ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms

Saeed Rashidi*, Srinivas Sridharan†, Sudarshan Srinivasan‡ and Tushar Krishna*
*Georgia Institute of Technology, Atlanta, USA
†Facebook, Menlo Park, USA
‡Intel, Bangalore, India
saeed.rashidi@gatech.edu, ssrinivas@fb.com, sudarshan.srinivasan@intel.com, tushar@ece.gatech.edu

***Abstract*—Modern Deep Learning systems heavily rely on distributed training over high-performance accelerator (e.g., TPU, GPU)-based hardware platforms. Examples today include Google's Cloud TPU and Facebook's Zion. DNN training involves a complex interplay between the DNN model architecture, parallelization strategy, scheduling strategy, collective communication algorithm, network topology, and the end-point accelerator. As innovation in AI/ML models continues to grow at an accelerated rate, there is a need for a comprehensive methodology to understand and navigate this complex SW/HW design-space for future systems to support efficient training of future DNN models. In this work, we make the following contributions (i) establish the SW/HW design-space for Distributed Training over a hierarchical scale-up fabric, (ii) develop a network simulator for navigating the design-space, and (iii) demonstrate the promise of algorithm-topology co-design for speeding up end to end training.**

***Index Terms*—Distributed training; Collective communication; Training parallelism; High performance training systems;**

I. INTRODUCTION

Deep Learning (DL) and Deep Neural networks (DNN) are driving the proliferation of Artificial Intelligence (AI) in a wide range of application domains such as image classification, natural language processing, and autonomous driving. As the popularity and the use cases expand, AI researchers are seeking to improve the capabilities and accuracy of DNNs by designing deeper networks and training them using millions, if not billions, of samples. However, these improvements come at the expense of increased training time and/or memory capacity, and hence drive the demand for scalable high-performance training platforms.

DL Training platforms today are built by interconnecting multiple accelerators together. Examples include Google's TPU that uses many TPUs interconnected in a 3D Torus [21], Facebook's Zion system [2] using CPUs and GPUs connected via alltoall topologies, and NVIDIA DGX systems [24] that use NVswitch to enable switch-based topologies. While numerous studies have benchmarked DL training on some of these training platforms [10], [14], [18], [25], [30], *there is limited or no work on broad design space exploration targeting future platforms*. Future platforms are expected to leverage emerging accelerators (say a next-generation GPU or TPU) that we call a Neural Processing Unit (NPU) for generality, connected via a hierarchical network fabric, going all the way from on-package

http://github.com/astra-sim/astra-sim

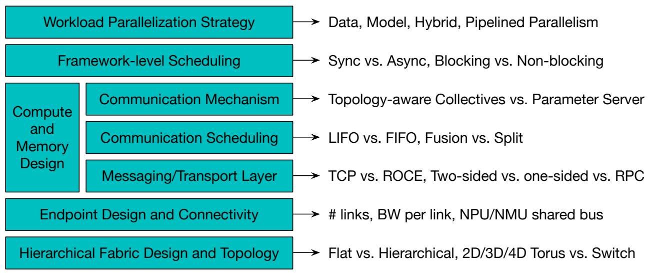

Workload Parallelization Strategy
Data, Model, Hybrid, Pipelined Parallelism
Framework-level Scheduling
Sync vs. Async, Blocking vs. Non-blocking
Compute and Memory Design
Communication Mechanism
Topology-aware Collectives vs. Parameter Server
Communication Scheduling
LIFO vs. FIFO, Fusion vs. Split
Messaging/Transport Layer
TCP vs. ROCE, Two-sided vs. one-sided vs. RPC
Endpoint Design and Connectivity
# links, BW per link, NPU/NMU shared bus
Hierarchical Fabric Design and Topology
Flat vs. Hierarchical, 2D/3D/4D Torus vs. Switch

Fig. 1: DL training SW/HW design space

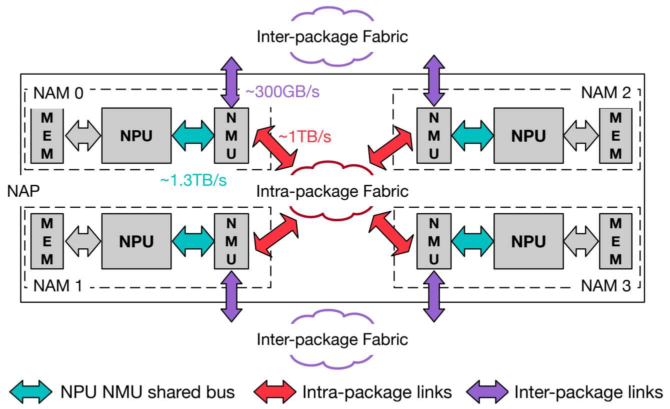
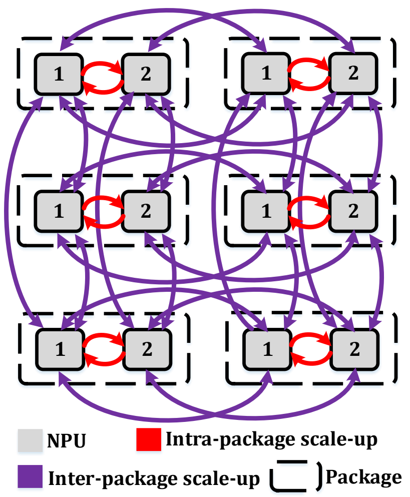


Fig. 2: Hierarchical accelerator system

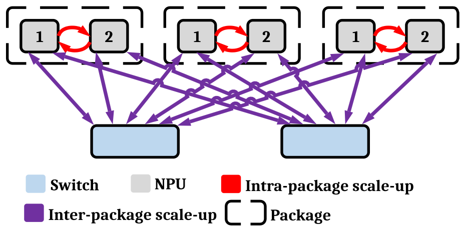
(MCM [3] or interposer) to on-rack (PCIe/NVlink) to across racks (ethernet/infiniband). Our work is focused on enabling researchers to quantify the research challenges involved in designing a highly scalable hierarchical accelerator systems for efficiently training future DNN models, and design efficient SW/HW co-design solutions.

Figure 1 presents the SW/HW design space for distributed DL training on future hierarchical DL accelerator systems and is inspired by today's DL training platforms. In this work, we focus on what we define as the "scale-up" network - i.e., the network connecting multiple accelerators hosted within a board or rack [2], [24].

We approach this in two steps. First we establish a comprehensive SW/HW design space for future accelerator systems, shown in Figure 1, and present a holistic evaluation methodology for enabling fast yet flexible and accurate exploration. Our infrastructure allows us to comprehend various requirements spanning DNN model (e.g. number of layers,



(a) Hierarchical 2D torus



(b) Alltoall

Fig. 3: Scale-Up topologies

size of GEMMs, etc.), DNN parallelism (e.g. data, model, hybrid) and corresponding communication patterns, framework-level optimizations (e.g. fusion vs. splitting messages, overlap vs. no overlap, etc.), design/mapping hierarchical collective algorithms, endpoint design (e.g. on-load vs. offload), fabric design (e.g. number of links per tile, latency/BW per link, etc.), fabric topology (e.g., pt-to-pt such as 2D/3D Torus vs. switch-based), and finally the interactions between these individual components.

Second, we present an end-to-end simulation methodology called ASTRA-SIM (Accelerator Scaling for TRAining Simulator), codifying the design-space described around a network simulator (Garnet [1], [19]). We allow parameterized descriptions of the DNN, system, and fabric, and enable end-to-end simulation of a DNN training loop. To demonstrate the power of our tool, we run case studies studying the effect of network topologies on the performance of collective communication algorithms, and also study the detailed compute-communication breakdown for a distributed training of ResNet-50 [16].

