# BEYOND DATA AND MODEL PARALLELISM FOR DEEP NEURAL NETWORKS

**Zhihao Jia** [1] **Matei Zaharia** [1] **Alex Aiken** [1]

## ABSTRACT

Existing deep learning systems commonly parallelize deep neural network (DNN) training using data or model parallelism, but these strategies often result in suboptimal parallelization performance. We introduce SOAP, a more comprehensive search space of parallelization strategies for DNNs that includes strategies to parallelize a DNN in the Sample, Operator, Attribute, and Parameter dimensions. We present FlexFlow, a deep learning engine that uses guided randomized search of the SOAP space to find a fast parallelization strategy for a specific parallel machine. To accelerate this search, FlexFlow introduces a novel execution simulator that can accurately predict a parallelization strategy's performance and is three orders of magnitude faster than prior approaches that execute each strategy. We evaluate FlexFlow with six real-world DNN benchmarks on two GPU clusters and show that FlexFlow increases training throughput by up to 3.3× over state-of-the-art approaches, even when including its search time, and also improves scalability.

## 1 Introduction

As deep learning methods have evolved, DNN models have gotten progressively larger and more computationally expensive to train. As a result, it is now standard practice to parallelize DNN training across distributed heterogeneous clusters (Dean et al., 2012; Abadi et al., 2016). Although DNN models and the clusters used to parallelize them are increasingly complex, the strategies used by today's deep learning frameworks (e.g., TensorFlow, Caffe2, and MXNet) to parallelize training remain simple, and often suboptimal.

The most common parallelization strategy is *data parallelism* (Krizhevsky et al., 2012), which places a replica of the entire neural network on each device, so that each device processes a subset of the training data, and synchronizes network parameters across replicas at the end of an iteration. 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).

Another common parallelization strategy is *model parallelism* (Dean et al., 2012), 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. ColocRL (Mirhoseini et al., 2017) uses reinforcement learning to learn efficient operator assignments for model parallelism but only explores parallelism in the operator dimension.

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.

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).

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.

[1]Stanford University. Correspondence to: Zhihao Jia <zhihao@cs.stanford.edu>.

*Proceedings of the $2^{nd}$ SysML Conference*, Palo Alto, CA, USA, 2019. Copyright 2019 by the author(s).

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.

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. We borrow the idea from OptCNN of measuring the performance of an operator once for each configuration and feed these measurements into a *task graph* that models both the architecture of a DNN model and the network topology of a cluster. The execution simulator estimates the performance of a parallelization strategy by simulating the execution on the task graph. 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×.

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%. 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.

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. The execution simulator can also work with other search strategies, such as learning-based search algorithms. When the search procedure is finished, the execution optimizer returns the best strategy it has discovered.

We evaluate FlexFlow on a variety of real-world DNNs including AlexNet (Krizhevsky et al., 2012), ResNet-101 (He et al., 2016), Inception-v3 (Szegedy et al., 2016), RNN Text Classification (Kim, 2014), RNN Language Modeling (Zaremba et al., 2014) and Neural Machine Translation (Wu et al., 2016). 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.

*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.

## 2 Related Work

**Data and model parallelism** have been widely used by existing deep learning systems to distribute training across devices. Data parallelism (Krizhevsky et al., 2012) is inefficient for operators with a large number of parameters (e.g., densely-connected layers) and becomes a scalability bottleneck in large scale distributed training. Model parallelism (Dean et al., 2012) splits a DNN into disjoint subsets and trains each subset on a dedicated device, which reduces communication costs for synchronizing network parameters but exposes limited parallelism.

**Expert-designed parallelization strategies** manually optimize parallelization for specific DNNs by using experts' domain knowledge and experience. For example, Krizhevsky (2014) introduces "one weird trick" that uses data parallelism for convolutional and pooling layers and switches to model parallelism for densely-connected layers to accelerate CNNs. To parallelize RNNs, Wu et al. (2016) uses data parallelism that replicates the entire RNN on each node and switches to model parallelism for intra-node parallelization. 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×.

