，对其他DNN发现了不同的策略，如图9b所示。对于这些具有非线性运算符图的DNN, FlexFlow通过利用不同运算符之间的并行性，实现了相较于OptCNN 1.2-1.6x的加速。

  FlexFlow and OptCNN found the same parallelization strategies for AlexNet and ResNet ***with linear operator graphs*** *and found different strategies for the other DNNs as shown in Figure 9b*



(b) OptCNN

8 Evaluation: 8.2 Execution Simulator

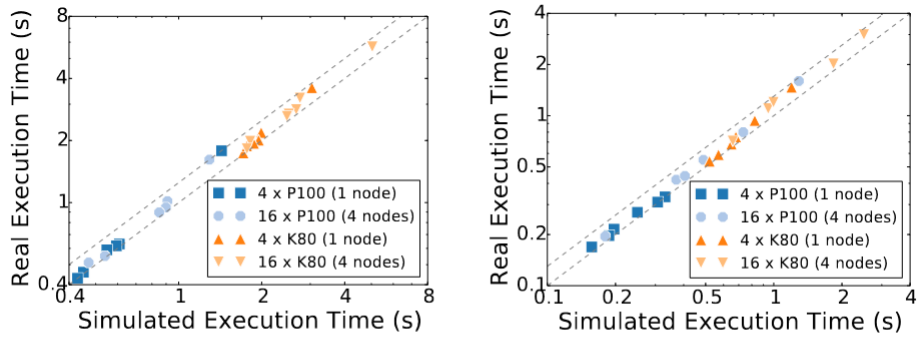
- 实验8.2用于展示执行模拟器的性能

- 第1段介绍评估方法。我们使用两个指标来评估模拟器的性能：模拟器精度和模拟器执行时间。

  We evaluate the performance of the simulator using two metrics: simulator accuracy and simulator execution time.

- 第2段介绍指标一：模拟器精度。
- 首先将执行模拟器预测的估计执行时间与实际测量的执行时间进行比较。图10显示了不同DNN和不同可用设备的结果。虚线分别表示0%和30%的相对差异，这涵盖了实际执行时间和预测执行时间之间的差异。
- 此外，对于具有相同算子图和设备拓扑(即图中形状相同的点)的不同并行化策略，其模拟执行时间保持了实际执行时间的顺序，这表明模拟执行时间是评估不同策略性能的合适指标。

*We **first compare** the estimated execution time predicted by the execution simulator with the real execution time measured by actual executions. Figure 10 shows the results for different DNNs and different available devices. **The dashed lines indicate** a relative difference of 0% and 30%, respectively, which encompasses the variance between actual and predicted execution time. **In addition**, for different parallelization strategies with the same operator graph and device topology (i.e., points of the same shape in the figure), **their simulated execution time preserves actual execution time ordering, which shows that simulated execution time is an appropriate metric to evaluate the performance of different strategies**.*



(a) Inception-v3 (b) NMT

Figure 10. Comparison between the simulated and actual execution time for different DNNs and device topologies.

8 Evaluation: 8.2 Execution Simulator

- 第3-4段介绍指标二：模拟器执行时间。
- 图11显示了在4个节点的16个P100 GPU上，不同仿真算法寻找NMT模型策略的搜索性能。完整仿真算法和增量（delta）仿真算法分别在16分钟和6分钟内结束。当允许时间预算小于8分钟时，全仿真算法会比增量仿真算法找到更糟糕的策略。（通过更改限定条件进一步凸显delta仿真算法的优越性）

  If the allowed time budget is less than 8 minutes, *the full simulation algorithm will find a worse strategy than the delta simulation algorithm.*
- 第4段进一步比较了不同模拟算法下执行优化器的端到端搜索时间。对于给定的DNN模型和设备拓扑，使用10种随机初始策略测量优化器的平均执行时间。结果如表5所示。delta仿真算法比全仿真算法快2.2~6.9倍。此外，随着设备数量的增加，整个模拟算法的加速速度也会增加。

  We compare the end-to-end search time of the execution optimizer with different simulation algorithms. ***Moreover***, *the speedup over the full simulation algorithm increases as we scale the number of devices.*

