Channel and Filter Parallelism for Large-Scale CNN Training

Nikoli Dryden[*†]
dryden2@illinois.edu

Naoya Maruyama[†]
maruyama3@llnl.gov

Tim Moon[†]
moon13@llnl.gov

Tom Benson[†]
benson31@llnl.gov

Marc Snir[*]
snir@illinois.edu

Brian Van Essen[†]
vanessen1@llnl.gov

ABSTRACT

Accelerating large-scale CNN training is needed to keep training times reasonable as datasets grow larger and models become more complex. Existing frameworks primarily scale using data-parallelism, but this is limited by the mini-batch size, which cannot grow arbitrarily. We introduce three algorithms that partition channel or filter data to exploit parallelism beyond the sample dimension. Further, they partition the parameters of convolutional layers, replacing global allreduces with segmented allreduces—smaller, concurrent allreduces among disjoint processor sets. These algorithms enable strong scaling, reduced communication overhead, and reduced memory pressure, enabling training of very wide CNNs.

We demonstrate improved strong and weak scaling, including up to 4.1x reductions in training time for residual networks and 4x reductions in allreduce overhead. We also show that wider models provide improved accuracy on ImageNet. We study the current limitations of our algorithms and provide a direction for future optimizations of large-scale deep learning frameworks.

KEYWORDS

Deep learning, CNN, convolution, algorithms, scaling

ACM Reference Format:
Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Marc Snir, and Brian Van Essen. 2019. Channel and Filter Parallelism for Large-Scale CNN Training. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356207

1 INTRODUCTION

Deep learning has achieved tremendous results in fields ranging from computer vision to climate analytics and astrophysics [30, 31, 33, 40]. A major component of this success has been the availability of enough compute power to train sufficiently large models on large datasets. GPUs have been critical in enabling this growth [46], and it is now common to employ clusters of GPUs to train a single model. Nonetheless, training a state-of-the-art convolutional neural network (CNN) to convergence can take days or weeks. As ever more data is produced, and as researchers iterate through larger varieties of more complex models in search of better accuracies, increasingly more compute power needs to be leveraged. Frequent retraining is also typical, to explore or update models based upon newly ingested data or simulation outputs [21, 34]. It is thus necessary to accelerate training on large clusters of GPUs.

Current approaches to accelerating CNN training focus on distributed data-parallelism, which we also refer to as *sample* parallelism, following the convention of Dryden et al. [17]. In this approach, a mini-batch, typically consisting of no more than a few thousand samples from a dataset, is partitioned among processors, which each have complete replicas of the model's parameters. These processors can independently perform forward and back-propagation for their local data samples and compute parameter updates based upon them. They must then synchronize these updates, which takes the form of a global allreduce on the parameter buffers, after which a local optimization step, typically stochastic gradient descent (SGD), is performed. Even with modern CNNs consisting primarily of convolutional layers, which have relatively few parameters compared to large fully-connected layers, these allreduces are a major bottleneck to efficient scaling [18, 28, 30], and this problem will only worsen in the future as compute power continues to increase faster than network latency and bandwidth.

The data-parallel approach can be both strong and weak scaled based upon the mini-batch size [6]. In strong scaling, a fixed mini-batch size is partitioned over more processors; in weak scaling, the number of samples per processor is fixed and the global mini-batch size grows as additional processors are added. With sample parallelism, strong scaling is fundamentally limited by the mini-batch size. Yet one cannot arbitrarily increase the mini-batch size during weak scaling, as it can negatively impact the generalization performance of the learned model [20, 27, 38, 52]. Thus, it is important to enable strong scaling beyond the sample dimension to take advantage of additional compute resources. This also helps mitigate memory pressure due to the very large samples present in emerging scientific deep learning applications [17, 30, 40], which may have many channels. In some applications, the memory requirements may be significant enough that partitioning is necessary merely to run. For both strong and weak scaling, it is also vital to reduce global allreduce overheads, which limit large-scale training.

To address these issues, we propose to exploit parallelism within convolutional layers beyond the sample dimension by also partitioning the channel and filter dimensions of CNNs. This is in the same spirit as Dryden et al. [17], who partitioned the sample and spatial

[*]Department of Computer Science, University of Illinois at Urbana-Champaign
[†]Lawrence Livermore National Laboratory

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356207

dimensions of CNNs to improve strong scaling and reduce memory pressure. To do this, we introduce a family of three algorithms for performing convolution with distributed input/output channel/filter data, resulting in the same output as non-distributed convolution. Each algorithm is differentiated by the data movement and computation patterns it performs. These algorithms can additionally be combined with sample parallelism or the spatial parallelism of [17] to form *hybrid* algorithms where multiple types of partitioning are combined hierarchically. The *stationary-x* algorithm avoids communication of input data during forward propagation and the *stationary-y* algorithm avoids communication of input data during backpropagation, along with the corresponding gradients. These are both special cases of the *stationary-w* algorithm, a more complicated algorithm that can control the amount of communication in both forward and backpropagation. These names are by analogy to stationary matrix product algorithms [51, 57]. Each algorithm requires some communication during forward or backpropagation, which adds unavoidable overhead and can prevent perfect strong scaling, but they do not require global communication. Indeed, it can typically be performed on-node, where high-bandwidth links (e.g. NVLink2) are available, depending on data partitioning. We also consider the impacts of using inter-node communication, which enables additional strong scaling and can reduce memory pressure.

These algorithms additionally partition the parameters of a convolutional layer instead of fully replicating them on every processor. Hence, they are both model- and data-parallel (parameters and activations are partitioned). This changes the communication pattern when synchronizing gradients from a global allreduce to a *segmented* allreduce: disjoint subsets of processors run concurrent allreduces on different portions of the parameters. These segmented allreduces therefore perform communication on smaller data buffers and among fewer processors, reducing communication overhead.

While other frameworks, such as DistBelief [15] and Project Adam [12], have exploited model parallelism in convolutions for distributed training, they have done so by partitioning the filters to workers and using parameter servers. In contrast, our work jointly partitions channels and filters, and does so in a completely distributed manner. This necessitates much more complex communication patterns during forward and backpropagation, which must be mapped efficiently to collective communication primitives.

We provide some additional background and define our notation in Section 2, describe our algorithms in Section 3, and develop a performance model in Section 4. Our implementation is described in Section 5 and we comprehensively evaluate performance of ResNet-50 [22] for ImageNet-1k [49] in Section 6. We also examine training the wider Wide ResNet-50-2 [62] and -50-4 networks. The latter could not be trained by the original authors due to memory and training time requirements; our work makes this training feasible. We summarize our contributions as follows:

- We describe a family of algorithms for parallelizing convolutional layers with channel and filter decompositions.
- We provide performant implementations of these algorithms in an open-source framework (LBANN [58]).
- We provide a performance model to help understand algorithm performance and scaling.
- We comprehensively evaluate our implementations with micro-benchmarks and end-to-end training.
- We provide accuracy results showing the benefit of wider models.

2 BACKGROUND AND NOTATION

We begin by providing a brief overview on training CNNs and related algorithms, and then describe our notation. We assume the reader has a basic familiarity with training deep learning models.

2.1 Convolutional neural networks

For simplicity, we will assume convolution is performed on 2D data with stride 1 and "same" padding (these assumptions are not necessary). We will say that a convolutional layer has F filters of size $K \times K$ that are applied to an input consisting of N samples, each with C channels, height H, and width W. The layer then has six 4D tensors associated with it: a $N \times C \times H \times W$ input x, a $F \times C \times K \times K$ weights w, and a $N \times F \times H \times W$ output y, plus the associated gradients $\frac{dL}{dx}$, $\frac{dL}{dw}$, and $\frac{dL}{dy}$ (where L is the loss function).

With this, convolution is defined as follows. To further simplify, we will assume that K is odd and write $O = \lfloor K/2 \rfloor$ to be the number of pixels outside the image border the filter needs, and that "out-of-bounds" subscripts are handled by padding. In forward propagation, a layer receives its input x and computes its output y as

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

In backpropagation, a layer receives an input *error signal* $\frac{dL}{dy}$ and computes its weights gradient $\frac{dL}{dw}$ and an output error signal $\frac{dL}{dx}$ as

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

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

These are the backward-filter and backward-data phases of computation, respectively. In this work, we do not focus on the implementation of convolution itself, and instead rely on cuDNN [11] to provide an optimized implementation for a single GPU. To emphasize this and simplify notation, we will use ConvForward, ConvBackFilt, and ConvBackData to refer to the computations of y, $\frac{dL}{dw}$, and $\frac{dL}{dx}$, respectively.

2.2 Distributed CNN training

From these equations, the various decompositions we discuss are formed by partitioning one or more dimensions (N, C, F, H, or W), and performing the necessary communication to ensure the correct result is computed. For example, the common data-parallel approach to distributed training, where samples are partitioned among different processors, is immediately clear from the above equations: computing y and $\frac{dL}{dx}$ can be done entirely locally, but only the part of the sum for $\frac{dL}{dw}$ that involves local samples can be done. Computing the remainder of the sum requires a global allreduce. Similarly, the H or W dimensions can be partitioned among processors, but in addition to the allreduce, halo exchanges on x and $\frac{dL}{dx}$ are required to compute the values of convolution near the partition boundaries [17].

Following the convention of [17] we refer to exploiting parallelism along a dimension as "parallelizing" that dimension. Thus, the methods just described are sample parallelism and height and width (or spatial) parallelism. Note that these methods are orthogonal and can be combined in hybrid methods. In this work, we introduce algorithms for channel and filter parallelism to exploit parallelism within the remaining dimensions: C and F.

2.3 Notation

We introduce the notation we use to describe our algorithms. This notation is meant to be a simplified, high-level description of how tensors are distributed while eliding many "lower level" details. It is inspired by the notation developed for the FLAME project [50, 51], and also inspired by the approach of High Performance Fortran [3].

2.3.1 Processor grid. A layer is distributed over a processor grid $\mathcal{G}$, which is constructed as a multi-dimensional grid over the available processors. $\mathcal{G}$ may vary between layers, since they can use different parallelization strategies or algorithms. Note that the order of dimensions for the processor grid has no bearing on the ordering of dimensions for local data, which can be laid out in whatever manner is most efficient.

2.3.2 Distributions. A distribution $\mathcal{D}_a^{\mathcal{G}}$ specifies which entries in a tensor a reside on which processors in a processor grid $\mathcal{G}$ (we will omit $\mathcal{G}$ when clear). Our algorithms use Cartesian distributions where each dimension N is either distributed over a processor grid dimension $\mathcal{N}$, denoted $\circ_{\mathcal{N}}^{N}$, or replicated, denoted $*^{N}$ (omitting N and $\mathcal{N}$ when clear). Thus, we can define the distribution of a by specifying $\circ$ or $*$ for each dimension.

For example, hybrid sample and spatial parallelism is given by a distribution over a $(\mathcal{N}, \mathcal{H}, \mathcal{W})$ processor grid. The input tensor x (with dimensions $N \times C \times H \times W$) is $x[\circ, *, \circ, \circ]$, indicating that the sample dimension N and the spatial dimensions H and W are distributed, while the channel dimension C is replicated. (The boundaries of the H and W distributions on each processor will need to overlap in order to properly describe the necessary halo exchanges.) This notation is intended to convey the high-level intuition about data distributions.

We now define the notation precisely. A distribution $\mathcal{D}_a^{\mathcal{G}}$ is a map from a multi-index in $\mathcal{G}$ to a set of multi-indices corresponding to entries of a. To be a valid distribution, we require that each element of a be assigned to at least one processor. Although this can define arbitrary distributions, our algorithms require only a few cases. Let N be a tensor dimension, $\mathcal{N}$ a processor grid dimension, $|N|$ the size of tensor dimension N, and $[|N|]$ the first $|N|$ integers. Then the function $\circ_{\mathcal{N}}^{N}(\text{proj}_{\mathcal{N}}(p))$ maps from $\mathcal{G}$ to a subset of $[|N|]$, giving a distribution on $\mathcal{N}$ for N. The projection ensures that the distribution is based solely on a processor's index in $\mathcal{N}$. The function $*^{N}(p)$ returns $[|N|]$ for any $p \in \mathcal{G}$. When $\circ$ and $*$ are specified for each tensor dimension, we can construct a distribution by taking the Cartesian product of these functions. Throughout this paper, we will use $\circ$ to denote a block, load-balanced distribution. While other distributions are possible in general, a block distribution is necessary for partitioning spatial dimensions, as convolution requires spatially adjacent data.

