

2024-06-03

# SmartFuse: reconfigurable smart switches to accelerate fused collectives in HPC applications

---

Pouya Haghi, Cheng Tan, Anqi Guo, Chunshu Wu, Dongfang Liu, Ang Li, Anthony Skjellum, Tong Geng, and Martin Herbordt. 2024. SmartFuse: Reconfigurable Smart Switches to Accelerate Fused Collectives in HPC Applications. In Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24). Association for Computing Machinery, New York, NY, USA, 413–425. <https://doi.org/10.1145/3650200.3656616>

<https://hdl.handle.net/2144/49416>

*"Downloaded from OpenBU. Boston University's institutional repository."*



# SmartFuse: Reconfigurable Smart Switches to Accelerate Fused Collectives in HPC Applications

Pouya Haghi  
phaghi@ur.rochester.edu  
University of Rochester  
USA

Cheng Tan  
chengtan@microsoft.com  
Microsoft  
USA

Anqi Guo  
anqigu@bu.edu  
Boston University  
USA

Chunshu Wu  
cwu88@ur.rochester.edu  
University of Rochester  
USA

Dongfang Liu  
dxleec@rit.edu  
Rochester Institute of Technology  
USA

Ang Li  
ang.li@pnnl.gov  
Pacific Northwest National  
Laboratory  
USA

Anthony Skjellum  
askjellum@tntech.edu  
Tennessee Tech University  
USA

Tong Geng  
tgeng@ur.rochester.edu  
University of Rochester  
USA

Martin Herbordt  
herbordt@bu.edu  
Boston University  
USA

## ABSTRACT

Communication switches have sometimes been augmented to process collectives, e.g., in the IBM BlueGene and Mellanox SHArP switches. In this work, we find that there is a great acceleration opportunity through the further augmentation of switches to accelerate more complex functions that combine communication with computation. We consider three types of such functions. The first is *fully-fused* collectives built by fusing multiple existing collectives like Allreduce with Alltoall. The second is *semi-fused* collectives built by combining a collective with another computation. The third are *higher-order collectives* built by combining multiple computations and communications, such as to perform matrix-matrix multiply (PGEMM).

In this work, we propose a framework called SmartFuse to accelerate fused collective functions. The core of SmartFuse is a reconfigurable smart switch to support these operations. The semi/fully fused collectives are implemented with a CGRA-like architecture, while higher-order collectives are implemented with a more specialized computational unit that can also schedule communication. Supporting our framework is software to evaluate and translate relevant parts of the input program, compile them into a control data flow graph, and then map this graph to the switch hardware. The proposed framework, once deployed, has the strong potential to accelerate existing HPC applications *transparently* by encapsulation within an MPI implementation. Experimental results show that this approach improves the performance of the PGEMM kernel, MINIFE, and AMG by, on average, 94%, 15%, and 13%, respectively.

## CCS CONCEPTS

• **Computer systems organization** → *Interconnection architectures; Reconfigurable computing.*

## KEYWORDS

In-Switch Computing, High Performance Computing, FPGAs

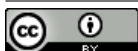
### ACM Reference Format:

Pouya Haghi, Cheng Tan, Anqi Guo, Chunshu Wu, Dongfang Liu, Ang Li, Anthony Skjellum, Tong Geng, and Martin Herbordt. 2024. SmartFuse: Reconfigurable Smart Switches to Accelerate Fused Collectives in HPC Applications. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24)*, June 04–07, 2024, Kyoto, Japan. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3650200.3656616>

## 1 INTRODUCTION

A growing trend in HPC is the increasing importance of the network in application support. Offload of collective processing into SmartNICs [4, 6, 18] is well established and has a number of benefits: first, it enables the bypassing of layers in the communication software stack; second, the hardware implementations are substantially faster than the software; third, it frees up the host processor for other tasks and, potentially, enables better communication-computation overlap; and fourth, some network-host communication is removed as the NIC handles additional send/receive operations. While SmartNICs are valuable, this scheme still forces processing into the endpoints. Another approach is to offload collective processing into the switches [10, 15, 56]. This has two additional benefits: first, latency is improved as computation is distributed rather than performed in a single source (broadcast) or endpoint (reduction); and, second, communication volume may be drastically reduced as messages are quickly merged (reduction) or slowly replicated (broadcast).

Currently, however, switch hardware support for collectives is limited to a small set of scalar operations and data types (e.g., [14, 15]). Moreover, beyond collectives there are additional acceleration opportunities. We hypothesize that it would be beneficial to



This work is licensed under a Creative Commons Attribution International 4.0 License.

ICS '24, June 04–07, 2024, Kyoto, Japan  
© 2024 Copyright held by the owner/author(s).  
ACM ISBN 979-8-4007-0610-3/24/06  
<https://doi.org/10.1145/3650200.3656616>

further augment switches to accelerate additional and more complex functions that integrate communication with computation; we refer to these as fused collective functions (FCFs). We propose in-switch hardware support for three types of FCFs. The first is *fully-fused collectives*, which are built by fusing multiple existing collectives such as Allreduce with Alltoall. The second is *semi-fused collectives*, which are built by combining a collective with a map computation. The third is *higher-order collectives*, which are built by combining multiple communications and computations. An example of a higher-order collective is support for a parallel generic matrix-matrix multiply (PGEMM): usually each node performs local computation and then sends input matrices to its neighbors; with appropriate hardware support for offloading higher-order collectives to switches, much of the computation can occur in transit [1, 20, 52]. Applications that benefit from the specification and acceleration of higher-order collectives include finite element methods [37], iterative solvers [13, 19, 44], and graph algorithms [7, 41].

Adding switch support for FCFs confers additional advantages to in-switch computing. First, additional processing can be removed from the host as the switch manages fused function operations *autonomously*. Second, performance can be improved through handling hardware-accelerated irregular computation and data caching. Third, multi-level communication-computation overlap is achieved: host-network and communication-computation overlap within the switch. Fourth, additional network-host communication is bypassed as multiple communication phases are offloaded to the switch. Finally, the programmer’s model (that is supported efficiently) is extended to reductions on user-defined data types, such as matrices and sparse data.

In-switch support of FCFs has certain requirements. First, to support line-rate communication, collective processing should be done in hardware, rather than, say, a network processor (e.g., ARM cores in SmartNICs). Second, since the processing is both non-trivial and application dependent, this hardware should be (at least partially) reconfigurable. Finally, communication and computation must be tightly coupled. These requirements are currently met by FPGA-augmented switches, e.g., from Arista [39]; by augmenting existing switches with reconfigurable logic; or by using the FPGA itself as a switch (e.g., New Wave [12]). Because of its accessibility, we use the latter approach in this study and demonstrate its utility in a direct network. We also evaluate our approach in an indirect network testbed with FPGA-augmented switches. Other off-the-shelf components may also be plausible, at least with enhancements; e.g. GPUs have been used to offload non-blocking collectives [53].

Multiple challenges need to be addressed. First, while FCFs can be programmed directly, more useful is that they be captured automatically from existing HPC applications. Second, the hardware support for these operations must be transparent to the existing communication middleware (e.g. MPI [16]). And third, FCF support in the switch must remain general-purpose while handling high-bandwidth communication.

Our solution, which we call *SmartFuse*, has two major parts. The first is a software framework that can evaluate and translate the relevant parts of the input program, compile them into a control data flow graph (CDFG), and then map this graph to *SmartFuse* switch hardware. The second is the *SmartFuse* switch hardware itself. This has multiple components for the different types of FCFs.

To support semi/fully fused collectives *SmartFuse* uses a coarse-grained reconfigurable array (CGRA) [54] overlay architecture for packet processing. A second, more specialized unit (we refer to as *kernel logic*) supports higher-order collectives. A feature of *SmartFuse* is that the switch augmentation is *transparent* to the MPI layer; legacy MPI application code is unchanged. The MPI communication library is modified with APIs that provide communication between hosts and *SmartFuse* switch hardware.

Fig. 1 compares *SmartFuse* and prior art (in-NIC and in-switch) with respect to various benefits of offloading computation into the network. The proposed in-switch approach has the potential to obtain gains along multiple dimensions.

We summarize the contributions of this work:

- Demonstrate the advantages of accelerating FCFs with reconfigurable switches (Sec. 3).
- A systematic framework to accelerate FCFs with reconfigurable switches transparent to MPI (Sec. 4).
- A network-optimized CGRA with SIMD and asynchronous execution features to fuse communications and computations (Sec. 5.2).
- A new compute-in-the-switch model to abstract communication and computation and to predict the performance of in-switch computing approaches (Sec. 6).
- Experimental results showing that this approach improves the performance of a variety of HPC applications. In a direct network setting, *SmartFuse* improves the performance of PGEMM, miniFE, and AMG by, on average, 94%, 15%, and 13%, and on an indirect network testbed it improves the performance of finite element method by, on average, 98% (Sec. 7).

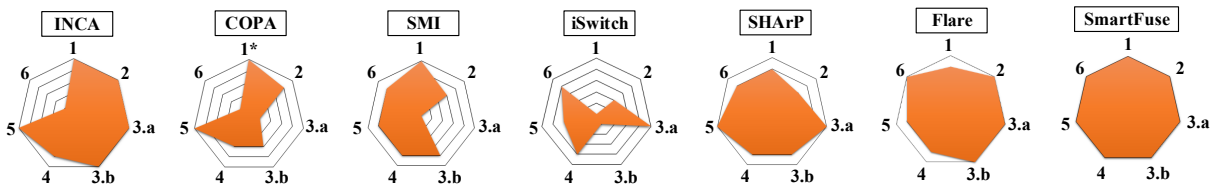
The organization of this paper is as follows. Section 2 gives preliminaries including the definition of FCFs and *SmartFuse* building blocks. Section 3 provides the motivation. Section 4 presents the software model and proposed framework. Section 5 describes the smart switch hardware support. Section 6 introduces an in-switch computing performance model. Section 7 evaluates *SmartFuse*. Related work is discussed in Sec. 8 while Sec. 9 provides a conclusion.

