2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

Improving Strong-Scaling of CNN Training by Exploiting Finer-Grained Parallelism

Nikoli Dryden*†, Naoya Maruyama*, Tom Benson*, Tim Moon*, Marc Snir†, Brian Van Essen*
*Lawrence Livermore National Laboratory
{maruyama3,benson31,moon13,vanessen1}@llnl.gov
†Department of Computer Science
University of Illinois at Urbana-Champaign
{dryden2,snir}@illinois.edu

***Abstract*—Scaling CNN training is necessary to keep up with growing datasets and reduce training time. We also see an emerging need to handle datasets with very large samples, where memory requirements for training are large. Existing training frameworks use a data-parallel approach that partitions samples within a mini-batch, but limits to scaling the mini-batch size and memory consumption makes this untenable for large samples. We describe and implement new approaches to convolution, which parallelize using spatial decomposition or a combination of sample and spatial decomposition. This introduces many performance knobs for a network, so we develop a performance model for CNNs and present a method for using it to automatically determine efficient parallelization strategies.**

We evaluate our algorithms with microbenchmarks and image classification with ResNet-50. Our algorithms allow us to prototype a model for a mesh-tangling dataset, where sample sizes are very large. We show that our parallelization achieves excellent strong and weak scaling and enables training for previously unreachable datasets.

***Index Terms*—Deep learning, HPC, convolution, algorithms, performance modeling**

I. Introduction

A key factor in the success of deep learning [1] has been the availability of sufficient compute power to meet the demands of training deep networks. GPUs have been particularly important in enabling this [2], [3]. Training a modern convolutional neural network (CNN) to convergence typically involves many iterations through large datasets, which can take days to weeks. Clusters of GPUs are often employed to help train models in more reasonable timeframes. This is especially important for researchers developing novel models or working with novel datasets, where rapid turnaround is needed. Further, dataset sizes continue to grow [4], [5], deep learning is being applied to new domains such as medicine and physics, and models are becoming more complex [6]. This necessitates continued scaling and performance improvements.

GPUs are becoming increasingly more prevalent in HPC systems; more than half of the new FLOPS added in the June, 2018 Top-500 list were due to GPUs [7]. These systems are well-suited to training CNNs, and there has been much interest in this. Commercial firms are also increasingly turning to large clusters for training. It is therefore important to scale training—for datasets with samples of all sizes—to such systems, allowing researchers to take advantage of their significant compute power.

Classic datasets have relatively small samples. ImageNet [8] images, for example, are typically resized to be 224×224 with three color channels. Emerging datasets may have significantly larger samples, especially those from large-scale numerical simulations. We consider here a dataset for detecting mesh-tangling in numerical simulations, where each sample is a 2048×2048 image with 18 channels, generated from a hydrodynamics code. Each sample requires $\sim$288 MiB (in single-precision). Larger sample sizes are easily possible depending on the simulation size and resolution, including the case where a *single* sample is too large to fit in GPU memory. This is typical of the scale of data produced by HPC applications. There are other domains where samples may be large, including medical imagery [9], drug discovery [10], video [11], and satellite imagery [12]. As training requires the activations of each layer to be preserved until backpropagation completes, and the size of activations depends on the input data size, large samples result in very high pressure on GPU memory (e.g. 16 or 32 GB on a V100).

Current deep learning software typically employs a data-parallel approach to scaling, wherein the samples in a mini-batch are partitioned between processors. Typical mini-batch sizes are 64-1000 samples, with 256 common. Processors then perform forward and backpropagation independently, only needing to synchronize parameter updates. However, data-parallel scaling is limited by the number of samples in a mini-batch, which can be difficult to increase due to generalization and convergence issues [13], [14]. Recent work has been successful at training on ImageNet with large mini-batches (e.g. 8192+ samples) [15], [16], but it is unclear if these techniques generalize to other datasets. Further, data-parallel scaling cannot reduce memory usage beyond what is required for a single sample, and therefore is not viable for very large samples.

We address both scaling and memory issues by exploiting parallelism in convolutional layers beyond data-parallelism (which we refer to as *sample* parallelism). Conceptually, we think of a convolutional layer as being specified by five dimensions: samples, height, width, channels, and filters. (See

The first and second authors contributed equally.

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00031

Section II-A. Generalizing to additional spatial dimensions is not hard, and we ignore the convolutional kernel size as it is typically small in modern CNNs.) Sample parallelism partitions layers only along the first dimension; we describe and implement algorithms to additionally partition the spatial dimensions by splitting up input samples. This enables additional parallelism, but requires extra communication in forward and backpropagation, which can make them less advantageous in some situations compared to pure sample parallelism. We also discuss algorithms for partitioning the channel and filter dimensions. These, and our implementation in LBANN [17], are described in detail in Sections III and IV.

These additional parallelism techniques make it complicated to determine how to parallelize a network to achieve good speedup on a system. Section V introduces a technique to automatically find good parallel execution strategies. Given a platform and a CNN architecture, our system uses a performance model to determine promising ways to parallelize the network, accounting for memory requirements.

In the remainder of the paper, we evaluate our algorithms with ResNet-50 [18] on ImageNet-1K classification [8] and proof-of-concept models for our mesh-tangling dataset. Training models on this dataset was previously infeasible due to memory requirements, so this is an initial demonstration of capability rather than a final model. Future models and evaluations will undoubtably improve upon this now that training is feasible.

We summarize our contributions as follows:

- We describe and implement algorithms for parallelizing convolutional layers by exploiting parallelism available from sample and spatial decompositions. We additionally describe extensions to channel and filter decompositions.
- We present a performance model for CNNs and an approach for determining good parallel execution strategies for CNNs.
- We comprehensively evaluate these implementations with microbenchmarks and end-to-end training.
- We demonstrate the feasibility of training on a dataset with very large samples.

II. Background and notation

Here we provide a brief overview of relevant background on CNNs and performance modeling, and introduce our notation. We assume the reader has a basic familiarity with deep learning.

A. Convolutional neural networks

Consider a 2D convolutional layer, with F filters of size $K \times K$, stride S, and padding P. For simplicity, we assume K is odd, and write $O = \lfloor K/2 \rfloor$ to be the number of filter entries on either side of the center. Its input consists of N samples, each with C channels, height H, and width W. Each filter is applied to each sample, resulting in N outputs with F channels, height $\tilde{H}$, and width $\tilde{W}$. These quantities define the layer, which we think of as six 4D tensors: a $N \times C \times H \times W$ input x, a $F \times C \times K \times K$ weights w, and a $N \times F \times \tilde{H} \times \tilde{W}$ output y, plus the associated gradients. We do not consider alternate storage layouts (e.g. $NHWC$) in this paper. The CNN is trained with a loss function L.

To simplify notation, we assume $S = 1$ and "same" padding. Subscripts of x and dL/dy may be "out of range" (e.g. negative); we assume these are handled with padding (these assumptions are not necessary for our work). Forward propagation is given by

$$y_{k,f,i,j} = \sum_{c=0}^{C-1} \sum_{a=-O}^{O} \sum_{b=-O}^{O} x_{k,c,i+a,j+b} w_{f,c,a+O,b+O} \quad (1)$$

In backpropagation, a layer has input dL/dy (of shape $N \times F \times \tilde{H} \times \tilde{W}$), which we refer to as an *error signal*. It computes the gradients for its weights, dL/dw, and an error signal for the next layer, dL/dx as

$$\frac{dL}{dw_{f,c,a,b}} = \sum_{k=0}^{N-1} \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} \frac{dL}{dy_{k,f,i,j}} x_{k,c,i+a-O,j+b-O} \quad (2)$$

$$\frac{dL}{dx_{k,c,i,j}} = \sum_{f=0}^{F-1} \sum_{a=-O}^{O} \sum_{b=-O}^{O} \frac{dL}{dy_{k,f,i-a,j-b}} w_{f,c,a+O,b+O}. \quad (3)$$

Note that the common data-parallel (sample-parallel) formulation for convolutional layers can be immediately seen from these formulas: the N dimension is partitioned, resulting in local computations for y and dL/dx, and the summation in dL/dw is aggregated with an allreduce.