In particular, we make the following novel contributions:

- Establish the SW/HW design space for hierarchical accelerator fabric design space exploration and identify key bottlenecks in scaling DL workloads.
- Develop an end-to-end network simulator, called ASTRA-SIM , for evaluating various aspects of the design space
- Present comprehensive analysis of 1D, 2D, and 3D topologies for all-reduce and all-to-all collectives for alltoall and Torus topologies using ASTRA-SIM .

The rest of the paper is organized as follows: Section II provides some background, Section III establishes the broad design space for DL training platform exploration. We present our simulator in Section IV. Section V describes the analytical and simulation results. We conclude the paper in Section VII.

II. Background - DL Training

The process of supervised training is adjusting the weights of a predictor $\hat{y} = F(x, w)$ (with the output $\hat{y}$,weight w, and input x) with respect the data-set of samples $D = y^*, x$ (where y^* is the ground truth corresponding to each input data x) in a way that minimizes the difference between the $\hat{y}$ and the ground truth y^* for each sample x [29]. This is usually done by forming a loss function over D, $L_D(y^*, F(x, y))$ that is differentiating between the ground truth and predictor output, and then trying to minimize the loss function by the iterative gradient-descent method. Mathematically, this means finding the gradients of the loss function with respect to the weights $\frac{\partial L_D}{\partial w}$ (called weight gradients) and then, updating the weights using gradient-descent on each iteration.

DNNs are special kinds of predictors that consists of many layers such as convolutional, fully connected, sub-sampling, and so on. The DNN training task consists of a layer-wise procedure with three different phases: (i) forward-pass, (ii) weight gradient computation, and (iii) input (error) gradient computation. The process begins with the forward pass for each sample data, the output activations of each layer is computed. This is followed by the back-propagation process that starts from the last layer and goes backward to compute the weight gradient and input gradient of each layer for all samples in B. Finally, the computed weight gradients update the existing weight values using gradient descent.

Distributed Training. When it comes to parallelizing the training task across multiple nodes (e.g. NPUs), two main questions arise: (i) how to synchronize the weight updates? (ii) how to distribute the parameters (e.g. training data, model parameters) across different nodes? The most common approach used to address the first question is called synchronous training, where each node works on its own data and produces its local gradients, which are then accumulated/reduced across all or a certain number of nodes to update the weights before the next iteration can start. The answer to the second question depends on the parallelization strategy employed, as discussed later in Section III-A.

III. SW/HW Co-design and Scaling Challenges

In this paper, we consider a future DL training platform comprising of multiple *Neural Accelerator Packages* (NAP) (e.g. NVIDIA V100 GPU, Google TPU [21], etc.) interconnected via a dedicated *scale-up* fabric. Figure 2 presents a high-level system design of a NAP with four *Neural Accelerator Modules* (NAM) integrated through multi-chip packaging technology

TABLE I: Different parallelism approach communications

Parallelism	Activations during the forward pass	Weight gradients	Input gradients
Data		✓	
Model	✓		✓
Hybrid	partially	partially	partially

[28][1]. Each NAM consists of a massively parallel compute engine; henceforth called *Neural Processing Unit* (NPU); high-bandwidth memory (e.g. HBM), and a *Neural Messaging Unit* (NMU). NMUs play the role of a traditional Network Interface Card (NIC) but simpler and purpose-built for accelerator fabrics. The hierarchical scale-up fabric comprises of very high bandwidth *intra-package NAM links* for module-to-module communication within package (∼500GB/s [28]) and longer distance *inter-package NAP links* (∼25 GB/s per link [6]) for enabling communication spanning multiple chassis or racks. Figure 3 shows example scale-up topologies.

As shown in Figure 1, distributed DNN training involves a complex inter-dependent SW/HW design-space. To best of our knowledge this is the first effort to systematically explore this vast design space for end-to-end-training for future accelerator system designs.

A. Parallelization Strategies

The common parallelization techniques for partitioning work across multiple nodes, are data parallelism (replicating the entire model, model parallelism (splitting the model), pipelined parallelism, or some combination of these. In data-parallel, each node is assigned a subset of samples and during each iteration, works on its minibatch (chosen from its own dataset) to produce the local gradients. In model parallel, the nodes have the same data-sets and work on the same minibatch, but since the model is divided, each model is responsible for a portion of model gradients. In, hybrid-parallel, nodes are divided into different groups and the training within the group is data-parallel/model-parallel while between groups is model-parallel/data-parallel.

The different parallelism approaches have different indications in terms of communication patterns between the nodes. Table I shows when data should be exchanged for different parallelism approaches. In data-parallel approach, weight gradients should be exchanged among all nodes since each node calculates the gradients over a subset of the inputs and hence, the gradients of all nodes should be accumulated to generate the updated weights for the next step. In model-parallel scheme, each node produces a part of the output activations and input gradients during the forward pass and back-propagation, respectively. Hence these values must be communicated across all nodes. The hybrid parallel is in between the data-parallel and model-parallel and its communication behavior is in the middle as well. Consequently, the nodes within a data-parallel/model-parallel group in the hybrid-parallel have the same communication pattern as the data-parallel/model-parallel schemes.

[1]Our classification is not limited to accelerators and can be extended to CPUs as well - a CPU socket is a NAM and a multi-socket system is a NAP.

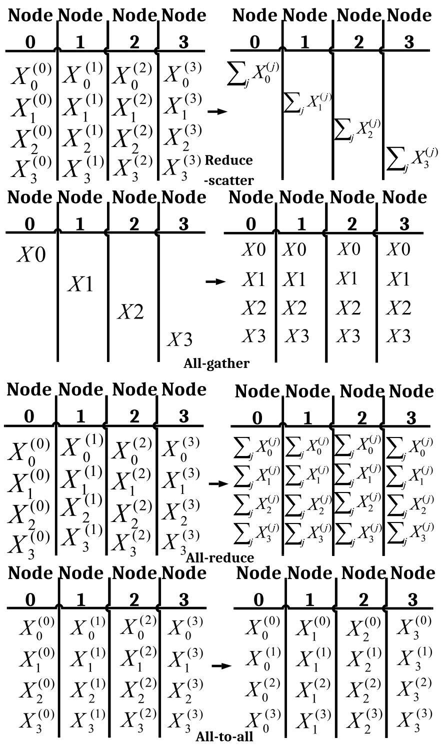


Fig. 4: Overview of collective communication operation used in DNN training networks.

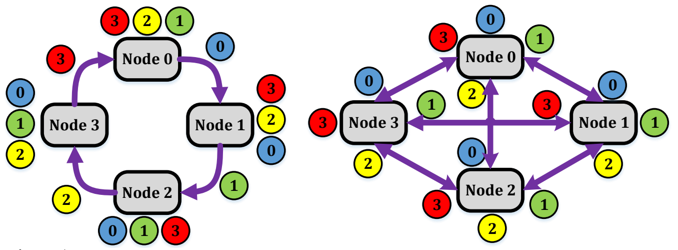


Fig. 5: The first step of reduce-scatter in a ring topology (left) and alltoall topology (right) with 4 nodes

B. Collective Communication Mechanisms

As Table I indicates, different communications are initiated at different phases for different parallelism approaches. However, all of these communications are handled by using some set of collective communication operations described in Figure 4. Collective communications refer to a set of operations in which multiple nodes participate in data exchange to perform a certain operation over the data. In general, four different collective communication algorithms are the main contributor in DNN training communication: (i) reduce-scatter, (ii) all-gather, (iii) all-reduce, and (iv) all-to-all. Figure 4 shows the initial state and the final state for an example of four nodes participating in the collective communication. Among these operations, all-reduce has the most frequent usage and can be done using a reduce-scatter followed by an all-gather. The usage of all-to-all is specific to some certain DNNs that has distributed key/value table across the nodes.

The efficient way of performing the collective communication is topology dependent and in the recent years, many proposals suggested specialized collective communication algorithms for specific topologies [11], [22], [23]. Here we briefly describe how collective algorithms work on ring and alltoall topologies since they act as the the building blocks for the modern topologies observed in the training configurations as well as our proposed hierarchical topology.

