Dynamic Scheduling For Dynamic Control Flow in Deep Learning Systems

Jinliang Wei[1] jinlianw@cs.cmu.edu
Garth Gibson[2,1,5] garth@cs.cmu.edu
Vijay Vasudevan[3] vrv@google.com
Eric Xing[1,4] epxing@cs.cmu.edu

[1]Carnegie Mellon University, [2]Vector Institute, [3]Google Brain, [4]Petuum Inc., [5]University of Toronto

Abstract

Today's deep learning systems are dominated by a dataflow execution model. Given a static dataflow graph and the shape of the input (e.g., mini-batch sizes and image dimensions), the system can fully determine its computation before execution. When the same static graph applies to every data sample, the system may search for an optimal computation schedule offline by trying out many schedules on a sample input, knowing the input values won't affect computation throughput. However, for many neural networks, data samples have variable shapes and the computation graph topology depends on input or parameter values. In this case, a static graph fails to fully describe the computation and a better schedule needs to be dynamically derived to take runtime information into account. Thus we argue for the importance of dynamic scheduling, especially regarding distributed device placement.

1 Dynamic Computation in Neural Networks

In a dataflow system, application programs first construct a dataflow graph that describes the computation, and then request the system to execute a subgraph or the whole graph. Although for many neural networks (e.g., AlexNet [7], Inception-v3 [13], and ResNet [3]), the computation can be described by a static acyclic directed graph (DAG) that applies to all data samples, there are many cases where the graph topology varies based on input or parameter values.

Recurrent Neural Networks [2] model sequences of data (e.g., sentences). A recurrent neural network (RNN) repeatedly applies a cell function, such as long-short-term-memory (LSTM) [4], to each element of the sequence. Since sequences may have variable length, the cell function is executed for different number of times for different sequences. A typical approach for expressing RNNs as a static DAG is to statically unroll the sequence for a finite number of steps, padding shorter sequences with empty values and likely chopping longer ones. An alternative approach is to construct a distinct graph for each input sequence, paying the graph construction overhead for each data sample.

Recursive Neural Networks [12] generalize recurrent neural network to model arbitrary topologies. For example, Tree-LSTM [14] models the syntactic tree of a sentence. Since the topology differs from sentence to sentence, Tree-LSTM constructs a distinct static DAG for each sentence. As shown by Xu et al. [16], per-sample graph construction constitutes a significant overhead (over 60% of runtime in some cases). Xu et al. [16] propose to resolve the graph construction overhead by reusing the graph structure that already exists in the dataset instead of programmatic construction, restricting its applicability.

Mixture of Experts (MoE) [11] is an example of conditional computation in neural networks. A MoE layer consists of a gating network and a large number (up to hundreds of thousands) of expert networks. Each data sample sparsely activates a small number of experts as determined by the gating

Preprint. Work in progress.

network based on runtime values. Therefore, for an input mini-batch, the input size of each expert is unknown until the gating network has been executed on the mini-batch.

Expressing dynamic computation via dynamic control flow. Yu et al. [17] present two dynamic control flow operations `cond` and `while_loop` in TensorFlow that represents conditional and iterateive computation respectively. Recursive (including recurrent) neural networks can be expressed as a while loop iterating over the nodes in a topologically sorted order. As the loop body is represented as a subgraph in a static DAG, all dynamic instances of the loop body (i.e., iterations) share the same dependency pattern. Therefore, for recursive neural networks, each iteration is conservatively specified to depend on its previous iteration to ensure correct ordering, resulting in a sequential execution, even though some iterations can potentially be executed in parallel. Jeong et al. [5] take advantage of the additional parallelism by introducing a recursion operation into TensorFlow. With recursion, a node recursively invokes the computation function on other nodes and waits until the recursive calls return to continue its execution. This allows a caller to dynamically specify its distinct dependency on the callees, permitting parallel execution of the functions on independent nodes.

