Supporting Very Large Models using Automatic Dataflow Graph Partitioning

Minjie Wang
New York University

Chien-chin Huang
New York University

Jinyang Li
New York University

Abstract

This paper presents Tofu, a system that partitions very large DNN models across multiple GPU devices to reduce per-GPU memory footprint. Tofu is designed to partition a dataflow graph of fine-grained tensor operators used by platforms like MXNet and TensorFlow. In order to automatically partition each operator, we propose to describe the semantics of an operator in a simple language inspired by Halide. To optimally partition different operators in a dataflow graph, Tofu uses a recursive search algorithm that minimizes the total communication cost. Our experiments on an 8-GPU machine show that Tofu enables the training of very large CNN and RNN models. It also achieves 25% - 400% speedup over alternative approaches to train very large models.

CCS Concepts • **Computer systems organization** → *Neural networks*; *Data flow architectures*.

ACM Reference Format:
Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3302424.3303953

1 Introduction

The deep learning community has been using larger deep neural network (DNN) models to achieve higher accuracy on more complex tasks over the past few years [1, 2]. Empirical evidence shows that, since the 80s, the number of parameters in the state-of-the-art neural network has doubled roughly every 2.4 years [3], enabled by hardware improvements and the availability of large datasets. As deployed DNN models remain many orders of magnitude smaller than that of a mammalian brain, there remains much room for growth. However, the size of a DNN model that can be explored today is constrained by the limited GPU device memory.

There have been many efforts to tackle the problem of limited GPU device memory. Some proposals try to fit larger models into a single GPU, e.g. by using the much larger CPU memory as a swap area for the GPU [4] or by discarding intermediate results to save memory at the cost of re-computation [5–7]. Another promising solution is to partition a DNN model across multiple GPU devices. Doing so reduces per-GPU memory footprint and comes with the additional benefit of parallel speedup. This is commonly referred to as "model parallelism" in the literature.

A DNN model consists of a large number of layers, each parameterized by its own weights. There are two approaches to realize model parallelism. One approach is to assign the computation of different layers to different devices. The second approach is to partition the tensors to parallelize each layer across devices. For very large DNN models, tensor partitioning is the better approach; not only it results in balanced per-GPU memory usage but also it necessary for speeding up popular models such as CNNs.

Tensor partitioning has been explored by existing work as a means for achieving parallel speedup [8–10] or saving memory access energy [11, 12]. Recent proposals [13–15] support partitioning a tensor along multiple dimensions and can automatically search for the best partition dimensions. The major limitation is that these proposals partition at the coarse granularity of individual DNN layers, such as fully-connected and 2D convolution layers. As such, they either develop specialized implementation for specific models [9, 13] or allow only a composition of common DNN layers [8, 10, 14, 15].

However, the vast majority of DNN development and deployment today occur on general-purpose deep learning platforms such as TensorFlow [16], MXNet [17], PyTorch [18]. These platforms represent computation as a dataflow graph of fine-grained tensor operators, such as matrix multiplication, various types of convolution and element-wise operations etc. Can we support tensor partitioning on one of these general-purpose platforms? To do so, we have built the Tofu system to automatically partition the input/output tensors of each operator in the MXNet dataflow system. This approach, which we call *operator partitioning*, is more fine-grained than layer partitioning. While we have built Tofu's prototype to work with MXNet, Tofu's solution is general and could potentially be applied to other dataflow systems such as TensorFlow.

In order to partition a dataflow graph of operators, Tofu must address two challenges. 1) How to partition the input/output tensors and parallelize the execution an individual operator? What are the viable partition dimensions? 2) how to optimize the partitioning of different

operators for the overall graph? Both challenges are made difficult by the fine-grained approach of partitioning operators instead of layers. For the first challenge, existing work [13–15] manually discover how to partition a few common layers. However, a dataflow framework supports a large and growing collection of operators (e.g. 139 in MXNet), intensifying the manual efforts. Manual discovery is also error-prone, and can miss certain partition strategies. For example, [14] misses a crucial partition strategy that can significantly reduce per-worker memory footprint (Sec 7.3). For the second challenge, existing proposals use greedy or dynamic-programming based algorithms [13, 14] or stochastic searches [15]. As the graph of operators is more complex and an order of magnitude larger than the graph of layers (e.g. the graph for training a 152-layer ResNet has >1500 operators in MXNet), these algorithms become inapplicable or run too slowly (Sec 5, Table 1).

Tofu introduces novel solutions to address the above mentioned challenges. To enable the automatic discovery of an operator's partition dimensions, Tofu requires developers to specify what the operator computes using a lightweight description language called TDL. Inspired by Halide [19], TDL describes tensor computation by specifying the output tensor value at each index with simple expressions on the input tensors. The Halide-style description is useful because it makes explicit which input tensor regions are needed in order to compute a specific output tensor region. Thus, Tofu can statically analyze an operator's TDL description using symbolic execution to determine what input regions must be transferred among GPUs when tensors are divided along a specific partition dimension. To partition each tensor in the overall dataflow graph, we propose several techniques to shrink the search space. These include a recursive search algorithm which partitions the graph among only two workers at each recursive step, and graph coarsening by grouping related operators.

We have implemented a prototype of Tofu in MXNet and evaluated its performance on a single machine with eight GPUs. Our experiments use large DNN models including Wide ResNet [1] and Multi-layer Recurrent Neural Networks [20], most of which do not fit in a single GPU's memory. Compared with other approaches to train large models, Tofu's training throughput is 25% - 400% higher.

To the best of our knowledge, Tofu is the first system to automatically partition a dataflow graph of fine-grained tensor operators. Though promising, Tofu has several limitations (Sec 9). Some operators (e.g. Cholesky) cannot be expressed in TDL and thus cannot be automatically partitioned. The automatically discovered partition strategies do not exploit the underlying communication topology. Tofu is also designed for very large DNN models. For moderately sized models that do fit in the memory of a single GPU, Tofu's approach of operator partitioning are likely no better than the much simpler approach of data parallelism. Removing these limitations requires further research.

2 Background

The problem. Training very large DNN models is limited by the size of GPU device memory today. Compared with CPU memory, GPU memory has much higher bandwidth but also smaller capacity, ranging from 12GB (NVIDIA K80) to 16GB (NVIDIA Tesla V100). Google's TPU hardware has similar limitations, with 8GB attached to each TPU core [21].

Partitioning each tensor in the DNN computation across multiple devices can lower per-GPU memory footprint, thereby allowing very large models to be trained. When partitioning across k devices, each device roughly consumes $\frac{1}{k}$ times the total memory required to run the computation on one device. Furthermore, partitioning also has the important benefit of performance speedup via parallel execution. As most DNN development today is done on dataflow platforms such as TensorFlow and MXNet, our goal is to automatically partition the tensors and parallelize the operators in a dataflow graph to enable the training of very large DNN models. The partitioning should be completely transparent to the user: the same program written for a single device can also be run across devices without changes.

System setting. When tensors are partitioned, workers must communicate with each other to fetch the data needed for computation. The amount of bytes transferred divided by the computation time forms a lower bound of the communication bandwidth required to achieve competitive performance. For training very large DNNs on fast GPUs, the aggregate bandwidth required far exceeds the network bandwidth in deployed GPU clusters (e.g. Amazon's EC2 GPU instances have only 25Gbps aggregate bandwidth). Thus, for our implementation and evaluation, we target a single machine with multiple GPU devices.