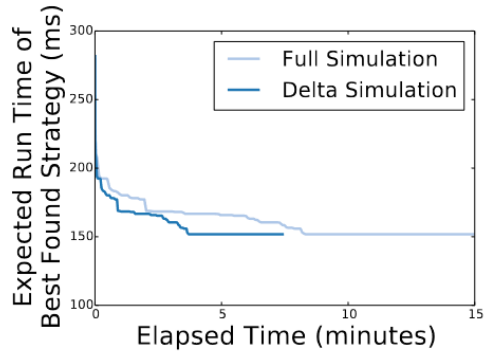


Figure 11. Search performance with the full and delta simulation algorithms for the NMT model on 16 P100 GPUs (4 nodes).

Table 5. The end-to-end search time with different simulation algorithms (seconds).

Num.	AlexNet			ResNet			Inception			RNNTC			RNNLM			NMT		
GPUs	Full	Delta	Speedup	Full	Delta	Speedup	Full	Delta	Speedup	Full	Delta	Speedup	Full	Delta	Speedup	Full	Delta	Speedup
4	0.11	0.04	**2.9**×	1.4	0.4	**3.2**×	14	4.1	**3.4**×	16	7.5	**2.2**×	21	9.2	**2.3**×	40	16	**2.5**×
8	0.40	0.13	**3.0**×	4.5	1.4	**3.2**×	66	17	**3.9**×	91	39	**2.3**×	76	31	**2.5**×	178	65	**2.7**×
16	1.4	0.48	**2.9**×	22	7.3	**3.1**×	388	77	**5.0**×	404	170	**2.4**×	327	121	**2.7**×	998	328	**3.0**×
32	5.3	1.8	**3.0**×	107	33	**3.2**×	1746	298	**5.9**×	1358	516	**2.6**×	1102	342	**3.2**×	2698	701	**3.8**×
64	18	5.9	**3.0**×	515	158	**3.3**×	8817	1278	**6.9**×	4404	1489	**3.0**×	3406	969	**3.6**×	8982	2190	**4.1**×

8 Evaluation: 8.3 Search Algorithm

- 实验8.3用于展示搜索算法的优秀

- 将发现的最佳策略与全局最优策略在small executions下进行比较。为了获得一个合理大小的搜索空间，我们将设备数量限制为4个，并考虑以下两个DNN。①LeNet是一个6层的CNN；②RNNLM的一种变体，其中每个循环层的展开步骤数被限制在2步以内。使用深度优先搜索来探索空间，并使用A*来剪枝。寻找LeNet和RNNLM的最优策略分别需要0.8和18个小时。对于两个DNN, FlexFlow在不到1秒的时间内找到了相同的全局最优策略。

  *We compare the best discovered strategies with the global optimal strategies for small executions. **To obtain a search space of reasonable size**, we limit the number of devices to 4 and consider the following two DNNs. LeNet (LeCun, 2015) is a 6-layer CNN. The second DNN is a variant of RNNLM where the number of unrolling steps for each recurrent layer is restricted to 2. We use depth-first search to explore the space and use A* (Cormen et al., 2009) to prune the search. Finding the optimal strategies for LeNet and RNNLM took 0.8 and 18 hours, respectively. For both DNNs, FlexFlow finds the same global optimal strategy in less than 1 second.*

① 硬件规模小
② DNN网络小

8 Evaluation: 8.4 Case Studies

- 本小节通过两个Case Studies展示FlexFlow搜索到的最优策略——Inception-v3和NMT。
- Inception-v3：图12显示了在四个P100 GPU上并行化Inception-v3的最佳策略，它利用了关键路径上的算子的内部并行性，并对不同分支上的算子使用算子间和算子内并行性的组合。这将产生平衡良好的工作负载，并减少用于参数同步的数据传输。与数据并行性相比，该策略降低了75%的参数同步成本和12%的每次迭代执行时间。
- 为了在4个具有非对称连接的GPU(见图5b)上并行化相同的Inception-v3模型，我们观察到最好的策略倾向于在相邻的具有直接连接的GPU上并行化操作，以降低通信成本。

*This **results in** a well-balanced workload and reduces data transfers for parameter synchronization. **Compared to** data parallelism, this strategy reduces the parameter synchronization costs by 75% and the per-iteration execution time by 12%.*