We do not focus on the implementation of convolution itself and instead rely on cuDNN [19] for an optimized implementation.

Other layers, such as batch normalization [20], ReLUs, pooling, and fully-connected (FC) layers are also present in CNNs. We do not focus on their details, as modifications are either simple or similar to those for convolutional layers. For FC layers, we use a model-parallel formulation based on distributed matrix products [17].

B. Performance modeling

We make use of analytic models for some of our performance estimates, particularly communication. We use a linear model [21] for communication, where α is the latency and β is the inverse bandwidth. Then the cost to send a message between two nodes is $\alpha + \beta n$. We additionally assume that the network is full-duplex and that there is no interference.

Collective communication operations such as allreduce will be important for some operations; for these, we use the performance models of Thakur et al [22]. For distributed matrix multiplication, we use the performance models developed for the Elemental library [23].

C. Notation

We now define some notation for distributed tensors that will be used throughout this paper. Our notation is heavily based on the tensor notation developed for the FLAME project [23]–[25].

A tensor is an M-dimensional array, where the size of dimension m is I_m, and we write $I = (I_0, \ldots, I_{M-1})$ to refer to the shape of an entire tensor.

1) Distributions: Let P be the set of processors over which we distribute data. In general, we assume that every processor has the same amount of data (excepting minor imbalances due to divisibility) for load-balancing reasons, but this is not required.

We refer to the distribution of a tensor as $\mathcal{D}$, and of a particular dimension m as $\mathcal{D}^{(m)}$. A distribution $\mathcal{D}^{(m)}$ can be thought of as an assignment of indices in that dimension to processors. Then a distribution for a tensor is $\mathcal{D} = \left(\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}\right)$. We require that every element of a tensor be assigned to at least one processor.

2) Processor and index sets: It is helpful to be able to refer to different sets of processors and indices for a distribution. We use $\mathcal{I}^{(p)}\left(\mathcal{D}\right)$ to refer to the set of elements on processor p under distribution $\mathcal{D}$, and $\mathcal{I}_m^{(p)}\left(\mathcal{D}^{(m)}\right)$ to the indices in dimension m on p. The number of indices in m on p is $I_m^{(p)}\left(\mathcal{D}^{(m)}\right)$. We omit the distribution when it is clear from context. We write $\mathcal{P}^{(p)}\left(\mathcal{D}^{(m_0)}, \ldots\right)$ for the set of processors that have the same indices as p under the distributions.

3) CNNs: We think of a CNN as a directed acyclic graph, where a layer may have multiple parents or children (e.g. residual connections). Each layer ℓ_i has six tensors associated with it: input x_i, output y_i, weights w_i, input error signal dL/dy_i, weight gradients dL/dw_i, and output error signal dL/dx_i. (If a layer has no parameters, w_i and dL/dw_i are empty.) For a "line" network architecture, the output from the previous layer is $x_i = y_{i-1}$ and the error signal from the subsequent layer is $dL/dy_i = dL/dx_{i+1}$.

Many of the dimensions for a layer's tensors are the "same"; for example, the sample dimension N of a convolutional layer's inputs and outputs. To avoid communication, it makes sense for dimensions to be distributed the same way in both tensors. Thus we will overload our notation to allow $\mathcal{D}$ to refer to the distribution of a layer, with the distributions for each dimension applied to the appropriate tensors.

III. Distributed memory convolution algorithms

We now present our algorithms for distributed-memory convolution. We refer to exploiting parallelism along a particular dimension as "parallelizing" that dimension. For convolutional layers, this results in five approaches: sample, height and width (together, spatial), channel, and filter parallelism. Sample parallelism is simply the familiar data-parallel approach. Note that these approaches are not mutually exclusive.

We begin by describing spatial parallelism, together with sample parallelism, and then discuss our implementation. Our algorithms exactly replicate convolution as if it were performed on a single GPU (up to floating point accumulation issues). We also briefly discuss channel and filter parallelism, but we defer implementation to future work.

One assumption we require is that spatial dimensions are distributed in a blocked manner. This is because applying convolution at a point requires spatially adjacent data, and a non-blocked distribution would require extensive communication.

A. Sample and spatial parallelism

At a high level, sample parallelism partitions x, y, dL/dy, and dL/dx along the N dimension, assigning complete samples to processors. The weights w are stored redundantly on every processor. Computation of y (forward propagation), dL/dx, and the local contributions of dL/dw (backpropagation) can then be computed independently. An allreduce is required to complete the sum to form the final dL/dw (the sum on N in Eq 2), after which SGD can proceed independently on each processor.

Spatial parallelism is more complicated. The spatial dimensions H and W of x, y, dL/dy, and dL/dx are split among processors. Most of forward propagation can be performed locally, but when a filter of size greater than 1×1 is placed near the border of a partition, remote data will be needed to compute the convolution. Thus a small number of rows and/or columns will need to be transferred from the remote processors in a halo exchange (as in a stencil computation). Backpropagation is similar, requiring a halo exchange on dL/dy to compute dL/dx, and using the data from forward propagation to compute the local contributions of dL/dw. Finally, an allreduce completes the sum in dL/dw, like in sample parallelism. It should be observed that sample and spatial parallelism are orthogonal and can be used simultaneously. We refer to this as hybrid sample/spatial parallelism.

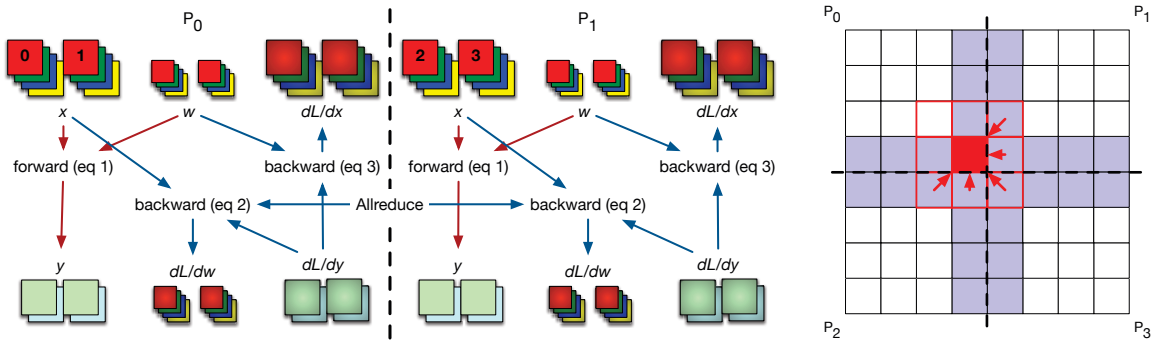
Figure 1 graphically illustrates sample parallelism and the halo exchange in spatial parallelism.