Note that a tensor cannot be distributed over the same grid dimension multiple times, or else some entries will not reside on any processor. If a processor grid dimension is not specified in $\mathcal{D}_a^{\mathcal{G}}$, the data is replicated over that dimension. We refer to such dimensions as the *redundant* grid dimensions. (This is equivalent to the notion of redundant communicators in Elemental [50].)

2.3.3 Segmented collectives. We will make use of *segmented* collectives, wherein disjoint sets of processors concurrently perform the same collective operation using only the data local to the processors. A segmented collective has the potential to be significantly faster than a global collective, since it is performed among a smaller set of processors, and may involve less data from each processor. The processors involved in such a collective correspond to the redundant grid dimensions for the tensor the collective is performed on.

2.3.4 Redistributions. We use $a[\overline{\mathcal{D}_a^{\mathcal{G}}}] \longleftarrow a[\mathcal{D}_a^{\mathcal{G}}]$ to denote a redistribution (typically, an allgather) of a from distribution $\mathcal{D}_a^{\mathcal{G}}$ to $\overline{\mathcal{D}_a^{\mathcal{G}}}$. On a per-dimension basis, there are four basic cases:

- $* \longleftarrow \circ$ is an allgather that assembles the complete set of indices for a dimension on every processor involved.
- $\circ \longleftarrow *$ *discards* local data.
- $\circ \longleftarrow \circ$ and $* \longleftarrow *$ are NOPs.

We may also need to specify a redistribution $\circ_{d'} \longleftarrow \circ_{d}$ that moves data between distributions over different processor grid dimensions; here we require that d be a proper subset of d' (or vice-versa). These are discards of local data or segmented allgathers.

2.3.5 Reductions. Data movement may also involve collective reductions, which we denote explicitly as $a[\mathcal{D}_a^{\mathcal{G}}] \longleftarrow \widetilde{\sum_{\mathcal{A}}} a[\mathcal{D}_a^{\mathcal{G}}]$. This is logically a segmented allreduce over the redundant grid dimensions. For clarity, we include a subscript to explicite indicate the dimensions being reduced over: The reduction will be over the processors that have different indices for the dimensions in $\mathcal{A}$, and the same indices for the remaining dimensions of $\mathcal{G}$ (this can be omitted). Note that one could define reductions entirely in terms of data distributions, but we do not use that approach here.

We mention an important special case that fuses a reduction and a redistribtion: $a[\overline{\mathcal{D}_a^{\mathcal{G}}}] \longleftarrow \widetilde{\sum} a[\mathcal{D}_a^{\mathcal{G}}]$ where the distribution change is $\circ \longleftarrow *$. This can be implemented as a segmented reduce-scatter, which is significantly cheaper than an allreduce.

3 ALGORITHMS

We now introduce our algorithms, beginning with an example of sample parallelism to illustrate our notation with a familiar algorithm. We assume that input tensors are distributed as required, and that output tensor distributions match what the algorithms "naturally" produce. Explicit redistributions can be added if needed.

3.1 Example: Sample parallelism

Recall that sample parallelism partitions only the N dimension of input/output tensors and that w is replicated on every processor. Our processor grid will consist of only one dimension, $\mathcal{N}$, to distribute the sample dimension. The distributions of x and $\frac{dL}{dy}$ are $[\circ, *, *, *]$, as are the output distributions for y and $\frac{dL}{dx}$. As the weights are replicated, w and $\frac{dL}{dw}$ have distribution $[*, *]$, where we omit the

$K \times K$ dimensions for simplicity (they are replicated in every case). We also make use of temporary variables denoted T_i. With this, sample parallelism in our notation is implemented as:

(1) $y[\circ, *, *, *] \longleftarrow \text{ConvForward}(x[\circ, *, *, *], w[*, *])$
(2) $\frac{dL}{dx}[\circ, *, *, *] \longleftarrow \text{ConvBackData}(\frac{dL}{dy}[\circ, *, *, *], w[*, *])$
(3) $T_3[*, *] \longleftarrow \text{ConvBackFilt}(\frac{dL}{dy}[\circ, *, *, *], x[\circ, *, *, *])$
(4) $\frac{dL}{dw}[*, *] \longleftarrow \widetilde{\sum_{\mathcal{N}}} T_3[*, *]$.

From this, we can see all the important parts of the algorithm and its communication: forward and backward propagation can be conducted entirely locally, and then a global reduction is performed to compute the final value of $\frac{dL}{dw}$ (completing the sum over N in Eq. 2). This reduction is performed over all processors that have different indices for the $\mathcal{N}$ dimension; since that dimension is the only one that is distributed, and based upon the transition rules in Section 2.3.5, this is a global allreduce.

Extending this algorithm to include spatial parallelism in addition to sample parallelism is simple (noting that we do not explicitly represent the halo exchange): x, y, $\frac{dL}{dx}$, and $\frac{dL}{dy}$ would instead be distributed as $[\circ, *, \circ, \circ]$ on a processor grid with dimensions $(\mathcal{N}, \mathcal{H}, \mathcal{W})$, and the reduction would be over the N, H, W dimensions. This would still involve a global allreduce.

3.2 Stationary-x

We now describe the first of our algorithms exploiting channel and filter partitioning, which we call *stationary-x*. We refer to it as "stationary" by analogy with stationary matrix product algorithms [51], which avoid communicating a particular matrix. We first present the algorithm in our notation and then discuss our reasoning behind it and some implications.

We will assume that input data is distributed channel-wise in addition to using sample and spatial parallelism: $x[\circ, \circ, \circ, \circ]$ over a processor grid $(\mathcal{N}, \mathcal{H}, \mathcal{W}, C)$. This choice of dimension ordering is not needed for correctness, but does result in good performance, as bandwidth-intensive communication of activations will tend to use faster links (e.g. NVLink2) due to channels being partitioned over a small number of close GPUs. Note that the order of processor grid dimensions need not match the tensor's, and that we will write tensors in the ordering given in Section 2.1 (e.g. $N \times C \times H \times W$). Similarly, we will assume that the desired distribution of y is $[\circ, \circ_C, \circ, \circ]$, where the F dimension of y is distributed according to the C dimension of the processor grid; this ensures that the inputs and outputs have matching data distributions. The tensors $\frac{dL}{dy}$ and $\frac{dL}{dx}$ are distributed similarly. Unlike sample or spatial parallelism, we distribute the weights by channels instead of replicating them: $w[*, \circ]$. Thus, w is distributed such that each rank has, for every filter, only the channel parameters for its local input channels. Note, when combining this with hybrid parallelism (e.g. sample), these may still be replicated, but not over every processor. With this, the algorithm is as follows:

(1) $T_1[\circ, *, \circ, \circ] \longleftarrow \text{ConvForward}(x[\circ, \circ, \circ, \circ], w[*, \circ])$
(2) $y[\circ, \circ_C, \circ, \circ] \longleftarrow \widetilde{\sum_C} T_1[\circ, *, \circ, \circ]$
(3) $\frac{dL}{dy}[\circ, *, \circ, \circ] \longleftarrow \frac{dL}{dy}[\circ, \circ_C, \circ, \circ]$
(4) $\frac{dL}{dx}[\circ, \circ, \circ, \circ] \longleftarrow \text{ConvBackData}(\frac{dL}{dy}[\circ, *, \circ, \circ], w[*, \circ])$
(5) $T_3[*, \circ] \longleftarrow \text{ConvBackFilt}(\frac{dL}{dy}[\circ, *, \circ, \circ], x[\circ, \circ, \circ, \circ])$
(6) $\frac{dL}{dw}[*, \circ] \longleftarrow \widetilde{\sum_{\mathcal{N}, \mathcal{H}, \mathcal{W}}} T_3[*, \circ]$.

As a high-level summary, this algorithm first does forward propagation locally (1), then performs a segmented reduce-scatter (2) among each set of processors that has different channels for the same sample and spatial region, in order to complete the sum over channels in Eq. 1 and then produce the correct distribution for y. The scatter is needed because each processor locally produces output for every filter. (See Figure 1, left.) Backpropagation begins with a segmented allgather (3) to assemble the filters of $\frac{dL}{dy}$. Backward-data (Eq. 3) can then be computed completely locally (4). Backward-filter (Eq. 2) can be partially computed locally (5), but completing the summations still requires aggregating over all sample and spatial distributions (6). (See Figure 1, right.) A key advantage of this algorithm is that this allreduce is no longer global: it is a segmented allreduce over only the local data, among processors that have the same channel data. Thus, we reduce both the amount of data that is allreduced, and the number of processors participating in each allreduce.

This algorithm trades additional communication overhead during forward and backpropagation (from the reduce-scatter and allgather) for additional parallelism among channels. Convolutions among different channels can be performed concurrently in forward propagation and for backward-data, reducing total compute time. In backpropagation, the allgather of $\frac{dL}{dy}$ before computing ConvBackData is necessary given our distribution of w. Other distributions would require either additional data or communication. It also significantly reduces memory usage, as the complete channels of each tensor need only be stored during the forward or backpropagation pass through their particular layer.

We also partition w by channels, as opposed to fully replicating it as is typical, for two reasons. First, as the input data is partitioned by channels, only the parameters corresponding to those channels would be used during forward and backpropagation on each processor. Second, if weights were replicated on every processor, the backward-filter stage would require a global allreduce to update the parameters. By partitioning w, we are able to instead use a segmented allreduce to reduce communication overheads, since updates need only be communicated among processors with the same parameters. Thus, for example, if a combination of sample and channel parallelism is used and channels are partitioned among two processors, we perform two disjoint allreduces, each among half the processors and over half the data. We also considered alternate communication patterns, but these result in excess communication or buffer space.

As a future optimization to reduce the communication overhead from reduce-scatters and allgathers, this algorithm (as well as the stationary-y and -w algorithms) is amenable to blocking and pipelining, similarly to stationary matrix product algorithms.

3.3 Stationary-y

The stationary-y algorithm is symmetric to the stationary-x algorithm, and uses a $(\mathcal{N} \times \mathcal{H} \times \mathcal{W} \times \mathcal{F})$ grid. It avoids communicating y and $\frac{dL}{dy}$ instead of x and $\frac{dL}{dx}$ (compare lines (2) and (3) of stationary-x to (4) and (1) below, respectively). The tensors x, y, $\frac{dL}{dx}$, and $\frac{dL}{dy}$

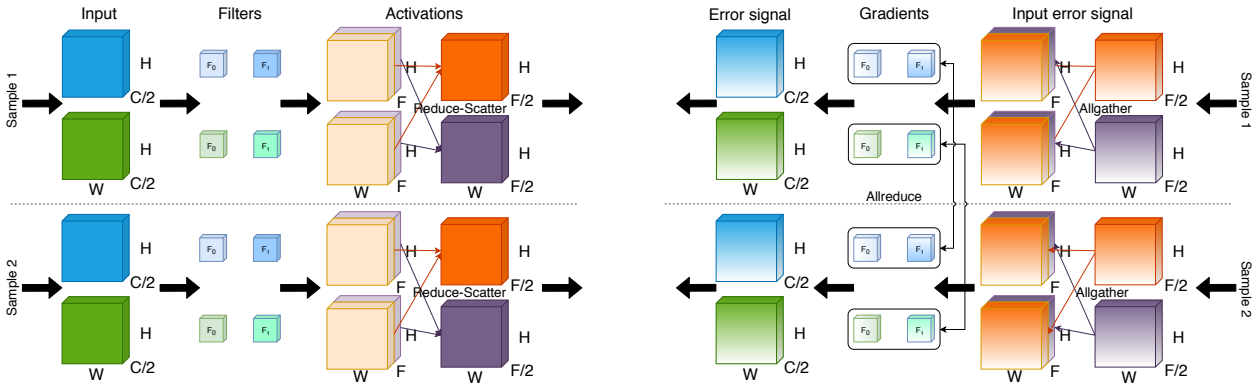

Figure 1: Illustration of the stationary-x algorithm's forward (left) and backpropagation (right) for one layer and two samples, using two-way channel/filter partitioning on four processors.

are distributed as in the stationary-x algorithm. w is distributed as $[\circ, *]$, so each processor has only its local filters. The algorithm is:

(1) $x[\circ, *, \circ, \circ] \longleftarrow x[\circ, \circ_{\mathcal{F}}, \circ, \circ]$
(2) $y[\circ, \circ, \circ, \circ] \longleftarrow \text{ConvForward}(x[\circ, *, \circ, \circ], w[\circ, *])$
(3) $T_2[\circ, *, \circ, \circ] \longleftarrow \text{ConvBackData}(\frac{dL}{dy}[\circ, \circ, \circ, \circ], w[\circ, *])$
(4) $\frac{dL}{dx}[\circ, \circ_{\mathcal{F}}, \circ, \circ] \longleftarrow \widetilde{\sum_{\mathcal{F}}} T_2[\circ, *, \circ, \circ]$
(5) $T_3[\circ, *] \longleftarrow \text{ConvBackFilt}(\frac{dL}{dy}[\circ, \circ, \circ, \circ], x[\circ, *, \circ, \circ])$
(6) $\frac{dL}{dw}[\circ, *] \longleftarrow \widetilde{\sum_{\mathcal{N},\mathcal{H},\mathcal{W}}} T_3[\circ, *]$.

