FlexFlow - Summary and Review

Minh Nguyen

TU Darmstadt

May 20, 2020

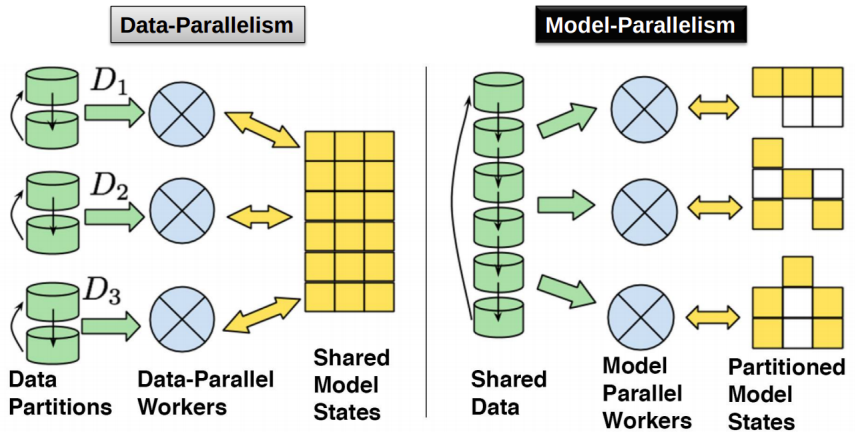
Outline

Motivation

- Data Parallelism and Model Parallelism in Deep Learning.
- Data Parallelism is supported by typical DL frameworks.
- Model Parallelism requires manual expert-designed strategies, otherwise sub-optimal training performance is yielded.

`FlexFlow` [JZA18]

Therefore, automatically proposed model parallelism strategy should be available to utilise the power of multiple worker nodes (GPUs).



FlexFlow

- The SOAP Search Space
- Execution Simulator
- Execution Optimiser
- FlexFlow Runtime

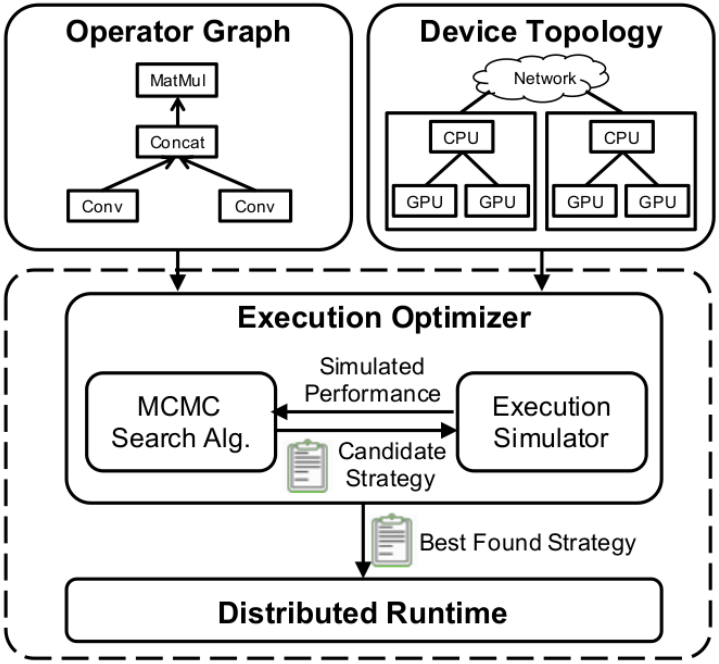


Figure 1. FlexFlow overview.

The SOAP Search Space

An *operator graph* $\mathcal{G}$ to describe all operators and state in a DNN. Each node $o_i \in \mathcal{G}$ is an operator.

A *device topology* $\mathcal{D} = (\mathcal{D}_N, \mathcal{D}_E)$ describing all available hardware devices and their interconnections.

A parallelisation strategy $\mathcal{S}$ describes one possible parallelisation of an application.