We now present our algorithm more formally, combining both sample and spatial parallelism. We keep the same simplifying assumptions as in Section II-A. Let $\mathcal{D}^{(C)}$ and $\mathcal{D}^{(F)}$ assign all indices to all processors (i.e. w and dL/dw are replicated on every processor). Let $\mathcal{D}^{(N)}$, $\mathcal{D}^{(H)}$, and $\mathcal{D}^{(W)}$ be distributions of the N, H, and W dimensions. The size of the halo region will be O rows or columns of length $I_W^{(p)}$ or $I_H^{(p)}$, respectively (we ignore padding issues here, but they are easy to handle). Let $q_H^{(p)} = \min \mathcal{I}_H^{(p)}$ be the lowest index in dimension H assigned to processor p and $r_H^{(p)} = \max \mathcal{I}_H^{(p)}$ the highest; define q and r similarly for W.

The non-local data in x that processor p requires (its halo) consists of the indices $(\mathcal{I}_N^{(p)}, \mathcal{I}_C^{(p)}, \{q_H^{(p)} - O, \ldots, q_H^{(p)} - 1, r_H^{(p)} + 1, \ldots, r_H^{(p)} + O\} \cup \mathcal{I}_H^{(p)}, \{q_W^{(p)} - O, \ldots, q_W^{(p)} - 1, r_W^{(p)} + 1, \ldots, r_W^{(p)} + O\} \cup \mathcal{I}_W^{(p)}) \setminus \mathcal{I}^{(p)}$. Note that when $K = 1$, $O = 0$ and no halo is needed. Forward propagation on p computes y similarly to Eq 1, except that the bounds on the indices for y change: $k \in \mathcal{I}_N^{(p)}$, f runs over all filters, $i \in \mathcal{I}_H^{(p)}$, and $j \in \mathcal{I}_W^{(p)}$. Notice that values of x in the halo region will be required.

Backpropagation is adapted similarly. The halo region for dL/dy is defined as for x, and the error signal dL/dx is computed as in Eq 3, where $k \in \mathcal{I}_N^{(p)}$, c runs over all channels, $i \in \mathcal{I}_H^{(p)}$, and $j \in \mathcal{I}_W^{(p)}$. Computing the local gradients dL/dw requires the halo region for x and follows Eq 2:

$$\frac{dL}{dw_{f,c,a,b}} = \sum_{k \in \mathcal{I}_N^{(p)}} \sum_{i \in \mathcal{I}_H^{(p)}} \sum_{j \in \mathcal{I}_W^{(p)}} \frac{dL}{dy_{k,f,i,j}} x_{k,c,i+a-O,j+b-O}$$



(a) Sample parallelism

(b) Spatial parallelism halo exchange

Fig. 1. (a) Forward and backpropagation phases (red and blue arrows) for sample parallelism with two processors and a global mini-batch of size 4. (b) Example halo exchange for spatial parallelism on four processors. The solid red box is where the 3×3 convolution filter is centered and red arrows indicate data movement. The shaded region is the entire halo region.

where f and c run over all filters and channels and a and b over $0, \ldots, K-1$. To compute the final value of dL/dw, an allreduce over all processors is performed. Note that dL/dw does not require the halo region for dL/dy, and therefore the entire computation for dL/dx, including the halo exchange, can be performed concurrently with dL/dw.

In the case that the stride is greater than 1, the process is similar, requiring that the halo region be adjusted to account for it. This may result in some processors not requiring a halo region. One important edge case to note is that spatial partitioning is complicated when a spatial dimension is the same size as the filter kernel. However, as spatial partitioning is most efficient when the kernel size (hence, halo area) is small compared to the partition size, as is typical, alternative parallelization approaches are preferred in this case.

B. Entire CNNs

The extension to an entire CNN is relatively straightforward. Input should be provided in the appropriate distribution for the first layer. Each convolutional layer can be parallelized as above. Pooling layers are parallelized similarly. Element-wise operations such as ReLUs parallelize trivially regardless of distribution. Batch normalization is typically computed locally on each processor; however, to our knowledge, performing batch normalization on subsets of the spatial dimensions has not been explored. Both purely local batch normalization and a variant that aggregates over the spatial distribution of a sample are easy to implement. FC layers use the model-parallel implementation of LBANN [17], although this can require a data redistribution, which we discuss next.

C. Data redistribution

It may occur that two layers ℓ_i and ℓ_j are adjacent, but use different data distributions $\mathcal{D}_i$ and $\mathcal{D}_j$ ($\mathcal{I}^{(p)}(\mathcal{D}_i) \neq \mathcal{I}^{(p)}(\mathcal{D}_j)$). This may be because it is more profitable to parallelize the layers using different approaches, or because one is a FC layer, which in our implementation uses an elemental distribution [24]. When this occurs, data must be shuffled between the two distributions on both forward and back-propagation. This shuffle can be implemented via an all-to-all collective, where a processor sends indices it no longer owns ($\mathcal{I}^{(p)}(\mathcal{D}_i) \setminus \mathcal{I}^{(p)}(\mathcal{D}_j)$), and receives its new indices ($\mathcal{I}^{(p)}(\mathcal{D}_j) \setminus \mathcal{I}^{(p)}(\mathcal{D}_i)$).

D. Channel and filter parallelism

We now briefly outline some approaches to channel and filter parallelism, although we leave implementation to future work. Channel and filter parallelism are closely related and data distributions must be chosen carefully; note, for example, if the input x to a layer is partitioned on its C dimension, the output y is partitioned on its F dimension. The choice of distributions significantly influences the communication involved, both within and between layers.

The computation of y in forward propagation is relatively simple, except that the summation over channels ($c \in \mathcal{I}_C^{(p)}$) may involve a global reduce-scatter. Computation of dL/dy is similar, with the summation over filters ($f \in \mathcal{I}_F^{(p)}$) possibly involving a global reduce-scatter. The gradients dL/dw can be complicated to compute, as the update for a filter depends on both x and dL/dy, which may require data to be gathered. A variety of data distributions are possible depending on the communication overheads and the amount of data replication that can be supported; we plan to explore these in future work.

IV. Implementation

In order to evaluate the proposed distributed convolution algorithms, we implement them by extending the LBANN toolkit for deep neural networks [17], which provides an underlying substrate for MPI-based parallel training with GPU acceleration. While LBANN can efficiently parallelize training using highly-optimized libraries such as cuDNN, NCCL, and Aluminum [26], it is limited to sample parallelism for convolution.

To extend LBANN for fine-grained distributed convolutions, we first develop a small C++ library for distributed tensor data structures that provides high-level abstractions for common

tensor primitives used in CNN training. The design of the tensor library is strongly influenced by our prior experience in designing a high-level framework for stencil computations [27]. It presents a partitioned global view of multidimensional tensors decomposed over distributed CPU and GPU memories. For supporting convolutions, a halo exchange among adjacent distributed sub-tensors is implemented as part of the library API. It uses Aluminum for inter-node communication, and CUDA inter-process communication on-node. Most of these operations do not require the programmer to explicitly manage the data distribution. The library automates the underlying bookkeeping of distributed data structures as much as possible so long as this does not negatively impact performance. Fortunately, similar to regular stencil computations in scientific simulations, CNN computations tend to be rather regular, and thus realizing automation without performance penalties is possible (e.g. [27] for stencils).

We implement a basic set of layers used in typical CNNs, including convolutions, pooling, batch normalization, and ReLU, on top of the tensor library. In this work, we focus on cluster systems with NVIDIA GPUs as the main computing platform for training and use NVIDIA's cuDNN library for optimized compute kernels. However, as cuDNN is not aware of the distribution of tensors, the library performs halo exchanges before convolutions and pooling. We replace LBANN's tensor representation with ours with necessary data shuffling so that the overall training pipeline can be used as is.

A. Optimization for strong scaling

While fine-grained parallelization allows us to use a larger set of parallel resources, careful optimization of data movement becomes more important, especially for strong scaling. Unlike sample partitioning, halo exchanges are required in spatial partitioning and can be prohibitively expensive when the spatial domain is not large enough.