For this, the communication patterns have been swapped between forward and backpropagation compared to the stationary-x algorithm. An allgather is performed at the beginning of forward propagation (1), followed by local convolution (2). The backward-data step can first be performed locally (3) and is then followed by a reduce-scatter (4) to complete the sum over filters (Eq. 3). The backward-filter computation (5, 6) is similar to the stationary-x algorithm, except now the segmented allreduce aggregates gradient updates among processors with the same filters. A similar discussion of communication choices also applies.

3.4 Stationary-w

We now present the stationary-w algorithm (the name is chosen for consistency despite the reduction on $\frac{dL}{dw}$). This is also the most complex of the algorithms we present, and, in fact, both the stationary-x and -y algorithms are special cases of this one. At a high level, this algorithm distributes the C dimension of x, the F dimension of y and both C and F dimensions of w, which requires communication during both forward and backpropagation. The parameter update allreduce remains segmented. The communication pattern essentially combines those of the stationary-x and y algorithms.

The processor grid for this algorithm is $(\mathcal{N}, \mathcal{H}, \mathcal{W}, \mathcal{C}, \mathcal{F})$. Input and output tensors are distributed over the linearization of the $\mathcal{C}$ and $\mathcal{F}$ dimensions of the processor grid. We will assume that x and $\frac{dL}{dx}$ are distributed according to $[\circ, \circ_{\mathcal{F}\times\mathcal{C}}, \circ, \circ]$ and y and $\frac{dL}{dy}$ according to $[\circ, \circ_{\mathcal{C}\times\mathcal{F}}, \circ, \circ]$. The weights are distributed according to $w[\circ, \circ]$, and so are partitioned over both the $\mathcal{F}$ and $\mathcal{C}$ dimensions of $\mathcal{G}$. The stationary-w algorithm is now:

(1) $x[\circ, \circ_{\mathcal{C}}, \circ, \circ] \longleftarrow x[\circ, \circ_{\mathcal{F}\times\mathcal{C}}, \circ, \circ]$

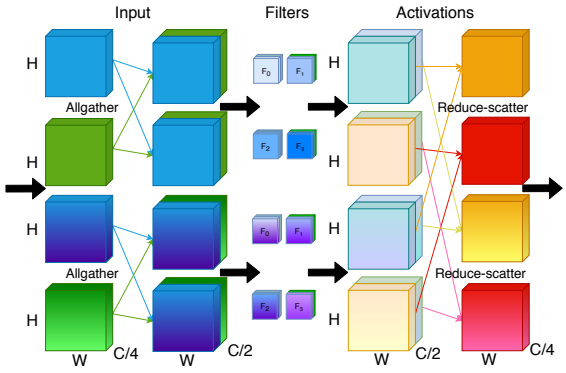

Figure 2: Illustration of the stationary-w algorithm's forward propagation for one layer and sample, four channels and filters, using a 2×2 distribution of channels and filters.

(2) $T_1[\circ, \circ_{\mathcal{F}}\circ, \circ] \longleftarrow \text{ConvForward}(x[\circ, \circ_{\mathcal{C}}, \circ, \circ], w[\circ, \circ])$
(3) $y[\circ, \circ_{\mathcal{C}\times\mathcal{F}}, \circ, \circ] \longleftarrow \widetilde{\sum_{\mathcal{C}}} T_1[\circ, \circ_{\mathcal{F}}, \circ, \circ]$
(4) $\frac{dL}{dy}[\circ, \circ_{\mathcal{F}}, \circ, \circ] \longleftarrow \frac{dL}{dy}[\circ, \circ_{\mathcal{C}\times\mathcal{F}}, \circ, \circ]$
(5) $T_2[\circ, \circ_{\mathcal{C}}, \circ, \circ] \longleftarrow \text{ConvBackData}(\frac{dL}{dy}[\circ, \circ_{\mathcal{F}}, \circ, \circ], w[\circ, \circ])$
(6) $\frac{dL}{dx}[\circ, \circ_{\mathcal{F}\times\mathcal{C}}, \circ, \circ] \longleftarrow \widetilde{\sum_{\mathcal{F}}} T_2[\circ, \circ_{\mathcal{C}}, \circ, \circ]$
(7) $T_3[\circ, \circ] \longleftarrow \text{ConvBackFilt}(\frac{dL}{dy}[\circ, \circ_{\mathcal{F}}, \circ, \circ], x[\circ, \circ_{\mathcal{C}}, \circ, \circ])$
(8) $\frac{dL}{dw}[\circ, \circ] \longleftarrow \widetilde{\sum_{\mathcal{N},\mathcal{H},\mathcal{W}}} T_3[\circ, \circ]$.

This algorithm first does a segmented allgather of the channels of x such that they match the channel distribution of w (1). This does not fully assemble the channels, but is required since the C dimension of x is distributed over $\mathcal{C} \times \mathcal{F}$ while the C dimension of w is distributed over only $\mathcal{C}$. Forward propagation is performed locally (2), and then a segmented reduce-scatter (3) completes the summation and scatters data back over the processor grid. (See Figure 2.) Note that this has transposed the channel dimension of y over the $\mathcal{F}\times\mathcal{C}$ subgrid, compared to x. This is due to the distribution of w and the communication patterns. No additional data movement is required for the subsequent layer, as its w can be transposed to account for this. The backward-data phase (4-6) is symmetric. The

backward-filter phase first does a local computation (7) and then aggregates the updates among corresponding processors (8).

The stationary-w algorithm generalizes both other algorithms, as we can recover them by making the $\mathcal{F}$ or C processor grid dimension be size 1, respectively. When this is done, some of the collective operations (e.g., steps (1) and (6) for stationary-x) can be elided, as they are performed over a single processor. When referring to a particular instance of the stationary-w algorithm, we indicate the $C \times \mathcal{F}$ grid it uses (e.g. 4×2).

3.5 Other layers

Partitioning data along the channel/filter dimension has implications for other commonly-used layers in CNNs. Most layers naturally adapt to this partitioning and require no additional communication while benefiting from reduced local computation. Element-wise layers, such as ReLU or dropout, are trivially adapted. Similarly, pooling is applied channel-wise and thus requires no further adaptation. Channel partitionings do not make sense for fully-connected layers, as they operate on a linearized input.

Batch normalization [26] is also applied channel-wise, so its computation requires no significant adjustment. It does have learnable parameters, but these are distributed channel-wise, so the corresponding allreduce becomes segmented. Thus, a channel partition can help reduce communication overhead for batch normalization layers in addition to convolutional layers.

4 PERFORMANCE MODEL

We now present a simple performance model of our algorithms and discuss some key performance implications. We will focus on the stationary-x algorithm, as the others are similar. The primary purpose of this model is to verify that the performance results we obtain are consistent with what we expect from the algorithms.

We use ConvForward, ConvBackData, and ConvBackFilt to refer to the local time taken to perform convolution on the local data of their arguments. We do not use an analytic model, as convolution libraries may select among many algorithms based upon the particular problem (direct, im2col [8], Winograd [32], FFT [39], etc.). We use the analytic communication models of [56] for collective operations; since again there are many possible algorithms (tree, butterfly, ring, etc.), we use $\text{AR}(n, p)$ to be the time for an allreduce on n words over p processors, $\text{RS}(n, p)$ for a reduce-scatter, and $\text{AG}(n, p)$ for an allgather. We write I_N^a to refer the local size of dimension N of tensor a (assuming a distribution from context), and use $|\mathcal{N}|$ to refer to the size of processor grid dimension $\mathcal{N}$.

With this, the time for stationary-x forward convolution is

$$\text{ConvForward}(x[\circ, \circ, \circ, \circ], w[*, \circ], S, P) + \text{RS}(I_N^y I_F^w I_H^y I_W^y, |C|)$$

where S and P are the stride and padding. To simplify, we have neglected the cost of halo exchanges for spatial parallelism, as they can typically be overlapped [17]. Note that due to the stride and padding, the spatial dimensions of y may be different from x.

Backpropagation time (including the allreduce) is given by

$$\begin{aligned}
&\text{AG}(I_N^{\frac{dL}{dy}} I_F^w I_H^{\frac{dL}{dy}} I_W^{\frac{dL}{dy}}, |C|) \\
&+ \text{ConvBackData}(\frac{dL}{dy}[\circ, *, \circ, \circ], w[*, \circ], P, S) \\
&+ \text{ConvBackFilt}(\frac{dL}{dy}[\circ, *, \circ, \circ], x[\circ, \circ, \circ, \circ], P, S) \\
&+ \text{AR}(I_F^w I_C^w K^2, |\mathcal{N}||\mathcal{H}||\mathcal{W}|).
\end{aligned}$$

While we have included it here, the allreduce time can be overlapped with ConvBackData and computation in other layers.

One can now see the tradeoffs within this algorithm: we perform additional communication in forward and backpropagation (which cannot be overlapped, except by blocking), reduce local computation, and perform allreduces over smaller sets of processors and buffers. Whether this is worthwhile depends on how local computation scales, the communication performance on the C processor grid dimension, and the decreased allreduce communication costs. This also implies that strong scaling will be ultimately limited by the unoverlapped communication overheads from the reduce-scatters and allgathers.

This also sheds light on how to choose between the stationary-x and y algorithms: it depends upon the relative number of channels and filters in a convolution, and the stride and padding parameters. If a layer has fewer channels than filters, the y algorithm may more efficiently partition the data and perform less communication. However, if the convolution has strides above one, or does not use "same" padding, the spatial domain of y will be smaller than x, meaning that the stationary-x algorithm performs less communication when the number of channels and filters are comparable. This matches the intuition that the larger tensor should be kept stationary. Whether the stationary-w algorithm is worthwhile depends on whether the decreased local computation and allreduce communication outweighs the additional reduce-scatters and allgathers and the decreased returns to partitioning one dimension too much.

Note that these algorithms require additional memory to hold data before or after reduce-scatters or allgathers, although the memory required to hold parameters is decreased. This memory need not be permanently allocated for the stationary-x algorithm, as it is only used by temporaries and could be allocated from a shared memory pool. The stationary-y and w algorithms requires the allgathered x data to be held until the layer completes backpropagation.

5 IMPLEMENTATION

We have implemented these algorithms by extending the open-source LBANN scalable deep learning toolkit [58] and its distributed tensor library [17]. This provides an underlying substrate for MPI-based parallel training, GPU acceleration, and tensor partitioning. While LBANN has efficient support for sample and spatial parallelism, it does not support partitioning the channels of activation tensors or convolution weights.

