

2023

ACiS: smart switches with application-level acceleration

<https://hdl.handle.net/2144/46648>

Downloaded from OpenBU. Boston University's institutional repository.

BOSTON UNIVERSITY
COLLEGE OF ENGINEERING

Dissertation

**ACIS: SMART SWITCHES WITH APPLICATION-LEVEL
ACCELERATION**

by

POUYA HAGHI

B.S., Iran University of Science and Technology, Iran, 2017
M.S., University of Tehran, Iran, 2019

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

2023

© 2023 by
POUYA HAGHI
All rights reserved

Approved by

First Reader

Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

Second Reader

Rabia Yazicigil, PhD
Assistant Professor of Electrical and Computer Engineering

Third Reader

Alan Liu, PhD
Assistant Professor of Electrical and Computer Engineering
Assistant Professor of Computer Science

Fourth Reader

Anthony Skjellum, PhD
Professor of Computer Science and Engineering
The University of Tennessee at Chattanooga

Acknowledgments

I am incredibly grateful to my advisor, Prof. Martin C. Herbordt. He provided valuable insights and unlimited assistance throughout my research journey, allowing me the freedom to explore the areas that interested me. He also encouraged me to set higher standards for my academic work. What sets Prof. Herbordt apart from other supervisors is that he not only guided me academically but also served as a mentor in my personal life. I feel honored to have him as my Ph.D. advisor. I want to also sincerely thank professor Anthony Skjellum for his tremendous assistance, valuable insights, and helpful suggestions throughout my Ph.D. journey.

I am also thankful to my thesis committee members, Prof. Rabia Yazicigil and Prof. Alan Liu, for their invaluable feedback, generous support, and precious time. I am extremely grateful to all my collaborators and co-authors, especially Prof. Tong Geng, Dr. Ang Li, Dr. Rushi Patel, Dr. Cheng Tan, Dr. Chen Yang, Dr. Qingqing Xiong, Anqi Guo, Chunshu Wu, Sahan Bandara, Po Hao Chen, William Krska, Justin Broaddus, Derek Schafer, and Ryan Marshall, for their contributions, expertise, and insights, which have significantly enhanced the quality of my research.

Additionally, I would like to express my appreciation to all my friends in the CAAD research group, the ICSG research group, and the Peac Lab research group, for their support and encouragement. I also want to thank my friends at Boston University, especially Zahra, Anqi, Chunshu, Leila, Rushi, Reza, Sahan, Hafsah, Zihao, Efe, and Chathura, for their friendship, which has made my stay at the university more enjoyable. I am deeply grateful to my family for their constant love, support, and encouragement throughout my academic journey. Finally, I would like to pay tribute to my mother who is no longer with us, but whose memory and values serve as a constant source of inspiration in my life.

ACIS: SMART SWITCHES WITH APPLICATION-LEVEL ACCELERATION

POUYA HAGHI

Boston University, College of Engineering, 2023

Major Professor: Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

ABSTRACT

Network performance has contributed fundamentally to the growth of supercomputing over the past decades. In parallel, High Performance Computing (HPC) peak performance has depended, first, on ever faster/denser CPUs, and then, just on increasing density alone. As operating frequency, and now feature size, have levelled off, two new approaches are becoming central to achieving higher net performance: configurability and integration. Configurability enables hardware to map to the application, as well as vice versa. Integration enables system components that have generally been single function-e.g., a network to transport data—to have additional functionality, e.g., also to operate on that data. More generally, integration enables compute-everywhere: not just in CPU and accelerator, but also in network and, more specifically, the communication switches.

In this thesis, we propose four novel methods of enhancing HPC performance through Advanced Computing in the Switch (ACiS). More specifically, we propose various flexible and application-aware accelerators that can be embedded into or attached to existing communication switches to improve the performance and scalability of HPC and Machine Learning (ML) applications. We follow a modular design disci-

pline through introducing composable plugins to successively add ACiS capabilities.

In the first work, we propose an inline accelerator to communication switches for user-definable collective operations. MPI collective operations can often be performance killers in HPC applications; we seek to solve this bottleneck by offloading them to reconfigurable hardware within the switch itself. We also introduce a novel mechanism that enables the hardware to support MPI communicators of arbitrary shape and that is scalable to very large systems.

