An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives

Benjamin Klenk  
*NVIDIA, USA*  
bklenk@nvidia.com

Nan Jiang  
*NVIDIA, USA*  
tedj@nvidia.com

Greg Thorson  
*NVIDIA, USA*  
gthorson@nvidia.com

Larry Dennison  
*NVIDIA, USA*  
ldennison@nvidia.com

Abstract—The slowdown of single-chip performance scaling combined with the growing demands of computing ever larger problems efficiently has led to a renewed interest in distributed architectures and specialized hardware. Dedicated accelerators for common or critical operations are becoming cost-effective additions to processors, peripherals, and networks. In this paper we focus on one such operation, the All-Reduce, which is both a common and critical feature of neural network training. All-Reduce is impossible to fully parallelize and difficult to amortize, so it benefits greatly from hardware acceleration.

We are proposing an accelerator-centric, shared-memory network that improves All-Reduce performance through in-network reductions, as well as accelerating other collectives like Multicast. We propose switch designs to support in-network computation, including two reduction methods that offer trade-offs in implementation complexity and performance. Additionally, we propose network endpoint modifications to further improve collectives.

We present simulation results for a 16 GPU system showing that our collective acceleration design improves the All-Reduce operation by up to 2x for large messages and up to 18x for small messages when compared with a state-of-the-art software algorithm, leading up to 1.4x faster DL training times for networks like Transformer. We demonstrate that this design is scalable to large systems and present results for up to 128 GPUs.

I. INTRODUCTION

As the amount of data we generate increases at staggering rates, compute resources on which data is processed need to keep up. While we observe that computing systems are becoming increasingly parallel, they are also being augmented by specialized hardware to accelerate performance-critical tasks. Examples of such specialized hardware include the Graphics Processing Unit (GPU), Field Programmable Gate Arrays (FPGAs) and various Machine and Deep Learning (ML/DL) accelerators like Google’s Tensor Processing Unit (TPU) [1]. These specialized processors not only increase performance but also provide much better power efficiency and therefore allow for higher compute density and scaling.

With increased parallelism, performance scalability becomes increasingly dependent on communication. Similar to how compute tasks can be accelerated by specialized hardware, certain communication patterns benefit from such specialization as well. A common pattern among scientific and especially distributed ML/DL applications are *collectives*. A simple example is the broadcast in which many processors receive the same data from one processor.

Although most of these collective patterns are only concerned with data distribution, the *Reduce* and *All-Reduce* operations also involve computation. In the Reduce operation, participating processors send their data to a root processor on which the data is reduced according to an arithmetic or relational operation. In the case of the All-Reduce operation the result is also broadcast to every participant. The All-Reduce operation is especially critical in parallel DL training algorithms on distributed systems. After each worker calculated the gradients of the parameters, these gradients need to be aggregated among all workers, resulting in one or more All-Reduce operations per training iteration. This operation can quickly become a bottleneck [2]–[4], motivating the need for specialized hardware to accelerate this common operation.

Previous work has shown that offloading reduction computation into the network can be an effective technique [2], [3], [5]. However, existing work is limited to CPU-initiated communication models where data movement between the accelerators are coordinated by the CPU using message passing protocols and often require explicit resource reservations. In a highly parallelized environment, this adds overhead that diminishes the performance gain offered by in-network reduction. Furthermore, in distributed shared-memory systems, network packets carrying memory operations are small and numerous, making reservation protocols prohibitively expensive.

In this work, we focus instead on an accelerator-centric communication model, where accelerators are directly attached to a distributed shared-memory fabric. Such a system architecture is exemplified by GPUs interconnected with NVIDIA’s NVLink/NVSwitch [6] network or AMD’s InfinityFabric [7] network. We propose a network/endpoint co-design architecture that accelerates various collective communication patterns, with a focus on the All-Reduce operation. Our approach is designed for highly parallel accelerators with millions of threads, sharing a global address space. The contributions of this work can be summarized as follows:

- We present a novel scalable shared-memory collective architecture for massively parallel accelerator systems.
- We present two different in-network reduction methods which differ in complexity of endpoints and switches.
- We show how we can extend DMA engines to efficiently use the in-network reduction methods to improve resource utilization and performance.
Our results show that our mechanisms are up to an order of magnitude faster than state-of-the-art software solutions, but also more suitable to large-scale, shared-memory systems than existing hardware proposals.

- We evaluate the benefits of our designs for the training of widely used and important deep neural networks.

The remainder of the paper starts with the background in § II. We review state-of-the-art software algorithms for the All-Reduce operation in § III. This is followed by a description of our system architecture in § IV, before results from our simulations are presented in § V and VI. Related work is given in § VII. The conclusion is presented in § VIII.

II. BACKGROUND AND MOTIVATION

A. Accelerated Computing Systems

Our work focuses on accelerator-centric systems with networks that directly interconnect massively parallel accelerators rather than CPUs. Particularly, we use NVIDIA’s DGX-2 [6] as a reference for such a system, but this work also applies to other shared-memory systems as well.

The DGX-2 system has 16 GPUs, each with 6 NVLink ports. The system implements a network with a Fat-Tree topology using twelve 16-port NVSwitches. GPUs and switches are evenly divided into two groups and within a group, each switch is connected to each of the 8 GPUs via one NVLink. Each switch’s remaining 8 ports are connected to a switch in the other group. This topology provides full-bisection bandwidth.