We implemented the appropriate tensor distributions for our algorithms and added support to the convolutional layers for such distributions. Local convolution is performed with NVIDIA's cuDNN library [11]. Each convolutional layer allocates temporary memory using a shared CUB memory pool [44] that releases memory after

the layer's forward or backpropagation work is done, in order to minimize memory overheads. NCCL [43] is used to implement the reduce-scatter and allgather communication, for both intra- and inter-node communication. Since our implementation locally stores data in an *NCHW* format, this communication is non-contiguous when $I_N^x > 1$. To mitigate this, we implement custom CUDA kernels for packing (for reduce-scatter) and unpacking (for allgather).

LBANN's optimization framework was also modified to support segmented allreduces, again using NCCL. To do this, we simply split the associated communicator appropriately. During testing, we observed that segmented tree allreduces using NCCL had very high performance variability, and that a similar effect was present with our system MPI. We therefore use only ring allreduces, which did not exhibit such variability. As many of the allreduces are large ($\geq$ 100 KiB), we do not expect this to fundamentally impact our results. We observed that concurrent NCCL allreduces and reduce-scatters or allgathers during backpropagation overlap well, so we are still able to effectively hide allreduce communication.

We validated our implementation by extensively testing it and comparing its results to convolution performed on a single GPU. Adapting other layers (e.g., pooling, batch normalization) was straightforward once the tensor library supported channel- and filter-parallel distributions.

6 EVALUATION

We now evaluate our algorithms using both microbenchmarks and end-to-end training. We examine independently the compute and segmented allreduce performance of our algorithms for representative layers of ResNet-50, and then the performance of the full algorithm. This allows us to understand the upper bound of possible performance improvements. We focus our microbenchmarks on the stationary-x algorithm in this paper, as the others exhibit similar trends. We then evaluate the strong and weak scaling performance for end-to-end training of ResNet-50 and two wider architectures using all our algorithms. Finally, we report accuracy results and speedups for our models.

We use the Lassen supercomputer [36]. It consists of 795 nodes, each with two IBM POWER9 CPUs and four NVIDIA V100 (Volta) GPUs with NVLink2. Nodes have 256 GB of memory and each GPU has 16 GB of memory. It is interconnected via dual-rail InfiniBand EDR. Each CPU has two GPUs attached to it. The NVLink2 interconnect does not cross the socket boundary; thus intra-node communication between GPUs on different sockets uses the socket interconnect, which has significantly lower bandwidth. Our implementation uses a recent development version of LBANN, GCC 7.3.1, Spectrum MPI 2019.01.30, CUDA 9.2.148, cuDNN 7.5.0, and NCCL 2.4.2. For all results we use single-precision floating point.

ResNet-50 [22] is a standard image recognition CNN with near state-of-the-art performance. It also forms the basis for many modern classification architectures (e.g. [24, 54, 59]) and architectures for other tasks (e.g. [9, 48, 63]). We therefore expect that improvements for ResNet-50 training will have broad impact. The network consists of a stack of five blocks of layers, `conv_1` through `conv_5`, which process progressively smaller spatial domains with larger filter banks. Our evaluation focuses extensively on this network.

To understand the impact of wider CNN architectures, we also evaluate end-to-end training for two Wide ResNet (WRN) [62] architectures, WRN-50-2 and -50-4. These networks have a similar structure to ResNet-50, but the 3×3 convolutions have the number of filters (their *width*) increased by a factor of 2 and 4, respectively. WRN-50-2 achieved superior accuracy to ResNet-50 in the authors' evaluation; they were unable to evaluate wider networks due to memory constraints. We are able to demonstrate the training of these wider models, which had not previously been done, and show the accuracy improvements they bring. We use fully-convolutional [37] versions of all models.

We evaluate both strong and weak mini-batch scaling. Strong scaling uses additional GPUs to train with a fixed mini-batch size; weak scaling fixes the number of samples per GPU and grows the global mini-batch size. Our evaluations will consist of *hybrid* sample and channel/filter parallelism: data is partitioned over an $\mathcal{N} \times C \times \mathcal{F}$ grid. Thus, when weak scaling, we fix a mini-batch size for each "row" of the $\mathcal{N}$ processor grid dimension. When using two- or four-way channel parallelism, we always partition channels within a single socket or node, respectively. Eight-way channel parallelism requires two nodes, and hence inter-node communication for reduce-scatters and allgathers. We also explore combinations of partitioning schemes, where different layers are partitioned with different techniques to best exploit parallelism.

6.1 Microbenchmarks

Compute. We first examine the computational strong scaling of the cuDNN kernels used for local convolution with a fixed mini-batch size as we partition a layer. Figure 3 presents the local convolution time on a single GPU as a layer is partitioned over additional GPUs. We select two layers, one near the beginning of ResNet-50 (in the `conv_3` block, which has relatively few channels and filters), and one near the end (in `conv_5`, which has many more channels and filters). Precise configurations are given in the figures. We examine three mini-batch sizes for strong scaling: $N = 1$, 2, and 32, which correspond to extreme sample parallelism and a more typical number of samples per GPU. Note that results with different mini-batch sizes are not directly comparable; it is the scaling trends that are important. To evaluate this, we used CUDA events to time the convolution kernel on GPU. We first did several warmup runs, then reported the mean of five runs. To avoid measuring kernel launch overheads, we used a spin kernel to ensure all work reached the GPU before beginning measurements. We use auto-tuning to select the fastest cuDNN algorithm for each configuration.

The $N = 1$ case is most indicative of the kernel scaling as channels are decreased. For the `conv_3` layer, while we observe improvements due to channel partitioning, the scaling is sub-linear. We also observe that forward propagation tends to scale better than backpropagation. Direct convolution should scale linearly regardless of how a dimension is partitioned, but optimized algorithms may exploit the locality in channel accesses (e.g. im2col); as we partition channels, there is less locality to exploit. Given the small total runtime, other overheads (e.g. kernel scheduling) may also be significant. A similar trend occurs for $N = 2$, although with such a small mini-batch it appears better to partition the channels instead of the samples. The $N = 32$ case exhibits a trend where sample

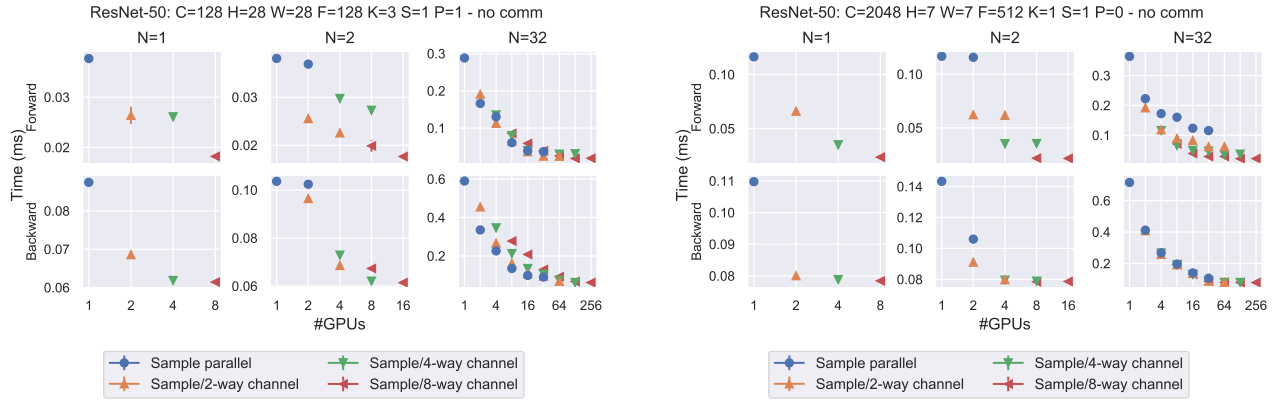

Figure 3: Microbenchmark results for a conv_3 (left) and conv_5 (right) layer of ResNet-50 without any communication.

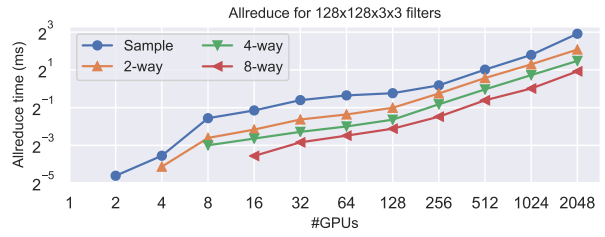

Figure 4: Segmented allreduce time for a conv_3 layer of ResNet-50.

parallelism initially scales better, but eventually hybrid sample and channel parallelism produces better results.

The trend is similar for the conv_5 layer, although forward propagation scaling is much closer to linear. For $N = 32$, we observe that channel parallelism consistently outperforms sample parallelism, due to the large number of channels. Backpropagation is more comparable, although there are slight benefits.

Communication. We next examine the allreduce performance in isolation, as it is the key communication bottleneck when scaling. We illustrate this with the conv_3 layer, as the trends in other layers are similar. Figure 4 presents these results. To obtain these measurements, we first conducted several warmup runs, then timed the appropriate segmented allreduce on a buffer sized appropriately for the amount of channel parallelism (e.g. half the size for two-way). We can see that for a given number of GPUs, the segmented allreduce is faster. The large jumps in runtime for sample and two-way allreduces are due to crossing socket and/or node boundaries. As bandwidth-bound ring allreduces are primarily sensitive to the vector size, much of this improvement is driven by the reduced buffer size. However, latency may become more of a factor at large scale or with smaller parameter buffers (e.g. for batch normalization). Thus, greater segmentation may be more beneficial for such cases.

We observe that while segmentation improves performance, it is again not a linear improvement. For a ring allreduce on a large buffer, runtime is approximately linear in the buffer size; we observe that eight-way segmentation results in only a 4x reduction in runtime. We investigated whether this was due to contention on shared InfiniBand HCAs, but this did not explain the results. Our hypothesis is that NCCL has all InfiniBand control performed by threads on a single core of each socket. For a segmented allreduce, the ranks on each socket are in different segments of the collective, creating contention. A communication framework optimized for segmented allreduces should be able to mitigate this impact. Nonetheless, we observe reductions in allreduce runtime of over 50% and up to 4x at large scale.

Summary. Based upon the individual computation and communication scaling trends, we expect to see limited gains in compute time, concentrated in layers with many channels or filters. Runtime gains will also be limited by the communication overheads of reduce-scatters and allgathers. However, we can expect to see significant improvements in allreduce overhead at large scale, particularly as the faster communication makes it easier to overlap communication and computation.

Layer scaling. We now examine the runtime of the conv_3 and conv_5 layers above using the full stationary-x algorithm. Our measurement methodology is as above, and we perform all communication. As we are looking at a single layer, the allreduce is not overlapped, and its runtime is included in backpropagation. Figure 5 presents strong scaling for these results. For conv_3, forward and backpropagation are flat or do not scale well when N is small; this is expected according to our performance model as the overhead of communication is high and this layer has few channels. Eight-way channel parallelism, in particular, suffers from the high cost of inter-node reduce-scatters and allgathers. Comparing Figures 3 and 5, we can see that this overhead is often 2x or more. For $N = 32$, we focus on the trends of each configuration and how they compare. Forward propagation scales relatively well, although this is likely due primarily to the scaling of sample parallelism. Backpropagation exhibits a parabolic shape due to the impact of allreduces at larger scales. We can observe that communication overhead is improved at large scale by the segmented collectives, but sample parallelism is still superior. Again, this is expected due to the few channels, but demonstrates that even when a layer is poorly suited to it, moderate channel parallelism does not add too much overhead.