Figure 5 shows the first step in reduce scatter for both the ring and alltoall topologies. In both cases with N nodes (here N=4), the data is partitioned into 4 messages. In the case of ring, during the first step, the node i sends data i to the next node and receives data $(i-1)\, mod\, N$ from its previous node. After receiving, it adds the received data with its corresponding local data and again sends it out to the next node. This process takes $N-1$ steps and after that, each node has one data that is globally reduced. The all-gather process, like reduce-scatter, takes $N-1$ steps but it does not contain any local-reduction and each node simply relays its received data to the next node. The all-to-all collective on the unidirectional ring is composed of (N-1) steps. In each step i, the NPU sends a message to its neighbour with the distance of i and receives a message from its neighbour with the distance of i. Each message that the node sends to a destination, contains the data that belongs to that destination node in the all-to-all collective.

In the case of alltoall topology, each node, say node i, initiate the process by sending its data j to node j and receiving data i from all other nodes at the same time. As an example, in Figure 5, node 0 sends its data 1,2, and 3, to nodes 1,2, and 3 respectively and receives nodes 0 from all other nodes. Then, each node reduces the received data with its local data and produces one data that is globally reduced. The all-gather is done simply by each node broadcasting its data to all other nodes. All-to-all is the same as reduce-scatter without any local reduction. In both topologies, the all-reduce, as mentioned before, is a reduce-scatter followed by an all-gather.

C. Hierarchical Fabric Design and Topology

By leveraging the primitive topologies discussed in earlier sections, it is possible to create more complex hierarchical topologies discussed in the previous sections. Given the large design space, we limit the fabric topology in this study to 3D torus (inspired by Google's TPU platform [21]) and all-to-all (inspired by Facebook's Zion platform [2]). Expanding this study to other scale-up topologies such as 4D/5D torus, switch-based, etc., should be straight-forward and will be explored as part of future work. Figure 3 shows how we can extend the ring topologies to create a hierarchical Torus and all-to-all topologies. Figure 3a shows a hierarchical torus topology with three dimensions: local, vertical , and horizontal. Each dimension simply contains one or many rings. The local dimension is made of fast and high bandwidth intra-package links that create one or more unidirectional rings. The horizontal and vertical dimensions are made of inter-package links, creating one or more bidirectional rings that connect NPUs, with the same number in the package, across different packages. Each bidirectional ring is divided into two unidirectional rings. In the Figure 3a, the local dimension size=2, vertical dimension size=3, and the horizontal dimension size =2. In our terminology, the hierarchical torus is described as the M×N×K where M is the local dimension, N is the horizontal dimension, and K is the vertical dimension. Hence, the size of the topology in Figure 3a is 2×2×3.

In Figure 3b, a traditional alltoall topology is extended by adding a local dimension consisting of the high bandwidth rings. In addition, switches enable the alltoall connectivity between the NPUs across different packages. In this configuration, there might be one or more global switches (two in Figure 3b) and each NPU is connected to all of the global switches using inter-package links. In the hierarchical alltoall topology, a system size is described by M×N where M is the local size and N is the alltoall size (i.e. the number of packages). In Figure 3b, the size of the network is 2×3.

D. Multi-phase Collectives

Due to the hierarchical nature of the topologies described so far, the collective algorithms also need to be updated to work efficiently on the hierarchical networks. This is typically done by making the collective algorithms be multi-phase and each phase working on a specific dimension. This provides a convenient way to pipeline the collectives and distribute the different dimensions. In the case of all-reduce on the hierarchical torus, a baseline approach is to perform the all-reduce on the local dimension first, followed by all-reduce in the vertical and then horizontal dimensions, respectively. It is possible to leverage the asymmetric bandwidth of the intra-package and inter-package links and enhance the baseline algorithm by sending less traffic to the inter-package links that have less bandwidth. It is done by first performing the reduce-scatter on the local dimension, followed by the all-reduce on the vertical and horizontal dimension and finally performing the all-gather on the local dimension. The local reduce-scatter distributes the job of all-reduce across different NPUs within the same package and in the inter-package phases, the NPUs with the same numbers work on the specific portion of the data. The final all-gather phase then distributes the data among all NPUs. The same procedure is applicable to enhance the all-reduce on the alltoall topology. It is performed by reduce-scatter on the local dimension, followed by all-reduce on the alltoall dimension (NPUs with the same number in Figure 3b work together), and finally all-gather on the local dimension.

The all-to-all collective can be adopted to have multiple phases. For example, the hierarchical all-to-all on the hierarchical torus in Figure 3a has three phases starting from all-to-all on the local dimension followed by all-to-all on the vertical and horizontal dimensions. The all-to-all in each phase is a multi-step operation as described earlier. However, in this case, in each step, in addition to the data that belongs to the destination node, the NPU also send all data that could be routed to their final destination through that destination node during the remaining phases. For example in phase 1 of the all-to-all in Figure 3a, the NPU #1 send all data with destination #2 to its local NPU #2, since those packets are able to reach their destination through NPU #2 using the remaining vertical and horizontal phases. This process continues for latter as well. For the hierarchical alltoall topology, the all-to-all collective has two phases: (i) all-to-all on the local dimension, and (ii) all-to-all on the alltoall network. In phase 1, the NPUs use the local dimension and perform ring all-to-all operation. In each step of this phase, like torus, the message they send to their destination contains all data that could be routed to their destination through that NPU using the remaining all-to-all phase on the alltoall dimension.

E. Communication Scheduling

Unlike model parallelism, in data parallelism there is significant opportunity to overlap communication with compute. Each node computes partial weight gradients for its mini-batch in the back-propagation step in each layer and aggregates these partial gradients across all nodes using an *allreduce* operation. These aggregated weight gradients are used to update the weights and only required right before the forward propagation step for that layer in the next iteration. This is captured in the compute to communication ratio and relies on networking library/HW's ability to *asynchronously progress* communication and framework's ability to schedule communication to maximize compute-communication overlap. While overlapping communication with computation across layers is indispensable, the overheads of the first layer's weight gradient communication in data parallelism is fully exposed given lack of useful compute to overlap communication. In other words, while network bandwidth is critical for all other layers, optimizing for network latency is essential for the first layer since size of the weight gradients are typically small(er). This motivates the need for further prioritizing and completing the first layers communication operations before communication operations from later layers even though they were issued earlier. Similarly, in the case of model/hybrid parallelism, activation communication must be prioritized as they may block the next layer's compute.

In summary, in this section we characterized the design-space of distributed DL training over a scale-up fabric comprising of intra-package and inter-package links.

TABLE II: Data granularity at different levels of ASTRA-SIM execution

Granularity	Size	Constraint
Set	Training Algorithm	Training Algorithm
Chunk	Parameter for Pipelining	Storage Element Size (Area/Power)
Message	Proportional to the - Number of Nodes	Topology
Packet	Link Technology	Technology
Flit	Network Buffer Size	Microarchitecture (Area/Power)
Phit	Link Width	Technology

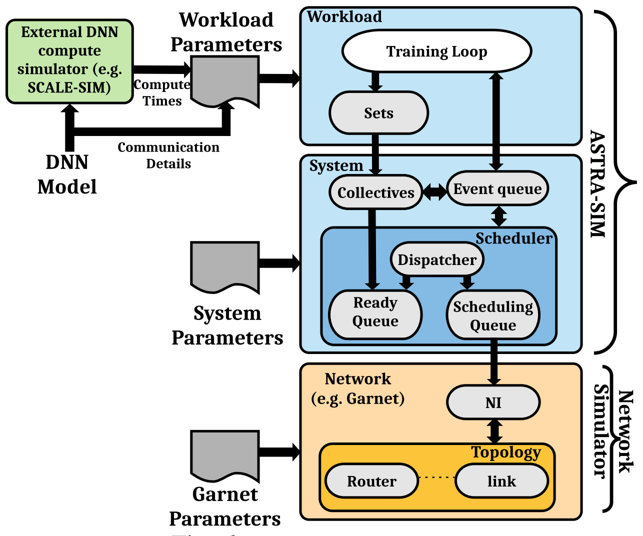


Fig. 6: Overview of ASTRA-SIM

IV. ASTRA-SIM

We present a network simulator called ASTRA-SIM for modeling the design-space presented in Section III. ASTRA-SIM is built on top of the Garnet [19] simulator (that comes as part of gem5 [20]) to use its networking infrastructure. Figure 6 shows the high level overview of the simulator components. We add two layers on top of the currently existing garnet simulator: (i) Workload layer, and (ii) System layer. Garnet is operated in a standalone manner [1]. ASTRA-SIM uses an event driven execution model - we use a separate event queue implemented in the system layer. The system layer then exposes its event queue to the workload layer to schedule events.