**Automated frameworks** have been proposed for finding efficient parallelization strategies in a limited search space. ColocRL (Mirhoseini et al., 2017) uses reinforcement learning to find efficient device placement for model parallelism. OptCNN (Jia et al., 2018) uses dynamic programming to parallelize linear CNNs. OptCNN's approach does not explore parallelism across operators and is not applicable to DNNs with non-linear computation graphs. Gao et al. (2017; 2019) exploited hybrid parallelization on tiled domain-specific

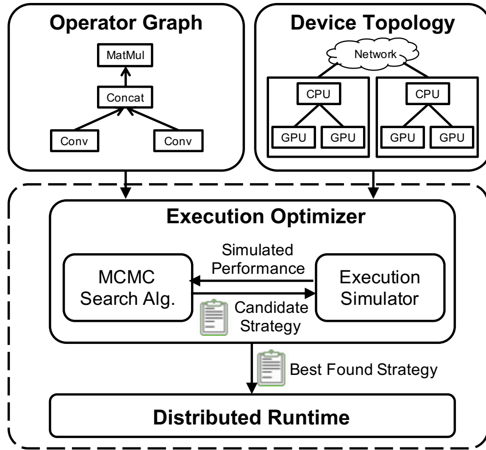



*Figure 1.* FlexFlow overview.

hardware and proposed various dataflow optimizations for both intra-layer and inter-layer data communication.

Table 1 summarizes the parallelism dimensions explored by existing approaches. Data parallelism uses the sample dimension to parallelize training, while model parallelism exploits the parameter and operator dimensions. Expert-designed strategies exploit parallelism in the sample or parameter dimension to parallelize an operator 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.

**Graph-based cluster schedulers.** 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. 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

Similar to existing deep learning frameworks (e.g., TensorFlow and PyTorch), FlexFlow uses an *operator graph* $\mathcal{G}$ to describe all operators and state in a DNN. Each node $o_i \in \mathcal{G}$ is an operator (e.g., matrix multiplication, convolution, etc.), and each edge $(o_i, o_j) \in \mathcal{G}$ is a tensor (i.e., a $n$-dimensional array) that is an output of $o_i$ and an input of $o_j$. In addition, FlexFlow also takes a *device topology* graph $\mathcal{D} = (\mathcal{D}_N, \mathcal{D}_E)$ describing all available hardware devices and their interconnections, as shown in Figure 1. Each node $d_i \in \mathcal{D}_N$ represents a device (e.g., a CPU or a GPU), and each edge $(d_i, d_j) \in \mathcal{D}_E$ is a hardware connection (e.g., a NVLink, a PCI-e, or a network link) between device $d_i$ and $d_j$. The edges are labeled with the bandwidth and latency of the connection.

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.

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.

*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 |

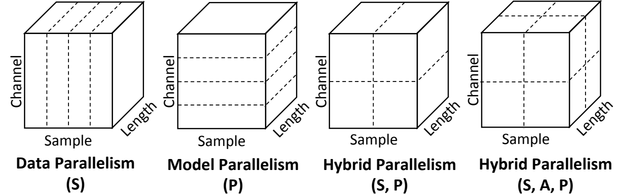



*Figure 2.* Example parallelization configurations for 1D convolution. Dashed lines show partitioning the tensor.

## 4 The SOAP Search Space

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 $o_i$ by defining how the output tensor of $o_i$ is partitioned.

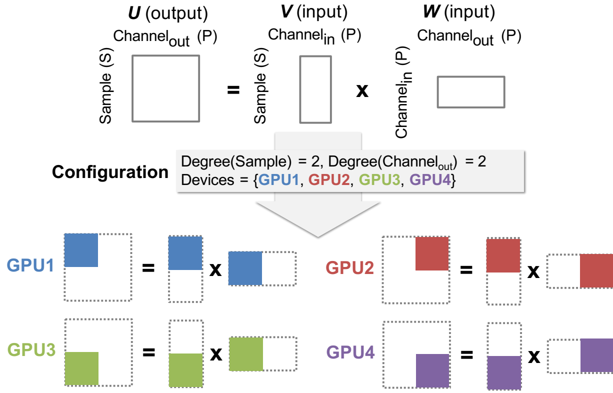



*Figure 3.* An example parallelization configuration for a matrix multiplication operator.

For an operator $o_i$, we define its *parallelizable dimensions* $\mathcal{P}_i$ as the set of all divisible dimensions in its output tensor. $\mathcal{P}_i$ always includes a *sample* dimension. For all other dimensions in $\mathcal{P}_i$, 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.