3 Challenges and our approach

In order to partition a dataflow graph of operators, we must tackle the two challenges mentioned in Sec 1. We discuss these two challenges in details and explain at a high level how Tofu solves them.

```
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.

3.1 How to partition a single operator?

To make the problem of automatic partitioning tractable, we consider only a restricted parallelization pattern, which we call "**partition-n-reduce**". Suppose operator c computes output tensor O. Under partition-n-reduce, c can be parallelized across two workers by executing the *same* operator on each worker using smaller inputs. The final output tensor O can be obtained from the output tensors of both workers (O_1, and O_2) in one of the two ways. 1) O is the concatenation of O_1 and O_2 along some dimension. 2) O is the element-wise reduction of O_1 and O_2. Partition-n-reduce is crucial for automatic parallelization because it allows an operator's existing single-GPU implementation to be re-used for parallel execution. Such implementation often belongs to a highly optimized closed-source library (e.g. cuBLAS, cuDNN).

Partition-n-reduce is not universally applicable, e.g. Cholesky [22] cannot be parallelized this way. Nor is partition-n-reduce optimal. One can achieve more efficient communication with specialized parallel algorithms (e.g. Cannon's algorithm [23] for matrix multiplication) than with partition-n-reduce. Nevertheless, the vast majority of operators can be parallelized using partition-n-reduce (Sec 4.1) and have good performance.

Tensors used in DNNs have many dimensions so there are potentially many different ways to parallelize an operator. Figure 1 shows an example operator, `conv1d`, which computes 1-D convolution over `data` using `filters`. The 3-D `data` tensor contains a batch (`b`) of 1-D pixels with `ci` input channels. The 3-D `filters` tensor contains a convolution window for each pair of `ci` input and `co` output channel. The 3-D `output` tensor contains the convolved pixels for the batch of data on all output channels.

There are many ways to parallelize `conv1d` using partition-n-reduce; Figure 2 shows two of them. In Figure 2(a), the final output is a concatenation (along the `b` dimension) of output tensors computed by each worker. Each worker reads the entire `filters` tensor and half of the `data` tensor. In Figure 2(b), the final output is a reduction (sum) of each worker's output. Figure 1 shows what input tensor region each work reads from. If tensors are partitioned, workers must perform remote data fetch.

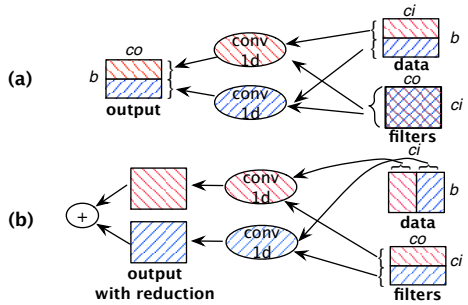


Figure 2. Two of several ways to parallelize `conv1d` according to partition-n-reduce. Each 3D tensor is represented as a 2D matrix of vectors. Different stripe patterns show the input tensor regions required by different workers.

Prior work [13–15] manually discovers the partition strategies for a few common DNN layers. Some [14, 15] have ignored the strategy that uses output reduction (i.e. Figure 2(b)), which we show to have performance benefits later (Sec 7.3). Manual discovery is tedious for a dataflow system with a large number of operators (341 and 139 in TensorFlow and MXNet respectively). Can one support automatic discovery instead?

Our approach. Tofu analyzes the access pattern of an operator to determine all viable partition strategies. As such, we require the developer of operators to provide a succinct description of what each operator computes in a light-weight language called TDL (short for Tensor Description Language). An operator's TDL description is separate from its implementation. The description specifies at a high-level how the output tensor is derived from its inputs, without any concern for algorithmic or architectural optimization, which are handled by the operator's implementation. We can statically analyze an operator's TDL description to determine how to partition it along different dimensions. Sec 4 describes this part of Tofu's design in details.

3.2 How to optimize partitioning for a graph?

As each operator has several partition strategies, there are combinatorially many choices to partition each tensor in the dataflow graph, each of which has different execution time and per-GPU memory consumption.

It is a NP-hard problem to partition a general dataflow graph for optimal performance [24–27]. Existing proposals use greedy or dynamic-programming algorithm to optimize a mostly linear graph of layers [13, 14], or perform stochastic searches [15, 28, 29] for general graphs. The former approach is faster, but still impractical when applied on fine-grained dataflow graphs. In particular, its running time is proportional to the number of ways an operator can be partitioned. When there are 2^m GPUs,

```
@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.

each input/output tensor of an operator can be partitioned along a combination of any 1, 2, ..., or m dimensions, thereby dramatically increasing the number of partition strategies and exploding the search time.

Our approach. We use an existing dynamic programming (DP) algorithm [14] in our search and propose several key techniques to make it practical. First, we leverage the unique characteristics of DNN computation to "coarsen" the dataflow graph and shrink the search space. These include grouping the forward and backward operations, and coalescing element-wise or unrolled operators. Second, to avoid blowing up the search space in the face of many GPUs, we apply the basic search algorithm recursively. In each recursive step, the DP algorithm only needs to partition each tensor in the coarsened graph among two "groups" (of GPUs). Sec 5 describes this part of Tofu's design in details.

4 Partitioning a single operator

This section describes TDL (Sec 4.1) and its analysis (Sec 4.2).

4.1 Describing an operator

Our Tensor Description Language (TDL) is inspired by Halide[19]. The core idea is "*tensor-as-a-lambda*", i.e. we represent tensors as lambda functions that map from coordinates (aka index variables) to values, expressed as a TDL expression. TDL expressions are side-effect free and include the following:

- 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.

Reducers are commutative and associative functions that aggregate elements of a tensor along one or more dimensions. Tofu supports `Sum`, `Max`, `Min` and `Prod` as built-in reducers. It is possible to let programmers define custom reducers, but we have not encountered the need to do so.

We implemented TDL as a DSL using Python. As an example, Figure 3 shows the description of `conv1d`, whose output is a 3D tensor defined by `lambda b, co, x: ...` Each element of the output tensor is the result of reduction (`Sum`) over an internal 2D tensor (`lambda ci, dx: ...`) over both `ci` and `dx` dimensions.

Opaque function. We have deliberately designed TDL to be simple and not Turing-complete. For example, TDL does not support loops or recursion, and thus cannot express sophisticated computation such as Cholesky decomposition. In such cases, we represent the computation as an *opaque function*. Sometimes, such an operator has a batched-version that can be partitioned along the batch dimension. Figure 3 shows the TDL description of the operator `batch_cholesky`. The output is a 3-D tensor (`lambda b,i,j:...`) where the element at (b, i, j) is defined to be the (i, j) element of the matrix obtained from performing Cholesky on the b-th slice of the input tensor. Note that, `batch_mat[b, :, :]` represents the $\texttt{b}^{th}$ slice of the `batch_mat` tensor. It is syntactic sugar for the lambda expression `lambda r, c: batch_mat[b, r, c]`.

Describing MXNet operators in TDL. Ideally, operator developers should write TDL descriptions. As Tofu is meant to work with an existing dataflow system (MXNet), we have written the descriptions ourselves as a way to bootstrap. We found that TDL can describe 134 out of 139 MXNet v0.11 operators. Out of these, 77 are simple element-wise operators; 2 use the opaque function primitive, and 11 have output reductions. It takes one of the authors one day to write all these descriptions; most of them have fewer than three LoC. Although we did not build Tofu's prototype for TensorFlow, we did investigate how well TDL can express TensorFlow operators. We found that TDL can describe 257 out of 341 TensorFlow operators. Out of these, 140 are element-wise operators; 22 use the opaque function. For those operators that cannot be described by TDL, they belong to three categories: sparse tensor manipulations, operators with dynamic output shapes and operators requiring data-dependent indexing. MXNet has no operators in the latter two categories.

TDL vs. other Halide-inspired language. Concurrent with our work, TVM [30] and TC [31] are two other Halide-inspired DSLs. Compared to these DSLs, TDL is designed for a different purpose. Specifically, we use TDL to analyze an operator's partition strategies while TVM and TC are designed for code generation to different hardware platforms. The different usage scenarios lead to two design differences. First, TDL does not require users to write intricate execution schedules – code for describing how to perform loop transformation, caching, and mapping to hardwares, etc. Second, TDL supports opaque functions that let users elide certain details of the computation that are not crucial for analyzing how the operator can be partitioned.

4.2 Analyzing TDL Descriptions

Tofu analyzes the TDL description of an operator to discover its basic partition strategies. A basic partition strategy parallelizes an operator for 2 workers only. Our search algorithm uses basic strategies recursively to optimize partitioning for more than two workers (Sec 5.2).

A partition strategy can be specified by describing the input tensor regions required by each worker to perform its "share" of the computation. This information is used later by our search algorithm to optimize partitioning for the dataflow graph and to generate the partitioned graph in which required data is fetched from different workers.

Obtaining input regions from a TDL description is straightforward if tensor shapes are known. For example, consider the following simple description:

```
def shift_two(A): B = lambda i : A[i+2]; return B
```

Suppose we want to partition along output dimension `i`. Given `i`'s concrete range, say $[0, 9]$, we can compute that the worker needs `A`'s data over range $[2, 6]$ (or $[7, 11]$) in order to compute `B` over range $[0, 4]$ (or $[5, 9]$).