We observe much better trends for conv_5. For small N, eight-way channel parallelism is again not profitable due to inter-node

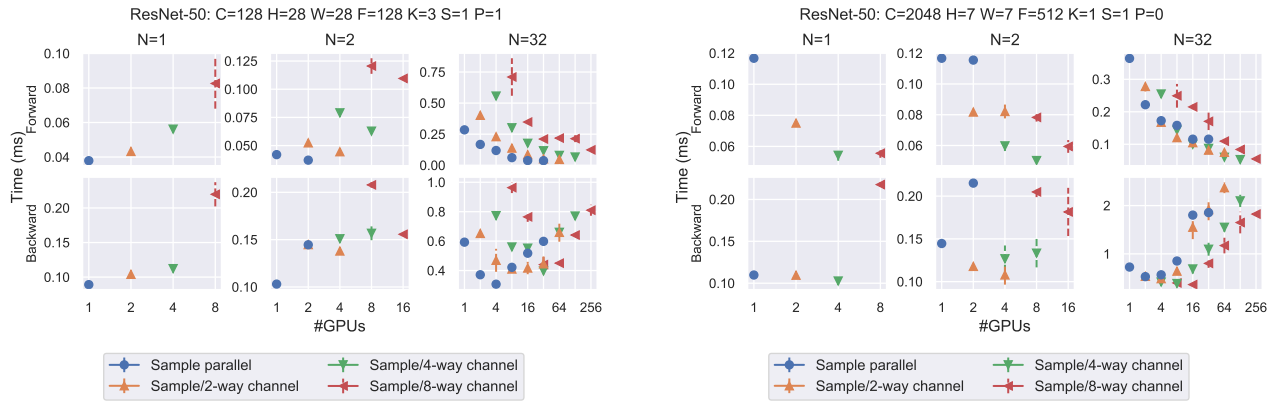

Figure 5: Strong scaling for a conv_3 (left) and conv_5 (right) layer of ResNet-50. Allreduces are included in backpropagation.

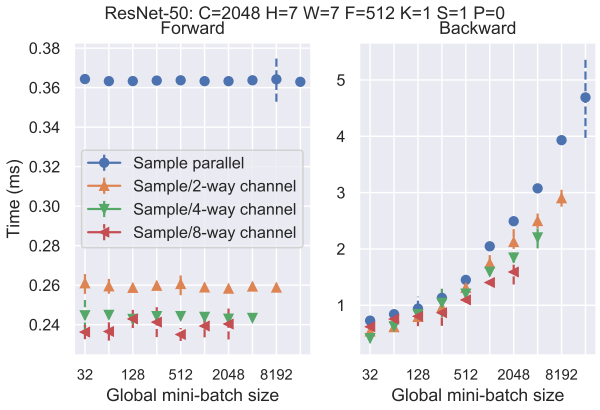

Figure 6: Weak scaling for a conv_5 layer of ResNet-50. Allreduces are included in backpropagation.

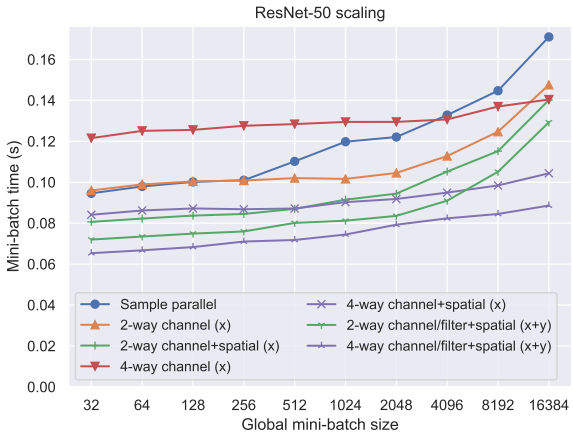

Figure 7: Scaling for end-to-end training of ResNet-50.

communication, but two- and four-way parallelism show more promise. For $N = 32$, channel parallelism scales better than sample parallelism, showing moderate improvements at larger scales, but allreduce overheads again rapidly dominate. Overall, we see moderate benefits from strong scaling via channel parallelism for layers with many channels, and a trend of reduced communication overheads at large scale.

With this in mind, we next consider weak scaling for the conv_5 layer in Figure 6. For this plot, we fix 32 samples per row of N and use hybrid parallelism; thus at the largest scale we consider 1024 GPUs (256 nodes). We use a maximum of 16k samples in sample parallelism, as this is roughly the maximum useful batch size for training ResNet-50 with current techniques [52]. In forward propagation, we observe scaling trends similar to what we would expect from Figure 5, as channel parallelism improves over sample parallelism. Weak scaling is excellent in this case. In backpropagation, we can observe that channel parallelism again outperforms sample parallelism. Further, the rate of increase in allreduce runtime is less with additional channel parallelism: we have successfully reduced communication overheads. In end-to-end training, we can therefore expect channel parallelism to enable easier communication/computation overlap.

Summary. There are some limitations to our results: cuDNN kernels and segmented allreduces do not always scale linearly. Nevertheless, we are able to demonstrate improvements for important layers, primarily through the reduced communication overhead. While the use of ring allreduces may limit weak scaling performance, since the segmentation reduces both the vector size and number of processors involved, we expect these results to translate similarly to tree or butterfly algorithms. Although we have focused on the stationary-x algorithm in this section, the other algorithms have a similar structure and their performance trends are similar.

6.2 End-to-End Training

We now evaluate the performance of end-to-end training on the ImageNet dataset. For simplicity, we use synthetic data that matches the dimensions of ImageNet data; since our compute nodes can fit the entire ImageNet-1K dataset in host memory, this is not significantly different than using real data. We report the average mini-batch time over an epoch, skipping the first mini-batch (which performs

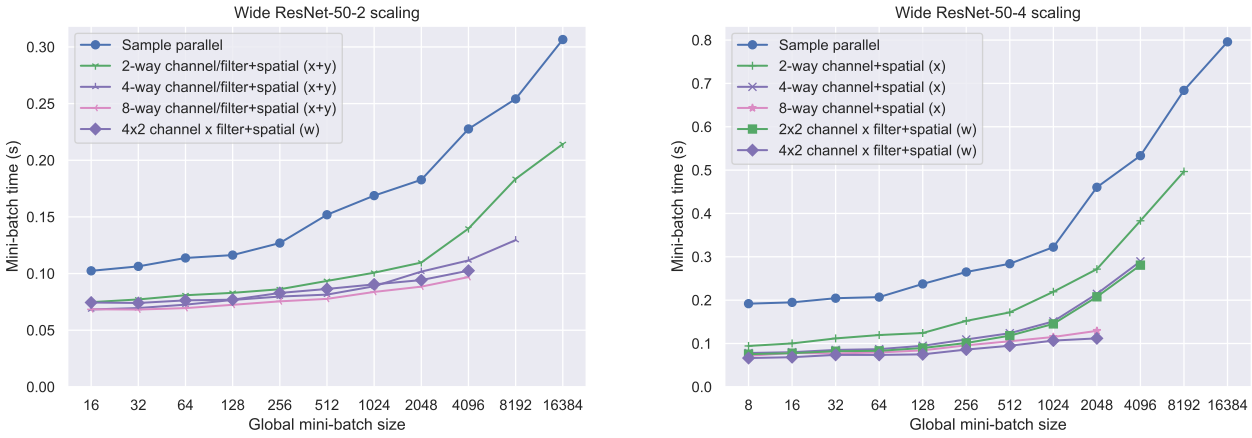

Figure 8: Scaling performance for end-to-end training of Wide ResNet-50-2 and Wide ResNet-50-4.

initialization). We evaluated 2-, 4-, and 8-way channel/filter parallelism using the stationary-x (channel-parallel), -y (filter-parallel), and -w (channel × filter parallel) algorithms with hybrid sample parallelism for each network. We also consider partitioning layers with combinations of schemes, including spatial parallelism, when it is profitable. For clarity, we plot only a subset of the results (see the supplementary material for complete results).

We show ResNet-50 scaling performance in Figure 7 with 32 samples per row of $\mathcal{N}$, again using a maximum mini-batch size of 16k. The four-way stationary-x algorithm uses up to 2048 GPUs (512 nodes). For small mini-batch sizes (32-256), sample and two-way channel parallelism weak scale very well, and have comparable performance. Beyond this, sample parallelism begins to degrade in performance due to allreduce overheads; two-way channel parallelism maintains its weak scaling for another two doublings. Four-way channel parallelism weak scales quite well throughout the entire range of mini-batch sizes, but is outperformed at smaller scales. We have observed that this is due to the overhead of inter-socket reduce-scatters and allgathers compared to compute scaling, primarily for the early layers of ResNet-50. This agrees with our performance model and microbenchmarks. At the largest scales, where allreduce overheads are greatest, four-way channel parallelism is fastest. We do not show eight-way channel parallelism, as it was not beneficial at these scales. The main cause of limited performance at smaller scales is the overhead of reduce-scatters and allgathers for channel parallelism, particularly for early layers in ResNet-50, outweighing the reduced allreduce overhead (which is mostly hidden by other compute).

We also evaluated using the stationary-y and -w for every layer. While they exhibit the same trends, they are outperformed by the stationary-x algorithm. This is because most layers in ResNet-50 have larger x tensors than y tensors, indicating the stationary-x algorithm is preferred. Further, there is insufficient compute in ResNet-50 for the additional parallelism in the stationary-w algorithm to be beneficial.

To reduce these overheads, we consider two additional sets of configurations. First, we used the spatial parallelism of [17] for the conv_1 and conv_2 blocks (a $\mathcal{N}\times\mathcal{H}$ grid) and then the stationary-x algorithm for the remaining layers. These results are also plotted in Figure 7 (with "+spatial"), where we observe that this has significantly mitigated overheads from the early layers while continuing to offer good weak scaling. The overhead from shuffling between different tensor distributions is negligible. At large scales, where allreduce overheads are most acute, we see up to 1.6x speedups with four-way spatial/channel parallelism over sample parallelism, comparable to the results in [17], with much better weak scaling compared to sample parallelism. To further reduce overheads, we extended this spatial/stationary-x hybrid to use the stationary-y algorithm for the layers with y tensors larger than x tensors (about $^1/_{10}$ of layers; plotted as "channel/filter" algorithms). This further reduces communication overheads, with the four-way algorithm outperforming all others, achieving up to 1.9x speedups.

In Figure 8, we show the scaling performance for end-to-end training of WRN-50-2 and -50-4, using 16 and 8 samples per row of $\mathcal{N}$, respectively. We use a smaller mini-batch per GPU due to the increased memory requirements of the networks. Based on the benefits observed for ResNet-50, we plot only configurations that use spatial parallelism for the first two ResNet blocks, as they consistently perform better; but note that due to the wider models, the benefit is less dramatic than for ResNet-50. For WRN-50-2, we show results using the combination of the stationary-x and -y algorithms, which achieves up to 2.4x speedups with eight-way partitioning (which performs inter-node reduce-scatters and allgathers). We also considered the hybrid spatial/stationary-w algorithm, and plot results for this with a 4×2 $\mathcal{C} \times \mathcal{F}$ sub-grid, where performance is comparable to the eight-way stationary-x/-y case. Using a 2×4 sub-grid performed slightly worse, as partitioning the y tensors further offers less advantage. For WRN-50-4, we plot results for spatial/stationary-x and spatial/stationary-w algorithms; due to the increasing width of 3×3 convolutions, this model has very few layers where y tensors are larger than x, so the advantage of the stationary-y algorithm is very small. We see large performance improvements at every scale and benefits from using the stationary-w algorithm due to the large compute requirements for this model.

Table 1: ImageNet validation accuracy and training time.

Model/Algorithm	Mini-batch	Top-1	Top-5	Time (min)
ResNet-50 (sample)	8192	77.3%	93.6%	34.1
+spatial+4-way x/y				19.9 (1.7x)
WRN-50-2 (sample)	4096	78.4%	94.3%	106.9
+spatial+8-way x/y				45.5 (2.3x)
WRN-50-4 (sample)	2048	80.0%	95.1%	432.3
+spatial+4×2 w				105.0 (4.1x)

The eight-way spatial/stationary-w algorithm with a 4×2 sub-grid achieves up to 4.1x performance improvements at large scale and is always at least 2x faster than sample parallelism.

Overall, our algorithms are able to significantly improve performance, especially at large scales or for wide models, which would be otherwise infeasible to train due to memory requirements. The varying benefits of the different stationary algorithms illustrates that they can complement one another and be used in combination to achieve significant training speedups.

6.3 Accuracy and Training Time

We evaluated the accuracy and total training time of these networks on ImageNet, using both sample parallelism and our best-performing algorithm configuration (Table 1). We used the learning rate schedule, hyperparameters, and data augmentation strategy of Goyal et al. [20] for all models and additionally included MixUp augmentation [64]. All models were trained for 90 epochs. Our baseline results for ResNet-50 exceed those of [20], and we show significant improvements in accuracy for wider models, comparable to significantly deeper and/or more complicated architectures (e.g. SE-ResNeXt101 [23]). For Wide ResNet-50-2 and -4, we report results using the mini-batch size that achieved the best accuracy; that this is smaller for wider networks agrees with recent results [45], although additional hyperparameter tuning may improve this. Note that, since our algorithms replicate convolution exactly, they achieve the same accuracy as sample parallelism.