A *parallelization configuration* $c_i$ of an operator $o_i$ 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.

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

Figure 3 shows an example parallelization configuration for a matrix multiplication operator (i.e., $U = VW$). 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.

A *parallelization strategy* $\mathcal{S}$ describes one possible parallelization of an application. $\mathcal{S}$ includes a parallelization configuration $c_i$ for each operator $o_i$, and each $o_i$'s configuration can be chosen independently from among all possible configurations for $o_i$.

# 5 Execution Simulator

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

The simulator depends on the following assumptions:

**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 $(d_i, d_j)$ between device $d_i$ and $d_j$ with bandwidth $b$, transferring a tensor of size $s$ from $d_i$ to $d_j$ 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.

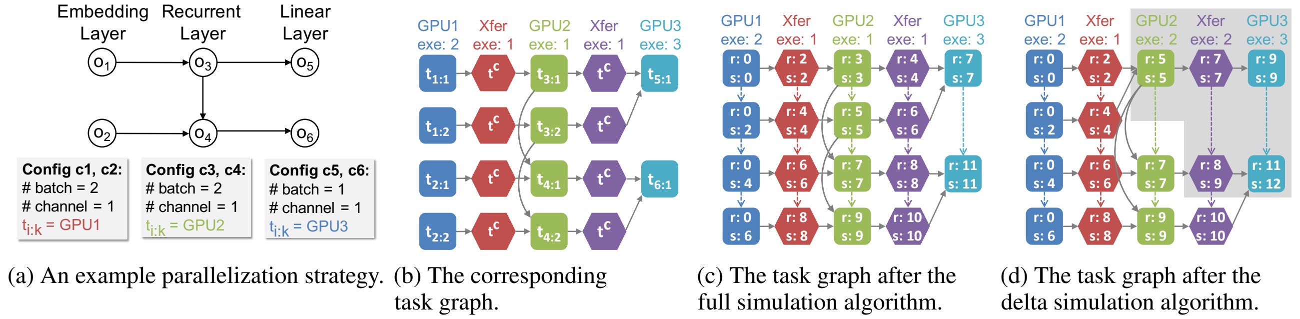
To simulate an execution, we borrow the idea from OptCNN (Jia et al., 2018) to measure the execution time of each distinct operator once for each configuration and include these measurements in a *task graph*, which includes all tasks derived from operators and dependencies between tasks. The simulator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

## 5.1 Task Graph

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.

Given an operator graph $\mathcal{G}$, a device topology $\mathcal{D}$, and a parallelization strategy $\mathcal{S}$, we use the following steps to construct a task graph $\mathcal{T} = (\mathcal{T}_N, \mathcal{T}_E)$, where each node $t \in \mathcal{T}_N$ is a task (i.e., a normal task or a communication task) and each edge $(t_i, t_j) \in \mathcal{T}_E$ is a dependency that task $t_j$ cannot start until task $t_i$ 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.

1. For each operator $o_i \in \mathcal{G}$ with parallelization configuration $c_i$, we add tasks $t_{i:1}, ..., t_{i:|c_i|}$ to $\mathcal{T}_N$.
2. For each tensor $(o_i, o_j) \in \mathcal{G}$, which is an output of operator $o_i$ and an input of $o_j$, we compute the output sub-tensors written by tasks $t_{i:k_i}$ $(1 \leq k_i \leq |c_i|)$ and the



(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`.

*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. |
| $\mathcal{I}(t)$ | $\{t_{in} \vert (t_{in}, t) \in \mathcal{T}_E\}$ |
| $\mathcal{O}(t)$ | $\{t_{out} \vert (t, t_{out}) \in \mathcal{T}_E\}$ |
| 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`. |

input sub-tensors read by tasks $t_{j:k_j}$ $(1 \leq k_j \leq |c_j|)$. For every task pair $t_{i:k_i}$ and $t_{j:k_j}$ with shared tensors, if two tasks are assigned to the same device, we add an edge $(t_{i:k_i}, t_{j:k_j})$ into $\mathcal{T}_E$, indicating a dependency between the two tasks, and no communication task is needed. If $t_{i:k_i}$ and $t_{j:k_j}$ with shared tensors are assigned to different devices, we add a communication task $t^c$ to $\mathcal{T}_N$ and two edges $(t_{i:k_i}, t^c)$ and $(t^c, t_{j:k_j})$ to $\mathcal{T}_E$. The new task $t^c$ is assigned to the communication device between the devices that perform $t_{i:k_i}$ and $t_{j:k_j}$.