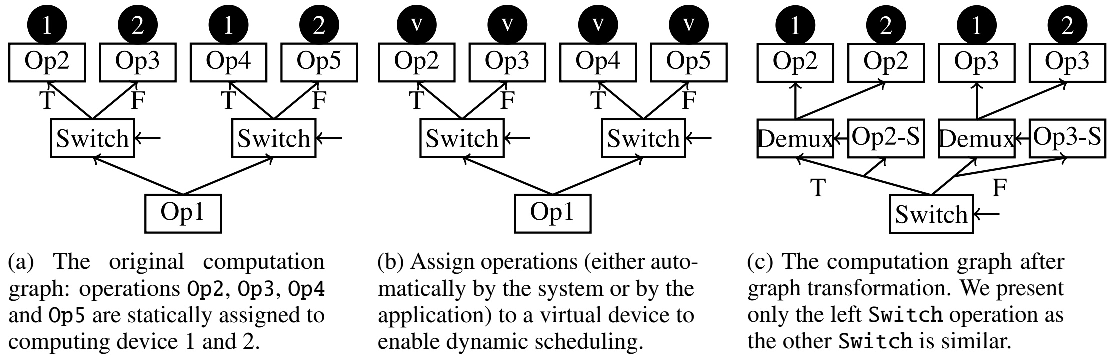
2 The Need for Dynamic Scheduling of Dynamic Control Flow

Despite the programming support for expressing dynamic computation, existing dataflow-based deep learning systems employ a static computation schedule derived prior to graph execution. A computation schedule determines how operations are placed on (possibly distributed) computing devices and compiles each device's graph partition to an executable program. Here we focus on distributed device placement.

When the same static computation graph applies to all data samples, it is possible to find an efficient computation schedule prior to execution. TensorFlow [1] relies on application programmers to manually place operations on devices; Mirhoseini et al. [10, 9] learn the device placement from repeated trial executions of various schedules. Jia et al. [6] simulates schedule execution to reduce the planning cost down to sub-seconds to tens of minutes depending on the scale (4 to 64 GPUs) and complexity of the network. Moreover, Jia et al. [6] exploit additional dimensions of parallelization, such as intra-operation parallelism. Nevertheless, existing approaches fail to consider that the computation may change based on input or parameter values. We discuss the inefficiency due to overlooking runtime information to motivate dynamic scheduling.

Conditional Computation. TensorFlow's `cond` is implemented using `Switch` which forwards an input tensor to one of two subgraphs. MoE generalizes `Switch` in two ways: (1) the forwarding decision is made separately for each row in the input tensor and (2) each row is forwarded to K out of N subgraphs. Due to MoE's large size (up to $\sim$131 billion parameters), existing implementations (e.g., Tensor2Tensor [15] and Shazeer et al. [11]) statically partition the expert networks to different GPUs. Such static placement faces two problems: (1) the memory for a subgraph (e.g., variables) is statically allocated regardless of whether a subgraph is actually executed; (2) the input sizes among different experts can be highly skewed. These issues lead to heavy over-provisioning of GPU memory while wasting GPUs' precious computing cycles. As reported by Shazeer et al. [11], a MoE layer consisting of 131072 experts requires 128 Tesla K40 GPUs to fit while achieving a computation throughput of 0.3TFLOPS per GPU (Nvidia's claimed peak throughput is 4.29TFLOPS/GPU). With dynamic scheduling, the system allocates memory for only subgraphs that are executed and may partition an overwhelmingly large input to an expert along with replicating the expert to multiple GPUs to balance load among GPUs.

Iterative and Recursive Computation. TensorFlow creates a *frame* for each dynamic instance of the `while_loop` loop body. Operations of different frames may run in parallel as long as their dependencies are satisfied. However, since each operation is statically placed onto one device, all frames of this operation is bound to this device. This can lead to saturating the computing power of a single device, thus missing the additional parallelism, such as observed by Jeong et al. [5]. Previous work on static device placement observes throughput improvement when placing different iterations of a statically unrolled RNN to different devices [10, 9, 6]. While static scheduling would be prohibitively expensive when different data samples require different graph topology, dynamic scheduling may dynamically schedule different frames to different devices to take advantage of the additional parallelism. Moreover, as recursion is restricted to trees, deep learning systems need a more general approach for precisely capturing the dependency among loop iterations in order to explore parallelism in arbitrary dependency topologies, such as Graph-LSTM [8].



(a) The original computation graph: operations `Op2`, `Op3`, `Op4` and `Op5` are statically assigned to computing device 1 and 2.

(b) Assign operations (either automatically by the system or by the application) to a virtual device to enable dynamic scheduling.

(c) The computation graph after graph transformation. We present only the left `Switch` operation as the other `Switch` is similar.

Figure 1: Graph transformation for dynamic scheduling.

3 Implementing Dynamic Scheduling in TensorFlow

In this section, we present one approach for performing dynamic device placement in TensorFlow. Please note that this is active onging research.