The NVLink fabric allows GPUs to access every GPU’s memory through ordinary memory operations, such as loads, stores, or atomics. While many existing in-network reduction proposals [2], [3], [5] are based on message-passing communication with a single source for each message, allowing them to use rather complex reservation protocols, GPUs and their shared memory fabric don’t fit this paradigm. Multi-GPU systems operate with millions of concurrently executing threads and a globally shared address space. The order in which these threads execute can be arbitrary and non-deterministic. Any network and reduction protocols that rely on a certain injection and ejection order are not feasible. Furthermore, packets in the network represent individual memory operations, which are both small and numerous. It is therefore essential for packets to have a lean format with minimal overheads, rendering existing message-passing and reservation-based protocols prohibitive. While GPUs do not implement complex network interfaces, they provide a plurality of simple DMA engines to which linear and multi-dimensional data transfers can be offloaded. In principle, these DMA engines create multiple requests that are issued to the memory system or network. DMA engines receive their commands through a command queue which is written by the host processor. Completed data transfers are signaled through the command interface as well. Although DMA engines might appear as a single source of a message, the resulting data stream still does not guarantee any ordering and remains non-deterministic.

B. DL Training

Deep neural nets (DNNs) have been shown to produce excellent and sometimes better-than-human results in tasks like image classification [8] or complex games like Go [9]. Before these networks can make accurate predictions, however, they need to be trained on large amounts of data. The training can take hours or even days and is extremely compute intensive. The following presents an overview on how neural nets are trained in parallel and highlights the importance of the All-Reduce operation.

1) Methodology: We analyze the communication requirements of DL training in two ways. For data-parallel training we use an accurate performance model to determine the time spent on every layer of a DNN. The model comprises a detailed description of the compute and memory system of a NVIDIA Volta GPU and matches real silicon performance closely. After we have calculated the compute time for each layer, we walk through the computational graph and determine how many weight gradients are being generated, as these need to be reduced among workers. The reduction can be implemented as a single, non-overlapped All-Reduce at the end of an iteration (one-shot), or as smaller All-Reduce operations for each gradient-producing layer. The latter allows overlap of communication with computation. Our model-based approach allows us to study various effects, such as hypothetical performance improvements of future GPUs, different network bandwidths, and various All-Reduce algorithms.

For model parallelism we focus on NVIDIA’s Megatron [10], one of the largest natural language model to date. We measure the iteration time with the All-Reduce and without the All-Reduce on a real Volta-based DGX-2 system. We report time spent in the All-Reduce operation. Megatron’s model parallelism does not allow for overlap and we can therefore rely on measurements on real systems.

2) All-Reduce in Data Parallelism (DP): Data-parallel training is the most common approach to parallel training. In stochastic gradient descent (SGD) a mini-batch of input samples is divided among processors. Each processor trains a model replica on its set of samples, which we refer to as sub-batch, and calculates weight gradients which are then reduced among all processors.

In order to assess the importance of the All-Reduce we need to take the training system into account. An NVIDIA Volta GPU [11] provides a theoretical performance of 120TFLOP/s and 150GB/s unidirectional NVLink bandwidth. While this is twice as much bandwidth as the previous Pascal architecture [12], the compute performance grew by a factor of 12x for DL training, mainly enabled by specialized hardware like tensor cores [13]. While hardware is getting faster and more specialized for important applications like the training of DNNs, software also improves rapidly. For example, many MLPerf v0.6 [14] submissions were run on the same hardware as MLPerf v0.5 [15], but average performance increased by 1.5x with a maximal improvement of 2.85x (ResNet-50, 32 DGX-2H) in just about 7 months. Consequently, instead of showing performance results for Volta only we consider future
An important aspect is the scale of $i$ while we compute gradients for $\text{backward pass}$ the weight gradient is calculated for each layer, allowing us to start with $\text{training}$ becomes more sensitive to bandwidth. Overall, the largest model on 16 GPUs spends about 30% of the step time in the All-Reduce operation.

Fig. 1b shows the results of our model-based analysis and the Transformer network on a DGX-2 system (16 GPUs). It presents the fraction of time spent in weight gradient reductions using the state-of-the-art ring algorithm and one-shot communication. On today’s Volta architecture we observe that up to 30% of the training time is spent on the All-Reduce, while projected future generation GPUs with higher compute capabilities exhibit higher fractions of up to 42%. The graphs also show the value of NVLink as high-bandwidth interconnect between GPUs, as communication dominates the training time on Ethernet-based systems. We note that overlap is not beneficial at larger scale and hence smaller sub-batches, as the latency of small message sizes is too high for ring-based algorithms and high kernel launch overheads on GPUs.

3) Model Parallelism (MP): Another form of parallelism is model parallelism, in which the model itself is distributed. Although there are various ways to implement this, one way is to split input matrices of a general matrix multiplication (GEMM) so that each processor calculates a partial output matrix which is then reduced across processor [10].

The time spent in the All-Reduce operation during forward and backward pass is shown in Fig. 1c. We measured various model sizes, from 345 million parameters to 8.3 billion parameters. Although one could expect the fraction of the All-Reduce to decrease as models grow, the opposite is observed. This is due to the computation that becomes more efficient for larger models and therefore the training becomes more sensitive to bandwidth. Overall, the largest model on 16 GPUs spends about 30% of the step time in the All-Reduce operation. Note that not all model sizes can be run on arbitrary number of GPUs due to partitioning and memory capacity constraints.

III. COLLECTIVE COMMUNICATION PRIMITIVES

As we have shown, collective communication primitives are important for distributed training algorithms in the ML/DL area, but they are also common in many other scientific applications [20]. The following provides a brief overview and discusses the All-Reduce specifically in more detail.
The most basic collective pattern is a Broadcast in which one process sends data to every other process. The Multicast is a more general form in that the set of receiving processes can be a subset of all processes.

The Gather operation collects data from a set of processes in one process, commonly referred to as the root. The All-to-All operation has every process be a root of a scatter operation. In other words, if a square matrix with \( p \) rows is divided such that each process holds a row, the All-to-All operation will transpose the matrix in that each process then holds a column.

The Reduce operation follows the reverse path of a multicast in that each of \( p \) processes sends \( m \) data elements to a root process, where the data is reduced by applying arithmetic or relational operations. The Reduce-Scatter combines the reduction with the All-to-All operation. If every process holds \( m \) elements, each process will hold \( \frac{m}{p} \) of the result after the Reduce-Scatter completes.