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.

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).

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

### 5.2 Full Simulation Algorithm

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.

### 5.3 Delta Simulation Algorithm

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.

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 prior-

ity 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.

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 $o_3$ to 1 (i.e., $|c_3| = 1$). 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.

# 6 Execution Optimizer

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.

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.

## 6.1 MCMC Sampling

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 $\mathcal{S}$ and randomly proposes a new strategy $\mathcal{S}^*$. $\mathcal{S}^*$ is accepted and becomes the new current strategy with the following probability:

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

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.

## 6.2 Search Algorithm

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

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.

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

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

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 (Russakovsky et al., 2015). 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.

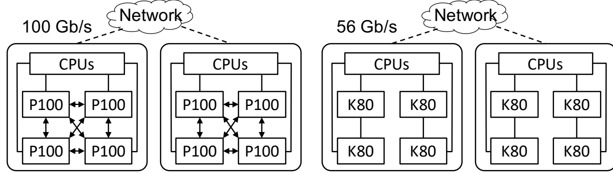
RNNTC, RNNLM and NMT are sequence-to-sequence RNN models for text classification, language modeling, and neural machine translation, respectively. RNNTC uses four LSTM layers with a hidden size of 1024. RNNLM uses two LSTM layers with a hidden size of 2048. Both RNN models include a softmax linear after the last LSTM layer. NMT includes an encoder and a decoder, both of which consist of 2 LSTM layers with a hidden size of 1024. 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.

[*] https://www. tensorflow.org/performance/benchmarks

*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).




(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.

We follow prior work (Krizhevsky et al., 2012; Szegedy et al., 2016; He et al., 2016; Kim, 2014; Zaremba et al., 2014; Wu et al., 2016) to construct operator graphs and set hyperparameters (e.g., learning rates, weight decays). 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.

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 E5-2680 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.

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.

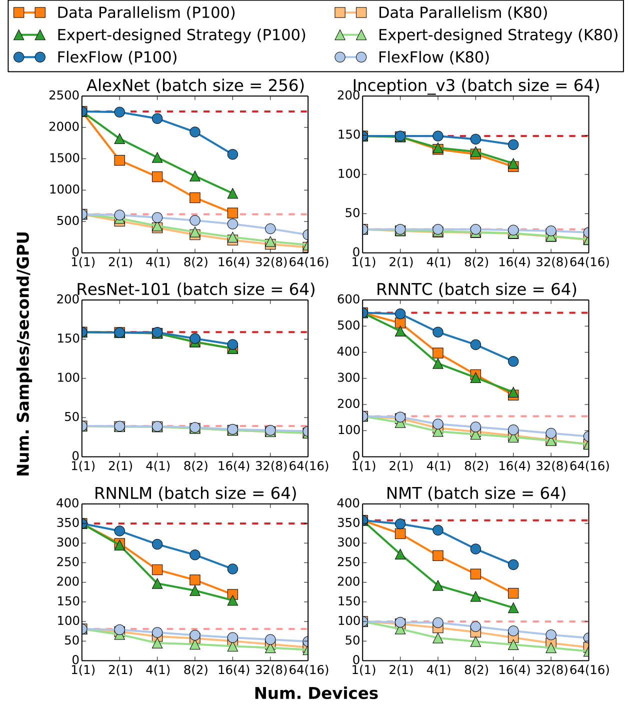


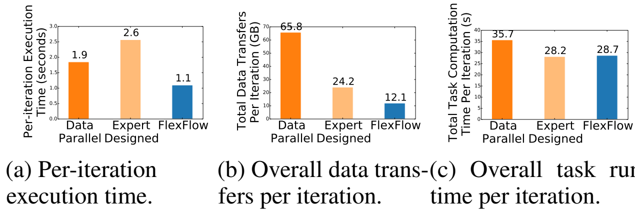

*Figure 6.* Per-iteration training performance on six DNNs. Numbers in parenthesis are the number of compute nodes used in the experiments. The dash lines show the ideal training throughput.

## 8.1 Parallelization Performance

### 8.1.1 Per-iteration Performance

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.