One of the well-known techniques for halo exchanges is overlapping with independent computations. Our implementation automatically decomposes an input tensor into its interior domain and boundary domains and calls cuDNN convolution kernels for each region separately so that halo exchanges can be run concurrently with the convolution of the interior domain. For backward convolutions, we exploit the task-level parallelism of backward data and filter convolutions to hide the halo exchange for the data convolution within the filter convolution. Note that the filter convolution does not require halo exchanges. We also minimize the latencies of halo exchanges by using asynchronous low-latency communication mechanisms such as GPUDirect RDMA if available.

V. Performance model

We now provide a simple performance model for sample and spatial parallelism. Then we describe our approach for estimating good parallel execution strategies. All of this can be easily extended to handle channel and filter parallelism.

A. Convolutional layer runtime

We use empirical estimates for convolution, as cuDNN may select among many algorithms, and it is difficult to model GPU performance. These come from a simple benchmark that times the appropriate cuDNN function; we perform several warmup runs, then take the average of ten runs. This is combined with an analytic communication model (see Section II-B). Using this approach requires access to the target system in order to gather data, but does not require large-scale runs. As communication is often a chief bottleneck for training CNNs, an analytic model additionally allows flexibility to consider hypothetical communication optimizations.

We let $C(n, c, h, w, f)$ be the runtime of forward propagation (Eq. 1) for f filters on n $h \times w$ samples with c channels, with kernel size, stride, and padding omitted for convenience. This is used to estimate the *local* runtime of convolution. Similarly, let $Cw(n, c, h, w, f)$ and $Cx(n, c, h, w, f)$ be the local runtimes for backpropagation (Eqs. 2 and 3 respectively). For communication, we will use $AR(p, n)$ to denote the time for an allreduce of n words over p processors, and $SR(n)$ for the time to send and receive n words between two processors. We separate these because allreduces use different algorithms (e.g., ring or butterfly) for different n and p, so its performance cannot be directly deduced from point-to-point performance.

With this, the time for forward convolution for a layer ℓ is

$$\begin{aligned} FP_\ell &= C(I_N^{(p)}, I_C^{(p)}, I_H^{(p)}, I_W^{(p)}, I_F^{(p)}) \\ &+ 2SR(OI_N^{(p)} I_C^{(p)} I_H^{(p)}) + 2SR(OI_N^{(p)} I_C^{(p)} I_W^{(p)}) \\ &+ 4SR(O^2 I_N^{(p)} I_C^{(p)}) \end{aligned}$$

where the send/recvs are for the east/west, north/south, and corner halo regions, respectively. If $I_W^{(p)} = W$, then the east/west and corner halo exchanges can be omitted; if $I_H^{(p)} = H$, then the north/south and corner halo exchanges can be omitted. Additionally, if the implementation supports it, the halo exchanges can be overlapped with interior computation.

For backpropagation, the time to compute dL/dx is

$$\begin{aligned} BPx_\ell &= Cx(I_N^{(p)}, I_C^{(p)}, I_H^{(p)}, I_W^{(p)}, I_F^{(p)}) \\ &+ 2SR(OI_N^{(p)} I_C^{(p)} I_H^{(p)}) + 2SR(OI_N^{(p)} I_C^{(p)} I_W^{(p)}) \\ &+ 4SR(O^2 I_N^{(p)} I_C^{(p)}). \end{aligned}$$

Note the similarity to forward propagation; the same discussion on halo exchanges and overlapping applies here. We will simply write $BPw_\ell = Cw(I_N^{(p)}, I_C^{(p)}, I_H^{(p)}, I_W^{(p)}, I_F^{(p)})$ to be the time to compute the *local* portion of dL/dw. Finally, the allreduce time to complete dL/dw is $BPa_\ell = AR(|\mathcal{P}^{(p)}\left(\mathcal{D}^{(C)}, \mathcal{D}^{(F)}\right)|, I_F^{(p)} I_C^{(p)} K^2)$. These are written separately because there are several additional, commonly used opportunities for overlap. The halo exchanges for computing dL/dx can additionally be overlapped with computing dL/dw; both local computations could be done concurrently; and the allreduce can be overlapped with computing dL/dx, as well as computations in other layers.

We will write $\text{Cost}_\mathcal{D}(\ell_i) = FP_{\ell_i} + BPx_{\ell_i} + BPw_{\ell_i} + BPa_\ell$ to be the cost of a layer under distribution $\mathcal{D}$, adjusting for

overlap if necessary.

From these performance models, we can see that in terms of communication overheads, sample parallelism is the "cheapest" approach: it requires only the allreduce time in BPa_ℓ. Spatial parallelism adds additional overhead from the halo exchanges (which can be overlapped).

B. Runtime for a CNN

Extending this to an entire CNN is relatively straightforward. We need to address three things: layers besides convolution, overlapping between layers, and data redistributions. As most layers other than convolution and FC layers are computationally cheap, we treat them as free, for simplicity. FC layers use the standard performance models for distributed matrix multiplication [23]. If a layer has learnable parameters (e.g. batch normalization), an allreduce is required, and modeled similarly to above (note that model-parallel FC layers do not need such an allreduce).

We estimate allreduce overlap between layers (e.g. BPa) by greedily overlapping as much computation as possible with an allreduce. Only one allreduce at a time is considered to run, which neglects the possibility of running multiple concurrently, but simplifies the model.

Data redistribtion may be required in forward and backpropagation when the distributions of adjacent layers differ (Section III-C). This uses an all-to-all to shuffle data; we use $\text{Shuffle}(\mathcal{D}_i, \mathcal{D}_j)$ for the cost of moving data from $\mathcal{D}_i$ to $\mathcal{D}_j$.

C. Parallel execution strategies

It may be advantageous to use different distributions of data for different layers in a CNN, in order to parallelize them differently. For example, spatial parallelism is unlikely to provide a significant benefit to a layer with a small spatial domain. This introduces a number of knobs that can be tuned to parallelize a network, and selecting the appropriate parallelization scheme for each layer is difficult. Further, the performance of convolution implementations can be hard to predict in advance. A *parallel execution strategy* for a network is an assignment of distributions to each layer. We now sketch a simple optimization approach for selecting good parallel execution strategies using our performance model.

First we generate candidate distributions for each layer. For convolutional layers, we heuristically select distributions that are load balanced and prefer cheaper partitioning methods (i.e. sample over spatial parallelism) when possible. We assume FC layers are either entirely sample- or model-parallel and that other layers simply inherit the distribution of their parent.

Given the set of candidate distributions for each layer, we find the best parallel execution strategy for the CNN by reducing to the single-source shortest path problem. For simplicity, we first consider the case where the CNN has a "line" architecture with no branches. We construct a graph with a vertex for every candidate distribution for every layer, plus source and sink vertices. There is an edge from each candidate distribution $\mathcal{D}_i$ for a layer ℓ_i to every candidate distribution $\mathcal{D}_j$ of its child layer ℓ_j, weighted with $\text{Cost}_{\mathcal{D}_i}(\ell_i) + \text{Shuffle}(\mathcal{D}_i, \mathcal{D}_j)$. Finally, there is an edge from the source vertex to every candidate distribution of the first layer with weight 0, and from every candidate distribution of the last layer ℓ_i to the sink vertex weighted by $\text{Cost}_{\mathcal{D}_i}(\ell_i)$.