A key feature of ASTRA-SIM is that it implements topology-aware collective operations and different parallelism approaches of training. It provides a high level interface to the user to define new DNN models and simulate distributed training on different network topologies and configurations. The simulator then generates a detailed analysis regarding the communication behaviour of the workload and the effect of communication overhead on training. In addition, the software architecture interfaces also allow the user to add new topology & collectives operations. The current version implements the most common topology/collective pairs.

ASTRA-SIM is highly portable, meaning that it can be ported on top of any network simulator using a lightweight interface that minimizes the change in the network side. Moreover, the layer-wise training compute times can be calculated by any DNN compute simulator that is able to find the compute delays of GEMM operations in training (the green box in Figure 6). More details on these parameters are provided in the next section.

A. Workload Layer

The workload layer runs the training loop algorithm for different networks and generates the sets of data to be communicated at different steps of training.

Compute Model. For each layer, we run a compute model to estimate the delay (in cycles) for the GEMM operation in both the forward and backward passes. The compute model can be any DNN accelerator or GPU simulator. Together with the layer-wise communication size (during forward-pass and back-propagation), parallelism approach and layer orders, these information specify the characteristics of the training workload and is fed to the workload layer as an input file. We provide more details about the workload layer input format later in Section IV-C.

In our simulations, we used an analytical DNN accelerator simulator [12] to model a 256x256 TPU-like Systolic Array accelerator, though it is possible to use alternate compute models (e.g. [7], [13]) or a GPU simulator as well. Since the DNN accelerator simulator we used computes only the GEMM delay, we added additional parameterized delays to model the rest of the DNN layer computations. We also accounted for any stalls that would result due to limited DRAM bandwidth.

Communication Data Sizes. Table II shows the granularity of data at different levels of the execution and factor that determines the size. One collective operation is initiated by generating a set. Each set is then divided into multiple chunks and begins processing & scheduling of each chunk individually and in a pipelined manner. The chunk itself decomposes into multiple messages and the collective algorithm runs on the message granularity. As an example, if the ring has four NPUs, then, the chunk is divided into four messages for the ring algorithm. The messages are decomposed into multiple packets when they enter the network level. Each packet may contain one/multiple flits and each flit is divided into one/multiple phits when it is traversing the link.

B. System Layer

The systems layer is the interface to garnet and is responsible for implementing the topology aware collective operations and generating traffic to the network layer. The system layer contains the scheduler component that pipelines the execution of the collectives across the different links. The system layer deals the *logical topology*, that might be completely different from the actual *physical network topology*. This allows the system layer to have its own illusion of the topology, and tune the collective execution based on the logical topology. This provides the flexibility to: (i) map a single logical topology on different physical topologies and compare the results (e.g. mapping a 3D logical topology on a 1D or 2D physical torus), or (ii) map different logical topologies on a single physical topology and compare the performance (e.g. map logical alltoall and 3D torus on an arbitrary physical topology). In the the default configuration of ASTRA-SIM , there is a one-to-one mapping between the logical and the physical topologies.

Figure 7 shows the different components of the dispatcher and how each component works. The scheduler keeps two sets of queues: (i) ready queue, that maintains the queue of chunks that are ready but not yet issued, and (ii) logical scheduling queue (LSQ), that maintains a set of in flight (running) chunks for a single phase of the algorithm. Since each LSQ is for a single phase of the algorithm, the number of LSQs should be at least equal to the number of phases in the collective

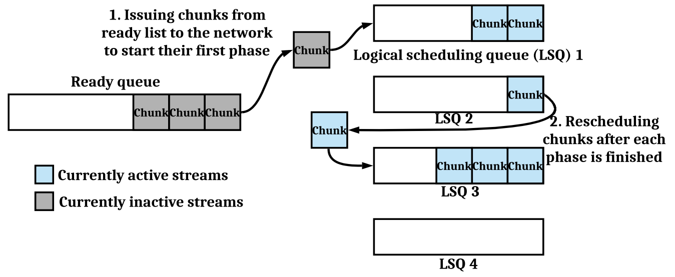
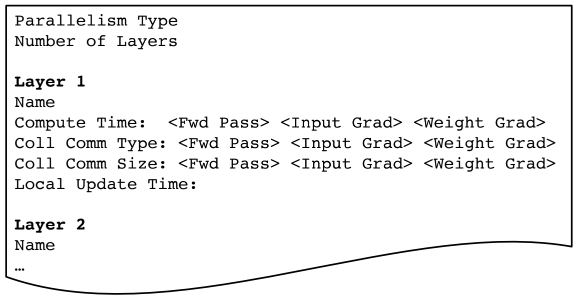

1. Issuing chunks from ready list to the network to start their first phase
Ready queue
Chunk
Chunk
Chunk
Chunk
Logical scheduling queue (LSQ) 1
Chunk
Chunk
LSQ 2
Chunk
2. Rescheduling chunks after each phase is finished
Chunk
LSQ 3
Chunk
Chunk
Chunk
LSQ 4
Currently active streams
Currently inactive streams

Fig. 7: The overview of the scheduler component and the two tasks of the dispatcher



```
Parallelism Type
Number of Layers

Layer 1
Name
Compute Time:  <Fwd Pass> <Input Grad> <Weight Grad>
Coll Comm Type: <Fwd Pass> <Input Grad> <Weight Grad>
Coll Comm Size: <Fwd Pass> <Input Grad> <Weight Grad>
Local Update Time:

Layer 2
Name
…
```

Fig. 8: Input file of the workload that describes the DNN and its compute behavior

algorithm. However, it is possible to have multiple LSQs per phase to further increase concurrency. In ASTRA-SIM 's default setting, the number of LSQs per phase is determined by the maximum number of chunks that could traverse the network using different dedicated links for that phase. For instance, if the horizontal dimension of a torus has two bidirectional rings, four LSQs are created for the horizontal phase of all-reduce on that phase, each one is dedicated to one uni-directional ring in that phase. In the case of alltoall topology, the number of global switches determine the number of LSQs for the alltoall dimension (the local dimension is treated as torus).

As Figure 7 shows, the dispatcher is responsible for issuing the chunks from ready queue to the LSQ and rescheduling the chunks among the LSQs once the current phase of the chunk is finished. The scheduler tries to interleave the execution of chunks within the same queue to fully utilize the bandwidth. In order to find out when to issue from ready queue, the dispatcher keeps track of the current active chunks at their first phase, if they fall below a certain threshold T, the dispatcher issues P new chunks from the ready queue. By implementing the collective operations, the system layer provides the collective APIs to the workload layer.