In the second work, we propose a look-aside accelerator for communication switches that is capable of processing packets at line-rate. Functions requiring loops and states are addressed in this method. The proposed in-switch accelerator is based on a RISC-V compatible Coarse Grained Reconfigurable Arrays (CGRAs). To facilitate usability, we have developed a framework to compile user-provided C/C++ codes to appropriate back-end instructions for configuring the accelerator.

In the third work, we extend ACiS to support fused collectives and the combining of collectives with map operations. We observe that there is an opportunity of fusing communication (collectives) with computation. Since the computation can vary for different applications, ACiS support should be programmable in this method.

In the fourth work, we propose that switches with ACiS support can control and manage the execution of applications, i.e., that the switch be an active device with decision-making capabilities. Switches have a central view of the network; they can collect telemetry information and monitor application behavior and then use this information for control, decision-making, and coordination of nodes.

We evaluate the feasibility of ACiS through extensive RTL-based simulation as well as deployment in an open-access cloud infrastructure. Using this simulation framework, when considering a Graph Convolutional Network (GCN) application as a case study, a speedup of on average $3.4\times$ across five real-world datasets is achieved

on 24 nodes compared to a CPU cluster without ACiS capabilities.

Contents

1	Introduction	1
1.1	Context and Thesis Statement	1
1.2	ACiS Taxonomy	4
1.3	Thesis Contributions	6
1.4	Organization	8
2	Fundamentals, Related Work, and Models	9
2.1	Fundamentals	9
2.1.1	Message Passing Interface (MPI)	9
2.1.2	Collectives	10
2.1.3	Field Programmable Gate Arrays (FPGAs)	11
2.2	Related Work	14
2.3	Models	16
2.3.1	Accelerator Integration Methods	17
2.3.2	Indirect/Direct Network Model	17
2.3.3	Fixed Function and Programmable Switches	17
2.3.4	Switch Configuration Methods	18
2.3.5	Methodology and Tools	18
3	Type 1 ACiS: User-Definable Inline Collectives	21
3.1	Introduction	21
3.2	Concepts	24
3.2.1	MPI Collectives	25

3.2.2	MPI-FPGA - Overview	25
3.2.3	Hardware Model	27
3.2.4	Communicators	28
3.2.5	Streaming Interface	30
3.3	In-Network Communicator Support	30
3.3.1	Communicator Table (CT) Design	31
3.3.2	Communicator Table (CT) Entry Creation	32
3.4	Implementation	33
3.4.1	Overview	33
3.4.2	Collective Control Module (CCM)	35
3.4.3	Two-Level Switch	37
3.4.4	Considerations for Indirect Networks	39
3.5	Performance Analysis of In-Network Collective Offload	40
3.6	Evaluation	43
3.6.1	Resource Utilization	44
3.6.2	Performance of MPI Collectives	45
3.6.3	MiniApp-Level Benchmarking	50
3.7	Conclusion	52
4	Type 2 ACiS: Look-Aside Acceleration	54
4.1	Introduction	55
4.2	Background, Motivation, Basics	58
4.2.1	Graph Convolutional Network (GCN)	59
4.2.2	Motivation	63
4.3	Hardware Design	66
4.3.1	FPin	66
4.3.2	FLASH	68

4.4	Software Design	75
4.4.1	Compiler	75
4.4.2	Supported Instructions	76
4.5	Application Acceleration	77
4.5.1	Distributed Matrix Multiplication	77
4.5.2	Graph Convolutional Network	80
4.6	Experimental Results	81
4.6.1	Distributed Matrix Multiplication	81
4.6.2	Graph Convolutional Network	83
4.7	Conclusion	91
5	Type 3 ACiS: Fused Collectives	92
5.1	Introduction	92
5.2	Preliminaries	95
5.2.1	Complex Communication-Computation Functions	95
5.2.2	<i>G-FPin</i> Basic Building Blocks	96
5.3	Motivation	96
5.4	<i>G-FPin</i> Software Support	100
5.4.1	Software Model	100
5.4.2	The Proposed Framework	101
5.5	<i>G-FPin</i> Hardware Support	103
5.5.1	Basic Collective Support	104
5.5.2	Network-Optimized CGRA	105
5.5.3	Kernel Logic	107
5.6	Model for In-Switch Computing	109
5.7	Experimental Evaluation	111
5.7.1	Experimental Setup	111