A shortest path from the source to the sink of this graph gives a parallel execution strategy with the fastest end-to-end runtime for the CNN. Since this is a directed acyclic graph, this path can be found in linear time. While a large number of vertices and edges are generated, we have found this is not an issue in practice. If necessary, additional heuristics could be used to prune them.

For networks that have branches (e.g. ResNets), we cannot directly apply this approach, as some layers have multiple parents or children. Instead, we first find the longest path from the beginning to the end of the CNN, and apply the above to every layer in this path. The distributions for these layers are fixed, and we repeat with the next longest path that contains as few of the already-used layers as possible until every layer has a distribution. The idea behind this approach is that the longest path is the most computationally intensive, and so it should be optimized first, to guarantee maximum flexibility in distribution choice.

VI. Evaluation

We now evaluate our algorithms via microbenchmarks and end-to-end training. We use Lassen, a CORAL-class supercomputer [28], which consists of 650 nodes, each with two IBM POWER9 CPUs and four Nvidia V100 (Volta) GPUs with NVLINK2 and 16 GB of memory per GPU, interconnected via dual-rail InfiniBand EDR. Our implementation uses a recent development version of LBANN and Aluminum. We use GCC 7.3.1, Spectrum MPI 2019.01.30, CUDA 9.2.148, cuDNN 7.4.1, and NCCL 2.4.2.

We consider two networks: a fully-convolutional ResNet-50 [18], [29] for ImageNet-1K classification and a proof-of-concept model for a 2D mesh-tangling problem which we formulate as semantic segmentation. The data consists of images representing a hydrodynamics simulation state at a timestep, and the problem is to predict, for each pixel, whether the mesh cell at that location needs to be relaxed to prevent tangling. Mesh tangling occurs when cells overlap, which is non-physical and results in the simulation degenerating or failing entirely. It is hoped that CNNs, which can incorporate global information from the simulation state, will offer better heuristics for predicting tangling than existing methods. However, as interesting simulations are high-resolution, it has not been possible to train neural networks on the data due to memory constraints.

For these tests, the input data is either 1024×1024 ($1K$) or 2048×2048 ($2K$) pixel images, with 18 channels consisting of various state variables and mesh quality metrics from a hydrodynamics simulation. We use 10,000 samples of each size. Our CNN is a very simple fully-convolutional model adapted from VGGNet [30] for our input sizes and semantic segmentation. It consists of six blocks of either three ($1K$) or five ($2K$) convolution-batch normalization-ReLU operations,

using 3×3 convolutional filters, and a final convolutional layer for prediction. Downsampling is performed via stride-2 convolution at the first convolutional filter of each block. The model for the $2K$ mesh data is large enough, when including intermediate activations, to exceed GPU memory when training with even one sample. For performance benchmarks on this problem, we use synthetic data, as our goal is to focus on the performance of our algorithms and demonstrate that models of this scale can be quickly trained on HPC systems. We leave developing optimized models to future work, now that training is feasible.

A. Layer benchmarks

We first present microbenchmark results for selected layers from ResNet-50 and the $2K$ mesh model on up to four Lassen nodes. These enable the performance characteristics of spatial convolution to be seen at a fine-grained scale. We time forward and backpropagation of each layer, with halo exchanges being overlapped. We exclude the allreduce to accumulate gradients to focus on the performance of convolution. Also, it is typically overlapped by other computation. For each measurement, we first do warmup runs, then report the mean and standard deviation of ten runs.

Figure 2 shows results for ResNet-50 layers `conv1` and `res3b_branch2a` with $N = 1$, 4, and 32 samples. The first two can occur when strong scaling sample-parallelism to few samples per GPU; the last is a typical target for efficient use of a GPU. `conv1` is the first layer of ResNet, with a relatively large (224×224) input, but only three channels and 64 filters. The kernel is large, $K = 7$, requiring large halo exchanges for spatial parallelism. For $N = 1$, forward propagation does not scale well, due to limited computation to hide halo exchanges; backpropagation fares better, and results in net improvements to forward and backpropagation of $\sim$1.35x with 8 GPUs. Performance degrades somewhat with 16 GPUs, due to communication overheads. `res3b_branch2a` is a 1×1 convolution from the middle of ResNet, with a fairly small spatial domain. The filter size means that no halo exchange is needed, avoiding communication overheads. Forward propagation does not show significant performance improvements beyond two GPUs, due to fixed kernel overheads. Backpropagation shows improvements up to 16 GPUs except that the 2 GPUs/sample case is significantly slower than 4 GPUs/sample at 4 GPUs due to the performance of the underlying cuDNN kernels. With larger numbers of samples, spatial decomposition remains competitive with pure sample parallelism, indicating halo exchanges are hidden.

Figure 3 presents results for two layers of our 2048×2048 mesh model. Here spatial domains are much larger, and we expect spatial parallelism to perform better. Results are for $N = 1$, 2, and 4 samples, since due to the size of the data, only one or two samples can be trained per node. (Since we benchmark only individual layers here, memory pressure is not as significant as with an entire network.) `conv1_1` is the first layer, and has extremely large spatial input. The $N = 1$ case has very good scaling on both forward and backpropagation, achieving $\sim$14.8x speedup on 16 GPUs, indicating inter-node halo exchange overheads are well-hidden. The two sample case is similar. With four samples, the overhead of the halo exchange is very minor, enabling competitive scaling to sample parallelism. `conv6_1` is from much deeper within the network, and has a smaller input spatial domain. Nonetheless we see continued benefit in the $N = 1$ case ($\sim$1.4x).

These results illustrate several important things. First, that the empirical performance of convolution can be complicated to predict. Second, that the intuition from our performance model is broadly correct: other things equal, sample parallelism typically has the least overhead. Finally, the small N case, observed when strong-scaling sample-parallelism, can benefit significantly from spatial parallelism.

B. Training

We now present scaling results for end-to-end training of our models. We consider both strong and weak scaling, although our focus is strong scaling. We define strong scaling as parallelizing a CNN over more GPUs, while keeping the mini-batch size and all other aspects fixed. Weak scaling parallelizes a CNN by keeping the mini-batch size per GPU fixed, and growing the global mini-batch size as more GPUs are added, working on the same problem but with larger batch sizes. Strong scaling has the advantage that one need not address issues of learning and generalization with large mini-batches, as the learning process does not change: once a good mini-batch size is selected, it can be strong-scaled to make training faster. Our results are primarily hybrid sample-spatial parallelism, where samples are first partitioned onto groups of GPUs, and then spatially parallelized within that group. We use the same data decomposition for every layer in a given configuration, although this is not necessarily optimal; we leave exploring more varied decompositions to future work.

1) Mesh model: Figure 4 shows scaling results for the $1K$ and $2K$ mesh models, up to 2048 GPUs (512 nodes). For the former, we present five cases: sample parallelism, and 2-, 4-, 8-, and 16-way hybrid sample/spatial parallelism with mini-batch sizes $N = 4$ to 2048 (omitting cases when they require too many GPUs). The model can fit only one sample per GPU, so we do not explore additional sample parallelism. We run the same configurations for the $2K$ mesh model, except pure sample parallelism is not possible due to memory constraints and our maximum mini-batch size is 1024. We compare strong-scaling at a fixed mini-batch size across parallelism cases, and weak scaling as the mini-batch size grows. Note that when using 8- or 16-way spatial parallelism, a sample is being partitioned across two or four nodes, requiring both intra- and inter-node communication for halo exchanges.