Most operations exist with an All- prefix, which adds a broadcast at the end so that every process holds the result. For example, the All-Reduce operation extends the Reduce by a broadcast and upon completion each of the \( p \) processes holds the reduced \( m \) elements. As this operation is the focus of this work, we will discuss various implementations next.

A common approach to implement the All-Reduce in software is to use a ring pattern, effectively implementing a Reduce-Scatter followed by an All-Gather. Each processor reduces and transmits \( \frac{m}{p} \) elements along the ring in each step of the algorithm. After \( p - 1 \) steps, each processor holds the final reduction result for \( \frac{m}{p} \) elements. Additional \( p - 1 \) steps are required to distribute the reduction result around the ring.

Both NVIDIA’s NCCL [21] and Baidu’s All-Reduce [22] implement this algorithm. Although it achieves optimal bandwidth, the latency is proportional to the number of processors. In shared-memory systems, each processor sets up mailboxes in which the data is received. Validation of the data is signaled by a flag. A commonly used relaxed memory model requires a memory fence in between data and flag. The fence’s scope must comprise the whole system and is therefore an expensive operation that also impacts the local memory system, negatively impacting other running kernels. Consequently, the ring is best for smaller systems, as synchronization costs increase linearly with every processor added.

For larger systems, the ring exchange pattern can be replaced by a tree, reducing the number of steps in the algorithm from \( O(p) \) to \( O(\log p) \). An efficient algorithm is the double binary tree, or two-tree [23]. Here, every processor is at most a root and a leaf in either tree and therefore can send and receive at the same time, maximizing network utilization. NCCL version 2.4 and later use this algorithm for larger scale inter-node reductions across InfiniBand [24]. Other types of tree algorithms, such as recursive doubling, are also commonly used in other communication libraries such as MPI [25].

What is common to all bandwidth-optimal software implementations of the All-Reduce is that each processor must send and receive an amount of data that is twice the size of the All-Reduce message, once for each of the Reduce-Scatter and All-Gather phases. As a result, the maximal achievable bandwidth is limited to only half of the available network bandwidth. Therefore, in-network reductions have the potential of accelerating All-Reduce operation by approximately \( 2x \) from a bandwidth perspective. In practice, the realized speedup can be higher by eliminating expensive synchronization, especially at small reduction sizes and larger scale.

IV. IN-NETWORK REDUCTIONS

A. Multicast

As the All-Reduce operation requires the distribution of the reduction result to all GPUs, network-supported multicast is essential to accelerating its performance. It can also be beneficial for other collectives like All-Gather, Multicast/Broadcast, or the Barrier.

As this work focuses on shared-memory systems, we propose extending the global address space by multicast regions (MCRs). Each GPU registers an existing memory allocation with a network-wide fabric manager (Fig. 2, marker ①), and the resulting addresses (②) are mapped into the GPUs’ virtual address space (VAS). The mapping can be implemented through Memory Mapped I/O (MMIO), similar to how PCIe Base Address Registers (BAR) are mapped into the GPU’s VAS in NVIDIA’s GPUDirect [26], [27].

The fabric manager initializes MCR tables in the switches which contain multicast addresses and target IDs associated with the particular MCR (③). Note that a target can either be a switch or GPU and each switch has its own table. We only need one entry per MCR in the table, as the target IDs are the same for any address within an MCR.

A multicast starts with a store operation to an MCR address (④). Once the packet arrives at a switch it is determined whether the address belongs to an MCR. If it does, the target IDs for the multicast are taken from the table, packets are replicated and then forwarded to their destination according to their target ID (⑤). Note that the source of the packet may be excluded from the destinations of the multicast. The target ID is used for the lookup in the routing table (⑥), which provides the arbiter with a list of ports. In hierarchical network topologies, packets are replicated in a tree fashion through the network, minimizing bandwidth on links between switches.

![Multicast concept for a system with four GPUs connected by a switch.](image-url)
The number of available MCRs is limited by the size of the table, which grows according to \( O(2^p) \) if all combinations among \( p \) processors need to be supported. However, large numbers of multicast groups are rarely needed by applications [20]. Initialization of the MCR tables is done once at the beginning of an application.

Inside the network fabric, multicast requests use the same virtual channels (VCs) as unicast requests. We assume a switch design that allows multicast packets to make progress to some output ports when not all required output ports are available. For example, if there are \( k \) ports (\( k > 1 \)) a packet needs to go to, and one port is busy, the arbiter will allow the packet to go to \( k - 1 \) ports and considers the remaining port in the next arbitration round. This ensures progress and avoids deadlocks, but also improves performance by maximizing crossbar utilization.

Since the multicast is supported through the native VAS, it can be triggered by GPUs using ordinary memory operations. A multicast is simply a write that is replicated in the network. A load operation, however, may not be simply propagated as a multicast to all GPUs, as the result would be undefined. Instead, implementations may choose to route a read operation to the memory system of the local GPU.

Embedding the multicast information into the address of a packet allows us to keep the packet overhead small, as we do not need to carry information about multicast destinations in the header of every packet.

### B. Pull Reduction

While having multicast capabilities in the system benefits existing software reduction algorithms and other collectives, we see an opportunity to further accelerate the All-Reduce operation by adding compute capabilities to switches. In this work we propose two methods of performing in-network reductions, *Pull* and *Push*, which differ in complexity of their implementation in GPUs and switches.

The first method is to have GPUs trigger reductions by injecting pull requests into the network, as illustrated in Fig. 3. A pull request is essentially a load operation with a reduction operator that is multicast to a plurality of GPUs by a single requester. The network applies the reduction operator to the responses before returning the final result to the requester. In the All-Reduce operation with \( m \) elements and \( p \) GPUs, every GPU would issue \( \frac{m}{p} \) pull requests and therefore get \( \frac{m}{p} \) results back. A final All-Gather, supported by the in-network multicast, completes the operation.