## 2 PRELIMINARIES

### 2.1 Communication-Computation Functions

FCFs provide substantial benefits by reducing communication volume, possibly making computation faster, and, at the same time, enabling overlap between them; they can potentially improve the performance (Sec. 3). That useful FCFs can be extracted and/or constructed, however, is not at all obvious. After experimenting with a variety of applications and kernels we have identified a number of such instances (Sec. 7). The current study examines the following FCFs:

- **Fully-fused collective:** A collective followed by computation (we refer to as *op*) followed by another collective (*collective\_op\_collective*). They are chained together; the receive buffer from the first collective is used during the computation and data generated during the computation is used in the second collective.



**1:** Software Layer Removal **2:** Hardware Implementation Speed **3:** Communication-Computation Overlap **(a)** between Host and Network **(b)** in Network Devices **4:** Network-Host Communication Bypassing **5:** End-to-End Bandwidth **6:** Composite Data Reduction

**Figure 1: Characterizing *SmartFuse* and related work qualitatively with respect to the benefits of collective offload to network devices; *COPA* [33], *INCA* [46], and *SMI* [9] are in-NIC approaches while *iSwitch* [36], *SHArP* [15], *Flare* [10], and *SmartFuse* are in-switch methods. \*Denotes a headless configuration in *COPA*.**

- **Semi-fused collective:** A collective followed by computation (*op*) (collective\_*op*), where the receive buffer from the first collective is used during the computation.
- **Higher-order collective:** A kernel with multiple computations and communications following a well-defined and repeatable communication pattern. A simple example is a distributed dot-product; more complex are distributed matrix-multiply and FFT.

## 2.2 *SmartFuse* Basic Building Blocks

To support general-purpose computation in the switch in conjunction to collective offload, *SmartFuse* is built around three fundamental building blocks: (1) basic collective operation units, (2) a network-optimized CGRA, and (3) kernel logic. The first block realizes primitive collective operations (reduction, gather, and multicast) on incoming packets from different switch ports. The second and the third support semi/fully fused collectives and higher-order collectives, respectively. The second block is inspired by CGRAs with the distinction that it processes packets (CDFGs are mapped to this block). This block enables *SmartFuse* to support loops, which may have conditions. The third block is a more specialized unit consisted of a 2D dataflow-based architecture to support HPC kernels.

## 3 MOTIVATION

We now give an example to illustrate FCFs. Fig. 2 (a) shows an example of a fully-fused collective for NAS parallel benchmark (IS) [40]. Communication (marked with red rectangles) and computation (marked with blue rectangles, referred as *op*) are chained together; the receive buffer in the MPI\_Allreduce (*stream\_in* array in the figure) is used during the computation and the array the computation generates (*stream\_out*) is used as the send buffer by MPI\_Alltoall. In *SmartFuse*, instead of sending the *stream\_in* array back to the host and performing the computation there, the switch performs this computation (*op*), as well as the next communication, without host involvement. Fig. 2 (b) shows another example of a fully-fused collective used in finite element method (FEM) applications [5]. Again, the receive buffer in the first collective routine (MPI\_Allgather) is used for the computation part (lines with blue rectangles) as well as the next communication routine (MPI\_Allgatherv). In this example, the result of the computation part is used as a displacement vector in the second collective during a prefix sum calculation. *SmartFuse* combines the collective-operation-collective as a single

function and offloads it to the switch accelerator; this will save intermediate communications among the nodes as the computation part is handled by the accelerator. Fig. 2 (c) shows the C++ binding for a higher-order collective, sparse matrix-vector multiplication (*SpMV*), in which all of the communications and computations are handled by the switch.

We now define the terminology necessary to characterize semi and fully fused collectives. We refer to *stream-in*, *inbound*, *stream-out*, and *outbound* as, respectively: the receive buffer (produced) of the first collective; a collection of all the array(s) consumed in the computation (but are not produced by the first collective); the updated buffer during the computation and used in the second collective; and a collection of all the array(s) produced in the computation, but not consumed in the second collective. These four arrays are identified by the *SmartFuse* compiler. In addition, computation operations (*op*) are compiled into instructions (stored in *op* buffer in the host, see Fig. 2 (a)), which are transferred to the switch in order to program the CGRA at runtime (Sec. 5.2).

To justify this project, we conducted some preliminary experiments where we compared emulated *SmartFuse* results on an FPGA cluster with the original CPU implementation measured on a CPU cluster with 128 nodes. Fig. 3 (a) summarizes *SmartFuse* improvements over the original CPU implementation with respect to some of the aforementioned in-switch computing benefits for the two FCFs shown in Fig. 2. It is evident that *SmartFuse* provides substantial improvements by fusing communications and offloading them to the switch. In order to further show the potential efficacy of this approach for a variety of semi/fully fused collectives, we evaluated *SmartFuse* with respect to a set of proxy benchmarks. Although these proxy benchmarks are not real HPC applications, they are representative of many communication-computation patterns in applications, including graph algorithms and iterative PDE solvers. Table 1 shows the specifications of these proxy benchmarks. Fig. 3 (b) depicts the average latency (among all ranks) for these benchmarks on a CPU cluster and *SmartFuse* (128 nodes). As is apparent, *SmartFuse* provides considerable improvement. For the CPU cluster specification and simulation setup used for FPGA cluster, refer to Sec. 7.1. The *SmartFuse* results are based on emulation.

## 4 SMARTFUSE SOFTWARE SUPPORT

In this section, we first present an overview of the software model used in this work, and then we describe the proposed framework

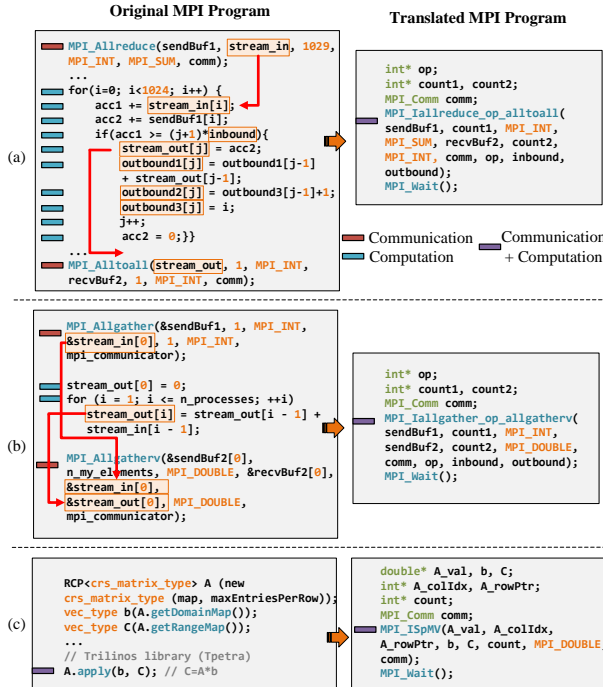


Figure 2: Examples of (a) a fully-fused collective `Allreduce_op_alltoall` in the IS application from the NAS parallel benchmark [40], (b) a fully-fused collective `Allgather_op_Allgatherv` in an open source finite element method library [5], and (c) a higher-order collective `SpMV` from Trilinos Library (Tpetra) [30]. Some variables are renamed from the original code for better readability.

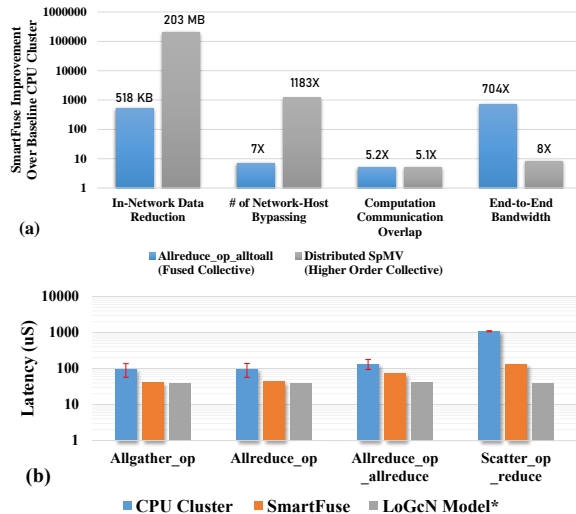


Figure 3: Evaluating *SmartFuse* vs. the original CPU implementation on 128 nodes with respect to (a) some in-switch computing benefits for the two FCFs introduced in Fig. 2, and (b) latency of the proxy benchmarks discussed in Table 1. \* LoGcN model is discussed in Sec. 6. The *SmartFuse* results are based on emulation.

Table 1: Specifications of the proxy benchmarks used.

Proxy Benchmark	Op	Sample Application	Stream-In Size	Inbound Size
Allgather_op	MAC (Multiply-Accumulate)	PAGERANK Algorithm [41]	128	128
Allreduce_op	Accumulation	Sorting Algorithms	512	0
Allreduce_op_allreduce	DOT-PRODUCT + SCALING	Norm Computation in GMRES [57]	1	16K
Scatter_op_reduce	Accumulation	Master-Slave (Workload Distribution)	512	0

that can fuse communications and computations from the input program and construct their offload to the reconfigurable switches.