For strong scaling, spatial parallelism allows us to use additional GPUs for the same mini-batch size. Table I gives mini-batch times and speedups for the $1K$ mesh model. We see near-linear speedup for 2 GPUs/sample over sample parallelism (2x is ideal), and significant further improvements with 4 GPUs/sample. Improvements continue with 8 and 16 GPUs/sample, although they are not as dramatic due to

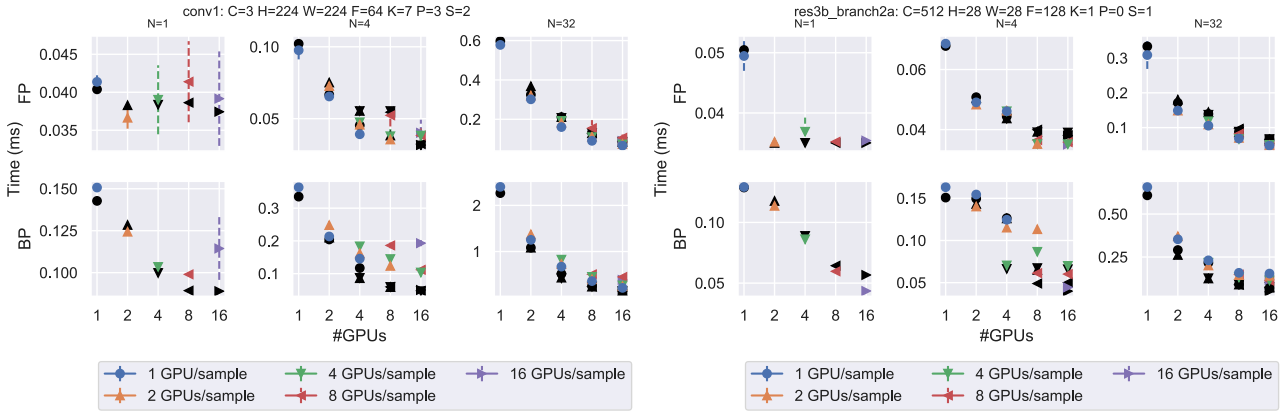


Fig. 2. Microbenchmark results for layers `conv1` and `res3b_branch2a` of ResNet-50 comparing parallelization schemes in forward (FP) and backpropagation (BP). Error bars are ± one standard deviation. Black shapes are performance model predictions. Specifications for each layer are above each figure.

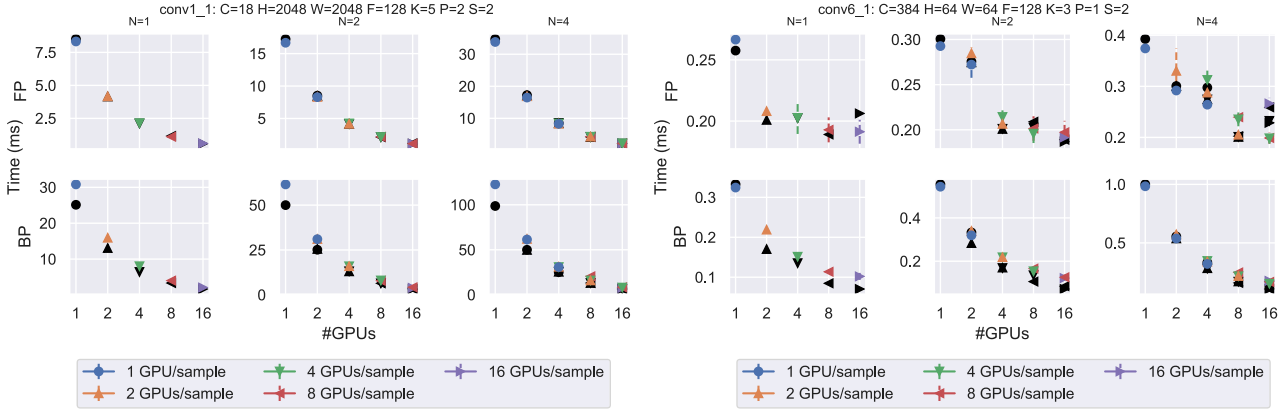


Fig. 3. Microbenchmark results for layers `conv1_1` and `conv6_1` from the $2K$ mesh model, comparing parallelization schemes in forward (FP) and backpropagation (BP). Error bars are ± one standard deviation. Black shapes are performance model predictions. Specifications for each layer are above each figure.

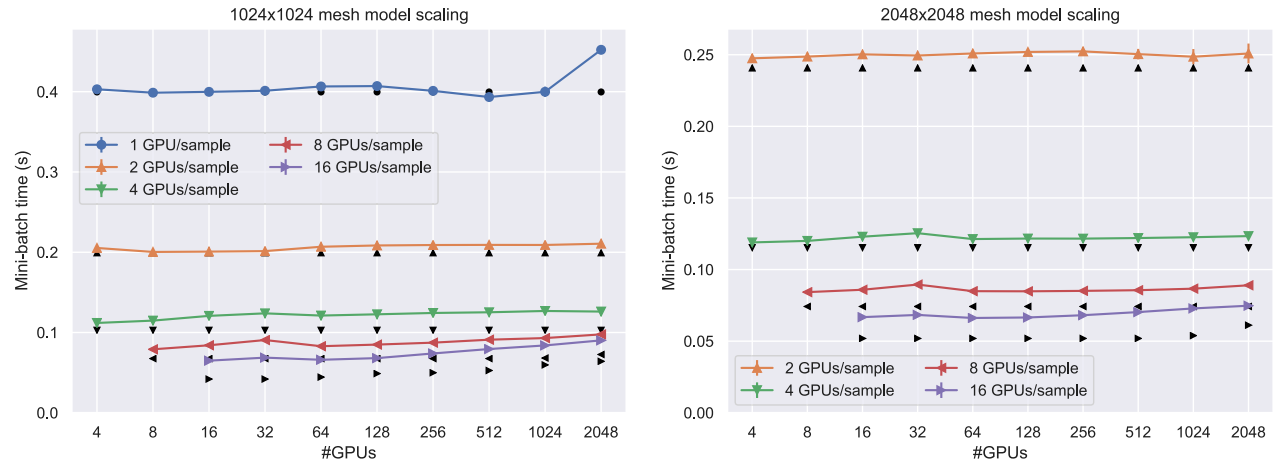


Fig. 4. Weak scaling for the $1K$ and $2K$ mesh models. Error bars are ± one standard deviation. Black markers are estimates from our performance model. 1 sample/GPU is is pure sample parallelism; other cases are hybrid sample/spatial parallelism. Note that there are 4 GPUs/node. Spatial parallelism is required for the $2K$ model due to memory requirements. Strong scaling can be seen between parallelization schemes (see Tables I and II).

TABLE I
1K MESH STRONG SCALING AT MINI-BATCH SIZE N, MINI-BATCH TIME AND SPEEDUP OVER 1 GPU/SAMPLE (SAMPLE PARALLELISM).

N	1 GPU/sample	2 GPUs/sample	4 GPUs/sample	8 GPUs/sample	16 GPUs/sample
4	0.403s	0.2s (2.0x)	0.121s (3.3x)	0.0906s (4.4x)	0.066s (6.1x)
8	0.399s	0.201s (2.0x)	0.124s (3.2x)	0.0829s (4.8x)	0.0681s (5.9x)
16	0.4s	0.201s (2.0x)	0.121s (3.3x)	0.085s (4.7x)	0.0739s (5.4x)
32	0.401s	0.207s (1.9x)	0.123s (3.3x)	0.0874s (4.6x)	0.0794s (5.1x)
64	0.407s	0.208s (2.0x)	0.124s (3.3x)	0.0911s (4.5x)	0.0839s (4.8x)
128	0.407s	0.209s (1.9x)	0.125s (3.3x)	0.0931s (4.4x)	0.0902s (4.5x)
256	0.401s	0.209s (1.9x)	0.127s (3.2x)	0.0977s (4.1x)	n/a
512	0.393s	0.209s (1.9x)	0.126s (3.1x)	n/a	n/a
1024	0.4s	0.211s (1.9x)	n/a	n/a	n/a

TABLE II
2K MESH STRONG SCALING AT MINI-BATCH SIZE N, MINI-BATCH TIME AND SPEEDUP OVER 2 GPUS/SAMPLE.