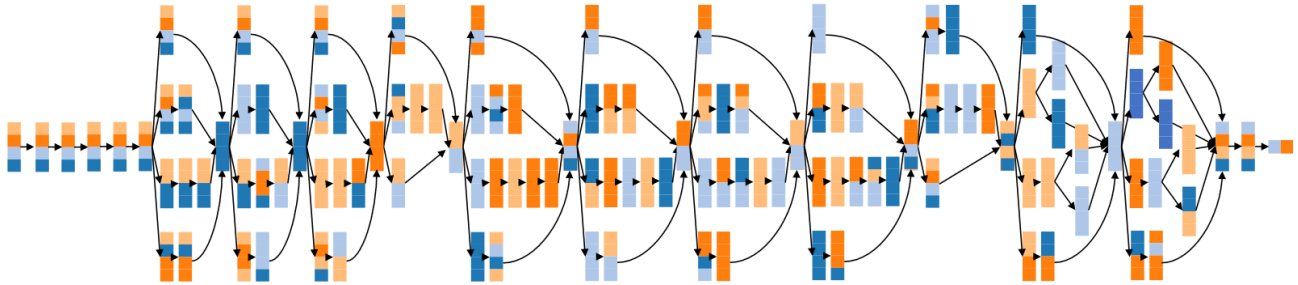
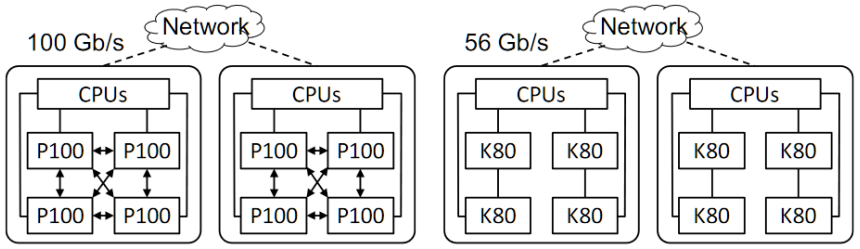


Figure 12. The best discovered strategy for parallelizing Inception-v3 on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each GPU is denoted by a color.



(a) The P100 Cluster (4 nodes). (b) The K80 Cluster (16 nodes).

Figure 5. Architectures of the GPU clusters used in the experiments. An arrow line indicates a NVLink connection. A solid line is a PCI-e connection. Dashed lines are Infiniband connections across different nodes.

8 Evaluation: 8.4 Case Studies

- 本小节通过两个Case Studies展示FlexFlow搜索到的最优策略——Inception-v3和NMT 。
- NMT：图13显示了在4个P100 GPU上并行化NMT的最佳策略。首先，对于网络参数多、计算量少的层(如嵌入层)，在单个GPU上进行计算，消除参数同步；其次，对于参数量大、计算量大的层(如softmax层)，FlexFlow在参数维度上使用并行性，将参数子集的计算分配给每个任务，这减少了参数同步成本，同时保持负载平衡。第三，对于多个循环层(如LSTM和注意层)，FlexFlow使用不同层之间的并发性以及每个算子之间的并行性来减少参数同步成本，同时保持负载均衡。

Figure 13 shows the best discovered strategy for parallelizing NMT on four P100 GPUs. ***First****, for a layer with a large number of network parameters and little computation…….* ***Second****, for a layer with a large number of parameters and heavy computation …….* ***Third****, for multiple recurrent layers…….*