5.7.2	Performance and Scalability	113
5.8	Conclusion	117
6	Type 4 ACiS: Control	119
6.1	Introduction	119
6.2	Sample Use Case for Type 4 ACiS: Global Support for Application Synchronization	122
6.2.1	Application Space	124
6.2.2	The Synchronous Extra Work Method	126
6.2.3	Definitions	128
6.2.4	Evaluation	130
6.3	Conclusion	137
7	Summary and Future Work	139
7.1	Conclusion	139
7.2	Future Work	142
	References	144
	Curriculum Vitae	167

List of Tables

3.1	Algorithms commonly used by MPICH for processing collectives (Thakur et al., 2005)	31
3.2	Inter-FPGA Latency and Bandwidth for Xilinx Alveo U280	43
3.3	Resource usage on Xilinx XCVU13P FPGA Devices	44
3.4	MPI collective latency variation on the Stampede2 compute cluster for message sizes 1 KB and 1 MB on 32, 64, and 128 nodes.	48
3.5	Performance comparison of NAS parallel benchmark, miniFE, and HPCCG, for MPI CPU cluster (SKX) vs MPI-FPGA for 64 and 128 nodes. *For the SP benchmark the number of processes must be a perfect square.	52
4.1	PGEMM Grid Parallelism for Float Datatype	82
4.2	Dataset Sizes	85
4.3	The number of vector instructions for different datasets and dataflow modes (OS: output stationary, WS: weight stationary)	86
4.4	Compilation time and other parameters of the back-end compiler for GCN packet handler	89
5.1	Specifications of the synthetic benchmarks.	99
5.2	Parameters used in the LoGcN model and our our simulation. * Denotes the Aurora IP latency.	113
6.1	Experimental Configuration for Proxy Applications	130

6.2	Fraction of execution time that a process is idle and average idle time per iteration in seconds.	133
6.3	Average work rates AWR for baseline and EWM for two extra work utility coefficients U	136

List of Figures

3-1	Subset of commonly used MPICH collective algorithms	26
3-2	<i>MPI-FPGA</i> Software Stack	27
3-3	FPGA Switch Hardware Model	29
3-4	<i>MPI-FPGA</i> switch with collective support: Collective Control Module (CCM) and Configurable Gather/Reduction (CGR) Unit.	34
3-5	Collective Control Module	36
3-6	Configurable Gather/Reduction (CGR) Unit: aggregation has two paths with vectorized AgLUs chained together and combined at the end.	39
3-7	MPI CPU cluster (SKX) vs <i>MPI-FPGA</i> execution times for 32, 64, and 128 nodes: (a) <code>osu_allgather</code> , (b) <code>osu_allreduce</code> , (c) <code>osu_bcast</code> , and (d) <code>osu_gather</code>	45
3-8	MPI CPU cluster (SKX) vs <i>MPI-FPGA</i> execution times for 32, 64, and 128 nodes: (a) <code>osu_reduce</code> , (b) <code>osu_reduce_scatter</code> , and (c) <code>osu_scatter</code>	46
3-9	Average <i>MPI-FPGA</i> speedup ratios for OSU Benchmarks running on 32, 64, and 128 nodes of Stampede2 for short and medium to long messages.	48
3-10	Performance comparison for short messages of <i>MPI-FPGA</i> with direct and indirect networks on 32-, 64-, and 128-node systems for Allgather and Allreduce operations.	49
3-11	Average speedup ratios of indirect network switches over direct network switches for short messages.	50

3.12	CPU cluster (SKX) execution time and <i>MPI-FPGA</i> speedup of Allreduce (Reduce) collectives used in the applications under study. Each collective instance is evaluated separately, e.g., with MG having six instances of Allreduce.	52
4.1	(a) GCN inference execution time on a CPU cluster (Skylake processors) without and with FLASH accelerators. (b) FLASH matrix partitioning for an SpMM kernel with tasks shown for process 0 (P0). Gray tiles in LHM belong to the diagonal part and the rest are off-diagonal.	62
4.2	FLASH results for ogbn-products dataset: (a) number of transferred elements, (b) number of hops, and (c) overlap.	64
4.3	FPin Overall Architecture	66
4.4	Systolic array, CGR core, PR, and HL progress unit Architecture . .	67
4.5	The proposed switch accelerator with three pipelined vector PEs. It is packaged with an AXI interface to facilitate integration with switch pipelines.	71
4.6	Memory map for AXI interfaces with three AXI MMs; the first is used for application instruction and two others for application data.	74
4.7	Framework to map a machine learning model with a user-provide packet handler to the reconfigurable switch accelerator. Gray components are proposed. White components are powered by existing tools.	76
4.8	(a) Workload decomposition and (b) scheduling for small and large memory modes.	78
4.9	Proposed tuning methodology to obtain the size of tiles.	79