Analyzing with concrete ranges is hugely inefficient as a dataflow graph can contain thousands of operators, many of which are identical except for their tensor shapes (aka index ranges). Therefore, we perform TDL analysis in the abstract domain using *symbolic interval analysis*, a technique previously used for program variable analysis[32], boundary checking[33], parameter validation[34].

Symbolic interval analysis. Suppose the output tensor of an operator has n dimensions and is of the form `lambda x1, ..., xn : ...`. We consider the range of index variable `xi` to be [0, $\mathcal{X}_i$], where $\mathcal{X}_i$ is a symbolic upper bound. We then symbolically execute the lambda function to calculate the symbolic intervals indicating the range of access on the operator's input tensors.

Symbolic execution should keep the range as precise as possible. To do so, we represent symbolic interval ($\mathcal{I}$) as an affine transformation of all symbolic upper bounds,

$$\mathcal{I} \triangleq [\Sigma_i l_i \mathcal{X}_i + c,\ \Sigma_i u_i \mathcal{X}_i + c], l_i, u_i, c \in \mathbb{R} \quad (1)$$

In equation 1, l_i, u_i and c are some constants. Thus, we can represent $\mathcal{I}$ as a vector of $2 * n + 1$ real values $\langle l_1, ..., l_n, u_1, ..., u_n, c\rangle$. Let $\texttt{ZV}[u_i = a]$ denote a vector of all 0s except for the position corresponding to u_i which has value a. By default, lambda variable `xi` for dimension i is initialized to $\texttt{ZV}[u_i = 1]$.

Our representation can support affine transformation on the intervals, as shown by the allowed interval arithmetic in Figure 4. Product or comparison between two intervals are not supported and will raise an error. We did not encounter any such non-affine operations among MXNet operators.

TDL description: `lambda x1, ..., xi, ..., xn: ...`

$$\mathcal{I} \triangleq \langle l_1, ..., l_n, u_1, ..., u_n, c\rangle$$
$$\mathcal{I} \pm k, k \in \mathbb{R} = \langle l_1, ..., l_n, u_1, ..., u_n, c \pm k\rangle$$
$$\mathcal{I} \times k, k \in \mathbb{R} = \langle l_1 k, ..., l_n k, u_1 k, ..., u_n k, c * k\rangle$$
$$\mathcal{I}/k, k \in \mathbb{R} = \langle l_1/k, ..., l_n/k, u_1/k, ..., u_n/k, c/k\rangle$$
$$\mathcal{I} \pm \mathcal{I}' = \langle l_1 \pm l_1', ..., u_1 \pm u_1', ..., c \pm c'\rangle$$

Figure 4. Tofu's symbolic interval arithmetic.

Discover operator partition strategies. Using the symbolic interval analysis, we infer the input regions required by each of the 2 workers for every partitionable dimension. There are two cases.

Case-1 corresponds to doing partition-n-reduce without the reduction step. In this case, each partition strategy corresponds to some output dimension. Suppose we are to partition `conv1d`'s output tensor along dimension `b`. We use two different initial intervals for lambda variable `b`, $\texttt{ZV}[u_\texttt{b} = \frac{1}{2}]$ and $\texttt{ZV}[l_\texttt{b} = \frac{1}{2}, u_\texttt{b} = 1]$, in two separate analysis runs. Each run calculates the input regions needed to compute half of the output tensor. The result shows that that each worker reads half of the `data` tensor partitioned on the `b` dimension and all of the `filter` tensor, as illustrated in Figure 2(a). Similarly, the analysis shows how to partition the other output dimensions, `co` and `x`. Partitioning along dimension `x` is commonly referred to as parallel convolution with "halo exchange" [9, 12, 13].

Case-2 corresponds to doing partition-n-reduce with the reduction step. In this case, we partition along a reduction dimension. In the example of Figure 3, the reduction dimensions corresponding to `ci` and `dx` in `Sum(lambda ci, dx: ...)`. The analysis will determine that, when partitioning along `ci`, each partially reduced tensor will require half of the `data` tensor partitioned on the second dimension and half of the `filter` tensor partitioned on the first dimension, as shown in Figure 2(b). Similar analysis is also done for dimension `dx`. Out of 47 non-element-wise MXNet operators describable by TDL, 11 have at least one reduction dimension.

5 Partitioning the dataflow graph

To partition a dataflow graph, one needs to specify which partition strategy to use for each operator. This section describes how Tofu finds the best partition plan for a dataflow graph.

Different plans result in different running time and per-worker memory consumption, due to factors including communication, GPU kernel efficiency and synchronization. Finding the best plan is NP-hard for an arbitrary dataflow graph [35]. Recent work has proposed an algorithm based on dynamic programming (DP) for partitioning a certain type of graphs. Sec 5.1 presents techniques

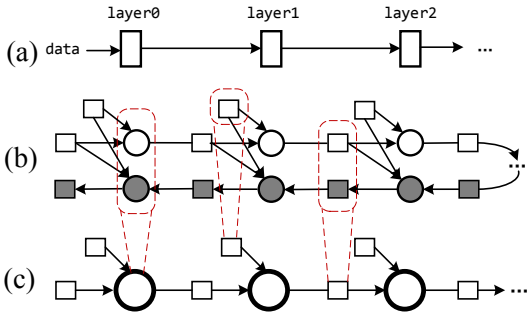


Figure 5. (a) Layer graph of a MLP model. (b) Its dataflow graph including forward and backward computation (in grey). (c) Coarsened graph. For cleanness, we only illustrate one operator group, one group for activation tensors and one group for weight tensor (dashed lines).

to make a dataflow graph applicable to DP, and Sec 5.2 improves search time via recursion.

Optimization goal. Ideally, our optimization goal should consider both the end-to-end execution time of the partitioned dataflow graph and the per-worker memory consumption. Unfortunately, neither metric can be optimized perfectly. Prior work [15] optimizes the approximate end-to-end execution time by minimizing the sum of total GPU kernel execution time and total data transfer time.

In Tofu, we choose to minimize the total communication cost based on two observations. First, the GPU kernels for very large DNN models process large tensors and thus have similar execution time no matter which dimension its input/output tensors are partitioned on. Consequently, a partition plan with lower communication cost tends to result in lower end-to-end execution time. Second, the memory consumed at each GPU worker is used in two areas: (1) for storing a worker's share of tensor data, (2) for buffering data for communication between GPUs. The memory consumed for (1) is the same for every partition plan: for k GPUs, it is always $1/k$ times the memory required to run the dataflow graph on one GPU. The memory consumed for (2) is proportional to the amount of communication. Therefore, a partition plan with lower communication cost results in a smaller per-worker memory footprint.

5.1 Graph coarsening

The algorithm in [14] is only applicable for linear graphs[1], such as the graph of DNN layers shown in Figure 5(a). Dataflow graphs of fine-grained operators are usually non-linear. For example, Figure 5(b) is the non-linear dataflow graph of the same DNN represented by Figure 5(a). Here, we propose to "coarsen" a dataflow graph into a linear one by grouping or coalescing multiple operators or tensors.

Grouping forward and backward operations. Almost all DNN models are trained using gradient-based optimization method. The training includes a user-written forward propagation phase to compute the loss function and a system-generated backward propagation phase to compute the gradients using the chain rule. Thus, we coarsen as follows:

- Each forward operator (introduced by the user) and its auto-generated backward operators (could be more than one) to form a group.
- Each forward tensor (e.g. weight or intermediate tensors) and its gradient tensor form a group. If a (weight) tensor is used by multiple operators during forward propagation and thus has multiple gradients computed during backward propagation, the chain rule requires them to be summed up and the summation operator is added to the group as well.

Figure 5(c) shows the coarsened dataflow graph for a MLP model. As forward and backward operators for the same layer are grouped together, the resulting graph becomes isomorphic to the forward dataflow graph. For MLPs and CNNs, their coarsened graphs become linear. We perform the DP-based algorithm [14] on the coarsened graph. When the algorithm adds a group in its next DP step, we perform a brute-force combinatorial search among all member operators/tensors within the group to find the minimal cost for adding the group.This allows tensors involved in the forward and backward operators to be partitioned differently, while [14] forces them to share the same partition configurations. As there are only a few operators (typically 2) in each group, the cost of combinatorial search is very low.