Expert-designed strategies optimize parallelization based




(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.

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.

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.

**Reducing overall communication costs.** Similar to existing deep learning systems, the FlexFlow distributed runtime supports overlapping data transfers with computation to hide communication overheads. 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.

**Reducing overall task computation time.** Data parallelism always parallelizes an operator in the batch dimension. However, as reported in (Jia et al., 2018), parallelizing an operator through different dimensions can result in different task computation time. For the matrix multiplication operator in the NMT model, parallelizing it in the channel dimension reduces the operator's overall computation time by 38% compared to parallelizing the operator in the batch dimension. Figure 7c shows that FlexFlow reduces the overall task computation time by 20% compared to data parallelism for the NMT model. The expert-designed strategy achieves slightly better total task computation time than FlexFlow. However, this is achieved by using model parallelism on each node, which disables any parallelism within each operator and results in imbalanced workloads. As a result, the expert-designed strategy achieves even worse execution performance than data parallelism (see Figure 7a). FlexFlow reduces the task computation time while enabling parallelism within an operator and maintaining load balance.

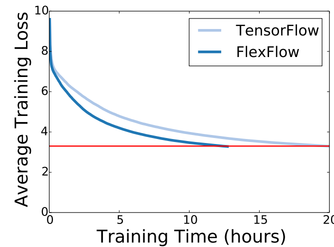
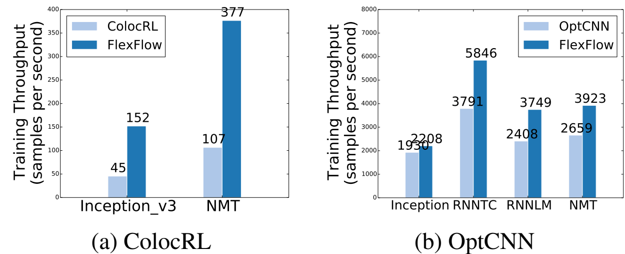



*Figure 8.* Training curves of Inception-v3 in different systems. The model is trained on 16 P100 GPUs (4 nodes).




(a) ColocRL

(b) OptCNN

*Figure 9.* Comparison among the parallelization strategies found by different automated frameworks.

### 8.1.2 End-to-end Performance

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.

In this experiment, we compare the end-to-end training performance between FlexFlow and TensorFlow on Inception-v3. 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.

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

| Num. GPUs | AlexNet Full | AlexNet Delta | AlexNet Speedup | ResNet Full | ResNet Delta | ResNet Speedup | Inception Full | Inception Delta | Inception Speedup | RNNTC Full | RNNTC Delta | RNNTC Speedup | RNNLM Full | RNNLM Delta | RNNLM Speedup | NMT Full | NMT Delta | NMT 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**× |

#### 8.1.3 Automated Frameworks

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

**ColocRL** uses reinforcement learning to learn device placement for model parallelism. 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.

Figure 9a compares the training throughput of the strategies found by FlexFlow and ColocRL for four K80 GPUs on a single node. 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.

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.

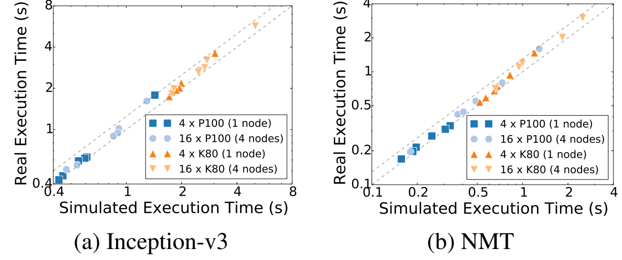
**OptCNN** (Jia et al., 2018) uses dynamic programming to parallelize linear DNNs. To evaluate OptCNN's performance on non-linear RNNs, we explicitly fuse all recurrent nodes sharing the same parameters to a single operator.

We compare the performance of FlexFlow and OptCNN for different DNNs on 16 P100 GPUs. 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. For these DNNs with non-linear operator graphs, FlexFlow achieves 1.2-1.6× speedup compared to OptCNN by using parallelization strategies that exploit parallelism across different operators. We show two examples in Section 8.4.