C. ASTRA-SIM *Parameters*

Table III shows the different simulator parameters at the workload, system and network layers. Parameter #1 determines the name of the input file that describes the network. Figure 8 describes the format of this input file. It specifies the parallelism approach (e.g. data parallel, weight parallel), the total number of layers in the DNN, and the compute time (see Section IV-A), collective communication type (all-to-all, all-

TABLE III: ASTRA-SIM Input Parameters

#	Parameter name	Level	Value	Comment
1	DNN_name	Workload	string	The input name of the file describing the DNN compute times and communication sizes
2	num-passes	Workload	int	The number of forward/backward iterations for simulation
3	algorithm	System	baseline/ enhanced	The collective communication algorithm
4	num-npus	System	int	The total number of NPUs
5	num-packages	System	int	The total number of packages
6	package-rows	System	int	The number of rows in the 2D torus
7	scheduling-policy	System	LIFO/FIFO	defines the order in which the collectives are executed
8	topology	System	Torus2D/ AllToAll	The logical topology of the network
9	local-rings	System	int	The number of local rings for the local dimension in the logical topology
10	vertical-rings	System	int	The number of vertical rings for the vertical dimension of the logical 2D torus
11	horizontal-rings	System	int	The number of vertical rings for the horizontal dimension of the logical 2D torus
12	global-switches	System	int	The number global switches int the alltoall logical topology
13	endpoint-delay	System	int	The endpoint delay constant delay after receiving a message
14	packet-routing	System	hardware /software	The way packets are routed in the the network
15	injection-policy	System	aggressive/ normal	How many messages should be injected in the case of hardware routing.
16	preferred-set-splits	System	int	The preferred number of the chunks each set should be divided into
17	local-link-efficiency	Garnet	int	The ratio between data-flits and (data-flits+header-flits) on the intra-package links
18	package-link-efficiency	Garnet	int	The ratio between data-flits and (data-flits+header-flits) on the inter-package links
19	flit-width	Garnet	int	The size of flits to be simulated
20	local-packet-size	Garnet	int	The size of the intra-package packets
21	package-packet-size	Garnet	int	The size of the inter-package packets
22	tile-link-width	Garnet	int	The width of the intra-package links
23	package-link-width	Garnet	int	The width of the inter-package links
24	vcs_per_vnet	Garnet	int	The number of the VCs per vnet
25	router-latency	Garnet	int	The latency of routers
26	local-link-latency	Garnet	int	The latency of the intra-package links
27	package-link-latency	Garnet	int	The latency of the inter-package links
28	buffers-per-vc	Garnet	int	The number of the buffers per vnet

reduce), size (computed from the layer dimensions) and local update time (i.e., the average time per 1KB of data it takes to process/reduce the communicated data after the forward-pass/input-gradient/weight-gradient communication is finished for that layer).

We do not describe the details of the rest of the parameters in Table III in the interest of space, but provide these details in the README of the tool's github page.

V. Experimental Results

This section is organized as follows: we begin by comparing alltoall and Torus topology for 8 NAPs (or packages) with one NAM per NAP i.e. 1D topology in Section V-A. We expand this to 64 packages and compare 2D/3D Torus topologies in Section V-B. Both these studies use one NAM per NAP and is representative of today's accelerator designs where the inter-package links are the same bandwidth (i.e. symmetrical).

TABLE IV: System Parameters

Parameter	Values
Intra-package	
Packet size	512 Bytes
Per link BW	200 GB/s
Link latency	90 cycles
Number of rings	2 (unidirectional)
Link efficiency	94%
Inter-package	
Packet size	256 bytes
Per link BW	25 GB/s
Link latency	200 cycles
Number of rings	2 (bi-directional)
Link efficiency	94%
NPU and NMU parameters	
Compute Accel.	256x256 TPU-like
Flit width	1024 bits
Router latency	1 cycle
VC/VNET	50
Message size	512B
Endpoint delay	10 cycles
Buffers per VC	5000

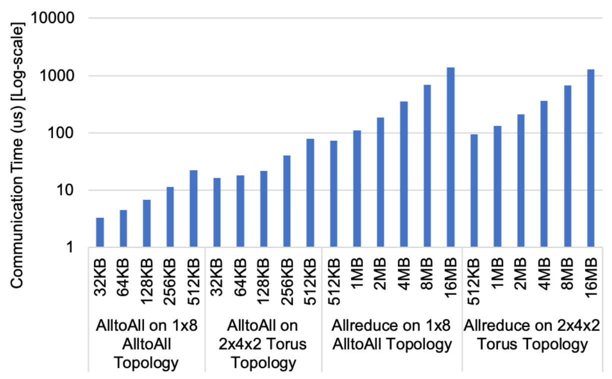


Fig. 9: 1D Topology: alltoall vs. Torus comparison

Section V-C introduces the benefits of having an asymmetrical and hierarchical topology with high bandwidth within a package; representing future accelerator designs with multi-chip packaging technology. Section V-D the benefits of asymmetric hierarchical topologies as we scale from 8 packages to 64 packages. We conclude with end-to-end Resnet-50 analysis in Section V-F.

Table IV describes the key simulation parameters used in our experiments. The message sizes for all-to-all and all-reduce collectives are based on typical sizes we observe in today's DL workloads such as Facebook's Deep Learning Recommendation Model [17] and Resnet-50 [16]. For the sake of consistency, we use the bandwidth test for comparison and assume software-based routing. In the interest of space, the results focus more on highlighting ASTRA-SIM capabilities and where possible describe the trade-offs of different solutions under consideration. A detailed study on specific system design recommendations

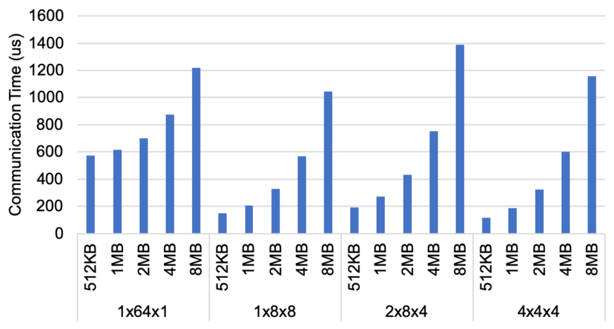


Fig. 10: Impact of 2D/3D Torus Topology

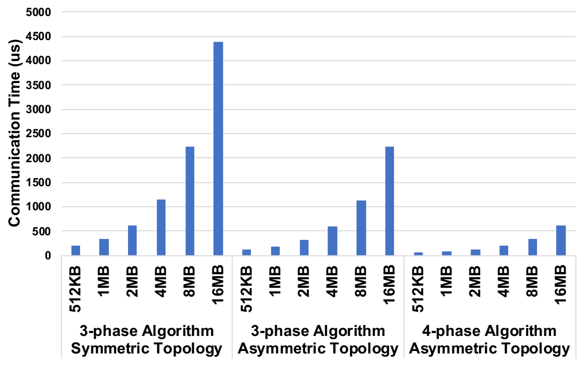
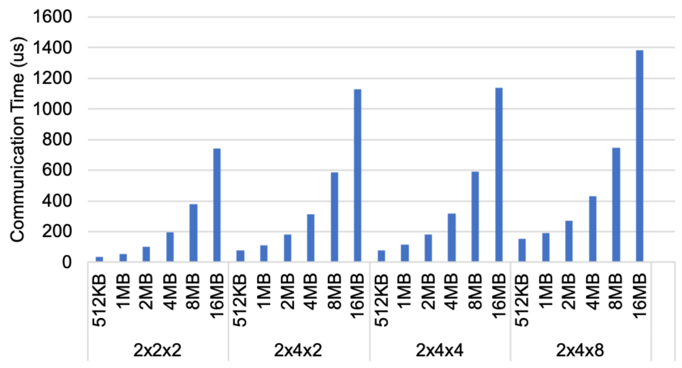
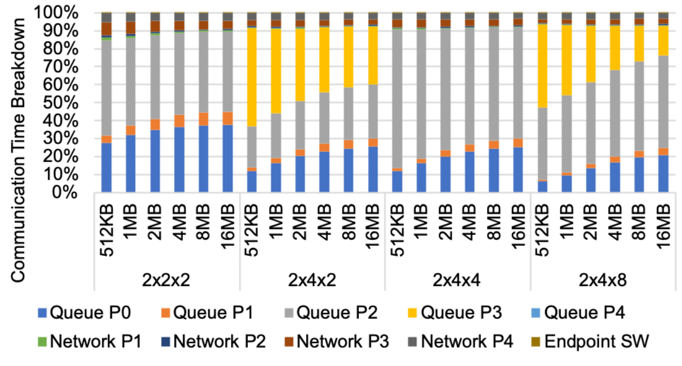


Fig. 11: Impact of Hierarchical Topology



(a) Communication Time in micro-seconds



(b) Communication Time breakdown

Fig. 12: Scaling on Torus topology

for DL training will be presented in future.

