Distributed Training

...

Presented by Aditya Roy and Alex Behr

The Problem

Training takes too long.

Training sometimes also takes more GPU memory than is typically (or ever) available.

State of the art models are only getting larger and larger.

We want to find ways to train models in less time and using smaller GPU footprints.

Current Distributed Solutions

Data parallelism

- Workers perform the full training process on a subset of the data and synchronize learned results.
- Reduces total training time.

Model parallelism

- Workers split the model into parts and perform only a subset of the computation. It requires communication between multiple workers to complete one forward-backward pass.

The Papers

Tofu - reducing the GPU footprint for very large DNNs

Prague - faster and more robust data parallel training

ByteScheduler - a generic communication scheduler leveraging data parallelism for accelerating distributed DNN training

Supporting Very Large Models using Automatic Dataflow Graph Partitioning

...

Minjie Wang, Chien-chin Huang, Jinyang Li
New York University

The Problem

Very large DNN models are at the forefront of cutting edge ML research.

The trend is that state of the art models are only going to get larger (doubling every 2.4 years).

Yet their size is constrained by GPU device memory.

Previous Solutions

Various techniques have been found to run models on GPUs which have smaller memory than the batch size of the model.

Capuchin and algorithms like it allow for data to be swapped to other memory or recomputed and reduce required footprint at the cost of throughput.

Model parallelism allows model data to be split across multiple devices for either an increase in throughput or to reduce GPU footprint.

Splitting the model by layers is also currently done, but this can prove to be too coarse as the tensors required to compute some layers (such as very large CNNs) can still be too large to compute on a single GPU.

Partitioning at the tensor level can provide finer partition granularity. This has been tested with some success but with manual configuration at the layer level.

The Tofu Solution

Tofu is a system for implementing tensor partitioning automatically without manual configuration. The steps are:

- Describe the model operations in TDL (Tensor Description Language).
- Generate a dataflow graph.
- Apply graph coarsening to simplify the graph.
- Generate partition strategies with recursive application of a Dynamic Programming (DP) algorithm.
- Optimize the model to assure proper memory management by ML framework.

Context / Limitations / Caveats

Only uses Partition-n-Reduce partition strategy.

TDL cannot describe certain operations (i.e. Cholesky decomposition).

Does not leverage communication topology among workers.

Can leave some GPUs unsaturated at some steps.

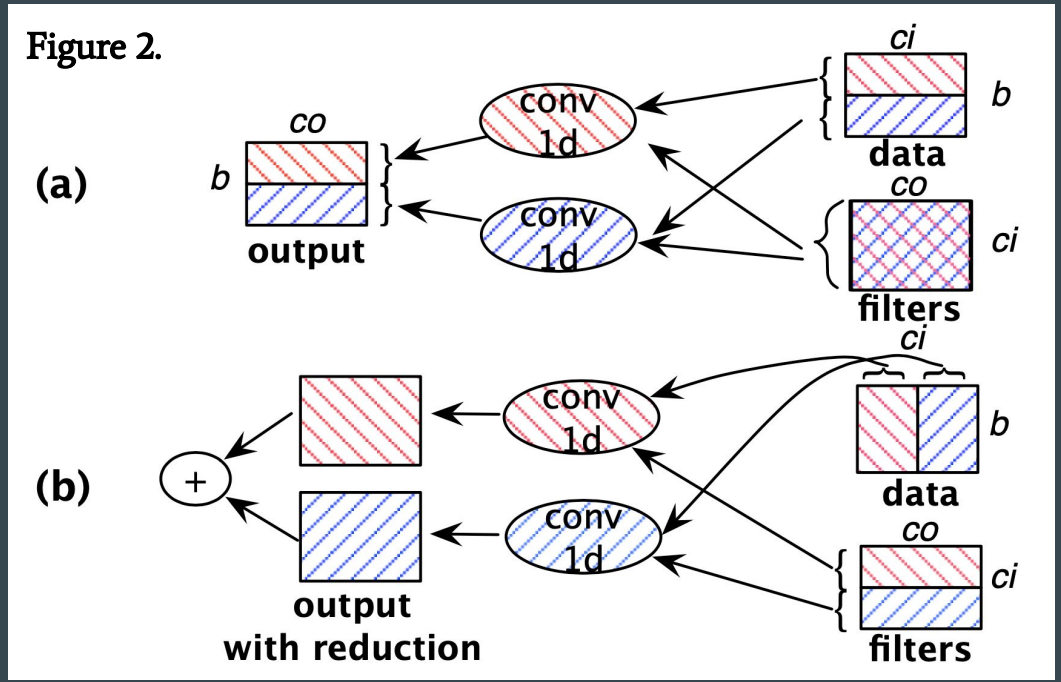
Targeted only at DNNs whose batches are too large to fit in a single GPU.