N	2 GPUs/sample	4 GPUs/sample	8 GPUs/sample	16 GPUs/sample
2	0.247s	0.12s (2.1x)	0.0859s (2.9x)	0.0683s (3.6x)
4	0.249s	0.123s (2.0x)	0.0895s (2.8x)	0.0662s (3.8x)
8	0.25s	0.125s (2.0x)	0.0849s (2.9x)	0.0665s (3.8x)
16	0.249s	0.121s (2.1x)	0.0848s (2.9x)	0.0681s (3.7x)
32	0.251s	0.122s (2.1x)	0.0851s (2.9x)	0.0703s (3.6x)
64	0.252s	0.122s (2.1x)	0.0856s (2.9x)	0.0729s (3.5x)
128	0.252s	0.122s (2.1x)	0.0867s (2.9x)	0.0748s (3.4x)
256	0.25s	0.123s (2.0x)	0.089s (2.8x)	n/a
512	0.249s	0.123s (2.0x)	n/a	n/a

the increased overheads of halo communication and local convolution kernels not scaling linearly. Weak scaling results can be seen in Figure 4 (left), where the flat mini-batch time for increasing numbers of GPUs (hence, increasing mini-batch size) shows near-perfect weak scaling. This implies that our spatial partitioning is not impacting the typical sample-parallel weak scaling trends. Weak scaling for 8 and 16 GPUs/sample does show a slight trend of increasing mini-batch time at large scale; due to the extensive data decomposition, each GPU has significantly less work, and our implementation cannot fully overlap global allreduces with backpropagation computation in these cases.

The performance degradation for sample parallelism at 2048 GPUs is due to memory pressure requiring a smaller workspace for cuDNN, impacting local convolution algorithm selection. The increased memory pressure is due to communication-related data structures taking increased GPU memory, and could be mitigated with future optimizations.

Results for the *2K* mesh model are shown in Table II (strong scaling) and Figure 4 (right, weak scaling). The strong scaling trend is similar to that for the *1K* model, with comparable speedups from each algorithm. Indeed, when increasing from 4 GPUs/sample to 8 GPUs/sample, the models observe roughly a $1.3x$ and $1.4x$ improvement in performance (respectively). Neither observe linear increases due to the high overhead of fine-grained inter-node halo communication, but our overlapping enables further improvement nonetheless. The *2K* model achieves slightly more speedup in this case, as there is more work to overlap communication with. The weak scaling trend is also comparable to that for the *1K* model, although we only observe weak scaling performance degrading with 16 GPUs/sample for the *2K* model.

That we can achieve both good strong and weak scaling for the mesh-tangling problem is important for being able to rapidly train and explore new models: good strong scaling means we can accelerate training of models without changing learning dynamics, and good weak scaling additionally helps with tuning mini-batch sizes.

2) ResNet-50: We present strong scaling results for ResNet-50, comparing pure sample parallelism to hybrid sample 2-way and 4-way parallelism, in Table III. We use 32 samples per GPU as our baseline, as this is a typical choice to saturate GPUs. Using spatial parallelism we achieve 1.4x speedups with 2x as many GPUs, and up to 1.8x with 4x as many GPUs. Prior work has shown that strong scaling this problem size via sample-parallelism past $\sim$8-16 nodes rapidly results in communication overhead making it unprofitable [26].

Thus, weak scaling is typically preferred, ensuring sufficient local work to hide communication costs. However, one cannot weak scale indefinitely, even with large mini-batch techniques, due to generalization issues. Goyal et al. [15] report ImageNet validation error begins to increase with mini-batch sizes above 8k when using ResNet-50. For different problems, datasets, or network architectures, the max mini-batch size will be different [14]. To continue to accelerate training beyond this, strong scaling must still be employed. Table III shows we get

TABLE III
STRONG SCALING RESNET-50, MINI-BATCH TIME AND SPEEDUP OVER SAMPLE PARALLELISM.

N	Sample (32 samples/GPU)	Hybrid (32 samples/2 GPUs)	Hybrid (32 samples/4 GPUs)
128	0.106s	0.0734s (1.4x)	0.0593s (1.8x)
256	0.106s	0.0732s (1.4x)	0.0671s (1.6x)
512	0.105s	0.0776s (1.4x)	0.0617s (1.7x)
1024	0.105s	0.0747s (1.4x)	0.0672s (1.6x)
2048	0.108s	0.0733s (1.5x)	0.0651s (1.7x)
4096	0.0984s	0.078s (1.3x)	0.066s (1.5x)
8192	0.109s	0.0785s (1.4x)	0.0725s (1.5x)
16384	0.108s	0.0844s (1.3x)	0.0792s (1.4x)
32768	0.109s	0.0869s (1.3x)	n/a

continued improvement from spatial parallelism with larger mini-batch sizes. Speedups decrease slightly at larger scale for more extensive decomposition, due to the implementation being unable to fully overlap the cost of allreduces.

Achieving near-linear speedup for ResNet is unlikely, as most layers have small spatial domains. This agrees with our microbenchmarks. (Channel/filter parallelism may be more promising, as many layers have many filters.) Despite this, we are still able to accelerate many problem sizes of interest.

3) Performance model: Our figures in this section have also included the corresponding predictions from our performance model. We can see that its predictions are quite accurate, and even when there are deviations, it still has the correct trend and ranking of algorithms. Much of the inaccuracy is due to lower-order computations that are not accounted for but matter with more extensive decompositions (e.g. 16 GPUs/sample). Network noise and other similar factors are also not accounted for. Nevertheless, this validates the accuracy of the performance model, and we can be confident that generating parallel execution strategies with it will be effective.

VII. RELATED WORK

There are many techniques for parallelizing CNNs at different scales. Ben-Nun and Hoefler [31] provide a comprehensive overview of approaches. Recent work has leveraged HPC resources for training CNNs on large simulation data (e.g. [32]–[34]). These works have primarily focused on scaling sample-parallel training via optimizing communication and I/O, large mini-batches, etc. They do not address the issues of very large samples. These optimizations are orthogonal to our algorithms, and they can be used together.

Scaling convolution. AlexNet [3] used an early form of model parallelism, where convolutional filters and fully-connected layers were partitioned between two GPUs to avoid memory limits. In this implementation, grouped convolutions were used for certain layers to reduce inter-GPU communication costs, instead of directly replicating the result of regular convolution. Coates et al [35] used a distributed tensor representation to spatially partition locally-connected layers; we use similar techniques for convolutional layers. Gholami et al. [36] also propose general decompositions of convolutional layers, but only provide simulated results. Some deep learning frameworks, such as DistBelief [37] and Project Adam [38] support partitioning the filters of convolutional layers onto workers and aggregating their results via parameter server. Oyama et al. [39] provide a framework to split mini-batches into smaller batches to use faster convolution algorithms.

Demmel and Dinh [40] give lower bounds on communication for convolutional layers and sequential algorithms that achieve them. However, their analysis is limited to forward propagation of a single layer, and does not consider optimizing training time for an entire CNN.