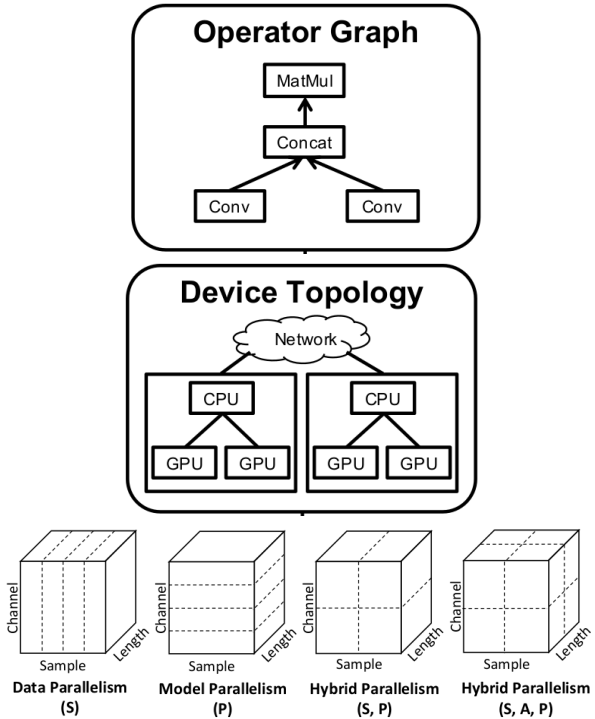


Figure 2. Example parallelization configurations for 1D convolution. Dashed lines show partitioning the tensor.

Execution Simulator

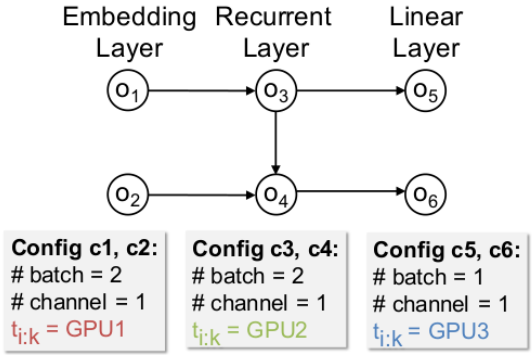
Task Graph

A task graph $\mathcal{T} = (\mathcal{T}_N, \mathcal{T}_E)$, where each node $t \in \mathcal{T}_N$ is a task and each edge $(t_i, t_j) \in \mathcal{T}_E$ is a dependency that task t_j cannot start until task t_j is completed. `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, for a communication task, `exeTime` is the time to transfer a tensor (of size s) between devices with bandwidth b and is estimated as s/b.

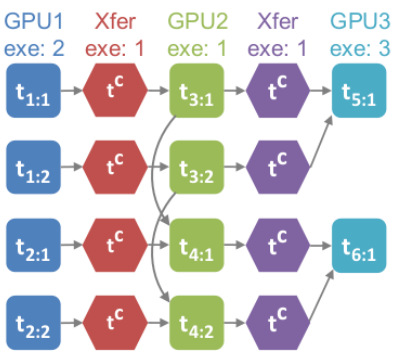
Full Simulation Algorithm

Tasks are enqueued into a global priority queue when ready and are dequeued in increasing order by their `readyTime`.

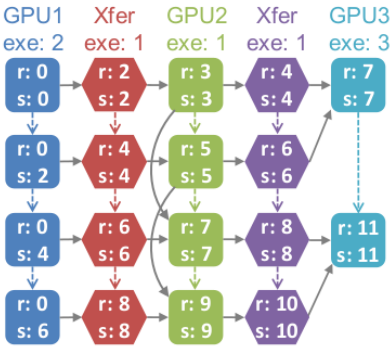
Execution Simulator



(a) An example parallelization strategy.



(b) The corresponding task graph.



(c) The task graph after the full simulation algorithm.

Execution Optimiser

- The *execution optimiser* takes an operator graph $\mathcal{G}$ and a device topology $\mathcal{D}$ as inputs and automatically finds an efficient parallelisation strategy.
- Because finding the optimal parallelisation strategy is NP-hard, to find a low-cost strategy, FlexFlow uses a cost minimization search to heuristically (an easy reduction from *minimum makespan*) explore the space and returns the best strategy discovered.

Markov Chain Monte Carlo sampling

The algorithm maintains a current strategy $\mathcal{S}$ and randomly proposes a new strategy $\mathcal{S}^*$. 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.