### 8.2 Execution Simulator

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

**Simulator accuracy.** 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.

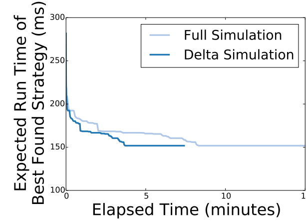



*Figure 11.* Search performance with the full and delta simulation algorithms for the NMT model on 16 P100 GPUs (4 nodes).

**Simulator execution time.** Figure 11 shows the search performance with different simulation algorithms for finding a strategy for the NMT model on 16 P100 GPUs on 4 nodes. The full and delta simulation algorithms terminate in 16 and 6 minutes, respectively. If the allowed time budget is less than 8 minutes, the full simulation algorithm will find a worse strategy than the delta simulation algorithm.

We compare the end-to-end search time of the execution optimizer with different simulation algorithms. For a given DNN model and device topology, we measure the average execution time of the optimizer using 10 random initial strategies. The results are shown in Table 5. The delta simulation algorithm is 2.2-6.9× faster than the full simulation algorithm. Moreover, the speedup over the full simulation algorithm increases as we scale the number of devices.

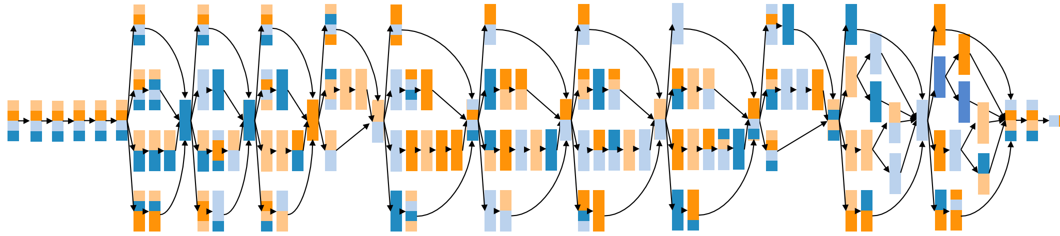


*Figure 12.* The best discovered strategy for parallelizing Inception-v3 on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each GPU is denoted by a color.

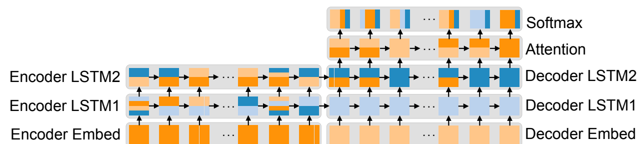



*Figure 13.* The best discovered strategy for parallelizing NMT on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each grey box denotes a layer, whose operators share the same network parameters. Each GPU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

### 8.3 Search Algorithm

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.

### 8.4 Case Studies

**Inception-v3.** Figure 12 shows the best discovered strategy for parallelizing Inception-v3 on four P100 GPUs, which exploits intra-operator parallelism for operators on the critical path and uses a combination of intra- and inter-operator parallelism for operators on different branches. 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%.

For parallelizing the same Inception-v3 model on four K80 GPUs with asymmetric connections between GPUs (see Figure 5b), we observe that the best discovered strategy tends to parallelize operators on adjacent GPUs with a direct connection to reduce the communication costs.

**NMT.** 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 (e.g., embed layers), it performs the computation on a single GPU to eliminate parameter synchronization. Second, for a layer with a large number of parameters and heavy computation (e.g., softmax layers), FlexFlow uses parallelism in the parameter dimension and assigns the computation for a subset of parameters to each task. This reduces parameter synchronization costs while maintaining load balance. Third, for multiple recurrent layers (e.g., LSTM and attention layers), FlexFlow uses concurrency among different layers as well as parallelism within each operator to reduce parameter synchronization costs while balancing load.

## 9 Conclusion

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.

## Acknowledgements

We thank the anonymous reviewers for their feedback on this work. This work was supported by NSF grant CCF-1409813, the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and is based on research sponsored by DARPA under agreement number FA84750-14-2-0006. This research was also supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, Intel, Microsoft, NEC, Teradata, SAP and VMware—as well as DARPA grant FA8750-17-2-0095 (D3M) and NSF grant CNS-1651570. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied of DARPA or the U.S. Government.

## References


Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. URL http://arxiv.org/abs/1410.0759.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

cuBLAS. Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas, 2016.

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. Large scale distributed deep networks. In *NIPS*, 2012.