A. *Impact of 1D Topology*

Figure 9 presents the communication time for all-to-all and all-reduce collectives on 1x8 alltoall topology (i.e. 1 NAM per NAP and 8 NAPs connected via alltoall topology) and 1x8x1 Torus topology (i.e. 1 NAM per NAP and 8 NAPs connected via 1D ring topology). Each NAM has 8 links with one link per peer NAM for alltoall topology (through 7 global switches, leaving 1 link unused) and four links per peer NAM for Torus topology (1D ring). In the case of all-to-all collective, alltoall topology always outperforms the Torus topology, although as the message size increases the performance difference between the two topologies shrinks. In the case of all-reduce, Torus topology performs better as we increase the message since because the alltoall topology uses only 7 of the 8 links. Further, alltoall topology has higher queuing delays given a single link between a pair of peers. On the other hand, Torus topology supports multiple rings and is bandwidth optimal since it supports better pipelining across chunks.

B. *Impact of 2D/3D Torus Topology*

Figure 10 presents the impact of 2D/3D Torus topologies at 64 packages for all-reduce collective with symmetric links (i.e. links with same BW) and running the baseline algorithm. Adding extra dimensions without increasing the number of links or BW per link results in: (a) lower average number of hops per ring dimension (positive effect), and (b) depending on the algorithm, it may increase the total amount of data a node sends out because the number of steps in the algorithm increases (negative effect). Going from 1x64x1 to 1x8x8 (i.e. 1D to 2D Torus), the number of hops decreases (63 vs. 2x7=14) outperforms the negative effect of increased data sent out by each node ($\frac{126}{64}N$ in 1X64X1 vs. $\frac{28}{8}N$ in 1X8X8, assuming N is the initial data size at each node). However, going from 1x8x8 to 2x8x4 results in worse overall time because of the dominant effect of increased data size, while the phase with longest latency (i.e. the latency bottleneck) remains constant in both configurations (the ring with 8 nodes). In contrast, going from 2x8x4 to 4x4x4 (3D topology) results in better performance since the worst case number of hops goes down. Note that 4x4x4 is even better than 1x8x8 for messages up to 4MB. But from 4MB, messages are bandwidth bound and having less data size to send out is beneficial ($\frac{28}{8}N$ in 1X8X8 vs. $\frac{36}{8}N$ in 4X4X4). We observe similar behavior for all-to-all collective.

C. *Impact of Asymmetric Hierarchical Topology*

Figure 11 presents all-to-all and all-reduce collective on a 64 module system with 4 NAMs per NAP and total of 16 NAPs, i.e. four modules per package and sixteen packages. For the asymmetric hierarchical system representative of future multi-chip packaged accelerator design, we have two uni-directional rings within the package and four bi-directional rings across packages forming a 4x4x4 topology. The local link bandwidth within a package is assumed to be 8x the inter-package links for asymmetric system and 1X for symmetric case. The main benefit of asymmetric hierarchical case arises from higher bandwidth local rings being able to feed the inter-package links

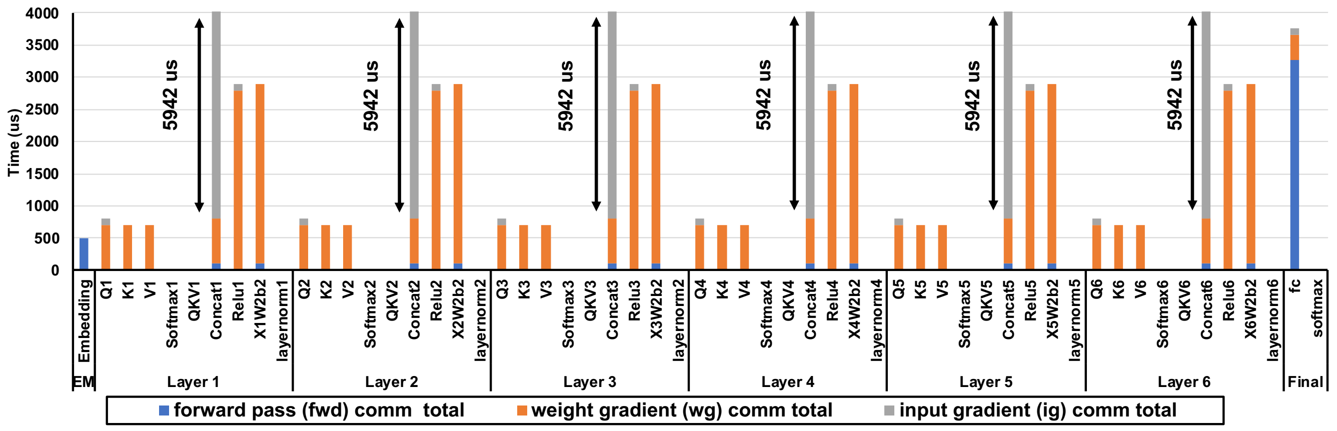


Fig. 13: Transformer layer-wise total raw communication time for two training iterations running on a 2X2X2 torus system. Since the parallelsim approach is hybrid-parallel, the output activations, input gradients and weight gradients should be communicated. Depending on the layer type and how model is split, some layers may not have communications.

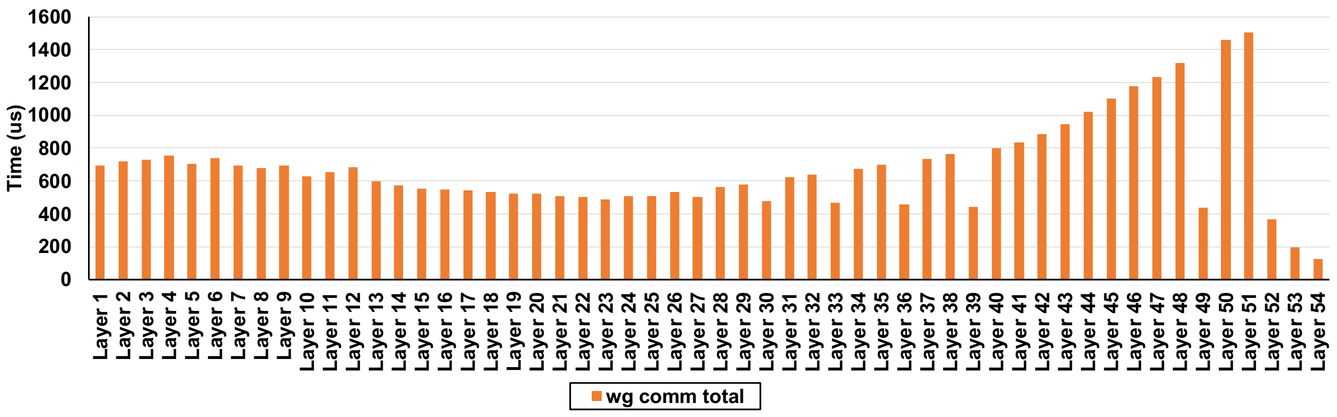


Fig. 14: Resnet-50 layer-wise total raw communication time for two training iterations running on a 2X4X4 torus system. Since the parallelism is data-parallel, only weight gradients need to be communicated during the back-propagation.

faster. Only changing from symmetric to asymmetric topology improves the performance significantly. It is possible to leverage the asymmetric system bandwidth and further improve the performance by using the four phase (enhanced) algorithm, as shown int the Figure 11. The three phase (baseline) algorithm performs three-level ring algorithm but the four-phase algorithm performs an hierarchical all-reduce: reduce-scatter within local dimension, all-reduce across two inter-package links, and all-gather across local dimension. This helps reduce the volume of data across inter-package links by 4x.