TensorFlow Graph Execution. When a computation graph is executed, TensorFlow partitions the graph among multiple computing devices and adds a `Send` and a `Recv` operation for each edge cut. TensorFlow partitions the graph according to its operations' device placement decisions specified by the application program. After partitioning, each graph partition is executed by an executor. The executor initializes a *ready queue* with operations that do not have any input. The executor executes operations from the ready queue and after an operation is executed, its downstream operations are visited and pushed onto the queue if an operation's dependencies are all satisfied. Most operations generate tensors as output and propagate their outputs to downstream operations, except that `Switch` generates a dead single to its unexecuted branch. TensorFlow skips executing operations that receive a dead signal as input and propogates the dead signal downstream, but `Merge` is executed as long as one of its input is a live tensor and forwards this tensor to its output.

We dynamically schedule operations whose input shape can not be determined statically and those that may be executed for a variable number of times. We statically schedule the other operations in the graph in order to bound the runtime overhead of dynamic scheduling. Our approach to dynamic scheduling performs static graph transformations and runtime graph refinements. A main advantage of our approach is that it avoids repartitioning the computation graph during execution, and only requires local changes to the graph partitions at runtime.

Static Graph Transformation for Dynamic Scheduling. In order to dynamically assign operations to compute devices without repartitioning the graph, we replicate each dynamially scheduled operation on all devices and dynamically decide which ones get executed by choosing where to send live tensors and dead signals. For each such operation `C`, we add a corresponding `Schedule` node `S` in the computation graph. The `Schedule` node have the same dependency as the computation node `C` and its output is fed to a `Demultiplexer` which determines which replicas are activated (an input tensor maybe split and forwarded to more than one replica). In this way, a `Schedule` operation is executed after all inputs to the computation operation become available and multiple `Schedule` operations can be executed in a batch for bulk scheduling. Fig 1 illustrates this graph transformation with an example.

Dynamic Graph Refinement for Precise Capture of Data Dependency. As previously discussed, some dependencies depend on data or parameter values which can not be statically determined. Statically expressing such dependencies results in an imprecise and conservative dependency representation (e.g., an enforced control dependency between each pair of consecutive loop iterations), leading to underutilized parallelism. Our idea is to refine such dependency at runtime when necessary values become available. For this purpose, we introduce a new node in the computation graph called `TensorStore`. `TensorStore` stores a set of tensors and has a similar programming interface to TensorFlow's `TensorArray`. `TensorStore`'s tensors are accessed by key, where each key is written once and read many times. An operation may read a tensor from a `TensorStore` `s` whose index `k` is computed at runtime. The operation has dependencies on both `s` and `k` and thus is executed only

after `k` is available. The first time such an operation become available for execution, it changes its dependency on `s` to a precise dependency on `s[k]`. The operation will be executed right away if `s[k]` is already available, otherwise its execution will be triggered when `s[k]` is written. Combined with the graph transformation described earlier, such an operation may be dynamically scheduled onto any computing device.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] J. L. Elman. Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211, 1990.

[3] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[4] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[5] E. Jeong, J. S. Jeong, S. Kim, G.-I. Yu, and B.-G. Chun. Improving the expressiveness of deep learning frameworks with recursion. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 19:1–19:13, New York, NY, USA, 2018. ACM.

[6] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. *CoRR*, abs/1807.05358, 2018.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[8] X. Liang, X. Shen, J. Feng, L. Lin, and S. Yan. Semantic object parsing with graph LSTM. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part I*, pages 125–143, 2016.

[9] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean. Hierarchical planning for device placement. 2018.

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

[11] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017.

[12] R. Socher, A. Perelygin, J. Y. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank.

[13] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR*, pages 1–9. IEEE Computer Society, 2015.

[14] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *ACL*, 2015.

[15] A. Vaswani, S. Bengio, E. Brevdo, F. Chollet, A. N. Gomez, S. Gouws, L. Jones, L. Kaiser, N. Kalchbrenner, N. Parmar, R. Sepassi, N. Shazeer, and J. Uszkoreit. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416, 2018.

[16] S. Xu, H. Zhang, G. Neubig, W. Dai, J. K. Kim, Z. Deng, Q. Ho, G. Yang, and E. P. Xing. Cavs: An efficient runtime system for dynamic neural networks. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, pages 937–950, 2018.

[17] Y. Yu, M. Abadi, P. Barham, E. Brevdo, M. Burrows, A. Davis, J. Dean, S. Ghemawat, T. Harley, P. Hawkins, M. Isard, M. Kudlur, R. Monga, D. G. Murray, and X. Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 18:1–18:15, 2018.