#### 4.1 Software Model

There are two ways to support reconfigurable network devices for MPI specifications. The first is *offline* [9] where collectives and *ops* are expressed in a high-level language (*i.e.* HLS, OpenCL). In this method, the high-level language is compiled into a bitstream for the hardware device for each MPI application (which is a lengthy process). The other is *online* (runtime reconfigurable) [45] where the reconfigurable network device (*i.e.* the FPGA) is transparent to MPI calls. In this work, we focus on the latter as it avoids regenerating the bitstream for each MPI program while addressing portability across HPC applications [16] without requiring the user to modify the application.

In order to make FPGAs transparent to MPI calls, we have written a transport layer that enables the host to communicate with the FPGA and load the configuration at runtime. As a proof-of-concept, we have experimented with ExaMPI [47]. ExaMPI is a light-weight MPI implementation, which focuses on key blocks of functionality and is designed for modularity and extensibility.

Existing collectives are usually implemented by building on a series of point-to-point operations. FCFs, however, bypass intermediate sends/receives; in this model, each rank has only up to one non-blocking send and receive routine. Other MPI functionality, including MPI rank management for the collective algorithms, is handled by the switch. In our implementation, each rank is associated with a distinct combination of two numbers: an IP address (given to the nodes) and a port number given to each process. This helps to calculate the rank IDs easier from incoming messages at the switch accelerator hardware.

#### 4.2 The Proposed Framework

**4.2.1 Overview.** The proposed framework translates an input MPI program into a new one enhanced with FCF APIs and compiles the relevant parts of the program to the reconfigurable switch. Fig. 4 shows the *SmartFuse* framework. An input MPI program is first fed into a source-to-source translator. The translator begins by parsing the code and inspecting the collectives and *ops* as well as the higher-order collective APIs. If it is a higher-order collective, then it is replaced with a new API (similar to Fig. 2 (c)). Otherwise, in the case of a semi/fully fused collective, an evaluator first evaluates whether fusing the collectives and operations is worth offloading to the network. In case of passing that criterion, a control data flow

graph is generated by *SmartFuse* compiler for the *op* extracted via the parser. We used LLVM infrastructure [35] for this purpose.

The next step is CDFG mapping. There, CDFG operations are mapped into CGRA processing elements (Sec. 5.2) with the aid of a configuration file, which embeds architecture-specific information (number of processing elements, etc). We use modulo routing resource graphs (MRRG) [38] for this mapping (the initiation interval is increased incrementally until a valid mapping is found). Subsequently, instructions are constructed using an instruction generator according to a dictionary of those supported instructions. The collective(s) and *ops* are replaced with new (non-blocking) FCF APIs with the correct arguments including inbound/outbound arrays and a pointer to a data structure holding the instructions. These new APIs are recognized by the MPI implementation. Note that the instructions are loaded into the switch at *runtime* for each semi/fully fused collective. In order to provide overlap between the FCF itself and the rest of computation in the program, an `MPI_Wait` is appropriately placed after it (considering the data dependency). This translated program is now ready to be compiled and launched with the usual MPI flow.

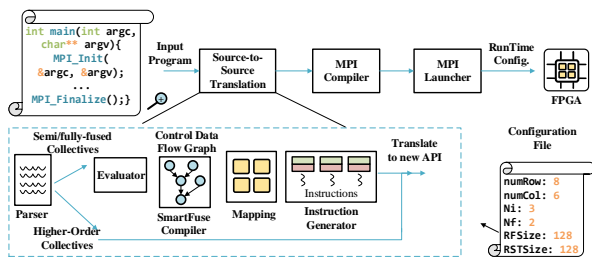


Figure 4: *SmartFuse* Framework

Now, we disclose a number of considerations and details for some of the framework parts.

**4.2.2 Evaluator.** There are two criteria to determine whether a FCF is worth offloading to the network. First, there should be sufficient reuse with at least one loop and sufficient parallelism, preferably larger than SIMD degree of CGRA. Second, inbound/outbound array size should be less than a threshold to lessen the overhead of transferring data to/from switch. We use a threshold of 16 KBytes.

**4.2.3 SmartFuse Compiler.** There are some considerations need to be addressed: (1) The compiler distinguishes the inbound and stream-in data and represents this information in the generated CDFG. (2) The proposed CGRA supports SIMD and reduction between SIMD lanes; this reduction is considered to be one of the operations for the CDFG. Also, index-based *if/else* statements (branches that depend on the loop iteration) should be handled through predication [25] when SIMD is involved.

**4.2.4 Mapping.** The mapper asserts higher priority for inbound data than for stream-in data. This is because stream-in data may arrive at CGRA later than inbound data due to having a communication phase; processing on the latter could be started without having to wait for the former.

## 5 SMARTFUSE HARDWARE SUPPORT

To support FCFs in a generic switch, *SmartFuse* is built around three main components: (1) basic collective support, (2) the CGRA, and (3) kernel logic. Fig. 5 shows the proposed smart switch design: it is based on a virtual output queue (VOQ) design [32] and shown with four input and output transceivers. We now describe each *SmartFuse* building block. Finally, we discuss reliability and deadlock mechanisms of this work. For this section, we consider a direct network where each node has an accelerator and these accelerators are directly connected through a secondary network in a given topology (e.g., 3D torus). Some of the discussions/details, however, are applicable to indirect networks as well (e.g., fat-tree topology).

### 5.1 Basic Collective Support

The basic collectives that *SmartFuse* supports are achieved through these four modules: (I) the collective control module handles communicator and MPI rank management; (II) the reduction unit supports reduce-type operations (`MPI_Reduce` and `MPI_Allreduce`); (III) the gather unit supports gather-type operations (`MPI_Gather`, `MPI_Allgather`, and `MPI_Allgatherv`); and, (IV) the multicast unit supports `MPI_Scatter` and `MPI_Bcast` operations. For the first two types of operations there is a collector module that synchronizes packets from different input ports. The following is an overview of each module.

**5.1.1 Collective Control Module.** To handle MPI rank connectivity for collective algorithms, a communicator table stores the parent-child MPI rank ID relationship. The currently supported collective algorithms are binomial tree [50], recursive doubling [50], and a tree-based algorithm optimized for a 3D-torus topology (used in this work).

**5.1.2 Reduction Unit.** This unit is capable of performing addition, MAX/MIN, and bit-wise AND/OR/XOR operations. This is done through aggregation logic units, each comprising parallel double-precision floating-point/integer ALUs since the phit size (data bitwidth of interface, 512 bits in this work) is wider than that bitwidth.

**5.1.3 Gather Unit.** There is a gather table in which the packets are first stored and then reordered and serialized based on the current MPI rank ID [16] and the rank ID of the child processes in the collective algorithm.

**5.1.4 Multicast Unit.** In order to support scatter-type operations, there is a flag in the packet header that indicates whether the packet is intended to be multicast. The crossbar logic is modified to support multicast operation. At the output of the crossbar logic is a packet re-assembler in which the destinations of the packets are appended to the packet header.

Describing the other two modules briefly: the route-compute unit determines the output port with the aid of communicator table and the packet-parser decodes packets to find the type of transport, operation, datatype, and packet size.

### 5.2 Network-Optimized CGRA

In order to support a variety of workloads for in-switch computing we propose a network-optimized CGRA. CGRAs are typically used

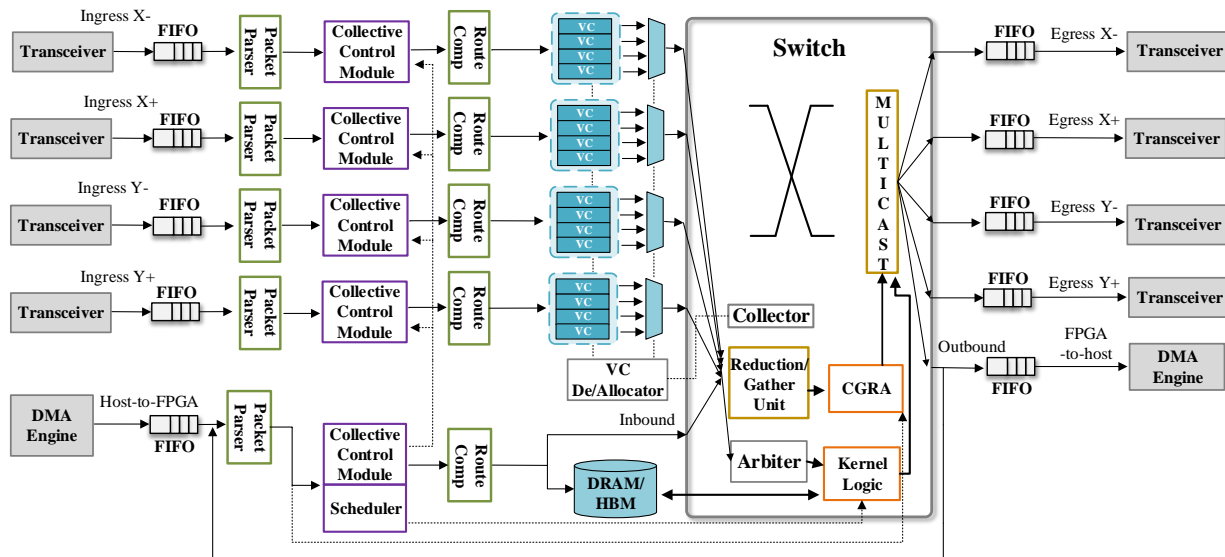


Figure 5: The proposed smart switch design with FCF support.

to accelerate computation-intensive loop kernels. In this work, we consider a dataflow-based CGRA architecture to fuse communication and computation in a streaming fashion. We have modified a traditional CGRA design with dynamic scheduling [54] through a runtime state table (RST) and a finite state machine to efficiently keep track of the loops and operations. To take advantage of the wide phit size, the proposed CGRA benefits from SIMD features with a reduction tree that performs the reduction among SIMD lanes.