Fig. 3 illustrates a multi-switch system with four GPUs (GPU 0 to GPU 3). For simplicity only GPU 0 issues two pull requests, \( R0 \) and \( R1 \), though our architecture supports pull requests issued by all GPUs simultaneously to implement an All-Reduce. Once a pull request arrives at the input port of the first-hop switch, an entry is allocated in a reduction table to hold the partial reduction results of returning responses (1).

We require pull requests to allocate an entry in the first-hop switch’s reduction table. If the port’s table is full, we stall the request until a space is freed by an outstanding request. Note that responses use a different VC than requests and therefore reductions always make progress and eventually release resources. Stalled requests create backpressure into GPUs and will therefore limit the injection of new requests.

After the reduction table allocates an entry, requests are sent through the network using the multicast mechanism described in § IV-A. In a multi-switch system, as a request traverses through the multicast tree, it will attempt to allocate a reduction table entry at each hop before it is forwarded. In these intermediate switches, if the allocation succeeds (2), returning responses from the subsequent multicast subtree (2) are collected and reduced at this intermediate reduction table before a single response is sent back towards the requesting GPUs (3). If the allocation at the intermediate switch is unsuccessful (4), these returning responses (4) will bypass the intermediate reduction table (5) and be individually routed toward the requesting GPU. This opportunistic allocation in a multi-switch system avoids complex management protocols of reduction table resources. Since the table allocation is required at the first-hop switch, all responses eventually arrive at the first-hop reduction table. Upon receiving the responses from all GPUs, the result is returned to the requesting GPU (6).

Reduction tables need to have the ability to count responses to determine whether the reduction is complete. When a request allocates an entry in the table it needs to register the number of expected responses with the entry. This information is taken from the switch’s MCR table (Fig. 2 marker 2) which is set by the fabric manager during the application’s initialization phase. An advantage of the pull method is that we can easily handle the case when a reduction table is full. We stall requests at the first-hop switch port if resources are busy and bypass the reduction table in intermediate switches if their resources are exhausted.

The downside, however, is that the pull method requires synchronization before the reduction to ensure that every GPU has completed any operation that manipulates the data to be reduced before these requests are injected into the network. Otherwise the requests might fetch stale data, leading to wrong results. Therefore, a barrier is needed before the All-Reduce.

Reduction tables need to be sized appropriately to sustain full bandwidth. This can be estimated through Little’s Law
Fig. 4. Concept of the push method in a system similar to NVIDIA’s DGX-2. [28]. If a GPU injects a pull request into the network all other GPUs within the multicast group will respond with data. If we assume that all GPUs in a system participate in the reduction, the minimum network reduction table size $C_{\text{min}}$ is given by the GPU’s injection bandwidth $B_{\text{in}}$, the maximal number of hops between two GPUs $d_{\text{max}}$, the latency $L$, and the number of GPUs $p$. The latency is a function of the per-hop latency of the network $L_{\text{hop}}$ and the GPU’s response latency $L_{\text{GPU}}$.

$$C_{\text{min}} = \frac{B_{\text{in}}}{p-1} \cdot \left(2 \cdot d_{\text{max}} \cdot L_{\text{hop}} + L_{\text{GPU}} \right)$$

(1)

The bandwidth needs to be divided by $p-1$ because each table entry captures $p-1$ returning responses. Requests are small and therefore negligible. Consequently, most bandwidth to the first-hop switch is consumed by responses rather than by requests. We will discuss the table size for two different systems in § VI-C when we evaluate our mechanisms.

C. Push Reduction

The second design option for in-network reductions uses write requests (push) instead of pull requests. We use different opcodes for the write to an MCR to distinguish between reduction write and multicast write requests. Every GPU injects $m$ elements of data using reduction writes. The packets are then scattered across the network in that every address has a home port in the network fabric to which a packet with a given address is routed to. This guarantees that packets with the same address are routed to the same reduction table.

Contrary to the pull method, the push method does not require synchronization beforehand. GPUs can inject the data as soon as it is ready. The reduction itself synchronizes GPUs as results are returned once all GPUs have sent their data.

The operation of the push method is shown in Fig. 4. It shows the same system as before with four GPUs and multiple switches. Two different data elements with different addresses are reduced, marked $D0$ and $D1$. When the requests first arrive in the network the home port is determined through hashing (9). In the example shown, $P0$ hosts $D0$ and $P1$ hosts $D1$. All requests are routed to the reduction table at their home port. The reduction table in the push method behaves as a fully associative cache. If the first arriving request is able to allocate an entry in the table (9) every subsequently arriving packet with the same address is reduced with the entry and a final result is multicast to every participating GPU (9).

When the reduction table is full and a new packet arrives that does not hit in the table, a Least Recently Used (LRU) policy can evict an existing entry. This eviction is required since we cannot stall requests like in the pull method because this could result in deadlocks. It is crucial for the push method to handle evictions efficiently as the order in which elements are injected is arbitrary, and for large reductions evictions are not rare. An evicted entry can be handled in two ways:

1) Multicast the partial result to every GPUs participating in the reduction.
2) Send the partial result to a ‘home GPU’ which collects all partials for that address and then multicasts the final result to every GPUs participating in the reduction.

The first strategy improves latency but is sensitive to evictions. If these are frequent it will cause many multicast requests being generated at the tables, which increases the load in the network. The second approach is less sensitive to the table size but increases latency. We adopt the latter approach in Fig. 4. Since the table in $P1$ is full, the partial results are sent to the home GPU, here GPU 1 (9), which collects partials, reduces them, and multicasts the result (9).