Large mini-batch training has been developed via linear scaling of learning rates [15] or layer-wise adaptive learning rates [16]. These approaches have been applied primarily to image classification, and can require extensive hyperparameter tuning. These are complementary to our work: when viable, large mini-batches can enable large-scale sample-parallel training, but we provide additional options for parallelization.

Parallel execution strategies. Yan et al. [41] develop a performance model for training DNNs with Adam [38] and an optimizer for finding good configurations. Their system does not include spatial partitioning, and their performance was evaluated on a CPU system, limiting its applicability to modern training regimes. Mirhoseini et al. [42] use reinforcement learning for TensorFlow graph placement on a small heterogeneous system, but do not consider partitioning convolutions beyond the sample dimension.

Memory pressure. Several approaches to alleviating memory pressure on GPUs have been used. If at least one sample can fit in GPU memory, an out-of-core "micro-batching" approach, where mini-batches are split into micro-batches and updates accumulated, can be used, but this can increase training time [43]. Other approaches utilize recomputation to avoid keeping intermediate values [44]. "Prefetching" and dynamic memory movement approaches, either through CUDA unified memory or in training frameworks [45]–[47], can be used, but this can require prefetch hints and result in additional CPU/GPU memory transfers, reducing the bandwidth available for communication.

VIII. CONCLUSIONS

We have demonstrated additional approaches for extracting parallelism from convolutional layers for training CNNs, en-

abling improved strong scaling. Exploiting parallelism within the spatial domain allows scaling to continue beyond the mini-batch size. This is necessary to train models on datasets with very large samples, which would otherwise be infeasible due to memory constraints, and enables further acceleration of training once the limits of sample-parallelism have been reached. As scientific domains and HPC simulations take greater advantage of deep learning, handling massive samples will only become more important. In particular, as 3D data becomes more widespread, spatial parallelism, which can be easily extended to 3D, becomes critical, and more advantageous, due to the more favorable surface-to-volume ratio.

We have also presented a performance model that reasonably approximates observed performance and provides intuition about different approaches to parallelizing convolution. Leveraging this to select good parallel execution strategies for CNNs will only become more important as they grow more complex and additional parallelism is exploited.

There remains significant additional sources of parallelism to explore. We sketched approaches to channel and filter parallelism, which need to be developed further. Reducing inter-node communication overheads for halo exchanges should enable greater spatial partitioning. There remain a host of classic communication optimization techniques, such as overdecomposition, that could be leveraged. Training deep nets on large data is an HPC problem, and tackling it requires exploiting as much parallelism as possible.

ACKNOWLEDGMENTS

Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-759919). Funding provided by LDRD #17-SI-003. Some testing/development support work funded by the LLNL Sierra Institutional Center of Excellence. Experiments were performed at the Livermore Computing facility. The authors thank the LBANN and Alkemi teams for their assistance.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, 2015.
[2] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *ICML*, 2009.
[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *NIPS*, 2012.
[4] C. Sun *et al.*, "Revisiting unreasonable effectiveness of data in deep learning era," in *ICCV*, 2017.
[5] D. Mahajan *et al.*, "Exploring the limits of weakly supervised pretraining," *arXiv preprint arXiv:1805.00932*, 2018.
[6] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.
[7] Top 500, "June 2018 TOP500," https://www.top500.org/lists/2018/06/, 2018.
[8] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, vol. 115, no. 3, 2015.
[9] G. Litjens *et al.*, "A survey on deep learning in medical image analysis," *Medical Image Analysis*, vol. 42, 2017.
[10] H. Chen *et al.*, "The rise of deep learning in drug discovery," *Drug Discovery Today*, 2018.
[11] S. Oh *et al.*, "A large-scale benchmark dataset for event recognition in surveillance video," in *CVPR*, 2011.
[12] T. N. Mundhenk *et al.*, "A large contextual dataset for classification, detection and counting of cars with deep learning," in *ECCV*, 2016.
[13] N. S. Keskar *et al.*, "On large-batch training for deep learning: Generalization gap and sharp minima," in *ICLR*, 2017.
[14] C. J. Shallue *et al.*, "Measuring the effects of data parallelism on neural network training," *arXiv preprint arXiv:1811.03600*, 2018.
[15] P. Goyal *et al.*, "Accurate, large minibatch SGD: training ImageNet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.
[16] Y. You *et al.*, "ImageNet training in minutes," in *ICPP*, 2018.
[17] B. Van Essen *et al.*, "LBANN: Livermore big artificial neural network HPC toolkit," in *MLHPC*, 2015.
[18] K. He *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016.
[19] S. Chetlur *et al.*, "cuDNN: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.
[20] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *ICML*, 2015.
[21] P. Fraigniaud and E. Lazard, "Methods and problems of communication in usual networks," *Discrete Applied Mathematics*, vol. 53, 1994.
[22] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, 2005.
[23] M. D. Schatz, R. A. Van de Geijn, and J. Poulson, "Parallel matrix multiplication: A systematic journey," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, 2016.
[24] J. Poulson *et al.*, "Elemental: A new framework for distributed memory dense matrix computations," *ACM TOMS*, vol. 39, no. 2, 2013.
[25] M. D. Schatz, "Distributed tensor computations: formalizing distributions, redistributions, and algorithm derivation," Ph.D. dissertation, University of Texas at Austin, 2015.
[26] N. Dryden *et al.*, "Aluminum: An asynchronous, GPU-aware communication library optimized for large-scale training of deep neural networks on HPC systems," in *MLHPC*, 2018.
[27] N. Maruyama *et al.*, "Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers," in *SC*, 2011.
[28] LLNL, "Lassen," https://hpc.llnl.gov/hardware/platforms/lassen, 2018.
[29] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *CVPR*, 2015.
[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015.
[31] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *arXiv preprint arXiv:1802.09941*, 2018.
[32] T. Kurth *et al.*, "Deep learning at 15PF: supervised and semi-supervised classification for scientific data," in *SC*, 2017.
[33] ——, "Exascale deep learning for climate analytics," in *SC*, 2018.
[34] A. Mathuriya *et al.*, "CosmoFlow: Using deep learning to learn the universe at scale," in *SC*, 2018.
[35] A. Coates *et al.*, "Deep learning with COTS HPC systems," in *ICML*, 2013.
[36] A. Gholami *et al.*, "Integrated model, batch and domain parallelism in training neural networks," in *SPAA*, 2018.
[37] J. Dean *et al.*, "Large scale distributed deep networks," in *NIPS*, 2012.
[38] T. M. Chilimbi *et al.*, "Project Adam: Building an efficient and scalable deep learning training system." in *OSDI*, vol. 14, 2014.
[39] Y. Oyama *et al.*, "Accelerating deep learning frameworks with micro-batches," in *CLUSTER*, 2018.
[40] J. Demmel and G. Dinh, "Communication-optimal convolutional neural nets," *arXiv preprint arXiv:1802.06905*, 2018.
[41] F. Yan *et al.*, "Performance modeling and scalability optimization of distributed deep learning systems," in *SIGKDD*, 2015.
[42] A. Mirhoseini *et al.*, "Device placement optimization with reinforcement learning," in *ICML*, 2017.
[43] Y. Ito, R. Matsumiya, and T. Endo, "ooc_cuDNN: Accommodating convolutional neural networks over GPU memory capacity," in *Big Data*, 2017.
[44] T. Chen *et al.*, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.
[45] M. Rhu *et al.*, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *MICRO*, 2016.
[46] C. Meng *et al.*, "Training deeper models by GPU memory optimization on TensorFlow," in *ML Systems Workshop @ NIPS*, 2017.
[47] L. Wang *et al.*, "Superneurons: dynamic GPU memory management for training deep neural networks," in *PPoPP*, 2018.