Only tested on single machine architecture.

Operator Partitioning

The basic idea is that the input and output tensors of an operator can be partitioned based on the the dataflow through that operator.

The figure on the right shows the two categories of partition-n-reduce.



TDL

Tensor-as-a-lambda

Describe each operation in the model in TDL. This codifies how the data will flow through the model.

Allows for the construction of a dataflow graph and with symbolic interval analysis, the relationship between ranges of features in the input / output of an operator.

```
def conv1d(data, filters):
 for b in range(output.shape[0]): #b is batch dimension
   for co in range(output.shape[1]): #co is output channel
    for x in range(output.shape[2]): #x is output pixel
     for ci in range(filters.shape[0]): #di is input channel
      for dx in range(filters.shape[2]): #dx is filter window
       output[b, co, x] += data[b, ci, x+dx]
                           * filters[ci, co, dx]
```

Figure 1. The naive implementation of `conv1d` in Python.

```
@tofu.op
def conv1d(data, filters):
  return lambda b, co, x:
    Sum(lambda ci, dx: data[b, ci, x+dx]*filters[ci, co, dx])

@tofu.op
def batch_cholesky(batch_mat):
  Cholesky = tofu.Opaque()
  return lambda b, i, j: Cholesky(batch_mat[b, :, :])[i,j]
```

Figure 3. Example TDL descriptions.

TDL Expressions

- Index variables (i.e. arguments of the lambda function).
- Tensor elements (e.g. filters[ci, co, dx]).
- Arithmetic operations involving constants, index variables, tensor elements or TDL expressions.
- Reduction over a tensor along one or more dimensions.

Symbolic Interval Analysis

Symbolic interval analysis allows Tofu to relate input and output tensor regions based on the operators between them.

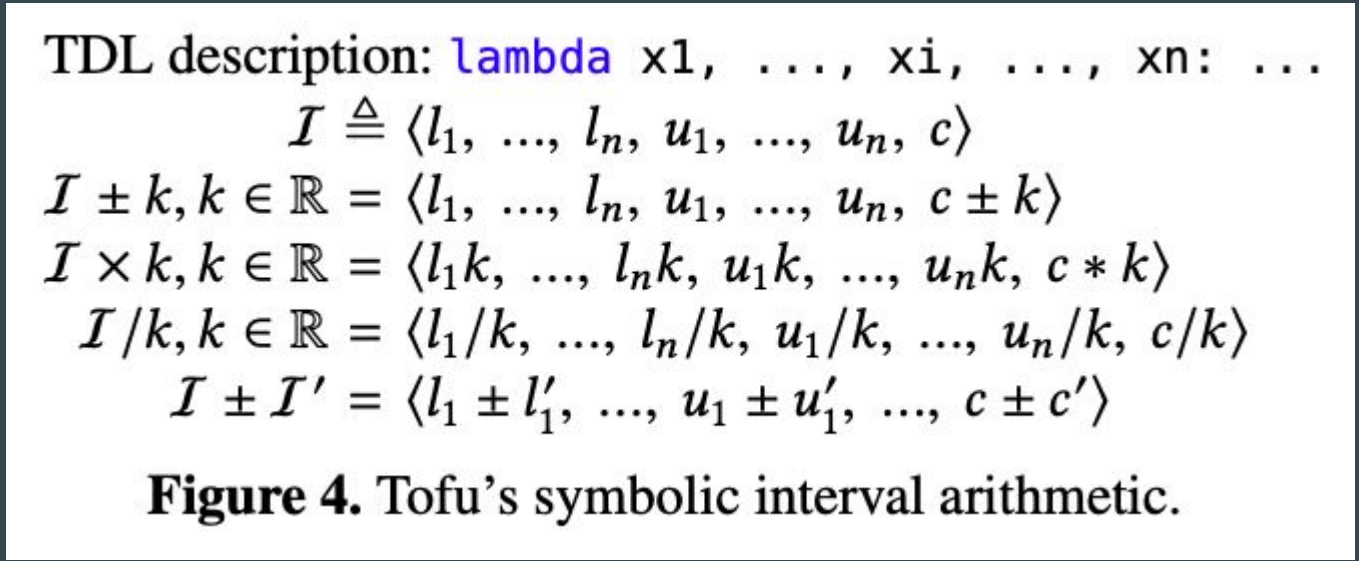


Figure 4. Tofu's symbolic interval arithmetic.