Coalescing operators. In DNN training, it makes sense for some operators to share the same partition strategy. These operators can be merged into one in the coarsened dataflow graph. There are two cases:

- *Merging consecutive element-wise operators*, because the input and output tensors of an element-wise operator should always be partitioned identically. We analyze the TDL description to determine if an operator is element-wise. Consecutive element-wise operators are very common in DNN training. For instance, almost all gradient-based optimizers (e.g. SGD, Adam, etc.) are composed of only element-wise operators.
- *Merging unrolled timesteps*. Recurrent neural networks (RNNs) process a variable sequence of token over multiple timesteps. RNN has the key property that different time steps share the same computation logic and

[1] We say a graph G is linear if it is homeomorphic to a chain graph G', meaning there exists a graph isomorphism from some subdivision of G to some subdivision of G' [36]. Note that a "fork-join" style graph is linear by this definition.

	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

Table 1. Time to search for the best partition for 8 workers. WRestNet-152 and RNN-10 are two large DNN models described in Sec 7.

weight tensors. Thus, they should be coalesced to share the same partition strategy. As a result, the dataflow graph of a multi-layer RNN becomes a chain of coalesced and grouped operators. To detect operators that belong to different timesteps of the same computation, we utilize how RNN is programmed in DNN frameworks. For example, systems like MXNet and PyTorch call a built-in function to unroll a basic unit of RNN computation into many timesteps, allowing Tofu to detect and merge timesteps.

5.2 Recursive partitioning

When there are more than two workers, each operator can be partitioned along multiple dimensions. This drastically increases the number of partition strategies available to each operator and explodes the running time of the DP-based search algorithm.To see this, consider the coarsened graph of Figure 5(b). Every operator group has two input tensor groups and one output tensor group. Each tensor group contains one forward tensor and one gradient tensor. At each step, the DP algorithm needs to consider all the possible configurations of an operator group including different ways to partition the six input/output tensors. For each 4D tensor used in 2D-convolution, there are in total 20 different ways to partition it evenly across 8 workers. Hence, the number of possible configurations of 2D-convolution's operator group is $20^6 = 6.4 \times 10^7$. Although not all the dimensions are available for partition in practice (e.g. the convolution kernel dimension is usually very small) , the massive search space still results in 8 hours of search time when partitioning the WResNet-152 model (Table 1).

Our insight is that the basic DP search algorithm can be *recursively* applied. For instance, a matrix, after being first partitioned by row, can be partitioned again. If the second partition is by column, the matrix is partitioned into a 2×2 grid; if the second partition is by row, the matrix is partitioned into four parts along the row dimension.

This observation inspires our recursive optimization algorithm to handle $k = 2^m$ GPUs:

1. Given a dataflow graph G, run the DP algorithm with coarsening to partition G for two worker groups, each consisting of 2^{m-1} workers. Note that each tensor is only partitioned along *one* dimension.
2. Consider the partitioned dataflow graph as consisting of two halves: G_0 for worker group#0 and G_1 for worker group#1. Each half also contains the data fetched from the other group as extra input tensors.
3. Repeat step 1 on G_0 and apply the partition result to G_1 until there is only one worker per group.

This recursive algorithm naturally supports partitioning along multiple dimensions. Figure 6 illustrates two recursive steps using an example dataflow graph (for brevity, we only show one matrix multiplication operator in the graph). Note the recursion must be done over the entire dataflow graph instead of a single operator, as the partition plan of the previous recursive step will influence the global decision of the current one.

While the recursive algorithm may seems straightforward, it is less obvious why the resulting partition plan has the optimal overall communication cost. In particular, the recursive algorithm chooses a sequence of basic partition plans $\{\mathcal{P}_1, \mathcal{P}_2, ...\mathcal{P}_m\}$ in m recursive steps, and we need to prove that no other sequence of choices leads to a better plan with a smaller communication cost. The main insight of our proof is that the partition plan decided in each recursive step is commutative (i.e, choosing partition plan $\mathcal{P}$ followed by $\mathcal{P}'$ results in the same total communication cost as choosing $\mathcal{P}'$ followed by $\mathcal{P}$.) Based on this insight, we derive the following property and use it to prove optimality.

Theorem 1. *Let the total communication cost incurred by all worker groups at step i be δ_i. Then $\delta_i \leq \delta_{i+1}$.*

Suppose $\{\mathcal{P}_1, \mathcal{P}_2, ...\mathcal{P}_m\}$ is the sequence of partition plans chosen and it is not optimal. Then there exists a different sequence $\{\mathcal{P}'_1, \mathcal{P}'_2, ...\mathcal{P}'_m\}$ with smaller total cost. Hence, there must be two consecutive steps $k-1$ and k, such that $\delta_{k-1} \leq \delta'_{k-1}$ and $\delta'_k < \delta_k$. We can show that, by choosing $\mathcal{P}'_k$ instead of $\mathcal{P}_k$ at step k, the search could have produced a better partition plan. This contradicts the optimality of the DP algorithm. The full proof is included in our techincal report [37].

If the number of GPUs k is not a power of two, we factorize it to $k = k_1 * k_2 * ... * k_m$, where $k_i \geq k_{i+1}$ for all i. At each step i in the recursive algorithm, we partition the dataflow graph into k_i workers in which each partition strategy still partitions a tensor along only one dimension but across k_i workers.

The benefits of recursion. Recursion dramatically cuts down the search time by partitioning along only one dimension at each step. For example, the number of configurations to be enumerated at each step for a 2D-convolution operator group is only $4^6 = 4096$. Therefore, the total number of partition strategies searched for the 2D-convolution operator with 8 workers (3 recursive

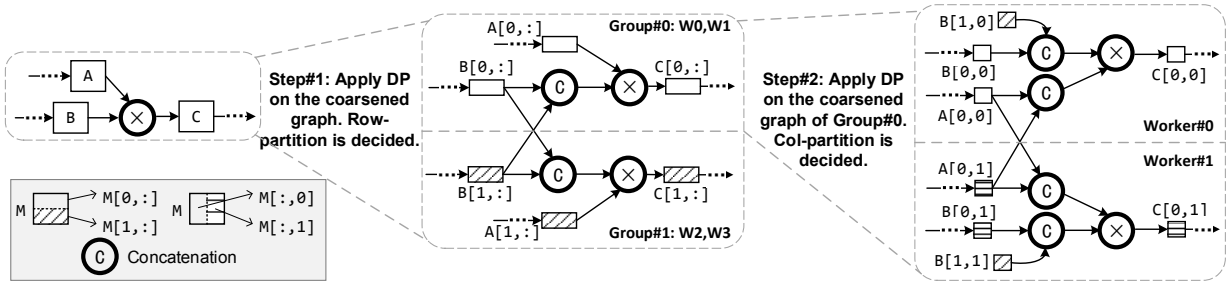


Figure 6. Recursively partition a dataflow graph to four workers. Only one matrix multiplication is drawn for cleanness. In step#1, every matrix is partitioned by row, and for group#0, `B[1,:]` is fetched from the other group. Because of this, `B[1,:]` becomes an extra input in step#2 when the graph is further partitioned to two workers. Because step#2 decides to partition every matrix by column, every matrix is partitioned into a 2x2 grid, with each worker computes one block.

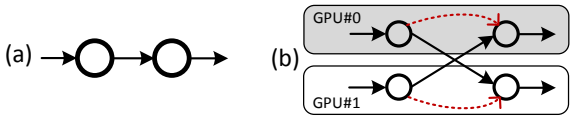


Figure 7. (a) Original dataflow graph; (b) Partitioned graph with extra control dependencies (dashed lines).

steps) is $3 * 4096$, which is far fewer than 20^6 when recursion is not used. Table 1 shows the search time for two common large DNN models when applying the original DP algorithm on coarsened graph without and with recursion.

As another important benefit, recursion finds partition plans that work well with common hierarchical physical interconnects which have less aggregate bandwidth near the top of the hierarchy. For example, many commercial servers group GPUs by faster PCI-e buses first and then connect the groups with slower QPI buses or Infinibands. As theorem 1 indicates, Tofu assigns worker groups with less communication near the top of the hierarchical interconnects in earlier steps of the recursion.