This demonstrates the advantage of wider models, even without additional hyperparameter tuning, to improve accuracy, while avoiding extensive CNN architecture development. This also shows that the warmup and linear scaling rule introduced by [20] can enable training with large batches on models other than ResNet-50.

7 RELATED WORK

There are a great many works on parallelizing CNN training at various scales. We refer the reader to Ben-Nun and Hoefler for a comprehensive overview [6]. In particular, there has been recent work on optimizing training for big scientific or computational simulation datasets on large HPC resources [30, 40]. These have focused on accelerating sample-parallel training via techniques such as optimized communication and I/O. These techniques are, in general, orthogonal to ours, and can be jointly leveraged.

Scaling convolution. AlexNet [29] introduced an early form of model parallelism, partitioning convolutional filters between two GPUs in order to avoid memory limits. However, this made use of grouped convolutions to reduce inter-GPU communication, instead of directly replicating regular convolution. Similarly, DistBelief [15] and Project Adam [12] supported partitioned filters using parameter servers, but did not partition input channels. When adapting this to decentralized settings, the communication patterns also become significantly more complicated. Coates et al. [13] spatially partitioned locally-connected layers, which are similar to convolutional layers, and also employed other model-parallel techniques. Dryden et al. [17] demonstrated efficient strong scaling of CNN training when using hybrid sample/spatial parallelism. In contrast, we partition channels and filters instead of the spatial domain, and also partition convolution parameters. Our work is orthogonal and can be used jointly to further improve both strong and weak scaling performance, as we have demonstrated (Section 6.2).

Demmel and Dinh [16] have developed lower bounds on communication complexity for forward propagation of a single CNN layer, and presented sequential algorithms that achieve them. However, they do not consider an entire training iteration. Gholami et al. [19] presents a general framework for parallelizing CNN training along multiple dimensions, but their results are limited to simulation, and their formulation of channel/filter parallelism differs.

Other neural network scaling. Model-parallelism has a long history, but modern work primarily targets fully-connected layers or distributes different layers to different processors. DistBelief, Project Adam, and TensorFlow [1], among others, support this. LBANN [58] included support for distributing fully-connected layer parameters by leveraging distributed linear algebra libraries. Recently, Mesh-TensorFlow [53] adopted a similar approach and introduced a framework for implementing more general decompositions layers via a set of collective communication primitives, but has focused on model-parallel fully-connected layers. TF-Replicator [7] is similar to Mesh-TensorFlow, but focuses on ease of scaling for existing models. Pipeline parallelism is also a common technique for scaling training, and has been implemented in frameworks such as GPipe [25] and Pipe-SGD [35]. These pipelining techniques are orthogonal to our algorithms and can be jointly implemented.

Accelerating ResNet-50. There has been significant interest in accelerating ResNet-50 [22] on the ImageNet-1K dataset [49], partly due to its inclusion in several deep learning benchmarks [5, 14, 42]. A sequence of works has steadily reduced the time to train ResNet-50 to convergence [2, 20, 41, 60, 61]. All of these works make use of pure sample parallelism when training. Their speedups come from a combination of framework optimizations, communication algorithm optimizations ([41, 60] use 2-D torus allreduces), special hardware ([60] use TPUv3s), and learning tricks to scale to extremely large batch sizes ([20] introduce linear learning rate scaling; [61] introduce LARS; [2] use RMSprop warmup and other tricks; [41] use batch size control and label smoothing). Many of these learning tricks require very extensive hyperparameter optimization before good results can be achieved. In contrast, we utilize channel/filter parallelism to strong scale convolution while also enjoying improved weak scaling due to reduced communication overheads, without changing the actual convolution operation being performed or the learning of the model. Thus, our work could be used jointly with other learning tricks to further accelerate ResNet-50 training, while also being applicable to other models.

Distributed matrix/matrix products. Our notation and algorithms are inspired by work done on distributed matrix/matrix product algorithms, particularly the SUMMA algorithms [57], stationary Elemental algorithms [51], and their extension to tensor contractions [50]. Cartesian data decompositions onto processor grids also have a long history within linear algebra. However, we wish to emphasize that our algorithms are designed explicitly for convolution, and not for approaches that implement convolution via matrix products (e.g. im2col), although all such algorithms may be used to implement the local convolution operation. Our notation also draws upon that of High Performance Fortran [3].

8 CONCLUSIONS

We have presented a family of algorithms for scaling CNN training by exploiting parallelism within the channel dimension. These further exploit model parallelism by partitioning the parameters of convolutional layers, reducing communication overheads and memory pressure. Our evaluation has demonstrated the promise of such methods, particularly at large scale, where communication overheads are large and it may not be feasible to increase mini-batch sizes further. We have also shown that it is possible to train very wide models, which were previously infeasible due to memory requirements. Essentially, these algorithms enable strong scaling while also reducing communication overheads for weak scaling, without requiring any additional hyperparameter tuning.

Parallelizing by channels is particularly relevant for future CNN architectures. While models have grown deeper over time, they are often refined into *wider* models (e.g. Inception v2 [55], Wide ResNets [62]). Further, as models grow larger (e.g. AmoebaNets [47]), exploiting all available parallelism becomes necessary to keep training times and memory use reasonable, particularly for large datasets or samples from novel applications (e.g. scientific or industrial domains, which may involve multi-spectral data with many channels). Supporting multiple parallelization approaches also provides flexibility (e.g. stationary-y for upsampling branches in semantic segmentation). Leveraging large-scale systems to accelerate training is critical to ensuring productive data science by keeping iteration time for exploring new models or integrating updated data low.

Our results also highlight several research directions to pursue to better exploit parallelism. GPU convolution kernels, which have been highly optimized for particular configurations, do not necessarily perform optimally when the channels or filters of a layer are partitioned. Hand-optimizing kernels for every possible configuration is infeasible [4], so exploring automatic methods (e.g. TVM [10]) may yield the best results. Similarly, segmented collective communication, an important primitive for more complex parameter distributions, does not perform optimally; optimizing communication frameworks to take advantage of these communication patterns is important. As many current and future HPC systems have multiple NICs per node, these communications could take advantage of network and compute node topology-aware algorithms for improved performance.

Now that it is feasible to train very wide CNN models, it is important to conduct experiments to understand the impact of such CNNs on problems of interest. One may also consider designing CNN architectures that are more amenable to distributed training; for example, grouped convolutions could be used to reduce the communication requirements of our algorithms. This co-design of neural network architecture and training can help enable improved models that can be trained faster, yielding new insights.

ACKNOWLEDGMENTS

Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-771796). Funding provided by LDRD #17-SI-003 and the ECP 2.2.6.08 ExaLearn Project. Experiments were performed at the Livermore Computing facility. The authors thank the LBANN team for their assistance.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[2] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. 2017. Extremely large minibatch SGD: Training ResNet-50 on ImageNet in 15 minutes. In *NeurIPS 2017 Workshop: Deep Learning at Supercomputer Scale.*

[3] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. 1997. A linear algebra framework for static High Performance Fortran code distribution. *Scientific Programming* 6, 1 (1997).

[4] Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems.*

[5] Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, and Torsten Hoefler. 2019. A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS).*

[6] Tal Ben-Nun and Torsten Hoefler. 2018. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *arXiv preprint arXiv:1802.09941* (2018).

[7] Peter Buchlovsky, David Budden, Dominik Grewe, Chris Jones, John Aslanides, Frederic Besse, Andy Brock, Aidan Clark, Sergio Gómez Colmenarejo, Aedan Pope, et al. 2019. TF-Replicator: Distributed Machine Learning for Researchers. *arXiv preprint arXiv:1902.00465* (2019).

[8] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition.*

[9] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. 2018. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European Conference on Computer Vision (ECCV).*

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

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[12] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[13] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. 2013. Deep learning with COTS HPC systems. In *International Conference on Machine Learning (ICML).*

[14] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2017. DAWNBench: An end-to-end deep learning benchmark and competition. In *NeurIPS ML Systems Workshop.*

[15] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems (NeurIPS).*

[16] James Demmel and Grace Dinh. 2018. Communication-optimal convolutional neural nets. *arXiv preprint arXiv:1802.06905* (2018).

[17] Nikoli Dryden, Naoya Maruyama, Tom Benson, Tim Moon, Marc Snir, and Brian Van Essen. 2019. Improving Strong-Scaling of CNN Training by Exploiting Finer-Grained Parallelism. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[18] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Andy Yoo, Mark Snir, and Brian Van Essen. 2018. Aluminum: An Asynchronous, GPU-Aware Communication Library Optimized for Large-Scale Training of Deep Neural Networks on HPC Systems. In *Proceedings of the Workshop on Machine Learning in HPC Environments (MLHPC)*.

[19] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. 2018. Integrated model, batch and domain parallelism in training neural networks. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[20] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch SGD: training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[21] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[23] Jie Hu, Li Shen, and Gang Sun. 2018. Squeeze-and-excitation networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[24] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[25] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, and Zhifeng Chen. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *arXiv preprint arXiv:1811.06965* (2018).

[26] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[27] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On large-batch training for deep learning: Generalization gap and sharp minima. In *Proceedings of the Fifth International Conference on Learning Representations (ICLR)*.

[28] Janis Keuper and Franz-Josef Preundt. 2016. Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[30] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. 2018. Exascale Deep Learning for Climate Analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)*.

[31] Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, Md Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, et al. 2017. Deep learning at 15PF: supervised and semi-supervised classification for scientific data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)*.

[32] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[33] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015).

[34] Steven Lee and Nathan Baker. 2018. *Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence.* Technical Report. US Department of Energy Office of Science.

[35] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. 2018. Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[36] LLNL. 2018. Lassen. https://hpc.llnl.gov/hardware/platforms/lassen.

[37] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[38] Linjian Ma, Gabe Montague, Jiayu Ye, Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael W Mahoney. 2019. Inefficiency of K-FAC for Large Batch Size Training. *arXiv preprint arXiv:1903.06237* (2019).

[39] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast training of convolutional networks through FFTs. *arXiv preprint arXiv:1312.5851* (2013).

[40] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Karna, Daina Moise, Simon J Pennycook, et al. 2018. CosmoFlow: Using Deep Learning to Learn the Universe at Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)*.

[41] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U-chupala, Yoshiki Tanaka, and Yuichi Kageyama. 2018. Massively distributed SGD: ImageNet/ResNet-50 Training in a Flash. *arXiv preprint arXiv:1811.05233* (2018).

[42] MLPerf Collaboration. 2019. MLPerf. https://mlperf.org/.

[43] NVIDIA. 2019. NVIDIA Collective Communications Library. https://developer.nvidia.com/nccl.

[44] NVIDIA Research. 2019. CUB. https://nvlabs.github.io/cub/.

[45] Daniel S Park, Jascha Sohl-Dickstein, Quoc V Le, and Samuel L Smith. 2019. The Effect of Network Width on Stochastic Gradient Descent and Generalization: an Empirical Study. *arXiv preprint arXiv:1905.03776* (2019).

[46] Rajat Raina, Anand Madhavan, and Andrew Y Ng. 2009. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*.

[47] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2018. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548* (2018).

[48] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[49] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015).

[50] Martin Daniel Schatz. 2015. *Distributed tensor computations: formalizing distributions, redistributions, and algorithm derivation.* Ph.D. Dissertation. University of Texas at Austin.

[51] Martin D Schatz, Robert A Van de Geijn, and Jack Poulson. 2016. Parallel matrix multiplication: A systematic journey. *SIAM Journal on Scientific Computing* 38, 6 (2016).

[52] Christopher J Shallue, Jaehoon Lee, Joe Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. 2018. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600* (2018).

[53] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-TensorFlow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[54] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, Inception-ResNet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence (AAAI)*.

[55] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[56] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005).

[57] Robert A Van De Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997).