Either strategy requires the table to count responses, similar to the pull method. However, each packet also needs to carry a count to allow GPUs to do the accounting of partial results. Only when all partials have been received can the GPU multicast the final result. We discuss the GPU’s architectural changes later in § IV-F. In the remainder of this work we assume the second eviction strategy. In addition, if an address is evicted from the table and another operand to the reduction arrives at the table at a later time, the table does not know that this address has been reduced before. Consequently, the table cannot rely on a simple count to determine whether a reduction is final or not. Therefore, entries also have separate timeouts to indicate when they need to be evicted.

Similar to the pull method, we estimate the required table sizes by applying Little’s Law. Every GPU needs to inject all the data, contrary to the pull method in which GPUs inject $m/p$ elements only. Bandwidth is only consumed by the injected writes.

$$C_{\text{min}} = \frac{B_{\text{in}} \cdot d_{\text{max}}}{p} \cdot L_{\text{hop}}$$

(2)

The table size solely depends on the network’s diameter and latency per hop. Furthermore, the push method is able to use all the reduction table in the network, whereas the pull is only guaranteed the GPU’s first-hop switch port.

D. Switch Reduction Tables

Common to both the pull and push method is the reduction table at each switch port to handle the computation and distribution of reduction results. While both reduction methods have different policies for allocating entries in the reduction table, as described in previous sections, the internal architecture of the reduction tables is the same. In our design the reduction
logic is composed of a table and ALUs to perform simple arithmetic and relational operations, as illustrated in Fig. 5.

When a reduction packet arrives at the switch, a lookup in the routing and MCR table (Fig. 5, 1) tells the arbiter that an entry in the reduction table is required (pull method), or which home port in the network the packet needs to be routed to (push method). When a packet arrives at the appropriate reduction table, the address is taken from the header and the reduction table is queried. If it hits, the data from the table is loaded into the operand registers (2) and at the same time the first data flit is forwarded to the second operand register (3). Both operands are then processed in the ALU array and the result is written back to the table (4). Meanwhile the next operands are loaded from the next flit of the packet. If the address does not hit in the table, the behavior is dependent on the reduction method. The push method will insert the flit into an empty entry of the table, while the pull method will either bypass the reduction logic (at intermediate switches) or raise an error at the first-hop switch.

The reduction logic needs to be designed so that a flit can be processed every cycle to support full line rate. Furthermore, the data needs to be aligned to simplify the logic and avoid complex packing and unpacking. If we assume 128B packets and 4B elements, all elements within the packet must be consecutive and the first byte must be aligned to 128B. Packets must also carry a count field, indicating how many operands have been reduced so far. This count field is stored together with the partial result in the table. Once the result is calculated by the ALUs, the count is also compared to a pre-set threshold (5). If the count matches the expected value, the result is written directly to the output of the reduction logic (6) and the table entry is reset to an initial value.

The reduction also needs to implement an eviction policy to prevent deadlocks for the push method. If a packet cannot be inserted due to a full table, the control logic will evict an entry, for example based on a timeout per entry.

E. Design Considerations

1) Switch Architecture: In-network collectives place a greater load on the switch internal bandwidth compared to a baseline switch. For example, assume an $m$ element All-Reduce among $p$ endpoints in a single switch system. Using the pull reduction method, the switch crossbar moves $(p - 1)m$ reduction responses to the reduction tables at the output ports. It also moves $(p - 1)m$ replicated write requests (multicast) to the output ports. In the output port, the number of reduction response packets are reduced to $m$ before leaving the switch. Therefore, the required internal bandwidth speedup $S_{\text{internal}}$ to sustain full output bandwidth is calculated as $S_{\text{internal}} = \frac{2(p-1)}{p} \approx 2$. We can compute a similar ratio for the push reduction method.

HPC switches typically have at least 2x internal speedup in order to sustain peak throughput for arbitrary traffic configurations. For example, Cray’s tiled switch architecture [29] has an 8x speedup. In comparison, our approach requires 2x internal bandwidth, which could be sustained by this architecture.

Much of complexity of network multicast occurs at the flow control and routing level, discussed in the next section. In-switch packet replication can be done using a variety of methods. For example, the switch’s ingress can replicate the multicast into multiple unicast packets in a side buffer before injecting them sequentially into the crossbar. Along with the switch internal speedup, this type of ingress replication is sufficient to sustain the peak bandwidth of the All-Reduce.

2) Deadlock Avoidance: The in-network multicast and reduction operations do not require additional VCs as long as request and responses use different VCs to avoid protocol deadlocks. We guarantee that multicast traffic is free from routing deadlock in three ways. First, cut-through flow control between switches and components within switches guarantees storage for the whole packet before advancing. This avoids many issues associated with multicast and worm-hole routing [30]. Second, we use deterministic routing for both multicast and unicast packets. Finally, the path of multicast packets always conforms to the path of the unicast packets, similar to the Base-Routing-Conformed-Path (BRCP) model [31].

3) Error Handling: Both the push and pull model assume a reliable data transfer layer underneath. That can be given by a link-level re-transmission (LLR) mechanism and sequence numbers to avoid duplicate packets. If a corrupted or duplicated packet is detected it is discarded and the sender issues it again. Permanent and critical errors are handled by software which can either re-configure the network or terminate the application. Larger networks require additional support, such as end-to-end re-transmission, multipathing, and congestion control mechanisms. Other shared-memory networks like Gen-
Z [32] offer solutions for these problems that can be added to NVLink-like GPU networks.

4) Other Considerations: Both of our reduction methods do not guarantee a deterministic order in which data is aggregated, resulting in non-deterministic results for floating point numbers. Both methods can be made deterministic by increasing memory, but our focus is performance and we assume non-deterministic reductions in the following.

The push method as described also prohibits concurrent All-Reduce operations with aliasing pointers. MCRs need to be set up to provide unique addresses.