As shown in Fig. 6, the proposed CGRA consists of input interfaces (stream-in and inbound); output interfaces (stream-out and outbound); an array of processing elements (PEs) grouped into different types, which are accompanied by register files (RFs); runtime configuration tables (RCTs); and an RST. RCTs store instructions (configurations) designated for PEs (generated by the instruction generator, Sec. 4). These instructions specify the correct operation, operands, RF address, and immediate data (IMM). RCTs are shared among PEs in the same SIMD lane. The RST holds the state of the configuration and determines the correct entry of the RCTs. Inbound buffers (IBs) store inbound data and are addressable. Other interfaces (stream-in, stream-out, and outbound) are streaming.

To maximize end-to-end network bandwidth we applied a number of optimizations. (1) Since stream-in data can arrive at the CGRA later than inbound data (due to having a communication phase), processing on the latter can begin without waiting for the former; this enables *asynchronous execution*. (2) SIMD lanes are used to boost network bandwidth and utilize data parallelism. (3) RFs are utilized to reuse data between loops and to store partial results between stream-in and inbound data.

There are several considerations for the CGRA design:

**5.2.1 Data Streaming.** As the CGRA emulates a network processor, it needs to be able to handle streaming data. At the interface level, this means that off-chip memory is not involved. Instead, on-chip FIFOs are used for the stream-in, stream-out, and outbound interfaces. Similarly, memory accesses for other types of buffers

(inbound, RFs, RST, and RCT) require single clock cycle latency. To achieve this, we have constrained the depth of these memories to *max\_depth*.

**5.2.2 Supporting Conditionals.** To facilitate conditionals, we support partial predication [25], in which both the *if*-path and *else*-path are executed in parallel and the correct outcome is selected by evaluating the condition.

**5.2.3 Supporting a variety of workloads.** To support different types of workloads, PEs inside the CGRA are assigned to be either (1) integer-processing (type-A) comprising integer multiplication, addition, and logical operations, (2) integer to double-precision floating-point conversion (type-B), (3) double-precision floating-point processing (type-C) including multiplication and addition, or (4) post-processing (type-D) such as division and square root. The wide MUXes on the inputs of these PEs enable processing data from different sources (stream-in, inbound buffers, RFs, and immediate data) to chain operations with high throughput.

### 5.3 Kernel Logic

Kernel logic is responsible for handling kernel-level operations (higher-order collectives). A condition that needs to be met to be considered a kernel is that it should have a well-defined and repeatable communication pattern (e.g., matrix multiplication). One of the challenges in designing such a unit for multiple nodes is to tightly couple communication and computation. This is handled by a scheduler unit (configured by the host) which drives the control signals of the kernel unit logic. Fig. 7 shows the proposed architecture. It is divided into two concurrent parts: computation and communication engines. To provide the asynchrony between these two engines, each can write their results to the *output buffers* independently. These two results will be summed up to provide the final output to a *packet assembler* and the corresponding queue once both have valid data. *Overlap registers* are responsible for tuning communication-computation overlap. They control and schedule

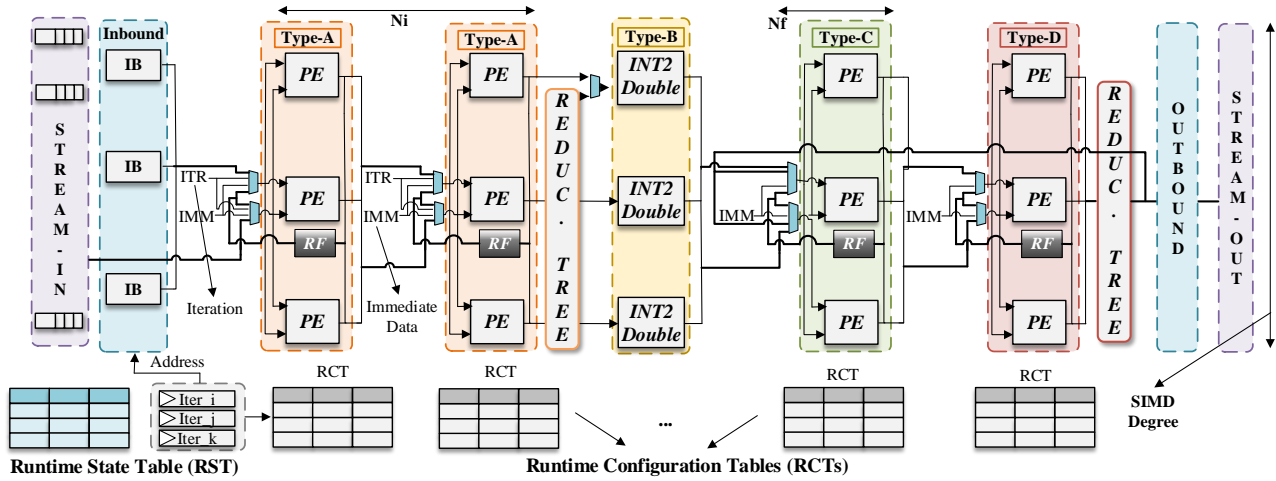


Figure 6: The proposed CGRA architecture; it is based on an array of SIMD-based processing elements.

the relative speed of the two engines. That is, if one engine is ahead at the end of a given period it has to wait for the other. The design is based on a dataflow architecture (horizontal, vertical, and diagonal connections) with double buffers (*D-buffers*) to achieve high off-chip memory bandwidth. While *D-buffers* are used to store the data elements of matrices, *banked buffers* are used to store the elements of vectors (i.e., in a SpMV kernel). The *bank resolver* module resolves the requests issued to *banked buffers* from PEs. One important feature of the kernel logic is that the switch is capable of issuing send/receive operations independent of the host. In other words, the switch is the *master* device.

We illustrate this capability with *SpMV* and *PGEMM*, but it is applicable to similar operations such as matrix transpose, dot-product, convolution, and many others.

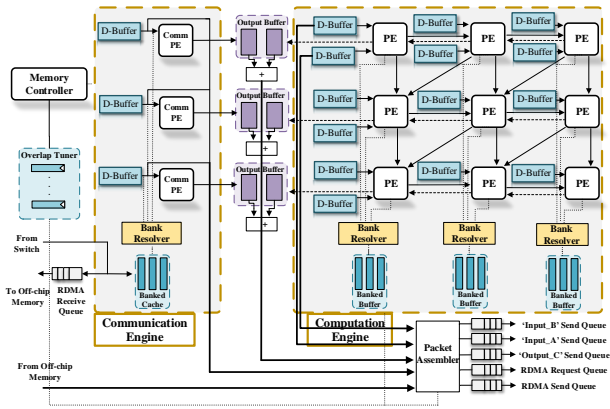


Figure 7: The kernel logic architecture is divided into two concurrent parts: computation and communication engines.

5.3.1 *SpMV*. We consider a row-wise partitioning [42] for the input matrix and input/output vectors across the accelerators and then apply a 2D partitioning scheme among PEs for the local computation. Diagonal (off-diagonal) matrix elements [42] are loaded

from off-chip memory into the *D-buffers* associated with the PEs in the computation (communication) engine, while the vector elements are loaded into banked buffers (see Fig. 7). To illustrate the process, PEs in the computation engine calculate the partial sums which get written into output buffers (Fig. 7). Meanwhile, column indices in the off-diagonal matrix provide the addresses for issuing RDMA requests to remote nodes. After fetching data, partial sums (for off-diagonal elements) are calculated in communication PEs, get written to the output buffers, and are then summed up with the corresponding computation result. To achieve good overlap of communication and computation, data elements that are ready for communication are not sent to remote nodes instantly; rather, they are lumped together (according to a threshold given by overlap tuner registers) and sent later.

5.3.2 *PGEMM*. We apply a pipelined version of the SUMMA algorithm [52]. Since our target topology is 3D-torus, we decompose the workload into 3D grids and assign each grid to an accelerator (i.e., both input matrices are partitioned in a block fashion). For this kernel, the computation engine is used as a systolic array for local computation. Communication is sending/receiving input matrices with neighbor nodes in a ring fashion as well as reductions in another direction (computations and communications are overlapped). It is possible to tune communication and computation by varying the size of memory tiling between matrix dimensions.