Gao, M., Pu, J., Yang, X., Horowitz, M., and Kozyrakis, C. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

Gao, M., Yang, X., Pu, J., Horowitz, M., and Kozyrakis, C. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, 2019.

Gog, I., Schwarzkopf, M., Gleave, A., Watson, R. N. M., and Hand, S. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 99–115, Savannah, GA, 2016. USENIX Association.

Hastings, W. K. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57 (1):97–109, 1970.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2016.

Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pp. 261–276. ACM, 2009.

Jia, Z., Lin, S., Qi, C. R., and Aiken, A. Exploring hidden dimensions in accelerating convolutional neural networks. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*. PMLR, 2018.

Kim, Y. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014. URL http://arxiv.org/abs/1408.5882.

Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014. URL http://arxiv.org/abs/1404.5997.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS, 2012.

Lam, S. and Sethi, R. Worst case analysis of two scheduling algorithms. *SIAM Journal on Computing*, 6, 1977.

LeCun, Y. LeNet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 2015.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. 2017.

Movies. Movie review data. https://www.cs.cornell.edu/people/pabo/movie-review-data/, 2005.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, 2002.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL http://arxiv.org/abs/1409.4842.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

Treichler, S., Bauer, M., Sharma, R., Slaughter, E., and Aiken, A. Dependent partitioning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA' 16. ACM, 2016.

WMT. Conference on machine translation. http://www.statmt.org/wmt16, 2016.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

Zaremba, W., Sutskever, I., and Vinyals, O. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014. URL http://arxiv.org/abs/1409.2329.

# A Full Simulation Algorithm

Algorithm 1 shows the pseudocode of the full simulation algorithm. It 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 (Section 24.3 of 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).

**Algorithm 1** Full Simulation Algorithm.

1: **Input**: An operator graph $\mathcal{G}$, a device topology $\mathcal{D}$, and a parallelization strategy $\mathcal{S}$.
2: $\mathcal{T}$ = BUILDTASKGRAPH($\mathcal{G}$, $\mathcal{D}$, $\mathcal{S}$)
3: readyQueue = {} *// a priority queue sorted by readyTime*
4: **for** t $\in \mathcal{T}_N$ **do**
5:     t.state = `NOTREADY`
6:     **if** $\mathcal{I}(\mathrm{t}) = \{\}$ **then**
7:         t.state = `READY`
8:         readyQueue.enqueue(t)
9: **while** readyQueue $\neq \{\}$ **do**
10:     **Task** t = readyQueue.dequeue()
11:     **Device** d = t.device
12:     t.state = `COMPLETE`
13:     t.startTime = max{t.readyTime, d.last.endTime}
14:     t.endTime = t.startTime + t.exeTime
15:     d.last = t
16:     **for** n $\in \mathcal{O}(\mathrm{t})$ **do**
17:         n.readyTime = max{n.readyTime, t.endTime}
18:         **if** all tasks in $\mathcal{I}(\mathrm{n})$ are `COMPLETE` **then**
19:             n.state = `READY`
20:             readyQueue.enqueue(n)
21: **return** max{t.endTime | t $\in \mathcal{T}_N$}

# B Delta Simulation Algorithm

Algorithm 2 shows the pseudocode of the full simulation algorithm. It first updates tasks and dependencies from an existing task graph and enqueues all modified tasks into a global priority queue (line 4-5). Similar to the Bellman-Ford shortest-path algorithm (Section 24.1 of Cormen et al. (2009)), the delta simulation algorithm iteratively dequeues updated tasks and propagates the updates to subsequent tasks (line 6-14). The full and delta simulation algorithms always produce the same timeline for a given task graph.

# C Artifact Appendix

## C.1 Abstract

This artifact appendix helps readers to reproduce the main experimental results in this paper. In the artifact evaluation, we compare the average training throughput of different parallelization strategies in FlexFlow.

## C.2 Artifact check-list (meta-information)

- **Compilation:** GCC 4.8 or above, CUDA 8.0 or above, cuDNN 6.0 or above
- **Run-time environment:** Linux Ubuntu 16.04 or above
- **Hardware:** A compute node with multiple GPUs, such as Amazon EC2 p2.x8large or p3.x8large instances. Note that a single GPU is able to verify the functionality of FlexFlow but cannot show FlexFlow's performance improvement over the baselines (e.g., data parallelism).
- **Metrics:** The primary metric of comparison is the average training throughput.