Graph Coarsening

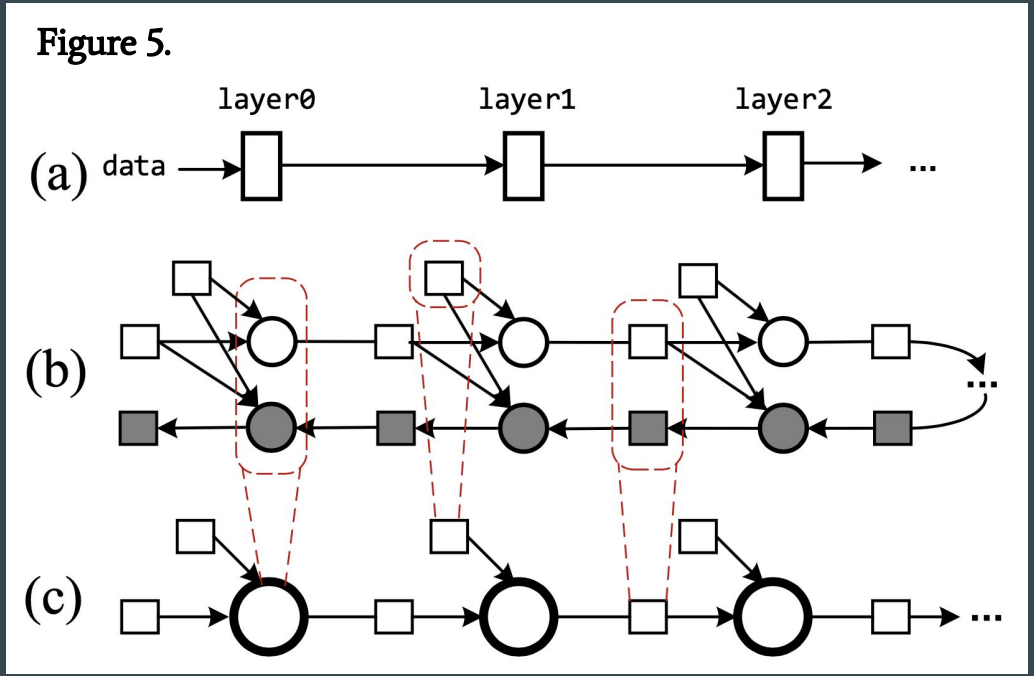
Simplifies the dataflow graph.

Group forward and backward operations.

Coalesce element-wise operators.

Coalesce unrolled timesteps (RNN).

All of the tensors within an example of these groups have the same dataflow relationships.



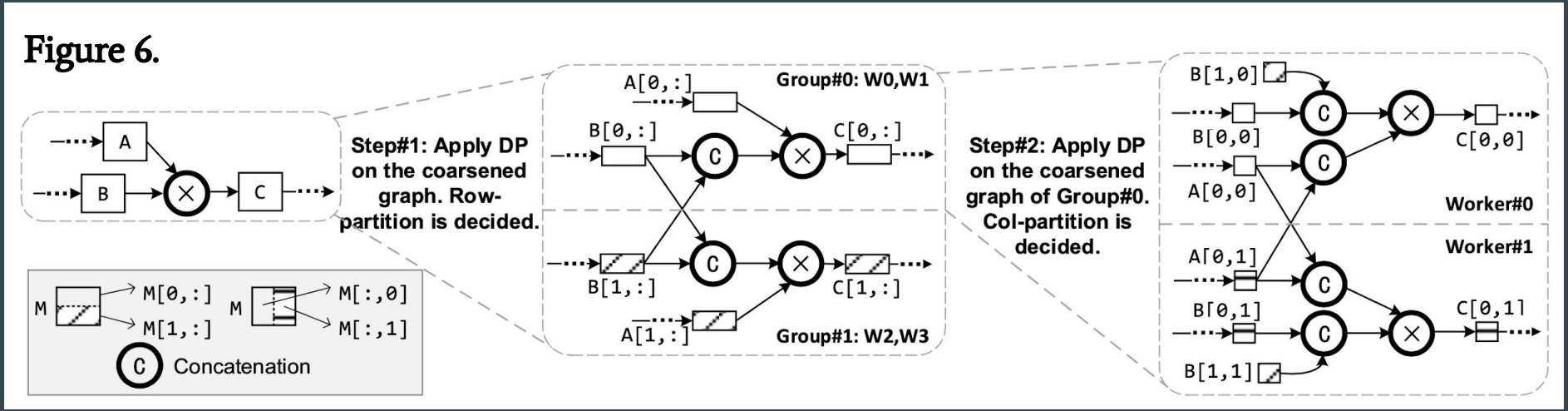
Generating Partition Strategies Using Recursive DP