6 Optimizations in generating the partitioned graph

Once the search algorithm determines how to partition for every tensor and operator, Tofu generates a corresponding partitioned dataflow graph. The graph generation process is mostly straightforward save for two optimizations, which are crucial to keep the per-worker memory consumption low.

Leveraging the existing memory planner. Systems like MXNet and TensorFlow have their own memory planners to statically allocate and re-use memory buffers among operators according to their dependencies. Ideally, the per-worker memory consumption for k workers should be $1/k$ of the original memory consumption. In our initial implementation, per-worker memory consumption far exceeded the expected amount. We found that this is because the partitioning of a dataflow graph changes the dependencies between original operators. Figure 7 illustrates an example. In the original graph, the second operator can reuse the memory buffer of the first one (such as the workspace of a convolution operator) due to the dependency between the two. Naive graph generation may result in the graph with solid edges in Figure 7(b), in which the two operators executed by each worker no longer have a direct dependency between them and thus allows no immediate memory-reuse. To fix this, Tofu maintains the original operator dependencies on each worker by generating the extra control dependencies (dashed lines), so that the memory planner can immediately re-use buffers across dependent operators.

Fusing operators for remote data fetch. For each operator in the original graph, Tofu generates a copy for each GPU worker in the partitioned graph. Often, these operators need to fetch data from a different worker. MXNet already supports copy, split, concatenate operators, which can be used to support data movements. A naively generated graph would use split to extract the required input regions from the other workers, copy data to the local worker, and concatenate them together to assemble the input region needed by the operator's GPU kernel. Extra reduce operators can also be generated if the output tensors of different workers need to be aggregated according to the partition strategy used. Execution of such graphs results in many intermediate memory blocks, increasing the per-worker memory consumption. To mitigate this, we wrote a custom GPU kernel called MultiFetch to retrieve remote data and assemble the input region in-place using CUDA Unified Virtual Addressing (UVA).CUDA UVA allows a kernel running on one GPU to directly access the memory on another, which avoids explicit data copying before kernel execution. Our MultiFetch kernel takes multiple pointers to the memory blocks of the input regions from the other GPUs and assembles them in one kernel launch.

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.

Beyond the two optimizations described above, we also spread out the reduction workload to all GPUs (all-reduce) when performing output reduction. This is important for avoiding any single aggregation bottleneck. We also find that the MXNet scheduler can execute the remote fetch operator much earlier than required, resulting in memory being occupied for longer than necessary. We adopt the same technique proposed by TensorFlow to delay the execution of the remote fetch operator.

7 Evaluation

This section evaluates Tofu and compares with various alternative approaches. The highlights of our results are the following:

- Tofu can train very large WResNet and RNN models across 8 GPUs with high throughput that is within 60%-98% of a hypothetical ideal baseline.
- Except for a few exceptions, Tofu outperforms existing alternative approaches including shrinking the mini-batch size used for training, swapping to CPU memory, and placing different operators on different GPUs.
- Tofu's recursive partition algorithm leads to better training throughput than existing partition algorithms [14, 35] and simple heuristics.
- The overall partition plan found by Tofu is highly non-trivial, even though the underlying DNN model has a regular structure.

7.1 Experimental setup

Prototype Implementation. We implement Tofu based on MXNet 0.11. The TDL components (operator descriptions and the region analyzer) are written in Python (2K LoC). The recursive search algorithm is implemented as a graph transformation pass in NNVM (4K LoC in C++). As we need information from gradient calculation and shape inference, we also made slight modifications to the corresponding NNVM passes.

Testbed: The experiments run on an EC2 p2.8xlarge instance. The instance has 8 K80 GPUs with 12GB memory each. GPUs are connected by PCI-e bus with 21GB/s peer-to-peer bandwidth. It has 32 virtual CPU cores and 488GB CPU memory. The CPU-GPU bandwidth is 10GB/s.

DNN Benchmarks: We evaluate the WResNet [1] convolutional neural network and recurrent neural network (RNN). We choose these two benchmarks because they correspond to very large models. We do not evaluate those well-known DNNs that fit into a single GPU's memory, such as AlexNet, VGGNet and Inception.

WResNet [1] is a widened version of the original residual network model [38]. It has a widening scalar to increase the number of channels on each convolution layer. The model size grows quadratically as each weight tensor is widened on both the input and output channel. WResNet has been shown to achieve a better accuracy when the model is widened by $10\times$. Due to the memory limitation, such improvement is only demonstrated on CIFAR-10 dataset of small images (32x32) using a 50-layer model. We experiment with WResNet on ImageNet dataset with images of size (224x224). We also test different model variations: widening scalar from 4 to 10 on networks with 50, 101 and 152 layers. We use notations like WResNet-101-8 to denote the 101-layer ResNet model widened by 8 times.

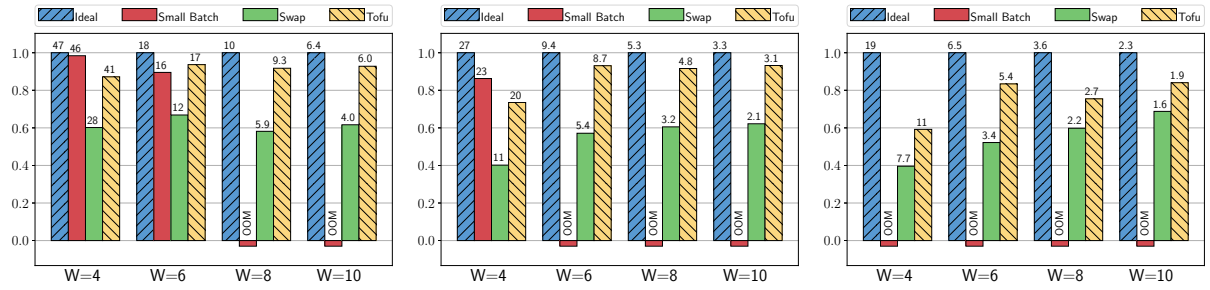
For RNN, there are two ways to increase model capacity. The number of neurons in each hidden layers can be increased, and multiple RNN layers can be stacked to form a deeper model. Researchers have explored very large RNNs by increasing the number of RNN layers to 8 [28, 29], or by using a large hidden layer size such as 8192 [20]. We use the model described in [20], and test it with different configurations varying from 6 to 10 layers with 4K, 6K, and 8K hidden sizes. All RNN model variants use LSTM cell [39] and are unrolled for 20 steps as in [20]. We use the RNN-8-8K to denote the 8-layer RNN model with 8K hidden size.

All the benchmarks are tested by running a full training iteration including forward/backward propagation and weight update. State-of-the-art weight optimizers such as Adam [40] and Adagrad [41] must maintain an extra buffer for storing the gradient history. Therefore, a model of weight size W needs to consume at least $3W$ size of memory for storing the weight, gradient and the history tensors. Table 2 shows the total weight memory consumption for all the benchmarks.

Baseline and Alternatives for Comparison. We consider an ideal baseline and several alternative approaches for comparison.

Ideal is a hypothetical baseline that assumes each GPU has infinite memory. We simulate this by modifying the memory allocator of MXNet to always return the same memory block. We measure the single-GPU throughput number and multiply it by 8 as the performance of running on 8 GPUs.

SmallBatch is a baseline that tries to fit the model in a single GPU by reducing the mini-batch size. Like the



(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.

ideal baseline, we scale the single-GPU throughput number by 8 for 8 GPUs. Thus, neither SmallBatch nor Ideal baseline consider the communication cost and represent performance upper-bounds.

Swapping [4, 42, 43] is a baseline that swaps in/out GPU memory blocks to CPU. There are many ways to design the swapping policy. Our baseline combines many of these techniques in order for a fair comparison. First, our baseline follows the design of [43], which includes a least recently used algorithm to decide the tensor to be swapped out and a prefetching unit based on the execution. This supports swapping in/out any memory block instead of only activation tensors as in [4]. Second, read-only tensors are copied to CPU only once and simply dropped the next time they are to be swapped out. Third, we combine dataflow analysis similar to [4] to disable swapping out memory blocks that will soon be used.