F. Endpoints

Transfers in shared-memory systems can either be initiated by processor cores or DMA engines. For example, in NVIDIA GPUs we can either issue memory operations from streaming processors (core-initiated) or use the copy engines (DMA-initiated) for bulk transfers. Note that both options do not guarantee any order in which packets are injected into the network. The following discusses both options, as well as how we regulate reduction request injection.

1) Injection Limiting or Wave Synchronization: In-Network multicast replicates packets and therefore leads to increased congestion if we inject traffic without any regulation. In the pull method this will stall the traffic source quickly and create backpressure, negatively affecting other traffic that might need to be injected. In shared systems that may lead to traffic interference. The network will accept the traffic in the push method, but frequent evictions increase congestion and therefore reduce the network’s performance and interfere with other traffic.

We regulate the injection by limiting the number of outstanding reductions. For example, with a total network reduction table capacity $C$, we can ensure that there are never more than $C$ data elements being reduced at the same time. For a reduction size of $m$ elements and $m > C$, we divide $m$ into waves $\{w_i | i \in \mathbb{Z}, 0 \leq i < \left\lceil \frac{m}{C} \right\rceil \}$. Synchronization is required between waves to ensure a wave is completed before the next one is injected. In order to hide this synchronization delay, we size waves to $\left\lceil \frac{r}{C} \right\rceil$, $k \in [1..C]$. We allow $k$ waves to be outstanding at the same time, in a pipeline fashion. Having multiple outstanding waves hides synchronization latencies.

2) Core-initiated Reductions: Although a detailed discussion is beyond the scope of this work, we want to highlight a few things. First, we note that software is less complex for the pull model as threads implicitly wait for the reduction ‘load’ to return before they multicast the result. In the push model, results are written to memory by the network and therefore additional synchronization is required. Second, the number of executed instructions is significantly less than with software algorithms, demanding less cores to be dedicated for communication. Memory accesses are also reduced by at least 50%, saving power and reducing interference within GPUs. This improves overlap with concurrently running kernels and is an additional benefit of in-network reductions.

3) DMA-initiated Reductions: One way of initiating reductions is to use DMA engines, which allows streaming processors to be available to the application for compute tasks, maximizing overlap between communication and computation.

In the pull method, the DMA engine needs to issue the pull requests and the multicast write of the response once it returns from the network. Segmenting the data into waves can be realized by issuing separate DMA requests for each wave. Requests need to be executed sequentially and only after the previous request is completed, requiring synchronization between DMA requests. Furthermore, DMA engines need the ability to re-issue load responses as multicast write requests, ideally by using two DMA engines that run in lock-step.

The push method differs in that it needs to handle partial results created from table evictions. After write requests are issued, the DMA engine needs to count how many responses it receives in order to determine the completion of a wave. Packets need to carry a reduction count indicating how many GPUs contributed to the result. Only when the GPU holds the final result can the multicast write be issued. At the same time, the next wave begins, and new write requests are injected into the network. Partial results are added in memory, similar to how GPUs support in-cache atoms [33].

We propose a light-weight mechanism to augment DMA engines with counters and the ability to withhold credits. This approach is shown in Fig. 6. The DMA engine receives DMA requests from the processor and attempts to allocate a counter in the wave controller. The request generator starts off with no credits. If the counter allocation succeeds, the wave controller returns credits for one wave. As long as the request controller has credits, it continues to issue memory requests either to the local memory system or the network. Responses are counted in the wave controller and only if the count reaches the specified wave size are the credits returned to the request generator. If no counter can be allocated a retry is issued after some time and eventually an error is sent back to the host processor.

Multiple counters are required to allow multiple outstanding waves as described in § IV-F1. Once a response is received, the associated counter needs to be determined. The lookup can be address-based or via a counter ID carried by packets.

G. Implementation Complexity Discussion

While the architecture of the reduction tables is the same in both the push and pull method, the complexity of additional
That would network simulator. The ring-based All-Reduce, evaluation cycles. The generators behaving like the DMA-initiated method (§ IV-F3). The flag is seen. In-network reductions are initiated by traffic to pass larger chunks of data around. Data is consumed once.

For the ring All-Reduce, we use the data-fence-flag semantic of individual memory operations, such as loads and stores. Implemented in NCCL and we found it to deliver the highest bandwidth among the various algorithms at the scale of our simulated systems. Our simulations operate on the granularity of individual memory operations, such as loads and stores. For the ring All-Reduce, we use the data-fence-flag semantic to pass larger chunks of data around. Data is consumed once the flag is seen. In-network reductions are initiated by traffic generators behaving like the DMA-initiated method (§ IV-F3).

The simulated switch has a latency of 150ns and a bandwidth of 25GB/s per port. The switch can sustain all-to-all uniform random traffic at greater than 95% throughput. The switch’s internal architecture is output-queued with two virtual channels to segregate request and response traffic. The output queues are sized to handle the round-trip latency between switches. In addition, we extended the switch by our proposals from § IV-A and IV-D to support the in-network collective acceleration. The maximum packet size is 144B of which 128B are used for the payload and the remaining 16B (1 flit) are reserved for the header. Read requests and write acknowledgments are single-flit packets. The GPU’s memory system is given a static latency of 180 cycles, which is comparable to a L2 hit in a Volta GPU [37]. In order to model cache misses and other dynamic effects, we add an additional randomly chosen latency between [0, 180) cycles.

We study two systems at different scale which are described next. The first system is NVIDIA’s DGX-2, as described in § II. Our approaches are not limited to a 16-GPU system, but could be implemented at larger scale. Therefore, we also simulate a 2-level Fat-Tree system with a total of 128 GPUs and 144 switches. As before, GPUs have 6 network ports and switches have 16 ports. The first level comprises 16 leaf groups with 6 switches and 8 GPUs per group. Each leaf switch is directly connected to each GPU in a leaf group via 1 link. The second level has 48 switches. Each switch in the second level is connected to 1 leaf switch in each of the 16 leaf groups.