## 5.4 Reliability and Deadlock

**Reliability:** For semi/fused collectives, reliability is ensured by pausing and restarting flows to prevent buffers on the switch accelerators from overflowing and dropping frames, respectively. The former happens if the buffer reaches a certain level of fullness and the latter occurs if the buffer empties to below a certain threshold. Ensuring reliability in higher-order collectives in the switch is more challenging than semi/fully fused collectives. This is because the messages received by the switch accelerator can be reused before receiving the next messages in the former. For instance, a single element fetched from remote nodes might be reused multiple times in the SpMV function depending on the distribution of non-zeros

in the matrix. This is in contrast with semi/fully fused collectives where messages are processed in a streaming fashion. The fact that messages are reused in higher-order collectives delays the task of receiving the next message(s). This, in turn, increases the chance of messages being dropped. However, it is possible to take advantage of communication-computation overlap in higher-order collectives. To ensure the reliability of messages while achieving higher performance, we set a deadline in which each received message has limited time to be processed before being able to listen to the next message's arrival. This deadline sets the threshold for overlapping communication and computation, as well as buffer sizing. In the case of buffer overflow, new incoming packets are dropped, the downstream notifies the upstream, and the upstream retransmits the message.

**Deadlock:** To avoid deadlock for semi/fused collectives, we designed SmartFuse hardware such that receiving packets do not block sending packets and vice-versa. At the software level, we use nonblocking sends/receives. For higher-order collectives, if one process is significantly ahead of the others (process-skew [21]), the accelerators may attempt to send a message. However, if it has not received any messages, it will pause and await synchronization within the network. Also, we note that we allocate enough buffer on the hardware for each rank separately.

## 6 MODEL FOR IN-SWITCH COMPUTING

To develop a fundamental understanding of compute-in-the-network, and to better analyze its performance, a simple yet accurate model is indispensable. The LogGP model [2] has primarily been used to abstract communication based on point-to-point messages. In this model,  $L$ ,  $o$ ,  $g$ ,  $G$ , and  $P$  represent the latency, overhead, gap between messages, gap per byte, and the number of processors, respectively. Although collectives could be represented in this model as a sequence of point-to-point communications, it is not able to model FCFs. Also, it is not possible to model computation in LogGP. We have therefore created a new version of this model, called LoGcN, to account for in-network computing on a collection of processes while also considering communication-computation overlap.

One benefit of the in-switch approach is that it eliminates the  $g$  term by pipelining the communication. For example, for MPI\_Bcast, the message is sent from the host to the switch only once. And for MPI\_Scatter, different messages are grouped into a single large message. Consequently, in our model  $g=0$  and we need only the  $G$  term for both short and long messages. Another advantage is that it effectively eliminates the overhead time (term  $o$ ) for intermediate ranks in a given collective algorithm (e.g., binomial tree) since the switches are able to perform reduction, gather, and multicast on the packets without the aid of the hosts.

We model the execution time of FCFs with overhead, latency (which depends on the number of switches), gap per byte (bandwidth), and an added term for computation (represented as  $c$ ). Computation itself comprises a latency term and another term for characterizing the message bandwidth. By modeling computation with latency and bandwidth terms we have abstracted the frequency of the switch logic, number of PEs, and other design details. To illustrate, we first consider the following equations:

$$T = (2 \times o) + L(N) + \frac{m}{G} + c(N, G) \quad (1)$$

$$c = (N \times \alpha) + \frac{(\gamma - 1) \times m}{G} \quad (2)$$

$$G = \beta \times BW \quad (3)$$

where  $T$ ,  $L$ ,  $N$ ,  $m$ , and  $BW$  are the time it takes to complete a FCF on a collection of processes, the maximum latency of the network from sending the first byte from a host to receiving the first byte by another host, the number of switches involved, the message size, and the maximum network bandwidth. The message size is the size of stream-in (array size sent from each host to the switch) in the case of fused collectives (higher-order collectives).  $\alpha$ ,  $\beta$ , and  $\gamma$  represent the latency of the computational units (*SmartFuse* building blocks), a bandwidth attenuating factor in case of gather-type or MPI\_Scatter communication (due to serialization), and data reuse for computation, respectively.

Note that coefficient 2 in Eq. 1 is needed due to the fact that the message is only transferred from (to) a sending (receiving) process on the host. Also, that communication latency is merged into term  $L$  instead of a series of point-to-point communications. For computation term  $c$ , we have considered a data reuse term: for kernel logic, this represents the amount of reused data; for the CGRA it means initiation interval (*II*). To represent the overlap of communication and computation, this term is reduced by one and used as a factor in the bandwidth term of the computation.  $\alpha$ ,  $\beta$ , and  $\gamma$  depend on the workload (kernel) but  $L$ ,  $o$ ,  $BW$ , and  $n$  are parameters that can be extracted. Our preliminary results show that the LoGcN model is effective in predicting the performance of semi/fully fused collectives (Sec. 3).

## 7 EXPERIMENTAL EVALUATION

In this section, we evaluate *SmartFuse*. We present the experimental setup, the resource utilization of the smart switch, performance and scalability of *SmartFuse* for HPC kernels and applications, and, finally, we evaluate SmartFuse in an indirect network testbed.

### 7.1 Experimental Setup

**SmartFuse Direct Network:** For proof-of-concept, we have implemented and tested *SmartFuse* on a two-node FPGA-based system in both direct and indirect network settings using the Xilinx *Vitis* unified software platform. In the direct network testbed, two Alveo U280 boards are directly connected using QSFP28 network interfaces (capable of 100 Gb/s). Each board is connected to an Intel Xeon E5-2620V2 server. To simulate a larger number of nodes (where SmartFusion provides performance benefits), we conduct an experiment to obtain some parameters used for the emulation of a larger-scale proxy system. In this experiment, a sender process sends 1408 bytes worth of data to a receiver process using TCP/IP network logic [28] handled by FPGAs. ExaMPI implementation [47] is used for this experiment. The parameters used in the emulation and LoGcN model (derived from our system setup) are shown in Table 2. MPI overhead is the average overhead of MPI\_send and MPI\_Recv in ExaMPI. ExaMPI overhead is larger than other MPI implementations due to the cost of creating progression threads. The last three parameters are used to evaluate the term  $L$  in the LoGcN

**Table 2: Parameters used in the emulation and LoGcN model.**  
\* Denotes the Aurora IP latency.

MPI Overhead (o)	14.8 usec
Maximum Network Bandwidth (BW)	95.9 Gbps
PCIe Latency	0.9 usec
FPGA-to-FPGA Latency*	0.44 usec
Minimum Port-to-Port Latency	52 nsec

model. We note that the latency of the *SmartFuse* components is not included in the reported port-to-port latency.

For the rest of this section, we evaluate the performance of *SmartFuse* based on the emulator as well as the LoGcN model. The emulation has the same requirements as [36]. That is, the emulation should possess: (1) the same volume of traffic in the network links, (2) an identical number of network hops, and (3) an accurate overhead of the accelerator. For the *SmartFuse* accelerator overhead, we use cycle-accurate RTL simulation through testbenches using the Xilinx Vivado Tool. We emulate an FPGA cluster (Xilinx VCU128) with up to 128 nodes with a direct network (3D-torus topology).

**SmartFuse Indirect Network:** We also evaluate our approach for an indirect network. The testbed is a two-node system on Cloud-Lab [26, 27] with a Xilinx Alveo U280 FPGA attached to a Dell Z9100-ON switch (total of three nodes including host). We use the Xilinx Vitis 2021.2 unified software platform to program the FPGA. Our accelerator is coupled with a modified version of [55] to send/receive packets from two leaf nodes. The operating frequency is 250 MHz. At the leaf nodes, packets are sent and received using socket APIs to communicate with the FPGA through the switch.

**CPU Implementation:** For the CPU reference, benchmarks were run on the TACC Stampede2 [48] Skylake (SKX) compute cluster with 48-cores per node (2 sockets) 2.1 GHz Intel Xeon Platinum 8160 CPUs, and a 100 Gb/s Intel Omni-Path (OPA) network. We used Intel MPI 18.0.2 as an Intel-compatible MPI is recommended for this cluster; we found it usually gives better performance than other MPI implementations.

## 7.2 Workloads Tested

The workloads tested comprise a number of standard HPC benchmark kernels and a complete application. The baselines are the unmodified codes run as described in the previous subsection.

First are two HPC kernels: SpMV from Trilinos Library (Tpetra package) [30] and PGEMM from the state-of-the-art COSMA algorithm [34]; they are the backbone of many scientific, graph analytic, and machine learning applications. Second, are a set of applications from the NAS parallel benchmark [40]: IS (Integer Sort), LU (Lower-Upper Gauss-Seidel solver), MG (Multi-Grid on a sequence of meshes), and SP (Scalar Penta-diagonal solver). Third, from the Mantevo project [29] is miniFE [3], which is an unstructured implicit finite element code. Fourth, is *deal.II* [5] from an open-source finite element method (FEM) library for solving partial differential equations. Finally, a complete application Algebraic Multigrid (AMG), which is a widely used iterative solver with irregular communication, is also studied. Table 3 shows FCF types of each benchmark/application used in this work.

**Table 3: Fused collective types for benchmarks/applications**

Benchmark/ Application	Fully-Fused Collectives	Semi-Fused Collectives	Higher Order Collectives
SpMV (Sparse matrix vector multiplication)	✗	✗	✓
PGEMM (Dense matrix multiplication)	✗	✗	✓
IS (Integer sort)	✓	✗	✗
LU (Lower-upper Gauss-Seidel solver)	✗	✓	✗
MG (Multi-grid)	✓	✗	✗
SP (Scalar penta diagonal solver)	✗	✓	✗
miniFE (unstructured grid)	✗	✓	✓
deal.II (finite element method)	✓	✗	✗
(AMG) Algebraic multigrid	✗	✗	✓

Collectively a variety of application types are represented, including dense matrix-matrix multiplication, sparse matrix-vector multiplication, structured grids, and unstructured grids.