At each step in a partitioning strategy a single dimension is chosen for splitting.

A key insight the authors had was that the search can be done recursively. Find the best single partition with DP and then repeat the search on either side until the there is a partition for every worker.

In some cases the number of workers is not a factor of 2. In these cases the number of workers is factored and partitions are made to fit the factors in descending order.

The resulting partitions:



Partitioning Search Time

Coarsening the dataflow graph and performing a search for a single dimension to partition at a time instead of a combination of dimensions shrinks the search space dramatically. For a set of 6-4D tensors to be split among 8 workers, the full search all at once has 64,000,000 possibilities, but finding a single split recursively for 8 workers requires searching 3 recursive layers for a total of 12,288 options.

Table 1.

	Search Time	
	WResNet-152	RNN-10
Original DP [14]	n/a	n/a
DP with coarsening	8 hours	>24 hours
Using recursion	8.3 seconds	66.6 seconds

Optimizations

Necessary procedures for the system to be usable in existing frameworks.

Tofu adds dependencies between the model nodes executed on a worker so that the framework memory planner allows for memory reuse.

They also wrote a kernel they call MultiFetch in CUDA UVA. This was necessary because some workers need to fetch data from other workers and the MXNet processes for this require many intermediate representations. Their MultiFetch kernel avoids the intermediate representations and allows a process running on one GPU to directly access data on another.

Evaluations - Throughput Benchmark Models

Two benchmark models are used. An LSTM RNN model and a Wide ResNet Model.

The RNN increases in size as the number of layers and the size of each layer's hidden output increases.

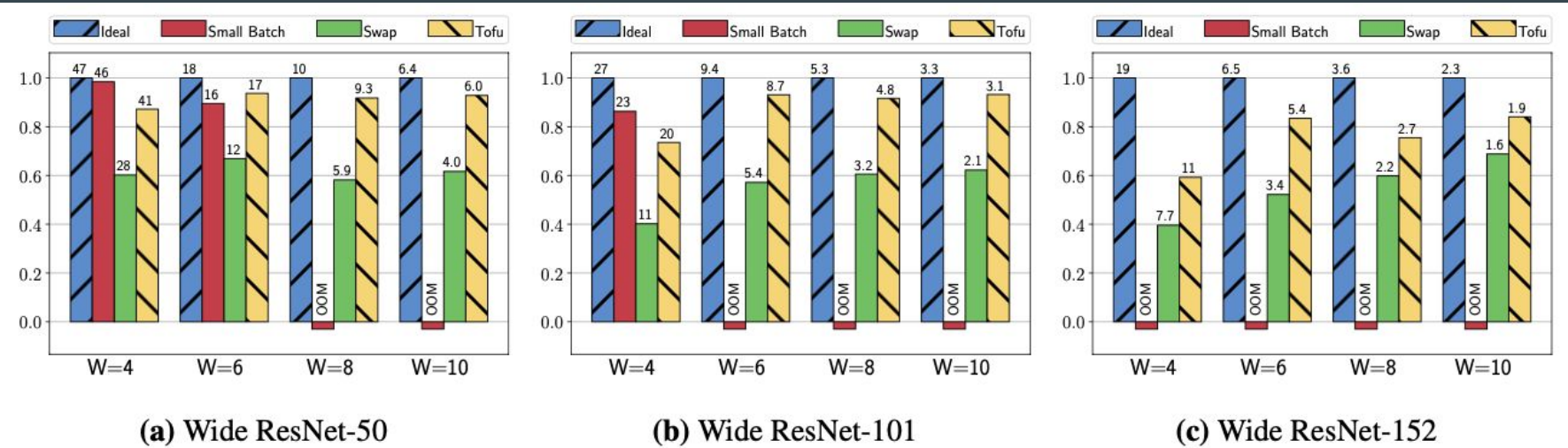
The WResNet increases in size as the scaling factor (W) increases and the depth of the model increases.