Operator Placement [2, 28, 44, 45] assigns operators to different devices to spread out memory usage. For RNN, this baseline assigns the computation of different layers to different GPUs to leverage the pipelining effect, as it is originally proposed in [44]. If there are more layers than the number of GPUs, we balance the assignment in a round-robin manner. Operator placement does not perform well for CNNs due the mostly serial layer-by-layer execution. Therefore, we skip this baseline for all WResNet benchmarks.

In our experiments, the ideal baseline uses a batch size that can saturate the GPU for the best performance. SmallBatch, Swapping and Tofu all use the largest batch size that make the execution fit in the GPU memory.

7.2 Training Large and Deep Models

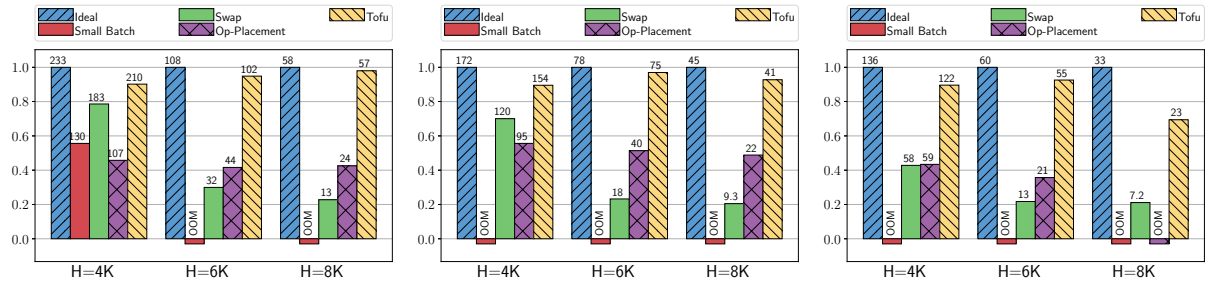
We show the performance of Tofu and compare it to the ideal baseline and alternatives. Since different systems use different batch sizes to achieve the best performance, we use throughput (samples/sec) instead of training time per iteration as the metric for comparison In Figures 8 and 9, each bar shows the throughput relative to the ideal baseline performance. The absolute throughput numbers are shown on top of each bar. *OOM* indicates out-of-memory error.

WResNet Performance. Figure 8 shows the WResNet throughput achieved by different systems. The ideal baseline uses a global batch size of 128. Only 3 models, WResNet-50-4,6 and WResNet-101-4 can be fit in a single GPU memory by shrinking the batch size (aka SmallBatch).

Tofu can achieve 60%-95% of the ideal performance for all the models. The largest model, WResNet-152, has the biggest performance gap. This is because we configured the ideal baseline to use a much larger mini-batch size for peak throughput without any consideration for memory consumption. For example, the ideal baseline uses base size 128 for WResNet-152-4 while Tofu can fit at most 32. The batch sizes used by Tofu ranges from 8 (for WResNet-152-10) to 128 (for WResNet-50-4). Tofu performs better than alternatives in all scenarios except for WResNet-50-4 and WResNet-101-4, in which SmallBatch achieves 12% and 15% better throughput than Tofu. This is because convolution kernels have good GPU utilization even for small batch sizes. However, SmallBatch runs out of memory for most of the models in Figure 8.

As shown in Figure 8, swapping is 20%-63% slower than Tofu across all the models. This is due to swapping's much larger communication amount. Although we implemented prefetching to "hide" communication latency in swapping, the CPU-GPU communication is the bottleneck as all 8 GPUs share the same bandwidth to communicate with the CPU.

RNN Performance. Figure 9 shows the throughput for RNNs. The ideal baseline uses a (global) batch size of 512. Tofu performs better than the other baselines in all RNN configurations, achieving 70% - 98% of ideal throughput. Unlike the WResNet experiments, SmallBatch does not achieve better throughput than Tofu in any RNN configuration. This is because the main RNN computation is matrix multiplication, which has much



(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.

less arithmetic density than convolution. Thus, performing matrix multiplication using small batch sizes results in decreased GPU utilization. The same reasoning explains why Tofu's relative performance with the largest model (RNN-10-8K) is worse than with other RNN models; Tofu uses a batch size of 128 in order to fit RNN-10-8K in memory while it uses larger batch sizes (256 or 512) with other RNN models. As is also the case with WResNet, SmallBatch results in OOM for larger RNN configurations.

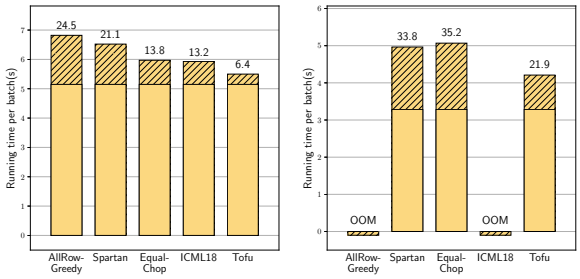
Operator placement achieves 38%-61% of Tofu's throughput and cannot train RNN-10-8K (OOM). Two reasons contribute to the lower performance. First, layer-wise placement results in imbalanced load because the number of layers is not a multiple of the number of GPUs. Second, layer-wise placement relies on pipelined parallelism: GPU-1 executes the first operator in the first layer and forwards its result to GPU-2. GPU-2 can execute the first operator in the second layer while GPU-1 concurrently executes the second node in the first layer. Pipelined parallelism cannot fully saturate GPUs at all times: e.g. GPU-2 is idle while GPU-1 executes its first operator. By contrast, Tofu parallelizes the execution of each operator and keeps all GPUs busy at all times.

Swapping achieves 23% - 30% throughput of Tofu and 48% - 53% throughput of operator placement when the weight size is large. The main reason is that many tensors may be used simultaneously in RNN training. To fully saturate a GPU, most deep learning frameworks, including MXNet and Tensorflow, schedule operators immediately when they are ready. RNN's mesh-like dataflow graph results in more tensors to be used at the same time. When the weight size is large, the amount of swapping increases significantly. Coupled with the CPU-GPU communication bottleneck, swapping is unable to achieve good throughputs for RNNs.

Comparing with TensorFlow. We compare with Tensorflow v1.8 (using Op-Placement) for training RNNs. Table 3 shows the throughputs for running on RNN-6-4K, RNN-8-4K, and RNN-10-4K. For additional comparison points, we also include MXNet (using Op-Placement). Note that the throughputs of Tofu and MXNet are same as those in Figure 9. Tensorflow's throughput is roughly half of MXNet and about 23% of Tofu. As Tensorflow and MXNet use the same operator kernel implementations, we originally expected the two systems to have similar throughput. However, further investigation shows that TensorFlow does not support in-place gradient aggregation which may be crucial for the performance of large RNNs.

	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.



(a) RNN-4-8K **(b)** WResNet-152-10

Figure 10. Comparison of different partition algorithms using RNN-4-8K and WResNet-152-10 on 8 GPUs. Striped parts show the overhead (percentage) due to communication.

7.3 Comparing different partition algorithms

We have compared Tofu's search time with the original DP algorithm [14] in Sec 5.2 (Table 1). We now compare the quality of partition plan found by Tofu vs. [14] and various other heuristics.

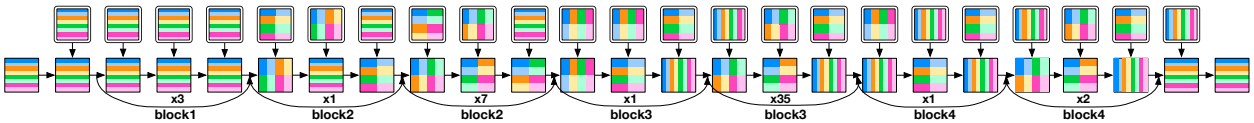


Figure 11. The partition found by Tofu for WResNet-152-10 on 8 GPUs. We draw the weight tensors (top row) and the activation/data tensors (bottom row) used by convolution operators. Partitioning is marked by the tiles and each color shows the tiles owned by the same GPU. The vertical and horizontal dimensions of an activation tensor indicate the batch and channel dimensions. 'xN' symbol means the corresponding block is repeated N times.