D. Impact of scaling on Torus topology

Figure 12 presents the total communication time, and the breakdown for all-reduce collective as we increase the number of modules from 8 to 64 in the Torus topology. Queue P0-P4 represents average queue delays at the different stages of 4-phase algorithm (P0 is the delay of ready queue described in Figure 7) and the Network P1-P4 represent average message delays inside network during different phases of collective algorithm. While communication time generally increases, in Figure 12a, we observe a slower growth in latency from 2x4x2 (16 modules) to 2x4x4 (32 modules). In the case of 2x4x2, the bottleneck is mostly the horizontal dimension (with 4 modules in the ring). With changing to 2X4X4, the bottleneck dimension size remains 4 (both horizontal and vertical dimension size are 4), but now the bottleneck is shifted to the vertical dimension since it is traversed sooner than horizontal dimension in the collective algorithm. This can be verified by observing that P2 queue delay becomes the dominant factor for the 2X4X4 in Figure 12b. However, going from 2x4x4 to 2x4x8 creates a new bottleneck (i.e. the dimension with 8 modules) and thus increases the communication time. This observation also demonstrates the importance of hierarchical torus topologies for scaling to large number of modules. We can effectively increase the size of the torus topology without incurring additional overheads.

E. Workload High-Level Analysis

In this section, we provide a high level workload analysis report for Transformer [8] and Resnet-50 [16] networks. In

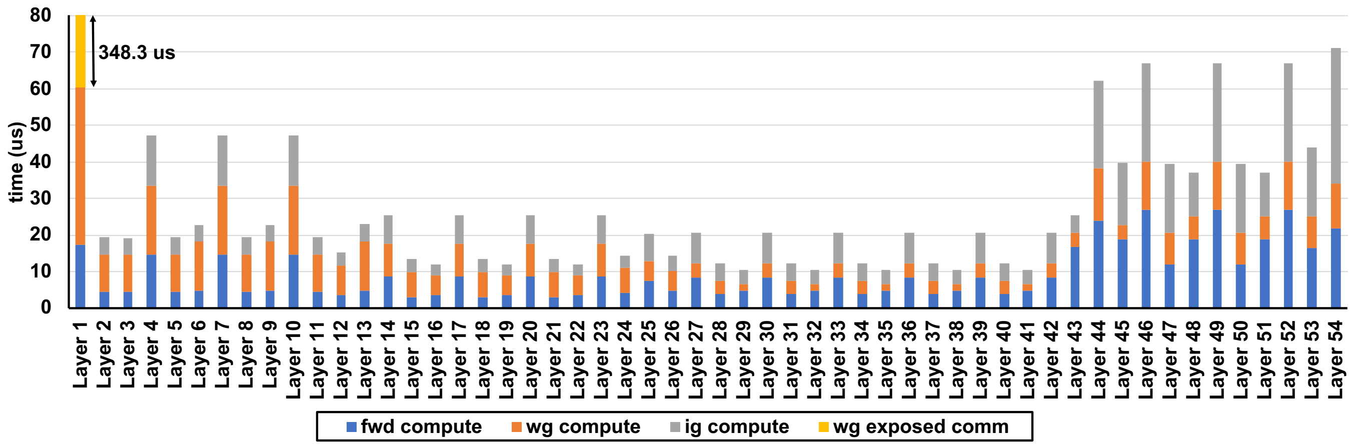


Fig. 15: Resnet-50 layer-wise compute and communication time

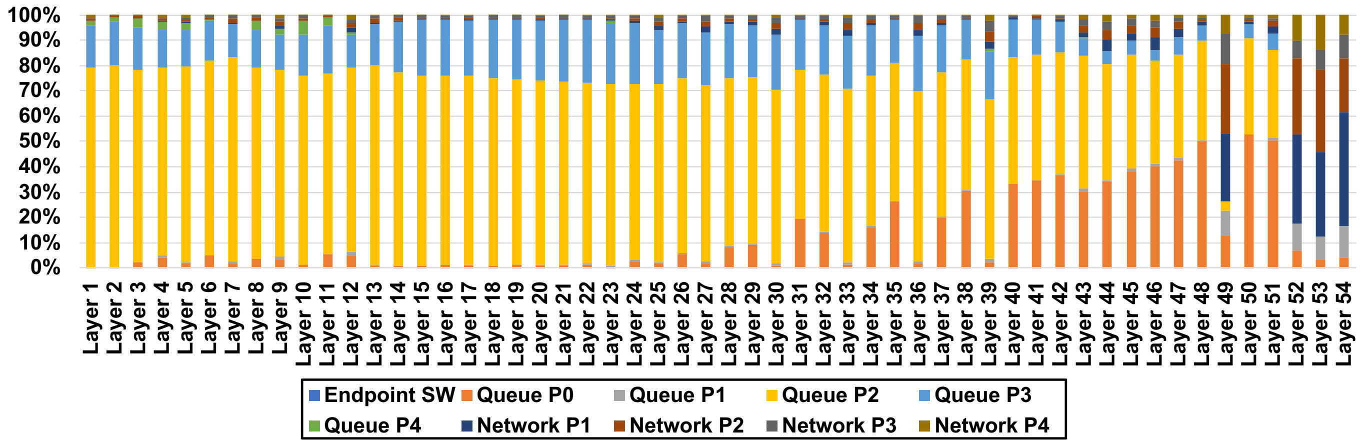


Fig. 16: Resnet-50 layer-wise communication time breakdown

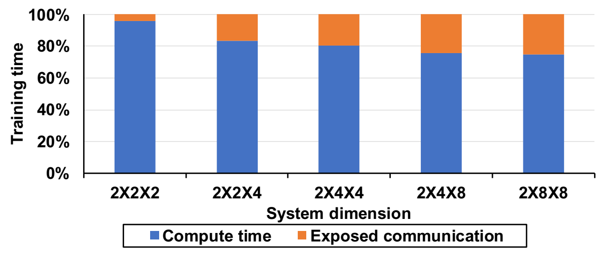


Fig. 17: Resnet-50 compute time and exposed communication time ratio for different network sizes

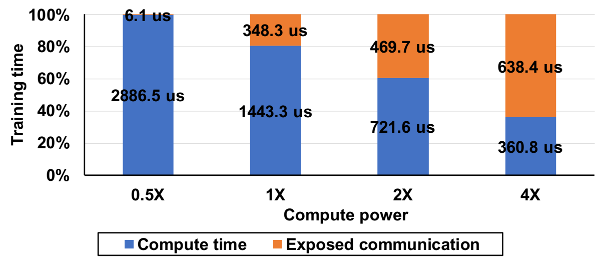


Fig. 18: Resnet-50 compute time and exposed communication time ratio for different compute powers compared to the baseline

Section V-F, we further breakdown Resnet-50 and provide more detailed reports along with sensitivity analysis to demonstrate the capabilities of ASTRA-SIM in reporting details at various levels for real DNN workloads. Figures 13 and 14 present the layer-wise total communication time spent for two training iterations of Transformer and Resnet-50, respectively. The network dimension for Transformer is a 2X2X2 torus system while Resnet-50 network is a 2x4x4 Torus topology, both with LIFO collective scheduling order and local mini-batch size of 32. The parallelism approach for Resnet-50 is data-parallel while Transformer training is hybrid-parallel (data-parallel across local and horizontal dimension, and model-parallel across vertical dimension).

As Figure 13 shows, the communication latency of Transformer remains uniform across different layers (layers 1-6 are the same structurally). This is because of strict communication dependencies that exist on the hybrid-parallel approach. For example, in forward pass, the output activation of a layer must be finished to proceed to the next layer. The same dependency exist for input gradients during the back-propagation. But for the data-parallel approach, execution of collectives across different layers can be overlapped during back-propagation.