FlexFlow Runtime

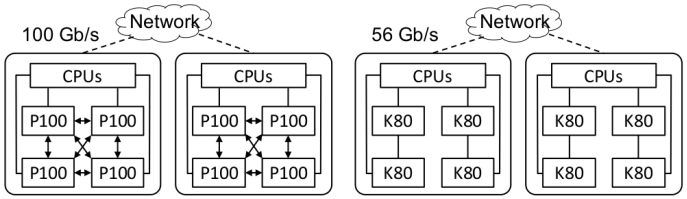
Existing deep learning systems

TensorFlow, PyTorch, Caffe2, and MXNet only support parallelising an operator in the sample dimension through data parallelism, and it is **non-trivial** to parallelise an operator in **other dimensions** or **combinations** of several SOAP dimensions in these systems.

Actual Runtime Implementation

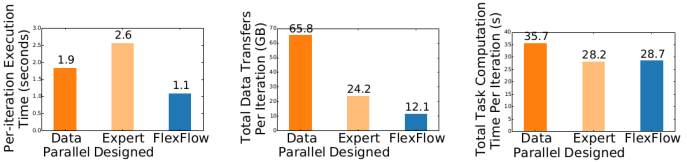
- Distributed runtime: Legion [BTSA12]
- High performance parallel runtime: cuDNN and cuBLAS
- Parallelizing an operator in any combination of SOAP dimensions: Legion high-dimensional partitioning interface dimensions [TBS+16]

Evaluation



(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.



(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.

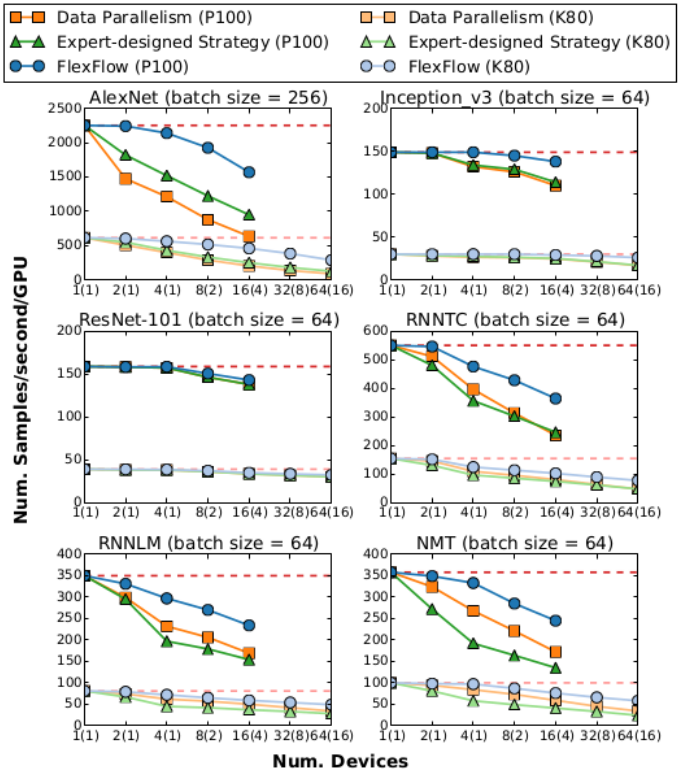


Figure 6. Per-iteration training performance on six DNNs. Numbers in parenthesis are the number of compute nodes used in the experiments. The dash lines show the ideal training throughput.

Discussion

Two recent publications has cited FlexFlow as following

Example

Mesh-TensorFlow [SCP+18]: A framework that uses cost modeling to pick the best parallelization strategy, including how to partition work for each operation. Their parallelizable dimensions are defined as the set of all divisible dimensions in the output tensor (owners compute), and therefore their mapping can be suboptimal in terms of communication.

Example

Megatron-LM [SPP+19]: A deep learning framework orchestrating such parallel computation, provides a method to pick the best parallelisation strategy. However, it requires rewriting the model, and rely on custom compilers and frameworks that are still under development.

References I

- M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, *Legion: Expressing locality and independence with logical regions*, SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1–11.
- Zhihao Jia, Matei Zaharia, and Alex Aiken, *Beyond data and model parallelism for deep neural networks*, arXiv preprint arXiv:1807.05358 (2018).
- Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al., *Mesh-tensorflow: Deep learning for supercomputers*, Advances in Neural Information Processing Systems, 2018, pp. 10414–10423.

References II


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


Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken, *Dependent partitioning*, Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2016, pp. 344–358.