RNN			
	L=6	L=8	L=10
H=4K	8.4	11.4	14.4
H=6K	18.6	28.5	32.1
H=8K	33.0	45.3	57.0

Wide ResNet			
	L=50	L=101	L=152
W=4	4.2	7.8	10.5
W=6	9.6	17.1	23.4
W=8	17.1	30.6	41.7
W=10	26.7	47.7	65.1

Table 2. Total weight tensor sizes (GB) of our benchmarks.

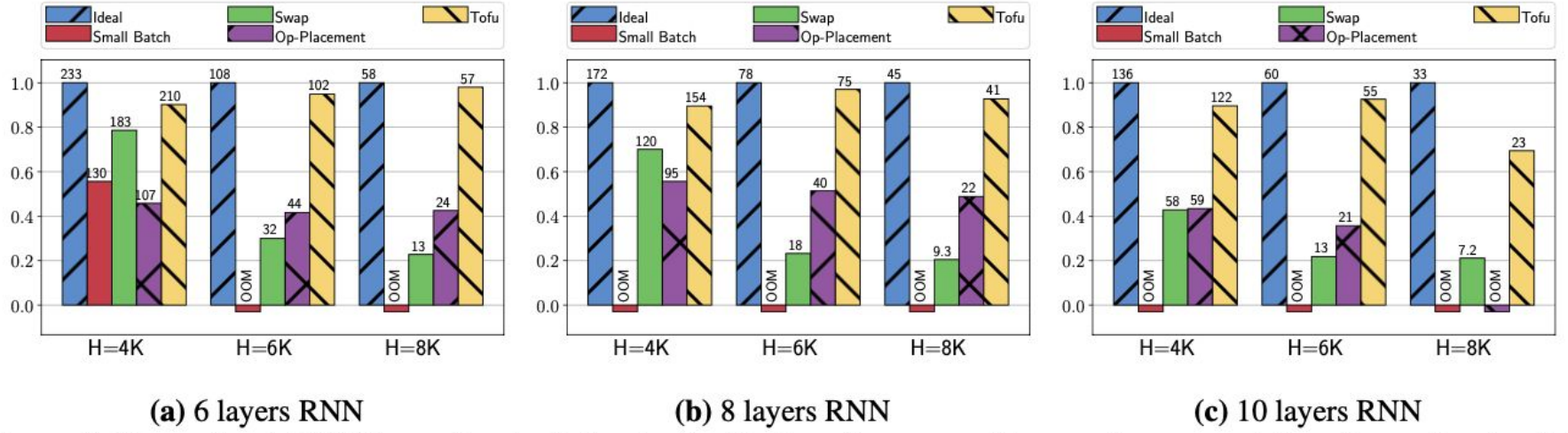
Evaluation - Throughput on Wide ResNet CNN



(a) Wide ResNet-50 **(b)** Wide ResNet-101 **(c)** Wide ResNet-152

Figure 8. Normalized WResNet throughput relative to the ideal performance. The number on each bar shows the absolute throughput in samples/sec.