## 7.3 Performance and Scalability

In this subsection, we evaluate the performance and scalability of *SmartFuse* for two kernels and a variety of applications.

**7.3.1 Higher-order collectives.** We consider two higher-order collectives with different sizes: SpMV and PGEMM. For SpMV, we study two sparse matrices from *SuiteSparse* [51]: *parabolic\_fem* (for SpMV-1) and *sme3Dc* (for SpMV-2). The former (latter) has 525825 (42930) rows with an average of 7 (73) nonzero elements per row. For comparison we use Trilinos [30] (Tpetra package) as the CPU reference. For PGEMM, we consider a square (16K×16K) and a tall-skinny (1K×128K) dense matrix and compare *SmartFuse* with the state-of-the-art COSMA [34] CPU implementation. We consider strong scaling for both PGEMM and SpMV. Also, the former is based on single-precision floating-point while the latter is double-precision.

Fig. 8 shows a performance comparison of *SmartFuse* and the original MPI implementation on the SKX cluster for SpMV and PGEMM kernels and also the performance estimated by the LoGcN model. SKX-1, SKX-24, and SKX-48 denote having 1, 24, and 48 OpenMP threads, respectively. For the PGEMM kernel, *SmartFuse* provides a considerable improvement as it is able to almost *fully* overlap communication and computation in the switch: *SmartFuse* provides 3.2× and 1.5× performance improvements compared with an optimized CPU implementation [34] for square and tall-skinny matrices, respectively. Similarly, for the SpMV kernel, *SmartFuse* delivers significant improvement as the whole kernel (irregular computation with series of MPI\_Sendrecv realized by RDMA) is handled by the reconfigurable switch. The problem is computation bound with a small number of nodes, which gradually becomes

communication bound for a larger number of nodes. Note that *SmartFuse* is able to maintain the speedup with scale.

We also estimated the performance using the LogGP model. However, it consistently provided smaller latencies than the actual emulated performance model (red bars) for higher-order collectives. This is because LogGP does not model the computation time in the network, and so is not suitable for modeling higher-order collectives. We note that another approach for the LoGcN modeling is to choose the value of the parameters to give the best fit for a training set of experimental results, rather than those shown in Table 2. While this approach can provide better estimations, we avoid complications in the performance modeling process in this work and resort to a lightweight model with the same procedure as LogGP.

**7.3.2 Applications.** Fig. 9 shows the performance benefit of *SmartFuse* over original MPI implementation on the SKX cluster (64 and 128 nodes) for the NPB and *MINIFE* proxy applications. The problem size used for IS and MG is based on class C and for LU and SP is class A [40]. SKX time and error bars represent the time it takes on SKX cluster and *std* for five runs. BENCHMARK-X-TY represents the benchmark with X nodes and Y OpenMP threads.

We find that the LoGcN model can effectively predict the performance of NPB, but not *MINIFE* (Fig. 9). One reason is that this model is not able to predict the performance of higher-order collectives used in *MINIFE*; these kernels often have complex communication and computation behavior, which may not be feasible to abstract them using a single model. Another reason is that there is a semi-fused collective with a large *inbound* array in *MINIFE*. But LoGcN model does not currently take *inbound* arrays into account.

According to Fig. 9, among NPB applications, the performance benefits for MG and IS are higher than for the others. For IS, one reason is that the message size of collectives is relatively high and *SmartFuse* can take advantage of communication-computation overlap and in-network data reduction. For MG, this is partly because there are a larger number of FCF calls. In contrast, SP manifests the smallest performance gain simply because the number of FCF calls is the smallest. Also, note that it is possible to utilize non-blocking FCFs, taking advantage of CPU idle time during the FCF offload. This happens for one of the FCFs in LU benchmark.

For *MINIFE*, the performance improvement percentage is typically higher than that of NPB. One reason is that higher-order collectives (SPMV), in addition to semi-fused collectives, are accelerated by the switch. Other reasons are the fact that semi-fused collectives constitute a larger fraction of runtime and it is possible to benefit from a non-blocking FCF.

Fig. 10 shows the performance and scalability of *SmartFuse* vs. the optimized CPU implementation (Hypre [31]) and SHArP [15] for AMG. We used the *lap3d* matrix [42] with size 1M rows. The numbers in front of SKX and *SmartFuse* denote the number of threads. The red bar indicates time saved by overlapping communication and computation in the reconfigurable switch for the SPMV kernel in *SmartFuse*. Note that by scaling up (weak scaling) the execution time increases gradually as more time is needed for the communication. However, there are some anomalies in which scaling up improves the performance. This is because the solver sometimes converges with fewer iterations. The improvement introduced by *SmartFuse* depends on the ratio of the time spent in the SPMV kernel

to that of total execution time. On average (among different numbers of nodes), *SmartFuse* improves the AMG execution time by 11%. For comparison with SHArP, we have calculated the performance improvement of MPI\_Allreduce times (supported by SHArP) in AMG. We superimposed their results published in [15] in the AMG application. Our approach provides considerable improvement compared with SHArP (see the lines for *SmartFuse* speedups that are above the SHArP speedup line in the figure).

## 7.4 Indirect Network Study

To evaluate the efficacy of the approach in indirect network settings we use CloudLab infrastructure [26] with three nodes (two leaf nodes and one node to host the FPGA) capable of 100 Gbps. A 100 Gbps switch interconnects all three nodes. Each process, two in this case, in addition to the FPGA itself, is assigned IP address and port number. This information is stored in the networking kernel of the FPGA to forward the messages to the correct destination according to the collective type and algorithm. Messages are sent from the leaf nodes to the FPGA through the switch, processed in the FPGA user kernel, and sent back to the corresponding leaf node(s). We also provide a runtime that automates and manages the execution of processes in basic/fused collectives. This includes connecting to leaf nodes from the master process (through SSH), creating processes there, assigning new port numbers for each process, and waiting for the completion.

Since the runtime and MPI support are based on Python, we compare this approach with a Python-based MPI, MPI4py [8]. We demonstrate its performance compared to traditional MPI for an instance of fused collectives used in FEM applications. Figure 11 shows the latency comparison of Allgather\_op\_allgather in both MPI4py *SmartFuse* for different message sizes. *Op* here is a prefix sum (see Fig. 2 (b)). The results are from taking the average of five runs. It clearly shows that *SmartFuse* provided superior performance, especially for larger message sizes with, on average, a 1.98× improvement. The performance benefit comes from the fact that intermediate communications are bypassed and computations sandwiched between collectives—are directly processed in the accelerator on FPGA.

## 8 RELATED WORK

**Collective offload:** Previous work has shown significant benefits of optimizing collectives and offloading them to the NIC. The authors in [4] present a framework for offloading MPI collectives to programmable logic on the NIC. The work in [36] integrates lossy compression into FPGA-based NICs to accelerate the distributed training of deep neural networks. Also, COPA [33] provides a software/hardware framework that makes the underlying FPGA hardware (endpoint device) agnostic to middleware. There is another line of work that targets offloading collectives to the switch. It is worth mentioning that perhaps in-switch processing originated from the NYU-Ultra [11] implemented in the distant past. Mellanox SHArP [15] has offloaded MPI collectives to ASIC-based switches using reduction trees. Their approach supports fixed functions and data types with no extensibility; also, few design details are provided. In contrast, the approach here provides support for user-defined collectives and other complex functions (FCFs). The authors

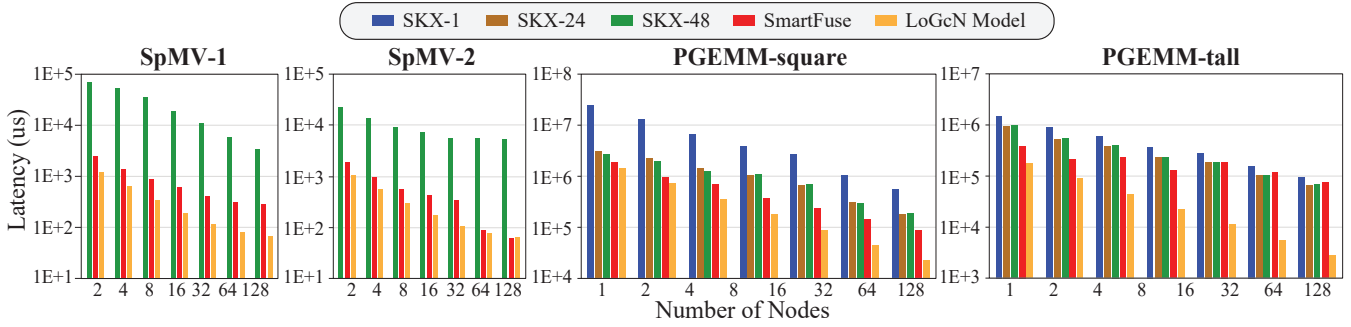


Figure 8: Scaling comparison of higher-order collectives on SKX vs. *SmartFuse*. The two left-most plots are the *SpMV* kernel (parabolic\_fem and sme3Dc matrices) and the two right-most are the *PGEMM* kernel (square and tall matrices). 1, 24, and 48 OpenMP threads were used for the CPU SKX cluster.