V. METHODOLOGY

We simulate our proposals in bksim2, a successor to the booksim network simulator [36]. The ring-based All-Reduce algorithm from § III serves as our baseline, as it is implemented in NCCL and we found it to deliver the highest bandwidth among the various algorithms at the scale of our simulated systems. Our simulations operate on the granularity of individual memory operations, such as loads and stores. For the ring All-Reduce, we use the data-fence-flag semantic to pass larger chunks of data around. Data is consumed once the flag is seen. In-network reductions are initiated by traffic generators behaving like the DMA-initiated method (§ IV-F3).

The simulated switch has a latency of 150ns and a bandwidth of 25GB/s per port. The switch can sustain all-to-all uniform random traffic at greater than 95% throughput. The logic needed in the switch and GPU differs. Nonetheless, the majority of costs stems from the added SRAM in the reduction buffers. Assuming TSMC’s 16nm FFT process, 18kB of SRAM occupies about 0.05mm² [34]. That would allow 1kB of reduction buffer per port and switch chip. For comparison, NVIDIA’s NVSwitch die measures 106mm² [6], resulting in less than 1% area overhead for the reduction buffers and logic. Also note that switch ports already have LLR buffers. For example, a 200Gbps link and a cable length of 15m requires about 4kB LLR buffer.

Another aspect is power consumption. While the ALU and the SRAM accesses add additional power, it is still relatively small compared to the power consumption of serializer and deserializer, which is multiple orders of magnitude higher [35]. Furthermore, the overall system power is not increased as memory accesses and ALU activities are reduced inside the GPU while data is reduced in the network.

VI. EVALUATION

A. All-Reduce Bandwidth

We first evaluate the bandwidth of the push and pull in-network reduction in a DGX-2 system. Bandwidth is calculated as message size divided by simulation time, independent of the number of nodes. The maximal achievable payload bandwidth is 120GB/s. Fig. 7a shows the All-Reduce performance of the pull and push methods, where the pull results are shown with and without synchronization at the beginning. We determine the synchronization cost as an All-Reduce on a flag, which is 1 packet per node. Parameters such as table size and number of waves are selected to achieve optimal performance.

The results show that the pull method achieves almost optimal performance for larger messages across all numbers of participating GPUs. As expected, synchronization reduces bandwidth for smaller messages. The push method scales well overall. In order to achieve optimal performance, the push method needs only 256B reduction table capacity per port, whereas the pull method uses 4kB per port. This is a result of the tight synchronization of our wave mechanism and the fact that the push method can use the entire network’s reduction table capacity efficiently. The explicit synchronization of the pull model has much more impact for smaller message sizes, but this impact is amortized as message sizes increase. The reason that pull cannot achieve the same performance as push even for large messages is due to other overhead such as read request and write acknowledgment control packets that compete for bandwidth with payload packets.

In Fig. 7b, we compare the performance to the software ring All-Reduce algorithm described in § III. The largest benefit of in-network reduction manifests itself at small message sizes,
where we observe speedups up to 18x. This large gain is due to reduced synchronization overhead of the in-network reduction methods when compared to the software solution. For larger All-Reduce operations, the speedup converges to 2x. Here, the performance is mostly sensitive to bandwidth and in-network reductions offers twice the bandwidth of the ring algorithm.

B. Network Scalability

We repeated the All-Reduce evaluation in the Fat-Tree system with 64 and 128 GPUs, as shown in Fig. 8. Performance trends at this scale follow those observed in the DGX system, with the push method outperforming pull for large message sizes. The maximum achieved bandwidth at 64 GPUs is consistent with results at smaller scale and a peak bandwidth of about 100GB/s. In the full-system with 128 GPUs, we note that peak bandwidth is approximately 70GB/s, and therefore lower than at smaller scale. We suspect the cause of this performance anomaly to be congestion, as the simulated network is near capacity and has no congestion control mechanism. Tree-saturation due to congestion can have a wide impact in the network, causing its effective bandwidth to be lower than the theoretical limit.

We also compared the performance to the software ring algorithm in the Fat-Tree system (not shown). For experiments beyond 16 GPUs, and especially small message sizes, the in-network reduction provides orders of magnitude higher bandwidth due to the synchronization overhead of the ring algorithm. However, at that scale, the software implementation of the All-Reduce operation would likely be a combination or hierarchy of ring and tree algorithms to reduce the synchronization overhead. Optimizing the software algorithm is beyond the scope of this work, but regardless of software synchronization improvements, in-network reduction inherently offers about 2x bandwidth advantage over software solutions.

C. Reduction Table Size Sensitivity

Results for the table size sensitivity are shown in Fig. 9. Given table sizes are per port and pull and push use tables differently. We show the DGX-2 system, as its smaller overall reduction table size shows greater sensitivity. We also do not use wave synchronization, which is evaluated separately.

It can be seen that bandwidth in the push method is much more sensitive to the table size than the pull method. That is because the pull method guarantees an entry in the table at the access port and stalls the GPU and injection if resources are exhausted. That has a self-regulating effect. In the push method, however, packets are returned to the GPUs if resources are exhausted, effectively consuming bandwidth and reducing the overall performance. Furthermore, as packets are injected in arbitrary order, tables quickly fill up and capacity misses become frequent. Without any injection limiting, the pull method seems to yield higher bandwidth even though the performance is still low, as the result multicasts barely overlap due to stalled pull requests and congested injection channels.

D. Wave Synchronization

Fig. 10 depicts the results for the pull and push methods with wave synchronization over a varying number of outstanding waves. We chose the Fat-Tree here but observed the same results on the DGX-2 system. We also chose table sizes of 1kB per port for the pull method and 256B for the push method. Although this is rather small, the push method yields high bandwidth with an adequate number of parallel waves. In this example, each wave comprises 8kB, a total of 512kB for 64 outstanding waves, roughly matching the total reduction table size of the Fat-Tree (590kB with 256B/port). On the DGX-2 system, high bandwidth is achieved with 16 waves, a total of 128kB. Although this is about 2.6x the reduction table size of 48kB, wave synchronization allows efficient table utilization.