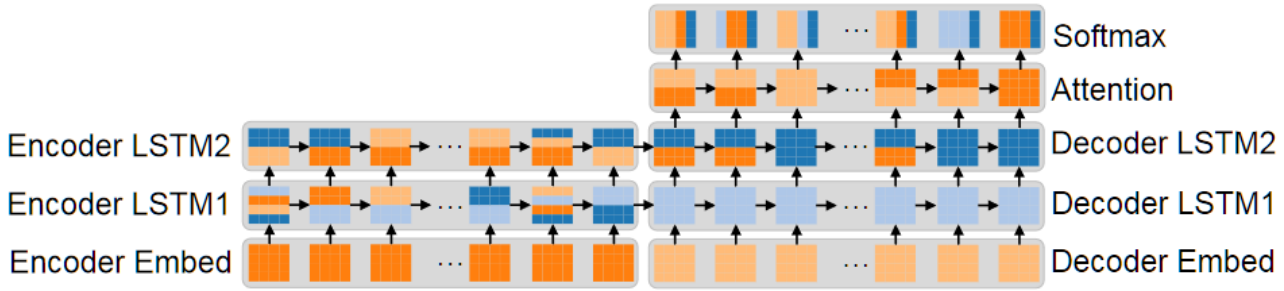


Figure 13. The best discovered strategy for parallelizing NMT on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each grey box denotes a layer, whose operators share the same network parameters. Each GPU is denoted by a color.

8 Evaluation——Summary

- 实验8.1为了凸显出FlexFlow的并行性能
  - 实验8.1.1以一个iteration为单位进行性能分析
  - 实验8.1.2以一次训练为单位进行性能分析
  - 实验8.1.3与两种自动框架进行比较

小 ↓ 大

FlexFlow 整体

=

- 实验8.2用于展示执行模拟器的性能

执行模拟器 部分

+

- 实验8.3用于展示搜索算法的优秀

搜索算法 部分

- 实验8.4给出Case Study帮助理解

9 Conclusion

- 论文工作：本文介绍了FlexFlow深度学习系统
- 主要功能：该系统可以在SOAP搜索空间中自动找到高效的并行化策略，用于DNN训练
- 实现方式：FlexFlow使用一种引导随机搜索过程来探索空间，并包括一个执行模拟器，它是DNN性能的高效和准确预测器
- 评估方法与结果：我们在两个GPU集群上用六个真实的DNN基准测试来评估FlexFlow，结果显示FlexFlow的性能明显优于最先进的并行方法。

This paper presents FlexFlow, a deep learning system that automatically finds efficient parallelization strategies in the SOAP search space for DNN training. FlexFlow uses a guided randomized search procedure to explore the space and includes an execution simulator that is an efficient and accurate predictor of DNN performance. We evaluate FlexFlow with six real-world DNN benchmarks on two GPU clusters and show FlexFlow significantly outperforms state-of-the-art parallelization approaches.

□ 一些感想

- 进行设计和实验时要为课题的目的服务，切忌舍本逐末，忽视了最核心的需求。
  - 比如FlexFlow的执行时间预估准确率，误差全在30%以内，作者一方面没有采取更讨巧的“平均准确率”，另一方面也没有在这方面盲目地追求更高的准确率。因为对于这项研究——并行策略的自动调优框架来说，需求仅仅是预估的时间与实际的时间拥有一致的趋势而已，准确率更高当然锦上添花，但准确率更低也不影响设计的成功。

- 在进行一个系统的测试与分析时，可以仿照这篇论文，先对整体的性能进行不同层次的评估（由小到大），然后再分别对系统的各个重要组件进行评估分析

- 本文的设计也是在已有工作OptCNN的基础之上进行的，站在现在FlexFlow的角度来看OptCNN当然有很多的不足，但如果不是作者真的有过之前的设计经验，或许也不会对系统理解得这么深。所以不要害怕自己的想法没有突破性进展，真正脚踏实地去做了或许会在尝试和迭代的过程中得到更深的感悟

- 本文的用词比较有趣，一方面会吊读者胃口，换着花样夸自己的工作；另一方面又很谦虚，把自己对现有框架不足之处的改进说成是non-trivial的

- 第1段介绍了现有深度学习系统的不足。我们发现，现有的深度学习系统(如TensorFlow、PyTorch、Caffe2和MXNet)仅支持通过数据并行在样本维度上并行化一个算子，而在这些系统中并行化其他维度上的算子或多个SOAP维度的组合是不平凡的。
  - 从其他地方找找优势：但专家设计的策略是通过在每个节点上使用模型并行来实现的，这将会禁用算子内的任何并行性，并导致不平衡的工作负载。
  - 还是有明显的优势！：因此，专家设计策略的执行性能甚至比数据并行更差(参见图7a)。
  - 总结并且夸一下自己的工作：FlexFlow减少了任务计算时间，实现了算子的并行性，并保持了负载均衡。