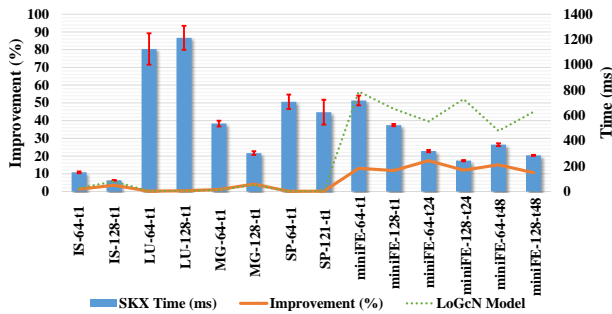


Figure 9: Performance improvement of *SmartFuse* over original MPI implementation on SKX cluster (64 and 128 nodes) for the NAS parallel benchmarks and *miniFE*. SKX time and error bars represent the time it takes on SKX cluster and *std* for five runs, respectively.

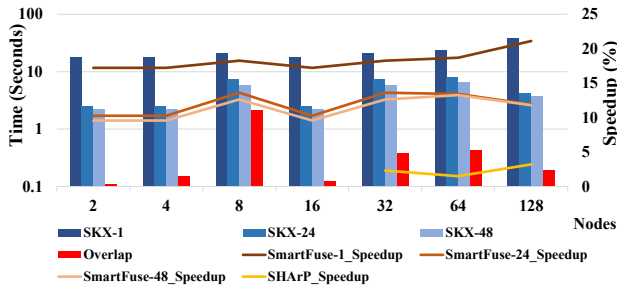


Figure 10: AMG performance and scalability comparison between *SmartFuse*, the optimized CPU implementation, and *SHARp* (with data adopted from [15]) for 2 up to 128 nodes with different number of threads (1, 24, and 48). “*SmartFuse-1\_Speedup*” is the speedup of our approach compared to SKX-1. Similarly, “*SmartFuse-24\_Speedup*” is the speedup of our approach compared to SKX-24.

in [36] propose an FPGA-based in-switch acceleration scheme for distributed reinforcement learning to move gradient aggregation from server nodes to the network switches. The authors in [22, 23]

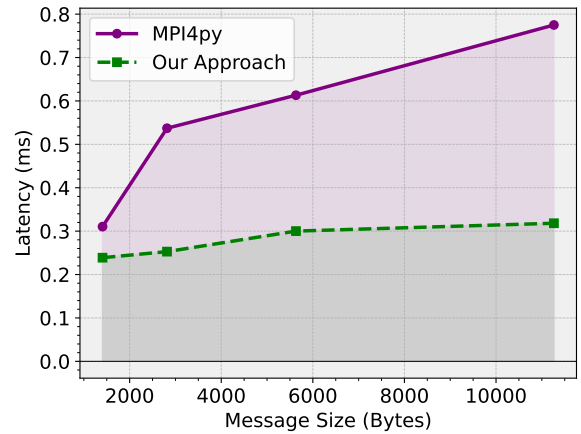


Figure 11: Latency comparison of *Allgather\_op\_Allgather* in *MPI4py* and our approach. *Op* is prefix sum (see Fig. 2 (b)). The X-axis shows the message size in bytes used for *Allgather*s and the Y-axis shows the latency in milliseconds.

designed a hardware accelerator to offload basic MPI collectives to the reconfigurable switch with flexible communicator support.

**General-purpose in-network processing:** INCA [46] proposes a compute assistance by building upon offload capabilities of state-of-the-art NICs to support a general-purpose computation in the network. The work [17] proposes a general-purpose data-centric computing framework for SmartNIC-based systems to accelerate various types of neural network kernels. However, these are in-NIC approaches (limitations discussed in Sec. 1). In [10], the authors design a flexible programmable switch architecture for in-network data reduction. Although it is possible to process custom operations through packet handlers, their evaluation is only limited to dense/sparse *MPI\_Allreduce*. The authors in [24] propose a programmable look-aside-type accelerator that can be embedded into, or attached to, existing communication switch pipelines and that is capable of processing packets at line rate. However, they do not support fused collectives. Also, supporting new applications in their approach is not automated from the MPI code; the user has to manually find out which parts of the application need to be off-loaded and compile them separately. The work in [49] adds custom

hardware based on a MapReduce pattern/abstraction (built upon a CGRA) to P4 switches to enable per-packet inference of machine learning. However, the fusion of collectives is not supported in this work. Prabhakar et al. [43] proposed a general-purpose architecture as a collection of compute and memory units to efficiently execute applications composed of parallel patterns. However, this work is based on a single-node acceleration (not distributed computing).

To the best of our knowledge, no prior work proposes a general-purpose framework that could be able to fuse MPI collectives without user involvement.

## 9 CONCLUSION

In this work, we propose a general-purpose, MPI-transparent, framework for in-switch computing in reconfigurable devices to accelerate three fused collective functions (FCFs). First, we propose that currently supported collectives in switches are extendable not just to communication phases but to series of communications and computations by identifying them in HPC applications. Then, we provide a framework to translate MPI programs and compile them into reconfigurable devices without user effort. Finally, we have designed various hardware building blocks to support FCFs. Our experimental results show that our approach on an FPGA cluster achieves on average 94%, 15%, and 13% improvement in execution time as compared to the original implementations running on TACC Stampede2 cluster for the PGEMM kernel, MINIFE, and AMG, respectively.

## ACKNOWLEDGMENTS

This work was supported, in part, by the NSF through awards CCF-1919130, CCF-2151021, and CCF-2326494; and by AMD and Intel both through donated FPGAs, tools, and IP. This research was also partially supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award No.66150: “CENATE - Center for Advanced Architecture Evaluation” and No.78284: “ComPort: Rigorous Testing Methods to Safeguard Software Porting”.

## REFERENCES

- [1] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* 39, 5 (1995), 575–582. <https://doi.org/10.1147/rd.395.0575>
- [2] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. 1997. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. *J. Parallel and Distrib. Comput.* 44, 1 (1997), 71–79. <https://doi.org/10.1006/jpdc.1997.1346>
- [3] ECP Proxy Applications. 2023. miniFE. <https://proxyapps.exascaleproject.org/app/minife/>.
- [4] Omer Arap and Martin Swamy. 2016. Offloading Collective Operations to Programmable Logic on a Zynq Cluster. In *2016 IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI)*. 76–83. <https://doi.org/10.1109/HOTI.2016.024>
- [5] Daniel Arndt, Wolfgang Bangerth, Maximilian Bergbauer, Marco Feder, Marc Fehling, Johannes Heinz, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Peter Munch, Jean-Paul Pelteret, Bruno Turcksin, David Wells, and Stefano Zampini. 2023. The deal.II Library, Version 9.5. *Journal of Numerical Mathematics* 31, 3 (2023), 231–246. <https://doi.org/10.1515/jnma-2023-0089>
- [6] M. Bayatpour, N. Sarkauskas, H. Subramoni, J. Maqbool Hashmi, and D. K. Panda. 2021. BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs. In *High Performance Computing*. Springer International Publishing, 18–37.
- [7] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing Breadth-First Search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–10. <https://doi.org/10.1109/SC.2012.50>
- [8] Lisandro Dalcin and Yao-Lung L. Fang. 2021. mpi4py: Status Update After 12 Years of Development. *Computing in Science & Engineering* 23, 4 (2021), 47–54. <https://doi.org/10.1109/MCSE.2021.3083216>
- [9] Tiziano De Matteis, Johannes de Fine Licht, Jakub Beránek, and Torsten Hoefer. 2019. Streaming message interface: high-performance distributed memory programming on reconfigurable hardware. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Denver, Colorado) (SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 82, 33 pages. <https://doi.org/10.1145/3295500.3356201>
- [10] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefer. 2021. Flare: Flexible in-Network Allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (St. Louis, Missouri) (SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 35, 16 pages. <https://doi.org/10.1145/3458817.3476178>
- [11] Susan Dickey and R. Kenner. 1992. Combining switches for the NYU Ultracomputer. 521 – 523. <https://doi.org/10.1109/FMPC.1992.234864>
- [12] New Wave DV. 2024. 32-Port Programmable Switch. <https://newwavedv.com/products/appliances/32-port-programmable-switch/>.
- [13] Robert D. Falgout. 2006. An Introduction to Algebraic Multigrid. *Computing in Science and Engg.* 8, 6 (Nov. 2006), 24–33. <https://doi.org/10.1109/MCSE.2006.105>
- [14] Ahmad Faraj, Sameer Kumar, Brian Smith, Amith Mamidala, and John Gunnels. 2009. MPI Collective Communications on The Blue Gene/P Supercomputer: Algorithms and Optimizations. In *2009 17th IEEE Symposium on High Performance Interconnects*. 63–72. <https://doi.org/10.1109/HOTI.2009.12>
- [15] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldener, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Werthim, and Eitan Zahavi. 2016. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. 1–10. <https://doi.org/10.1109/COMHPC.2016.006>
- [16] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.* 22 (1996), 789 – 828.
- [17] Anqi Guo, Tong Geng, Yongan Zhang, Pouya Haghi, Chunshu Wu, Cheng Tan, Yingyan Lin, Ang Li, and Martin Herbordt. 2022. A Framework for Neural Network Inference on FPGA-Centric SmartNICs. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 01–08. <https://doi.org/10.1109/FPL57034.2022.00071>
- [18] Anqi Guo, Yuchen Hao, Chunshu Wu, Pouya Haghi, Zhenyu Pan, Min Si, Dingwen Tao, Ang Li, Martin Herbordt, and Tong Geng. 2023. Software-Hardware Co-design of Heterogeneous SmartNIC System for Recommendation Models Inference and Training. In *Proceedings of the 37th International Conference on Supercomputing (Orlando, FL, USA) (ICS '23)*. Association for Computing Machinery, New York, NY, USA, 336–347. <https://doi.org/10.1145/3577193.3593724>
- [19] Pouya Haghi, Tong Geng, Anqi Guo, Tianqi Wang, and Martin Herbordt. 2020. FP-AMG: FPGA-Based Acceleration Framework for Algebraic Multigrid Solvers. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 148–156. <https://doi.org/10.1109/FCCM48280.2020.00028>
- [20] Pouya Haghi, Anqi Guo, Tong Geng, Justin Broaddus, Derek Schafer, Anthony Skjellum, and Martin Herbordt. 2020. A Reconfigurable Compute-in-the-Network FPGA Assistant for High-Level Collective Support with Distributed Matrix Multiply Case Study. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 159–164. <https://doi.org/10.1109/ICFPT51103.2020.00030>
- [21] Pouya Haghi, Anqi Guo, Tong Geng, Anthony Skjellum, and Martin C. Herbordt. 2021. Workload Imbalance in HPC Applications: Effect on Performance of In-Network Processing. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–8. <https://doi.org/10.1109/HPEC49654.2021.9622847>
- [22] Pouya Haghi, Anqi Guo, Qingqing Xiong, Rushi Patel, Chen Yang, Tong Geng, Justin T. Broaddus, Ryan Marshall, Anthony Skjellum, and Martin C. Herbordt. 2020. FPGAs in the Network and Novel Communicator Support Accelerate MPI Collectives. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–10. <https://doi.org/10.1109/HPEC43674.2020.9286200>
- [23] Pouya Haghi, Anqi Guo, Qingqing Xiong, Chen Yang, Tong Geng, Justin T. Broaddus, Ryan Marshall, Derek Schafer, Anthony Skjellum, and Martin C. Herbordt. 2022. Reconfigurable switches for high performance and flexible MPI collectives. *Concurrency and Computation: Practice and Experience* 34, 6 (2022), e6769. <https://doi.org/10.1002/cpe.6769> <https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6769>
- [24] Pouya Haghi, William Krska, Cheng Tan, Tong Geng, Po Hao Chen, Connor Greenwood, Anqi Guo, Thomas Hines, Chunshu Wu, Ang Li, Anthony Skjellum, and Martin Herbordt. 2023. FLASH: FPGA-Accelerated Smart Switches with GCN Case Study. In *Proceedings of the 37th International Conference on Supercomputing (Orlando, FL, USA) (ICS '23)*. Association for Computing Machinery, New York, NY, USA, 450–462. <https://doi.org/10.1145/3577193.3593739>