The pull method also benefits from waves and is able to achieve peak performance with just 4 outstanding waves, a total of 32kB. However, the same experiment with 256B per port (not shown) shows significantly lower bandwidth, despite wave synchronization. This highlights the difference between push and pull in that the push method is able to efficiently use the all the tables, whereas the pull method mostly uses the access port tables. Consequently, ports of symmetric switches need less memory in the push method than in the pull method.

We also measured the average packet latency as an indicator for QoS. On the DGX-2 system with 16 GPUs and an 8MB All-Reduce, wave synchronization reduces the average latency by about 90% compared to no wave synchronization. This further shows the need for injection regulation mechanisms.

When we calculated the required table size for the pull method according to Eq. (1) and a 16 GPU DGX-2 configuration, it indicated the table needs to be about 1.4kB per port (6 access ports) to sustain full bandwidth. This is confirmed by our simulations. The same applies to the push method and Eq. (2), which indicated about 270B per port (192 ports).

E. DL Training

As shown in § II, the All-Reduce can have significant impact on the training time of DNNs. Based on the same models we can evaluate the benefit of in-network reductions in terms of application performance. Fig. 11 shows the speedup of the push method over the NCCL ring algorithm for a DGX-2 system. We model one-shot and overlapped communication and compare the best of both approaches when calculating the speedup. Every kernel launch is assumed a latency of 6μs. Network parameters like link latencies are the same as
described in § V. Large models like Transformer benefit the most, with up to 1.4x faster training on NVLink-connected systems and a speedup of about 1.8x on Ethernet-based systems. ResNet-50, however, shows lower improvements because of its small model size of about 50MB. Projected future GPUs exhibit higher speedups as they are more sensitive to bandwidth. Speedups are also higher for larger scale systems as the sub-batch size decreases. For example, the latest MLPerf v0.6 submission included Transformer on 480 GPUs with a sub-batch of 1280 samples. A sub-batch of 640 is equivalent to doubling the number of GPUs.

The push method’s performance is slightly better than the pull model due to higher bandwidth, but the difference on application level is small and the lower implementation complexity of the pull model remains compelling.

Another interesting observation is that the in-network reduction with layer-wise overlap is faster than one-shot reductions for all data points for Transformer, for example. However, this is not the case with NCCL’s ring algorithm. Here, overlap is only beneficial for token sizes larger than 640 for Volta and larger than 5120 for Volta 4:1. We believe a 4x increase in compute-to-bandwidth ratio to be reasonable as hardware specialization and particularly software improvements progress rapidly. One example is the optimizer, which is the bottleneck at small sub-batches. Our model runs the optimizer on every GPU, but a distributed algorithm can significantly decrease the optimizer time and increase bandwidth sensitivity further.

The All-Reduce size in the smallest Megatron model with 345M and 16-way model parallelism is about 8MB. As we increase the model or the batch size, All-Reduce sizes increase. Based on Fig. 7b, the speedup of in-network reductions is about 2x at these sizes. As shown in Fig. 1c, the All-Reduce takes up to 30% of the step time, resulting in an expected 15% improvement on today’s Volta GPU if we had in-network reductions. We can expect larger benefits on future GPUs.

VII. RELATED WORK

Adding hardware support for collective communication primitives has been proposed before. IBM’s BlueGene [38], [39] system and PERCS [40] interconnect both facilitated accelerated collectives. Anton2 [41] had opportunistic request reservation and response reduction, similar to our pull method. These approaches are tailored to CPUs and message passing, rather than accelerators like GPUs and shared-memory fabrics. More recently, in-switch computation gained more attention as a means to accelerate the gradient All-Reduce for ML/DL.
applications. Sapio et al. [3] propose to use programmable switches to implement a reduction data path. They also propose something similar to our wave synchronization, but implemented in software and without pipelining. Mellanox introduced the SHArP [5] protocol and offers in-network compute acceleration for newer InfiniBand switches. Again, these approaches aim for message passing, requiring queue-pairs to be set up between endpoints and the in-network accelerator, as well as an explicit reservation of resources that comes with too much overhead for a shared-memory fabric. Our proposals can work alongside SHArP, as today’s systems often deploy multiple network domains. Reductions can be done in phases in a pipeline fashion. Li et al. [2] propose to add an accelerator to each switch that performs the reductions. However, our network has much higher bandwidth and higher radix switches. A central accelerator in each switch would quickly become a bottleneck if we wanted to operate at line-rate and adding more data paths is difficult due to wiring limitations in a packed crossbar chip. A rack-level parameter server has been introduced by Lui et al. [42]. The reduction logic is attached to a switch rather than incorporated. Similar to the push method, others have proposed to use the network as a cache [43], [44]. None of these approaches combine that with the reduction as we are proposing. There also exist several proposals to optimize the All-Reduce in software [45]–[51]. Many approaches are orthogonal to our work.

VIII. CONCLUSION

In this work we discussed how in-network computation can benefit the All-Reduce operation that is becoming increasingly important in ML/DL training and in many scientific applications. We presented two different reduction mechanisms for accelerator-centric, shared-memory systems that differ in their complexity. The pull method is simpler in its implementation, but the performance and scalability of the push method is superior if combined with tight synchronization. We showed both mechanisms can be implemented with networks of GPUs and demonstrated that both improve performance and scalability.

Compared to a software ring All-Reduce, the in-network reduction yields up to 2x better bandwidth for larger messages. It further improves bandwidth by 18x for small messages at larger scale, as the ring’s latency becomes a bottleneck. This can lead to DL training speedups of 1.4x on systems that already deploy high-speed interconnects like NVLink.

REFERENCES