[58] Brian Van Essen, Hyojin Kim, Roger Pearce, Kofi Boakye, and Barry Chen. 2015. LBANN: Livermore big artificial neural network HPC toolkit. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments (MLHPC)*.

[59] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[60] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. 2018. Image Classification at Supercomputer Scale. In *NeurIPS Systems for ML Workshop*.

[61] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. ImageNet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP)*.

[62] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide residual networks. In *Proceedings of the British Machine Vision Conference (BMVC)*.

[63] Amir R Zamir, Alexander Sax, William Shen, Leonidas J Guibas, Jitendra Malik, and Silvio Savarese. 2018. Taskonomy: Disentangling task transfer learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[64] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. 2017. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412* (2017).

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We run four sets of experiments: 1. Convolution with no communication: We run the 'distconv_benchmark' associated with our modified distributed tensor library, using the arguments '–conv-algo AUTOTUNE –skip-allreduce –skip-chanfilt-comm' (plus associated arguments to specify the particular problem being benchmarked). 2. We adapted the Aluminum 0.2.1 'benchmark_allreduces' benchmark to benchmark segmented allreduces. Each configuration was run with the NCCL backend and the environment variables 'AL_PROGRESS_RANKS_PER_NUMA_NODE=2' and 'NCCL_TREE_THRESHOLD=0'. 3. Stationary-x benchmarks: We run the 'distconv_benchmark' associated with our modified distributed tensor library, using the arguments '–conv-algo AUTOTUNE' (and '–chanfilt-algo X' when needed, plus associated arguments to specify the particular problem being benchmarked). We set the environment variables 'AL_PROGRESS_RANKS_PER_NUMA_NODE=2' and 'NCCL_TREE_THRESHOLD=0'. 4. End-to-end training experiments used our modified version of LBANN and specified the appropriate model file and partitioning strategy. Models were generated using LBANN's Python interface for ResNet-50 and Wide ResNets. We set the following environment variables: DISTCONV_WS_CAPACITY_FACTOR = 0.1, LBANN_DISTCONV_CONVOLUTION_FWD_ALGORITHM = AUTOTUNE, LBANN_DISTCONV_CONVOLUTION_BWD_DATA_ ALGORITHM = AUTOTUNE, LBANN_DISTCONV_CONVOLUTION_BWD_FILTER_ALGORITHM = AUTOTUNE, LBANN_DISTCONV_NUM_PRE_GENERATED_SYNTHETIC_DATA = 1, LBANN_DISTCONV_EVALUATE_PERFORMANCE = 1, LBANN_DISTCONV_TENSOR_SHUFFLER = HYBRID, NCCL_TREE_THRESHOLD = 0, AL_PROGRESS_RANKS_PER_NUMA_NODE = 2, OMP_NUM_THREADS = 4.

Exact problem configurations are given within the paper. Every experiment was run on a quiet system in a dedicated allocation of 512 compute nodes. Each run was launched using 'jsrun -n <num nodes> -d packed -b packed:10 -r 1 -c 40 -g 4 -a <ranks per node>'.

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: No author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

N/A

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Lassen, 650 nodes, 2x IBM POWER9 + 4x NVIDIA V100 16 GB/NVLINK2 + 2x IB EDR per node

Operating systems and versions: BlueOS 3.2-24

Compilers and versions: GCC 7.3.1

Applications and versions: LBANN-distconv

Libraries and versions: SpectrumMPI 2019.01.30, CUDA 9.2.148, cuDNN 7.5.0, NCCL 2.4.2, Aluminum 0.2.1, Hydrogen 1.2.0

Key algorithms: Stochastic gradient descent

Input datasets and versions: ILSVRC-2012 (ImageNet 1K)

Paper Modifications: We modified LBANN's distributed tensor library to support tensors with distributed channel or filter dimensions. We then modified its convolution implementation to support the stationary-x, -y, and w algorithms (as described in the paper). We did not need to modify its other layers (batchnorm, ReLU, etc.) as they naturally handled the distributed channel dimension. We implemented optimized CUDA kernels for packing/unpacking non-contiguous reduce-scatters/allgathers on the channel dimension. We added appropriate communicators and hooks for the necessary (segmented) reduce-scatters, allgathers, and allreduces.

We then integrated this updated library into the LBANN-distconv branch. This necessitated modifications to tensor dimension setup and allocation in convolution and batch normalization layers. We also added support for segmented allreduces to its optimizer framework.