Evaluation - Throughput on LSTM cell RNN



(a) 6 layers RNN **(b)** 8 layers RNN **(c)** 10 layers RNN

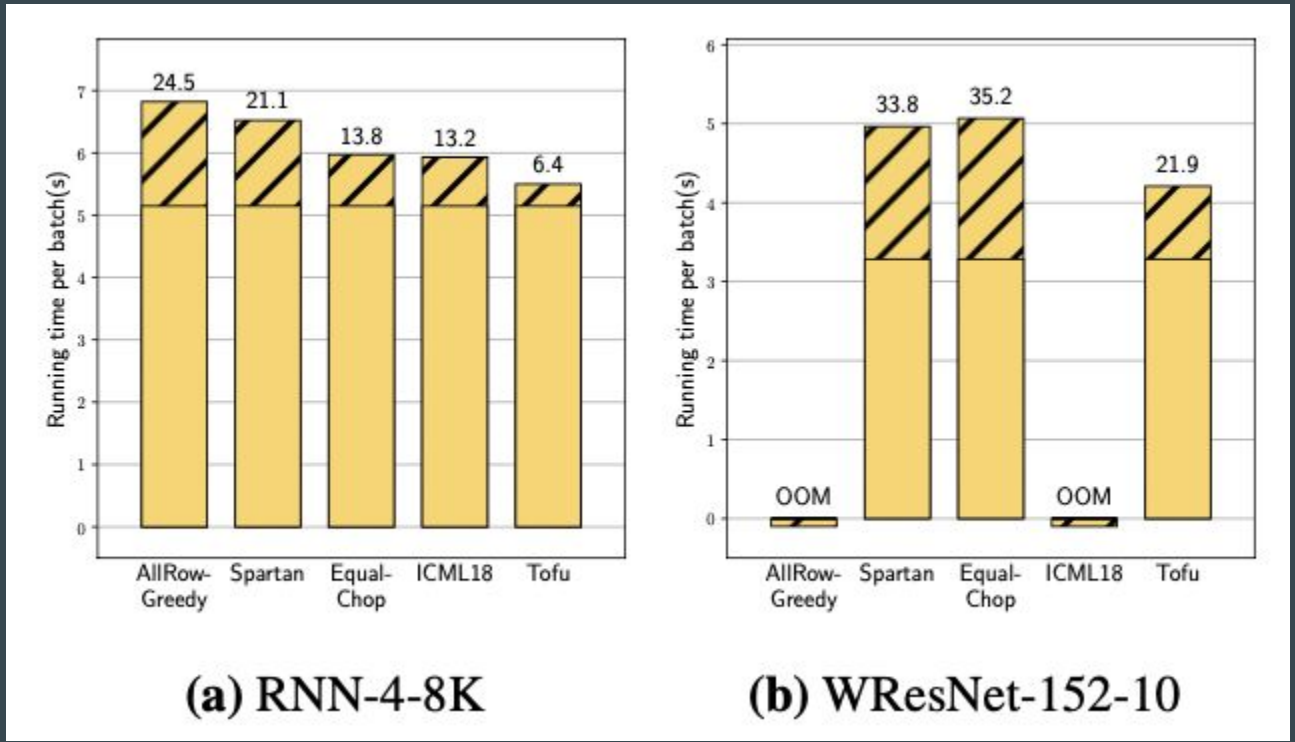
Figure 9. Normalized RNN throughput relative to the ideal performance. The number on each bar shows the absolute throughput in samples/sec.

Evaluation - Throughput vs. Framework Op Placement

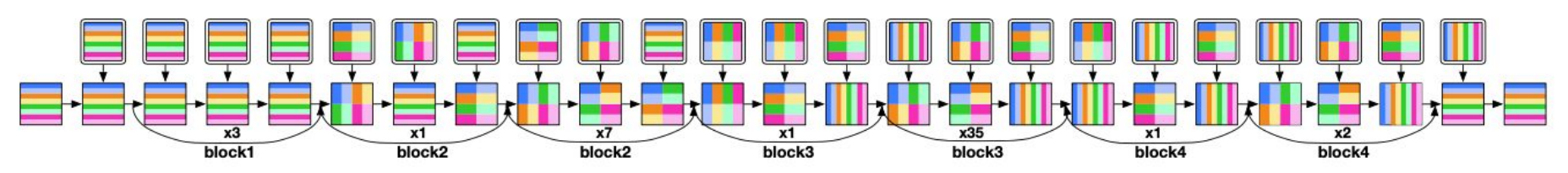
	RNN-6	RNN-8	RNN-10
Tofu	210	154	122
MX-OpPlacement	107	95	59
TF-OpPlacement	50	36	30

Table 3. Comparison of throughput (samples/second) for RNN models. The hidden size is 4096.

Evaluation - Running Time by Partition Strategy



Tofu Partition on WResNet-152-10 on 8 GPUs



Notice the widely varied partitionings. They do not follow an easily predictable pattern outside of repeated groupings.

Conclusion

- Tofu implements automatic graph partitioning at the tensor level
- TDL and symbolic interval analysis allow Tofu to generate a dataflow graph
- Coarsening the dataflow graph shrinks the search space for partitioning
- Partitioning is done quickly and efficiently with recursive optimal DP
- Partitioning optimizes for minimal communication across workers.
- Implementations using Tofu outperform other current methods when the models are very large