The simplest heuristic (`AllRow-Greedy`) partitions all tensors along the first dimension and partitions each operator using the best strategy given that its input/output tensors are partitioned on the first dimension. Note that, for the case of WResNet, this gives similar result as the *one-weird-trick* strategy proposed in [46], because all the convolution layers are partitioned by the batch dimension and the only fully-connected layer in WResNet occupies <1% of the total time. Our next heuristic is to greedily partition the largest tensor first (along any dimension), followed by its incident operators, followed by the second largest tensor and so on. This is equivalent to what is proposed by Spartan [35]. We also compare with Tofu's DP algorithm applied to chop each tensor equally along only one dimension (`EqualChop`). Finally, we compare with the algorithm in [14](`ICML18`) which does not consider the partition strategy of aggregating output tensors (aka output-reduction).

Figure 10 shows the execution time of training one batch on 8 GPUs for RNN-4-8K (batch size is 512) and WResNet-152-10 (batch size is 8). To see the impact of communication on the execution time, we modify the backend to skip memory copy among GPUs and measure the resulting pure computation time, which is shown as the lower light-colored portion of the bars in Figure 10.

`AllRow-Greedy` performs worse among all the algorithms and run out of memory for WResNet-152-10 because it needs to fetch too much data from the other GPUs. `Spartan` and `EualChop` reduce the communication overhead by 3%-10% but are still worse than Tofu. This result shows the benefit of partitioning a tensor along multiple dimensions. `ICML18` is 7% slower than Tofu for RNN-4-8K and results in OOM for WResNet-152-10 due to the lack of output-reduction. After adding output-reduction, `ICML18` can find the same strategy as Tofu, albeit with a much longer search time (see Table 1).

7.4 Partition Results

Figure 11 shows the partition found by Tofu for WResNet-152-10. ResNet-152 contains 4 groups of residual blocks: each block includes 3 convolutions and is repeated 3, 8, 36, and 3 times for each group respectively. The lower residual blocks (those close to the input layer) have larger feature map but smaller weight tensors while the higher ones are the opposite.

We make the following observations:

- Tofu partitions both the batch and channel dimensions and the resulting partition plan is a complicated combination of different partition strategies.
- Tofu chooses different partition plans for different convolution layers within one residual block. Repeated residual blocks are partitioned in the same way except for the first block in the group which has a different configuration to shrink the initial input feature map size by half.
- As the activation tensors in lower layers are larger and the weight tensor smaller, Tofu chooses to fetch weight tensors from remote GPUs to save communication. As the weight tensors are larger in the higher layers, Tofu switches to partition strategies that fetch the relatively smaller activation tensors.

8 Related Work

Parallel DNN training. Many parallel strategies have been developed to speedup DNN training. Some strategies such as the popular data parallelism [47–50] cannot be used for training very large models because the parameters are replicated to each device. Model parallelism spreads out the model parameters to multiple GPUs, thus is suitable for training very large models. Early work[8, 9, 46] parallelizes specific classes of DNN models, and is limited in flexibility and generality. Minerva[51] and Strads[52] require users to implement extra interfaces to partition model parameters while Tofu requires no change to the user program. Another approach is to assign different layers/operators to different devices via heuristics [45] or stochastic search [28, 44]. However, operator placement only works well only when there are sufficiently many concurrent operators, and thus is not suitable for DNN models with a deep stack of layers.

Out-of-core DNN training. This includes recomputation on demand [5–7] , swapping and prefetching from host memory [4, 42, 43]. Recomputation is not viable for large weight tensors. Swapping with host memory reduces the opportunity of co-locating computation and data, and scales poorly when there are multiple GPUs. None of them can efficiently utilize the aggregated memory capacity of multiple cards as Tofu does. Moreover, Tofu can also be combined with these techniques.

Model compression. This includes network pruning [53, 54] (which removes small weight values), quantization[55] and reduced precision[56]. The compressed model can then be deployed on mobile or edge devices or to speed up the inference. However, these approaches affect model accuracy while Tofu allows exploring very large models without changing the model behavior.

Parallel tensor computing. There is a long history in developing efficient parallel systems for tensor computing. The very first effort starts from developing low-level, optimized, parallel matrix/tensor libraries [57–61]. These libraries implement efficient parallel matrix algorithms [23, 62] and tensor operations [63]. However, they have very limited programmability support and adding new operators requires tremendous manual efforts.

Many frameworks or tools have been built to ease the programming of parallel tensor computation. In the low-level, ZPL [64], Chapel [65] and Unified Parallel C [66] are parallel language supports. In the higher-level, systems such as [35, 67–71] let users write programs in high-level primitives like map and reduce. MadLinq [22] and Presto [72] let user describe operators using parallel loop primitives. Users need to express parallelism using the proper combination of these primitives. For example, implementing a parallel matrix multiplication needs to call the `shuffle` primitive in Spartan [35] or the `Collect` primitive in [71]. However, these primitives are limited (e.g. it is hard to express halo-exchange in convolution). Distributed Halide [73] lets user describe the algorithm in their DSL and specifies how it is paralleled. As there are usually multiple ways of partitioning data and computation, the efficiency varies with different implementations. Spartan [35] and Kasen [70] propose algorithm to automatically optimize array/matrix partitioning to reduce communication. [71] further improves this by also considering different parallel patterns via transformations of nested high-level primitives.

More recent proposals aim to fully automate the whole stack – user programs are written in array language and the system can distribute the data and computation automatically. There are several approaches. Cylops Tensor Framework [74] and Tensor Contraction Engine [75] are specialized systems for automatically parallelizing tensor contraction. Spartan tries to map Numpy operators to high-level map and reduce primitives and then partitions them accordingly. Others tried to leverage the parallelism among array operators. For example, Pydron [76] translates Python program into an internal dataflow graph to parallelize independent loops. [28, 44] tries to dispatch array operators to different devices automatically based on the dataflow graph. However, they are not suitable for DNN computation that is mostly sequential. Compared with previous systems, Tofu automatically discovers the partition-n-reduce parallel patterns of operators using TDL description and optimizes partitioning for the entire dataflow graph.

Data layout optimization. There have been extensive work on optimizing communication (aka remote memory access) on the multiprocessor architecture (e.g. [77–87]) or the new hardware [11–13]. Since searching the optimal solution is NP-Complete [24–27], heuristics are used in practice [27, 79]. By contrast, Tofu analyzes the relatively simpler operator description language instead of the source code, and exploits the DNN computation structure for its optimization.

9 Discussion, limitations, and future work

Fundamental limitations. Tofu only supports parallelization via partition-n-reduce, which restricts each worker to perform a coarse-grained task identical to the original computation. This pattern is not applicable to all parallelizable computation (e.g. Cholesky [22]). Furthermore, the partition-n-reduce parallel strategies do not necessarily minimize communication, and do not take advantage of the underlying interconnect topology. By contrast, parallel algorithms developed for specific computation (e.g. matrix multiplication [23, 62], tensor contraction [74]) are explicitly structured to minimize communication and exploit the interconnect topology.

Limitations of TDL. TDL is a simple language without control flow primitives and data-dependent indexing. Furthermore, Tofu does not support sparse tensor operations due to load-imbalance, even though they can usually be described in TDL. For certain operations, these limitations may be removed by supporting data-dependent partitioning (e.g. as in parallel graph computation [88]) or by sampling runtime information (e.g. as in parallel range sort [89]).

Tofu does not verify that the operator implementation matches its TDL description. Such verification is an open research problem even if the underlying implementation is open sourced. A more promising direction is to leverage recent operator code-generation tools such as TVM [30] and TC [31]. As TVM and TC are also based on Halide, our analysis techniques can be ported to analyze operators implemented in these languages.

Partition flexibility and hardware heterogeneity. Tofu always partitions every operator and tensor across all workers. For moderately sized DNN models, partitioning across all workers lead to small GPU kernels that leave a GPU unsaturated. In such scenarios, it may be beneficial to leave certain operators un-partitioned or partially partitioned among a subset of workers. Furthermore, Tofu has no support for non-uniform partitioning

when GPUs have different computing and memory capacity. Although Tofu's search algorithm tries to accommodate bandwidth differences in a hierarchical interconnect, it does not explicitly optimize communication according to the interconnect topology.

Unfortunately, Tofu's recursive search cannot be extended to address the above limitations. This is because the underlying DP algorithm cannot optimally search different device placement choices for un-partitioned, or non-uniformly-partitioned operators. Exploring stochastic search mechanisms [15, 28, 29] is a direction of future work.