**Algorithm 2** Delta Simulation Algorithm.

1: **Input**: An operator graph $\mathcal{G}$, a device topology $\mathcal{D}$, an original task graph $\mathcal{T}$, and a new configuration $c_i'$ for operator $o_i$.
2: updateQueue = {} *// a priority queue sorted by readyTime*
3: /*UPDATETASKGRAPH *returns the updated task graph and a list of tasks with new* `readyTime`*/
4: $\mathcal{T}$, $\mathcal{L}$ = UPDATETASKGRAPH($\mathcal{T}$, $\mathcal{G}$, $\mathcal{D}$, $c_i$, $c_i'$)
5: updateQueue.enqueue($\mathcal{L}$)
6: **while** updateQueue $\neq$ {} **do**
7: **Task** t = updateQueue.dequeue()
8: t.startTime = max{t.readyTime, t.preTask.endTime}
9: t.endTime = t.startTime + t.exeTime
10: **for** n $\in \mathcal{O}$(t) **do**
11: **if** UPDATETASK(n) **then**
12: updateQueue.push(n)
13: **if** UPDATETASK(t.nextTask) **then**
14: updateQueue.push(t.nextTask)
15: **return** max{t.endTime | t $\in \mathcal{T}_N$}
16:
17: **function** UPDATETASK(t)
18: t.readyTime = max{p.endTime | p $\in \mathcal{I}$(t)}
19: */*Swap t with other tasks on the device to maintain FIFO.*/*
20: t.startTime = max{t.readyTime, t.preTask.endTime}
21: **if** t's readyTime or startTime is changed **then**
22: **return True**
23: **else**
24: **return False**

- **How much disk space required (approximately)?:** About 2 GB of disk storage should be sufficient for all experiments.
- **How much time is needed to prepare workflow (approximately)?:** About one hour to install all dependencies and compile the FlexFlow runtime.
- **How much time is needed to complete experiments (approximately)?:** About 20 minutes for all experiments.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache License, Version 2.0.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.2564262

### C.3 Description

#### C.3.1 Hardware dependencies

The experiments in the paper were performed on two GPU clusters, as described in Figure 5. To reproduce the experiments, we suggest to run this artifact evaluation on a compute node with multiple GPUs, such as Amazon EC2 p2.x8large or p3.x8large instances. This will be sufficient to demonstrate FlexFlow's performance improvement over the widely used data parallelism baseline.

#### C.3.2 Software dependencies

FlexFlow depends on the following software libraries:

- NVIDIA cuDNN and cuBLAS libraries are used to perform DNN operations.
- Legion (Bauer et al., 2012) is the underlying runtime FlexFlow built on.
- (Optional) GASNet [†] is used for distributed executions.

The following software versions were used in our experiments: cuDNN 7.3, CUDA 9.0, Legion 18.02.0, and GASNet 1.28.0.

### C.4 Installation

The `README.md` file includes detailed instructions on how to install the FlexFlow runtime. The Legion and GASNet submodules can be initialized by the following command lines:

```
git submodule init
git submodule update
```

The `ffcompile.sh` script compiles a DNN model in FlexFlow:

```
./ffcompile.sh dnn.cc
```

where `dnn.cc` defines the operators in the DNN model.

### C.5 Experiment workflow

The `run_experiments.sh` script automatically builds and evalautes two example DNN models (i.e., AlexNet (Krizhevsky et al., 2012) and ResNet (He et al., 2016)) in FlexFlow. All experiments were run with synthetic data in GPU memory to remove the side effects of data transfers between CPU and GPU.

For each DNN model, we compare the training throughputs of data parallelism and FlexFlow's optimized parallelization strategies on 1, 2, and 4 GPUs on a compute node.

### C.6 Evaluation and expected result

The `run_experiments.sh` script prints the training throughputs of different parallelization strategies. By running the script on a multi-GPU node, you should observe that FlexFlow's optimized strategies consistently outperform the data parallelism baseline.

### C.7 Experiment customization

FlexFlow can be used to optimize parallelization for arbitrary DNN models. We refer users to the `README.md` file in this artifact evaluation for detailed instructions on how to use FlexFlow for other DNN models.

[†] http://gasnet.lbl.gov/