This high-level analysis gives a rough estimation about the efficiency of the system in handling communication. However, two factors complicate this analysis. First, the total communication time depends on the interplay between the parallelism approach, communication size, computation time, network performance, scheduling policy, chunk-size and so on. Second, the communications are executing in parallel and also overlapped with computation. Thus further investigation to determine which factors are playing a major role needs more detailed analysis, which we provide in the next section.

F. Resnet-50 Detailed Analysis

Figure 15 shows the layer-wise end-to-end time spent associated with Resnet-50 analysis described in previous section. In addition it shows the layer-wise exposed communication latency (the yellow bar) that is the amount of communication time that is not overlapped and the training algorithm is forced to stop for such communications to be finished.

Figure 16 shows the layer-wise time breakdown for Resnet-50 similar to Figure 12b. Interestingly, we observe similar behavior for both FIFO and LIFO scheduling schemes and the reasoning is as follows. At every cycle, the scheduler issues 16 new chunks from the ready queue if there are fewer than 8 chunks yet to complete phase 1 of the algorithm (over local dimension). Now assume we are executing Layer 50. Given 8x higher bandwidth in the local dimension and good compute-communication overlap with input gradient compute Layer 50 and weight gradient compute of Layer 49, all chunks Layer 50 complete phase 1 of the collective algorithm before new chunks from Layer 49 become available. This can be further validated by inspecting the per-layer communication time breakdown across different queue and network links. After few initial iterations, the majority of delay is in Queue P2 waiting for the scale-up fabric to complete previously issued chunks. In a nutshell, the very high local bandwidth enforces in-order execution and turns LIFO scheduling to behave similar to FIFO scheduling.

Figure 17 shows the compute time and exposed communication time ratio of the Resnet-50 as the Torus dimensions varies between 2X2X2 (8 NPUs) to 2X8X8 (128 NPUs). As we expect, increasing the size of the system increases the exposed communication ratio, from 4.1% in the 8 node system to 25.2% in the 128 node system.

Figure 18 shows how the role of communication overhead changes for various compute efficiencies. These kinds of analysis are specially useful to predict the performance improvement of next generation NPUs with higher performance and find the point in which the communication becomes the bottleneck. ASTRA-SIM allows. According to Figure 18, the exposed communication is less than 1% when the compute power is 0.5X of baseline. This is because there is enough time for collectives to be completely overlapped with computation. But at 4X compute power, 63.9% of the latency comes from communication, resulting in diminishing effect of further improving the compute efficiency.

In summary, ASTRA-SIM allows us to study various scheduling policies, collective algorithms, and hierarchical topologies for DL training.

VI. Related Work

In recent years, many schemes proposed simulators and designs to model and enhance communication between different nodes, mostly targeting the scale-out network. For instance, the concept of designing a NIC/protocol pair for offloading the network tasks in the scale-out network has been proposed by prior works [15], [27]. More specifically, Underwood *et al.* [15] proposed designing a triggered based architecture that will trigger collective communications based on listening to the certain network events defined by the programmer. sPIN NIC (proposed in [27]) allows the programmer to offload packet handlers that processes different types of packets upon arrival using available compute resources inside the NIC. LogGOPSim [26] proposed a simulator that models the collective communication based on the LogGP abstract model [9]. Degomme et al. [5] models the MPI library collectives on a distributed system. Compared to ASTRA-SIM , the main difference of the previous schemes are: (i) They target scale-out network while ASTRA-SIM targets the scale-up network, (ii) ASTRA-SIM provides built-in support for implementing different parallelism approaches and training loops.

In terms of collective algorithms, many works suggested and examined different algorithms based on different model assumptions. Thakur *et al.* [23] analyzed algorithms based on the constant overhead between each two communicating node. Chan *et al.* [11] extended the collective operations for nodes with the ability to send/multiple messages. Patarasuk *et al.* [22] introduced bandwidth efficient all-reduce for clusters of workstations. However, these algorithms mainly targeted the scale-out network or they analyzed the algorithms using simplified assumptions for the systems that are not realistic in the actual systems.

MCM-GPU [3] and Dally et al. [28] discussed the circuit level techniques to enhance the memory bandwidth for deep learning training/inference workloads. Arunkumar *et al.* [4] proposed energy cost model for multi-chip scale-up design. Energy-cost model could be integrated to our work and we leave this to future work.

VII. Conclusions and Future Work

This work introduces a simulator called ASTRA-SIM to elaborate and navigate the HW/SW design-space of distributed DL training platforms. We focus on detailed modeling of collective communication algorithms over a hierarchical scale-up fabric, and enable end-to-end network simulations. The tool has been open-sourced. We also plan to extend it to a scale-out fabric (modeling the transport layer, e.g., Ethernet) as part of future work.

Acknowledgment

This work was supported by a Facebook Faculty Research Award. We thank Srikant Bharadwaj from AMD for sharing an updated version of Garnet that models both on-chip and interposer/MCM networks, and Eric Qin from Georgia Tech for sharing his analytical compute model.

References

[1] Garnet2.0. http://synergy.ece.gatech.edu/tools/garnet/, 2017.

[2] K. Arnold. Application-specific hardware accelerators, Mar 2019.

[3] A. Arunkumar et al. Mcm-gpu: Multi-chip-module gpus for continued performance scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 320–332, New York, NY, USA, 2017. ACM.

[4] A. Arunkumar et al. Understanding the future of energy efficiency in multi-module gpus. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 519–532, Feb 2019.

[5] A. Degomme et al. Simulating mpi applications: The smpi approach. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2387–2400, Aug 2017.

[6] A. Li et al. Evaluating modern GPU interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *CoRR*, abs/1903.04611, 2019.

[7] A. Samajdar et al. Scale-sim: Systolic CNN accelerator. *CoRR*, abs/1811.02883, 2018.

[8] A. Vaswani et al. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[9] D. Culler et al. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[10] D. Das et al. Distributed deep learning using synchronous stochastic gradient descent. *CoRR*, abs/1602.06709, 2016.

[11] E. Chan et al. Collective communication on architectures that support simultaneous communication over multiple links. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 2–11, New York, NY, USA, 2006. ACM.

[12] E. Qin et al. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2020.

[13] H. Kwon et al. Understanding reuse, performance, and hardware cost of dnn dataflows: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 754–768. ACM, 2019.

[14] J. Dean et al. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[15] K. D. Underwood et al. Enabling flexible collective communication offload with triggered operations. In *2011 IEEE 19th Annual Symposium on High Performance Interconnects*, pages 35–42, Aug 2011.

[16] K. He et al. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[17] M. Naumov et al. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[18] M. Papadonikolakis et al. Performance comparison of gpu and fpga architectures for the svm training problem. In *2009 International Conference on Field-Programmable Technology*, pages 388–391, Dec 2009.

[19] N. Agarwal et al. Garnet: A detailed on-chip network model inside a full-system simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 33–42, April 2009.

[20] N. Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[21] N. P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[22] P. Patarasuk et al. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124, February 2009.

[23] R. Thakur et al. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005.

[24] S. A. Mojumder et al. Profiling dnn workloads on a volta-based dgx-1 system. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 122–133, Sep. 2018.

[25] S. Sridharan et al. On scale-out deep learning training for cloud and HPC. *CoRR*, abs/1801.08030, 2018.

[26] T. Hoefler et al. Loggopsim: Simulating large-scale applications in the loggops model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 597–604, New York, NY, USA, 2010. ACM.

[27] T. Hoefler et al. spin: High-performance streaming processing in the network. *CoRR*, abs/1709.05483, 2017.

[28] W. J. Dally et al. Hardware-enabled artificial intelligence. In *2018 IEEE Symposium on VLSI Circuits*, pages 3–6, June 2018.

[29] Y. Li et al. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. pages 175–188, 10 2018.

[30] R. McDonald. Distributed training strategies for the structured perceptron. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, pages 456–464, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.