10 Conclusion

We present the Tofu system, which enables the training of very large DNN models by partitioning a dataflow graph of tensors across multiple GPU devices. To automate this process, Tofu infers each operator's valid partition strategies by analyzing its semantics written in a simple description language (TDL). Tofu uses a recursive search algorithm based on dynamic programming and DNN-specific heuristics to find the best partition plan that minimizes communication for the entire dataflow graph.

Acknowledgements

This work is supported in part by the National Science Foundation under award CNS-1816717, NVIDIA AI Lab (NVAIL) at NYU, and AWS cloud credits for research. Our shepherd, Chris De Sa, and other anonymous reviewers have given helpful feedback that improved this work. We also thank Jeff Hammond for pointing us to related work in the HPC community, esp. work on tensor contraction engines.

References

[1] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *arXiv:1605.07146*, 2016.

[2] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, and Mohammad Norouzi. Google's neural machine translation system: Bridging the gap between human and machine translation. In *arxiv.org:1609.08144*, 2016.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[4] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In *Proc. of ML Systems Workshop in NIPS*, 2017.

[5] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In *Advances in Neural Information Processing Systems*, pages 4125–4133, 2016.

[6] James Martens and Ilya Sutskever. Training deep and recurrent networks with hessian-free optimization. In *Neural networks: Tricks of the trade*, pages 479–535. Springer, 2012.

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[8] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Neural Information Processing Systems (NIPS)*, 2012.

[9] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1337–1345, 2013.

[10] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, 2014.

[11] Xuan Yang, Jing Pu, Blaine Burton Rister, Nikhil Bhagdikar, Stephen Richardson, Shahar Kvatinsky, Jonathan Ragan-Kelley, Ardavan Pedram, and Mark Horowitz. A systematic approach to blocking convolutional neural networks. *arXiv preprint arXiv:1606.04209*, 2016.

[12] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 380–392. IEEE, 2016.

[13] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. *ACM SIGOPS Operating Systems Review*, 51(2):751–764, 2017.

[14] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 2279–2288, 2018.

[15] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

[16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[17] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[18] PyTorch. http://pytorch.org.

[19] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[20] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[21] Google Cloud. Tpu: System architecture.

[22] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. MadLINQ:

large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, 2012.

[23] L. E. Cannon. *A cellular computer to implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

[24] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):869–916, 1998.

[25] Ulrich Kremer. Np-completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers, Delft, The Netherlands*, 1993.

[26] Jingke Li and Marina Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the*, pages 424–433. IEEE, 1990.

[27] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of parallel and distributed computing*, 13(2):213–221, 1991.

[28] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. *arXiv preprint arXiv:1706.04972*, 2017.

[29] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A hierarchical model for device placement. In *ICLR*, 2018.

[30] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[31] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. In *arXiv:1802.04730v2*, 2018.

[32] Arnaud J Venet. The gauge domain: scalable analysis of linear inequality invariants. In *International Conference on Computer Aided Verification*, pages 139–154. Springer, 2012.

[33] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM Sigplan Notices*, volume 35, pages 182–195. ACM, 2000.

[34] Xueguang Wu, Liqian Chen, and Ji Wang. An abstract domain to infer symbolic ranges over nonnegative parameters. *Electronic Notes in Theoretical Computer Science*, 307:33–45, 2014.

[35] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. Spartan: A distributed array framework with smart tiling. In *USENIX Annual Technical Conference*, 2015.

[36] J.A Bondy and U.S.R. Murty. *Graph Theory with Applications*. Elseyier Science Publishing, 1976.

[37] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. *arXiv preprint arXiv:1807.08887*, 2018.

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[39] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[40] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[41] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[42] Taro Sekiyama, Takashi Imamichi, Haruki Imai, and Rudy Raymond. Profile-guided memory optimization for deep neural networks. *arXiv preprint arXiv:1804.10001*, 2018.

[43] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[44] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[45] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[46] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. In *arXiv:1404.5997*, 2014.

[47] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *USENIX OSDI*, 2014.

[48] H. Cui, J. Cipar, Q. Ho, J.K. Kim, S. Lee, A. Kumar, J.Wei, W. Dai, G. R. Ganger, P.B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX Annual Technical Conference*, 2014.

[49] J. Wei, W. Dai, A. Qiao, H. Cui, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E.P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *ACM Symposium on Cloud Computing (SoCC)*, 2015.

[50] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Eurosys*, 2016.

[51] Minjie Wang, Tianjun Xiao, Jianpeng Li, Jiaxing Zhang, Chuntao Hong, and Zheng Zhang. Minerva: A scalable and highly efficient training platform for deep learning. In *NIPS Workshop, Distributed Machine Learning and Matrix Computations*, 2014.

[52] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth Gibson, and Eric Xing. Strads: A distributed framework for scheduled model parallel machine learning. In *Eurosys*, 2016.

[53] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[54] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[55] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[56] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.

[57] Edward Anderson, Zhaojun Bai, J Dongarra, A Greenbaum, A McKenney, Jeremy Du Croz, S Hammerling, J Demmel, C Bischof, and Danny Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings*

of the 1990 ACM/IEEE conference on Supercomputing, pages 2–11. IEEE Computer Society Press, 1990.

[58] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127. IEEE, 1992.

[59] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, 39(2):13:1–13:24, feb 2013.

[60] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[61] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[62] Robert A. van de Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. Technical report, Austin, TX, USA, 1995.

[63] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014.

[64] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In *Languages and Compilers for Parallel Computing*, pages 96–114. Springer, 1994.

[65] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 2007.

[66] UPC Consortium. UPC language specifications, v1.2. Technical report, Lawrence Berkeley National Lab, 2005.

[67] Joe B. Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. Scihadoop: array-based query processing in hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[68] Murray Stokely, Farzan Rohani, and Eric Tassone. Large-scale parallel statistical forecasting computations in r. In *JSM Proceedings, Section on Physical and Engineering Sciences*, Alexandria, VA, 2011.

[69] SparkR: R frontend for Spark. http://amplab-extras.github.io/SparkR-pkg.

[70] Mingxing Zhang, Yongwei Wu, Kang Chen, Teng Ma, and Weimin Zheng. Measuring and optimizing distributed array programs. *Proc. VLDB Endow.*, 9(12):912–923, August 2016.

[71] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, 2016.

[72] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S. Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems (Eurosys)*, 2013.

[73] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. Distributed halide. In *Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[74] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 813–824. IEEE, 2013.

[75] So Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[76] Stefan C. Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 645–659, Broomfield, CO, October 2014. USENIX Association.

[77] David E Hudak and Santosh G Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 187–200. ACM, 1990.

[78] Kathleen Knobe, Joan D Lukas, and Guy L Steele Jr. Data optimization: Allocation of arrays to reduce communication on simd machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, 1990.

[79] Michael Philippsen. *Automatic alignment of array data and processes to reduce communication time on DMPPs*, volume 30. ACM, 1995.

[80] Igor Z Milosavljevic and Marwan A Jabri. Automatic array alignment in parallel matlab scripts. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 285–289. IEEE, 1999.

[81] J Ramanujam and P Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *Parallel and Distributed Systems, IEEE Transactions on*, 2(4):472–482, 1991.

[82] J Ramanujam and P Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 637–646. ACM, 1989.

[83] David Bau, Induprakas Kodukula, Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Solving alignment using elementary linear algebra. In *Languages and Compilers for Parallel Computing*, pages 46–60. Springer, 1995.

[84] ERIKH D'HOLLANDER. Partitioning and labeling of index sets in do loops with constant dependence vectors. In *1989 International Conference on Parallel Processing, University Park, PA*, 1989.

[85] Chua-Huang Huang and Ponnuswamy Sadayappan. Communication-free hyperplane partitioning of nested loops. *Journal of Parallel and Distributed Computing*, 19(2):90–102, 1993.

[86] Y-J Ju and H Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Languages and Compilers for Parallel Computing*, pages 344–358. Springer, 1992.

[87] Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, Ponnuswamy Sadayappan, Yongjian Chen, Haibo Lin, et al. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 348–357. IEEE, 2009.

[88] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[89] Jeff Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI)*, 2004.