- [25] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. 2014. Branch-aware loop mapping on CGRAs. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.
- [26] S. Handagala, M.C. Herbordt, and M. Leeser. 2021. OCT: The Open Cloud FPGA Testbed. In *31st International Conference on Field Programmable Logic and Applications (FPL)*.
- [27] S. Handagala, M. Leeser, K. Patle, and M. Zink. 2022. Network Attached FPGAs in the Open Cloud Testbed (OCT). In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 1–6.
- [28] Z. He, D. Korolija, and G. Alonso. 2021. EasyNet: 100 Gbps Network for HLS. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE Computer Society, Los Alamitos, CA, USA, 197–203. <https://doi.org/10.1109/FPL53798.2021.00040>
- [29] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. *Improving Performance via Mini-applications*. Technical Report SAND2009-5574. Sandia National Laboratories.
- [30] M. A. Heroux and et al. 2005. An Overview of the Trilinos Project. *ACM Trans. Math. Softw.* 31, 3 (2005), 397–423. <https://doi.org/10.1145/1089014.1089021>
- [31] HYPRE. 2024. Scalable Linear Solvers and Multigrid Methods. <https://computing.llnl.gov/projects/hypre-scalables-linear-solvers-multigrid-methods>.
- [32] J. Kim, W.J. Dally, B. Towles, and A.K. Gupta. 2005. Microarchitecture of a high radix router. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 420–431. <https://doi.org/10.1109/ISCA.2005.35>
- [33] Venkata Krishnan, Olivier Serres, and Michael Blocksome. 2020. COConfigurable Network Protocol Accelerator (COPA): An Integrated Networking/Accelerator Hardware/Software Framework. In *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*. 17–24. <https://doi.org/10.1109/HOTI51249.2020.00018>
- [34] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Denver, Colorado) (SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 24, 22 pages. <https://doi.org/10.1145/3295500.3356181>
- [35] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (Palo Alto, California) (CGO '04)*. IEEE Computer Society, USA, 75.
- [36] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating Distributed Reinforcement learning with In-Switch Computing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 279–291.
- [37] Andre Massing, Mats Larson, and Anders Logg. 2013. Efficient implementation of finite element methods on non-matching and overlapping meshes in 3D. *SIAM Journal on Scientific Computing* 35 (01 2013), C23–C47.
- [38] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2003. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *2003 Design, Automation and Test in Europe Conference and Exhibition*. 296–301. <https://doi.org/10.1109/DATE.2003.1253623>
- [39] Arista Networks. 2024. 7130 FPGA-enabled Network Switches - Quick Look. [www.arista.com/en/products/7130-fpga-enabled-network-switches-quick-look](http://www.arista.com/en/products/7130-fpga-enabled-network-switches-quick-look).
- [40] NPB. 2023. NAS Parallel Benchmarks. <https://www.nas.nasa.gov/software/npb.html>.
- [41] L. Page, S. Brin, R. Motwani, and T. Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. TR 1999-66. Stanford InfoLab. <http://ilpubs.stanford.edu:8090/422/>
- [42] Jongsoo Park, Mikhail Smelyanskiy, Ulrike Meier Yang, Dheevatsa Mudigere, and Pradeep Dubey. 2015. High-performance algebraic multigrid solver optimized for multi-core based distributed parallel systems. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. <https://doi.org/10.1145/2807591.2807603>
- [43] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 389–402. <https://doi.org/10.1145/3079856.3080256>
- [44] Y. Saad and M. H. Schultz. 1986. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Statist. Comput.* 7, 3 (1986), 856–869. <https://doi.org/10.1137/0907058> arXiv:<https://doi.org/10.1137/0907058>
- [45] Manuel Saldaña, Arun Patel, Christopher Madill, Daniel Nunes, Danyao Wang, Paul Chow, Ralph Wittig, Henry Styles, and Andrew Putnam. 2010. MPI as a Programming Model for High-Performance Reconfigurable Computers. *ACM Trans. Reconfigurable Technol. Syst.* 3, 4, Article 22 (nov 2010), 29 pages. <https://doi.org/10.1145/1862648.1862652>
- [46] Whit Schonbein, Ryan E. Grant, Matthew G. F. Dosanjh, and Dorian Arnold. 2019. INCA: in-network compute assistance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Denver, Colorado) (SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 54, 13 pages. <https://doi.org/10.1145/3295500.3356153>
- [47] A. Skjellum and et al. 2020. ExaMPI: A Modern Design and Implementation to Accelerate Message Passing Interface Innovation. *Communications in Computer and Information Science* 1087 CCIS (2020), 153–169. [https://doi.org/10.1007/978-3-030-41005-6\\_11](https://doi.org/10.1007/978-3-030-41005-6_11)
- [48] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, S. Mehringer, Eric Wernert, H. Tufo, D. Panda, and P. Teller. 2017. Stampede 2: The Evolution of an XSEDE Supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact (New Orleans, LA, USA) (PEARC '17)*. Association for Computing Machinery, New York, NY, USA, Article 15, 8 pages. <https://doi.org/10.1145/3093338.3093385>
- [49] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. 2022. Taurus: a data plane architecture for per-packet ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Lausanne, Switzerland) (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1099–1114. <https://doi.org/10.1145/3503222.3507726>
- [50] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of Collective Communication Operations in MPICH. *Int. J. High Perform. Comput. Appl.* 19, 1 (Feb. 2005), 49–66. <https://doi.org/10.1177/1094342005051521>
- [51] Texas A&M University. 2024. SuiteSparse Matrix Collection. <https://sparse.tamu.edu/>.
- [52] R. A. Van De Geijn and J. Watts. 1997. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274. [https://doi.org/10.1002/\(SICI\)1096-9128\(199704\)9:4<255::AID-CPE250>3.0.CO;2-2](https://doi.org/10.1002/(SICI)1096-9128(199704)9:4<255::AID-CPE250>3.0.CO;2-2)
- [53] A. Venkatesh, K. Hamidouche, H. Subramoni, and Dhableswar K. Panda. 2015. Offloaded GPU Collectives Using CORE-Direct and CUDA Capabilities on Infini-Band Clusters. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. 234–243. <https://doi.org/10.1109/HiPC.2015.50>
- [54] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 703–716.
- [55] Xilinx. 2023. XUP Vitis Network Example (VNx). [https://github.com/Xilinx/xup\\_vitis\\_network\\_example](https://github.com/Xilinx/xup_vitis_network_example).
- [56] Qingqing Xiong, Chen Yang, Pouya Haghi, Anthony Skjellum, and Martin Herbordt. 2020. Accelerating MPI Collectives with FPGAs in the Network and Novel Communicator Support. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 215–215. <https://doi.org/10.1109/FCCM48280.2020.00046>
- [57] Ichitaro Yamazaki, Mark Hoemmen, Piotr Luszczek, and Jack Dongarra. 2017. Improving Performance of GMRES by Reducing Communication and Pipelining Global Collectives. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1118–1127. <https://doi.org/10.1109/IPDPSW.2017.65>