Output from scripts that gathers execution environment information.

```
LSB Version:        :core-4.1-noarch:core-4.1-ppc64l
↪ e:cxx-4.1-noarch:cxx-4.1-ppc64le:desktop-4.1-noa
↪ rch:desktop-4.1-ppc64le:languages-4.1-noarch:lan
↪ guages-4.1-ppc64le:printing-4.1-noarch:printing-
↪ 4.1-ppc64le
Distributor ID:        RedHatEnterpriseServer
Description:        Red Hat Enterprise Linux Server
↪ release 7.5 (Maipo)
Release:        7.5
Codename:        Maipo

Linux lassen1 4.14.0-49.18.1.bl6.ppc64le #1 SMP Tue
↪ Dec 11 16:29:11 PST 2018 ppc64le ppc64le ppc64le
↪ GNU/Linux

Architecture:          ppc64le
Byte Order:            Little Endian
CPU(s):                176
On-line CPU(s) list:   0-175
Thread(s) per core:    4
```

```
Core(s) per socket:    22
Socket(s):             2
NUMA node(s):          6
Model:                 2.1 (pvr 004e 1201)
Model name:            POWER9, altivec supported
CPU max MHz:           3800.0000
CPU min MHz:           2300.0000
L1d cache:             32K
L1i cache:             32K
L2 cache:              512K
L3 cache:              10240K
NUMA node0 CPU(s):     0-87
NUMA node8 CPU(s):     88-175
NUMA node252 CPU(s):
NUMA node253 CPU(s):
NUMA node254 CPU(s):
NUMA node255 CPU(s):

MemTotal:       333899392 kB
MemFree:        307218624 kB
MemAvailable:   306914112 kB
Buffers:            27904 kB
Cached:            273856 kB
SwapCached:         75776 kB
Active:            562816 kB
Inactive:          253056 kB
Active(anon):      415616 kB
Inactive(anon):    134976 kB
Active(file):      147200 kB
Inactive(file):    118080 kB
Unevictable:     16777216 kB
Mlocked:         16777216 kB
SwapTotal:       16777152 kB
SwapFree:        16093120 kB
Dirty:               960 kB
Writeback:             0 kB
AnonPages:       17276800 kB
Mapped:            327360 kB
Shmem:              36416 kB
Slab:             3158400 kB
SReclaimable:      591232 kB
SUnreclaim:       2567168 kB
KernelStack:        36272 kB
PageTables:         52608 kB
NFS_Unstable:           0 kB
Bounce:                 0 kB
WritebackTmp:           0 kB
CommitLimit:    183726848 kB
Committed_AS:    18174976 kB
VmallocTotal:   549755813888 kB
VmallocUsed:            0 kB
VmallocChunk:           0 kB
HardwareCorrupted:      0 kB
AnonHugePages:      65536 kB
ShmemHugePages:         0 kB
ShmemPmdMapped:         0 kB
CmaTotal:       13434880 kB
CmaFree:        13433088 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB

NAME        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
nvme0n1     259:0    0  1.5T  0 disk
|-bb-swaplv 253:1    0   16G  0 lvm  [SWAP]
`-bb-rootlv 253:0    0  150G  0 lvm  /

+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.39       Driver Version: 418.39                              |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000004:04:00.0 Off |                    0 |
| N/A   28C    P0    36W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  On   | 00000004:05:00.0 Off |                    0 |
| N/A   27C    P0    34W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  On   | 00000035:03:00.0 Off |                    0 |
| N/A   27C    P0    35W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  On   | 00000035:04:00.0 Off |                    0 |
| N/A   24C    P0    34W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

H/W path      Device      Class      Description
=============================================
                          system     8335-GTW
                                     (ibm,witherspoon)
/0                        bus        Motherboard
/0/16                     processor  02AA966
```

```
/0/16/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/16/1                 memory     32KiB L1 Cache
↪ (data)
/0/16/2                 memory     512KiB L2 Cache
↪ (unified)
/0/16/3                 memory     10MiB L3 Cache
↪ (unified)
/0/20                   processor  02AA966
/0/20/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/20/1                 memory     32KiB L1 Cache
↪ (data)
/0/20/2                 memory     512KiB L2 Cache
↪ (unified)
/0/20/3                 memory     10MiB L3 Cache
↪ (unified)
/0/24                   processor  02AA966
/0/24/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/24/1                 memory     32KiB L1 Cache
↪ (data)
/0/24/2                 memory     512KiB L2 Cache
↪ (unified)
/0/24/3                 memory     10MiB L3 Cache
↪ (unified)
/0/28                   processor  02AA966
/0/28/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/28/1                 memory     32KiB L1 Cache
↪ (data)
/0/28/2                 memory     512KiB L2 Cache
↪ (unified)
/0/28/3                 memory     10MiB L3 Cache
↪ (unified)
/0/32                   processor  02AA966
/0/32/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/32/1                 memory     32KiB L1 Cache
↪ (data)
/0/32/2                 memory     512KiB L2 Cache
↪ (unified)
/0/32/3                 memory     10MiB L3 Cache
↪ (unified)
/0/36                   processor  02AA966
/0/36/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/36/1                 memory     32KiB L1 Cache
↪ (data)
/0/36/2                 memory     512KiB L2 Cache
↪ (unified)
/0/36/3                 memory     10MiB L3 Cache
↪ (unified)
/0/40                   processor  02AA966
/0/40/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/40/1                 memory     32KiB L1 Cache
↪ (data)
/0/40/2                 memory     512KiB L2 Cache
↪ (unified)
/0/40/3                 memory     10MiB L3 Cache
↪ (unified)
/0/44                   processor  02AA966
/0/44/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/44/1                 memory     32KiB L1 Cache
↪ (data)
/0/44/2                 memory     512KiB L2 Cache
↪ (unified)
/0/44/3                 memory     10MiB L3 Cache
↪ (unified)
/0/48                   processor  02AA966
/0/48/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/48/1                 memory     32KiB L1 Cache
↪ (data)
/0/48/2                 memory     512KiB L2 Cache
↪ (unified)
/0/48/3                 memory     10MiB L3 Cache
↪ (unified)
/0/52                   processor  02AA966
/0/52/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/52/1                 memory     32KiB L1 Cache
↪ (data)
/0/52/2                 memory     512KiB L2 Cache
↪ (unified)
/0/52/3                 memory     10MiB L3 Cache
↪ (unified)
/0/56                   processor  02AA966
/0/56/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/56/1                 memory     32KiB L1 Cache
↪ (data)
/0/56/2                 memory     512KiB L2 Cache
↪ (unified)
/0/56/3                 memory     10MiB L3 Cache
↪ (unified)
/0/60                   processor  02AA966
/0/60/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/60/1                 memory     32KiB L1 Cache
↪ (data)
/0/60/2                 memory     512KiB L2 Cache
↪ (unified)
/0/60/3                 memory     10MiB L3 Cache
↪ (unified)
/0/64                   processor  02AA966
```

```
/0/64/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/64/1                 memory     32KiB L1 Cache
↪ (data)
/0/64/2                 memory     512KiB L2 Cache
↪ (unified)
/0/64/3                 memory     10MiB L3 Cache
↪ (unified)
/0/68                   processor  02AA966
/0/68/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/68/1                 memory     32KiB L1 Cache
↪ (data)
/0/68/2                 memory     512KiB L2 Cache
↪ (unified)
/0/68/3                 memory     10MiB L3 Cache
↪ (unified)
/0/72                   processor  02AA966
/0/72/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/72/1                 memory     32KiB L1 Cache
↪ (data)
/0/72/2                 memory     512KiB L2 Cache
↪ (unified)
/0/72/3                 memory     10MiB L3 Cache
↪ (unified)
/0/76                   processor  02AA966
/0/76/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/76/1                 memory     32KiB L1 Cache
↪ (data)
/0/76/2                 memory     512KiB L2 Cache
↪ (unified)
/0/76/3                 memory     10MiB L3 Cache
↪ (unified)
/0/80                   processor  02AA966
/0/80/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/80/1                 memory     32KiB L1 Cache
↪ (data)
/0/80/2                 memory     512KiB L2 Cache
↪ (unified)
/0/80/3                 memory     10MiB L3 Cache
↪ (unified)
/0/84                   processor  02AA966
/0/84/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/84/1                 memory     32KiB L1 Cache
↪ (data)
/0/84/2                 memory     512KiB L2 Cache
↪ (unified)
/0/84/3                 memory     10MiB L3 Cache
↪ (unified)
/0/88                   processor  02AA966
/0/88/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/88/1                 memory     32KiB L1 Cache
↪ (data)
/0/88/2                 memory     512KiB L2 Cache
↪ (unified)
/0/88/3                 memory     10MiB L3 Cache
↪ (unified)
/0/92                   processor  02AA966
/0/92/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/92/1                 memory     32KiB L1 Cache
↪ (data)
/0/92/2                 memory     512KiB L2 Cache
↪ (unified)
/0/92/3                 memory     10MiB L3 Cache
↪ (unified)
/0/8                    processor  02AA966
/0/8/0                  memory     32KiB L1 Cache
↪ (instruction)
/0/8/1                  memory     32KiB L1 Cache
↪ (data)
/0/8/2                  memory     512KiB L2 Cache
↪ (unified)
/0/8/3                  memory     10MiB L3 Cache
↪ (unified)
/0/2048                 processor  02AA966
/0/2048/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2048/1               memory     32KiB L1 Cache
↪ (data)
/0/2048/2               memory     512KiB L2 Cache
↪ (unified)
/0/2048/3               memory     10MiB L3 Cache
↪ (unified)
/0/2052                 processor  02AA966
/0/2052/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2052/1               memory     32KiB L1 Cache
↪ (data)
/0/2052/2               memory     512KiB L2 Cache
↪ (unified)
/0/2052/3               memory     10MiB L3 Cache
↪ (unified)
/0/2056                 processor  02AA966
/0/2056/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2056/1               memory     32KiB L1 Cache
↪ (data)
/0/2056/2               memory     512KiB L2 Cache
↪ (unified)
/0/2056/3               memory     10MiB L3 Cache
↪ (unified)
/0/2060                 processor  02AA966
```

```
/0/2060/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2060/1               memory     32KiB L1 Cache
↪ (data)
/0/2060/2               memory     512KiB L2 Cache
↪ (unified)
/0/2060/3               memory     10MiB L3 Cache
↪ (unified)
/0/2064                 processor  02AA966
/0/2064/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2064/1               memory     32KiB L1 Cache
↪ (data)
/0/2064/2               memory     512KiB L2 Cache
↪ (unified)
/0/2064/3               memory     10MiB L3 Cache
↪ (unified)
/0/2068                 processor  02AA966
/0/2068/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2068/1               memory     32KiB L1 Cache
↪ (data)
/0/2068/2               memory     512KiB L2 Cache
↪ (unified)
/0/2068/3               memory     10MiB L3 Cache
↪ (unified)
/0/2072                 processor  02AA966
/0/2072/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2072/1               memory     32KiB L1 Cache
↪ (data)
/0/2072/2               memory     512KiB L2 Cache
↪ (unified)
/0/2072/3               memory     10MiB L3 Cache
↪ (unified)
/0/2076                 processor  02AA966
/0/2076/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2076/1               memory     32KiB L1 Cache
↪ (data)
/0/2076/2               memory     512KiB L2 Cache
↪ (unified)
/0/2076/3               memory     10MiB L3 Cache
↪ (unified)
/0/2080                 processor  02AA966
/0/2080/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2080/1               memory     32KiB L1 Cache
↪ (data)
/0/2080/2               memory     512KiB L2 Cache
↪ (unified)
/0/2080/3               memory     10MiB L3 Cache
↪ (unified)
/0/2084                 processor  02AA966
/0/2084/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2084/1               memory     32KiB L1 Cache
↪ (data)
/0/2084/2               memory     512KiB L2 Cache
↪ (unified)
/0/2084/3               memory     10MiB L3 Cache
↪ (unified)
/0/2088                 processor  02AA966
/0/2088/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2088/1               memory     32KiB L1 Cache
↪ (data)
/0/2088/2               memory     512KiB L2 Cache
↪ (unified)
/0/2088/3               memory     10MiB L3 Cache
↪ (unified)
/0/2092                 processor  02AA966
/0/2092/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2092/1               memory     32KiB L1 Cache
↪ (data)
/0/2092/2               memory     512KiB L2 Cache
↪ (unified)
/0/2092/3               memory     10MiB L3 Cache
↪ (unified)
/0/2096                 processor  02AA966
/0/2096/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2096/1               memory     32KiB L1 Cache
↪ (data)
/0/2096/2               memory     512KiB L2 Cache
↪ (unified)
/0/2096/3               memory     10MiB L3 Cache
↪ (unified)
/0/2100                 processor  02AA966
/0/2100/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2100/1               memory     32KiB L1 Cache
↪ (data)
/0/2100/2               memory     512KiB L2 Cache
↪ (unified)
/0/2100/3               memory     10MiB L3 Cache
↪ (unified)
/0/2104                 processor  02AA966
/0/2104/0               memory     32KiB L1 Cache
↪ (instruction)
/0/2104/1               memory     32KiB L1 Cache
↪ (data)
/0/2104/2               memory     512KiB L2 Cache
↪ (unified)
/0/2104/3               memory     10MiB L3 Cache
↪ (unified)
/0/2108                 processor  02AA966
```

```
/0/2108/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/2108/1                 memory     32KiB L1 Cache
↪ (data)
/0/2108/2                 memory     512KiB L2 Cache
↪ (unified)
/0/2108/3                 memory     10MiB L3 Cache
↪ (unified)
/0/2112                   processor  02AA966
/0/2112/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/2112/1                 memory     32KiB L1 Cache
↪ (data)
/0/2112/2                 memory     512KiB L2 Cache
↪ (unified)
/0/2112/3                 memory     10MiB L3 Cache
↪ (unified)
/0/2116                   processor  02AA966
/0/2116/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/2116/1                 memory     32KiB L1 Cache
↪ (data)
/0/2116/2                 memory     512KiB L2 Cache
↪ (unified)
/0/2116/3                 memory     10MiB L3 Cache
↪ (unified)
/0/2128                   processor  02AA966
/0/2128/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/2128/1                 memory     32KiB L1 Cache
↪ (data)
/0/2128/2                 memory     512KiB L2 Cache
↪ (unified)
/0/2128/3                 memory     10MiB L3 Cache
↪ (unified)
/0/2132                   processor  02AA966
/0/2132/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/2132/1                 memory     32KiB L1 Cache
↪ (data)
/0/2132/2                 memory     512KiB L2 Cache
↪ (unified)
/0/2132/3                 memory     10MiB L3 Cache
↪ (unified)
/0/2136                   processor  02AA966
/0/2136/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/2136/1                 memory     32KiB L1 Cache
↪ (data)
/0/2136/2                 memory     512KiB L2 Cache
↪ (unified)
/0/2136/3                 memory     10MiB L3 Cache
↪ (unified)
/0/2140                   processor  02AA966
/0/2140/0                 memory     32KiB L1 Cache
↪ (instruction)
/0/2140/1                 memory     32KiB L1 Cache
↪ (data)
/0/2140/2                 memory     512KiB L2 Cache
↪ (unified)
/0/2140/3                 memory     10MiB L3 Cache
↪ (unified)
/0/12                     processor  02AA966
/0/12/0                   memory     32KiB L1 Cache
↪ (instruction)
/0/12/1                   memory     32KiB L1 Cache
↪ (data)
/0/12/2                   memory     512KiB L2 Cache
↪ (unified)
/0/12/3                   memory     10MiB L3 Cache
↪ (unified)
/0/0                    memory     319GiB System memory
/0/0/0                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/1                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/2                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/3                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/4                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/5                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/6                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/7                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/8                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/9                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/a                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/b                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/c                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/d                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/e                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/0/f                    memory     16GiB RDIMM DDR4
↪ 2666 MHz (0.4ns)
/0/1                      generic
↪ bmc-firmware-version
/0/2                      generic    buildroot
/0/3                      generic    capp-ucode
```

```
/0/4                    generic    hostboot
/0/5                    generic    hostboot-binaries
/0/6                    generic    linux
/0/7                    generic    machine-xml
/0/9                    generic    occ
/0/a                    generic    petitboot
/0/b                    generic    sbe
/0/c                    generic    skiboot
/0/d                    generic    version
/0/100                  bridge     IBM
/0/100/0                storage    NVMe SSD
↪  Controller 172Xa
/0/101                  bridge     IBM
/0/101/0                bus        TUSB73x0
↪  SuperSpeed USB 3.0 xHCI Host Controller
/0/102                  bridge     IBM
/0/102/0                bridge     AST1150
↪  PCI-to-PCI Bridge
/0/102/0/0              display    ASPEED Graphics
↪  Family
/0/103                  bridge     IBM
/0/103/0      hsi0      network    MT28800 Family
↪  [ConnectX-5 Ex]
/0/103/0.1    hsi1      network    MT28800 Family
↪  [ConnectX-5 Ex]
/0/104                  bridge     IBM
/0/104/0                bridge     PLX Technology,
↪  Inc.
/0/104/0/2              bridge     PLX Technology,
↪  Inc.
/0/104/0/2/0            storage    88SE9235 PCIe 2.0
↪  x2 4-port SATA 6 Gb/s Controller
/0/104/0/a              bridge     PLX Technology,
↪  Inc.
/0/104/0/a/0            display    GV100GL [Tesla
↪  V100 SXM2]
/0/104/0/b              bridge     PLX Technology,
↪  Inc.
/0/104/0/b/0            display    GV100GL [Tesla
↪  V100 SXM2]
/0/104/0/c              bridge     PLX Technology,
↪  Inc.
/0/104/0.1              generic    PLX Technology,
↪  Inc.
/0/104/0.2              generic    PLX Technology,
↪  Inc.
/0/104/0.3              generic    PLX Technology,
↪  Inc.
/0/104/0.4              generic    PLX Technology,
↪  Inc.
/0/105                  bridge     IBM
/0/105/0      enP5p1s0f0  network    NetXtreme BCM5719
↪  Gigabit Ethernet PCIe
/0/105/0.1    enP5p1s0f1  network    NetXtreme BCM5719
↪  Gigabit Ethernet PCIe
/0/106                  bridge     IBM
/0/107                  bridge     IBM
/0/108                  bridge     IBM
/0/109                  bridge     IBM
/0/10a                  bridge     IBM
/0/10b                  bridge     IBM
/0/10c                  bridge     IBM
/0/10d                  bridge     IBM
/0/10e                  bridge     IBM
/0/10f                  bridge     IBM
/0/110                  bridge     IBM
/0/111                  bridge     IBM
/0/112                  bridge     IBM
/0/113                  bridge     IBM
/0/113/0      hsi2      network    MT28800 Family
↪  [ConnectX-5 Ex]
/0/113/0.1    hsi3      network    MT28800 Family
↪  [ConnectX-5 Ex]
/0/114                  bridge     IBM
/0/115                  bridge     IBM
/0/115/0                bridge     PLX Technology,
↪  Inc.
/0/115/0/4              bridge     PLX Technology,
↪  Inc.
/0/115/0/4/0            display    GV100GL [Tesla
↪  V100 SXM2]
/0/115/0/5              bridge     PLX Technology,
↪  Inc.
/0/115/0/5/0            display    GV100GL [Tesla
↪  V100 SXM2]
/0/115/0/d              bridge     PLX Technology,
↪  Inc.
WARNING: output may be incomplete or inaccurate, you
↪  should run this program as super-user.
```

ARTIFACT EVALUATION

Verification and validation studies: Correctness of algorithms was ensured by validating that the results produced by the stationary-x algorithm matched the results of convolution performed on a single GPU within a fixed threshold (0.00001) for every layer configuration used within the networks considered, and a broader set of test cases.

Accuracy and precision of timings: Benchmarks were run several times, averaged, and checked for outliers.

Used manufactured solutions or spectral properties: None.

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: None.

Controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system. None.