4.10	Strong scaling performance comparison of COSMA implementation on SKX cluster and PGEMM kernel accelerated with MPI_Gemm on FPGA-based cluster for square, tall, and flat matrices with complex float (a)-(c), double (d)-(f), and float (g)-(i) data types.	82
4.11	Communication performance and scalability comparison of GCN for a baseline CPU cluster (SKX) vs. FLASH with different configurations.	87
4.12	Application performance and scalability comparison of GCN on a baseline CPU cluster (SKX) vs. FLASH.	88
4.13	Overall application throughput measured at the LA's input interface for different datasets.	89
4.14	Resource utilization for AiS configuration with 16 pipeline vector PEs on Alveo U280.	90
5.1	Characterizing <i>G-FPin</i> and related work with respect to benefits of collective offload to network devices; <i>COPA</i> (Krishnan et al., 2020), <i>INCA</i> (Schonbein et al., 2019), and <i>SMI</i> (De Matteis et al., 2019) are in-NIC approaches while <i>iSwitch</i> (Li et al., 2019), <i>SHArP</i> (Graham et al., 2016), <i>Flare</i> (De Sensi et al., 2021), and <i>G-FPin</i> are in-switch methods. *Denotes a headless configuration in <i>COPA</i>	93
5.2	Examples of (a) a fused collective (<code>Allreduce_op_alltoall</code> in the IS application from the NAS parallel benchmark) and (b) a distributed kernel (SPMV).	97
5.3	Evaluating our approach vs. the original CPU implementation on 128 nodes with respect to (a) some in-switch computing benefits for the two CCCFs introduced in Fig. 5.2.	98
5.4	Latency of the synthetic benchmarks discussed in Table 5.1. * LoGcN model is discussed in Sec. 5.6.	99

5.5	Proposed Framework	102
5.6	The proposed smart switch design with CCCF support.	103
5.7	The proposed NGRA architecture; it is based on an array of SIMD-based processing elements.	106
5.8	The proposed kernel logic architecture; it is divided into two concurrent parts: computation and communication engines.	108
5.9	Time execution breakdown of a sender and a receiver process for 1408 bytes worth of data using <i>Vitis</i> framework and TCP/IP as the transport protocol. Error bars indicate <i>std</i> for five runs.	112
5.10	Scaling comparison of distributed kernels on SKX vs. <i>G-FPin</i> . The two left-most plots are the SPMV kernel (parabolic_fem and sme3Dc matrices) and the two right-most are the <i>PGEMM</i> kernel (square and tall matrices). 1, 24, and 48 OpenMP threads were used for the CPU SKX cluster.	115
5.11	Performance improvement of <i>G-FPin</i> 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 <i>std</i> for five runs, respectively.	116
5.12	AMG performance and scalability comparison between <i>G-FPin</i> , the optimized CPU implementation, and SHArP (with data adopted from (Graham et al., 2016)) for 2 up to 128 nodes with different number of threads (1, 24, and 48).	117
6.1	General model of control-in-the-switch. Telemetry information and other signals are sent to the switch and the switch controls leaf nodes, for example, by synchronizing them.	120

6.2	Control-in-the-switch model for the first use case: global support for time synchronization algorithms. Leaf nodes send data for collective communication as before and the switch keeps track of deadline and it sends a signal to leaf nodes when the counter is triggered.	121
6.3	Control-in-the-switch model for the second use case: stale synchronous parallel for DNN training. Leaf nodes send a done signal to the switch and switch sends back a continue signal if at least the done signal of P processes are received.	122
6.4	The three cases for non-laggard process execution.	130
6.5	time measurement (application work) of HPCG per iteration for laggard and leader processes and the deviation.	131
6.6	Laggard process occurrences sorted by the frequency.	132
6.7	Deadline Prediction (in seconds) with different Coefficient U.	135
6.8	Probability Density Function (PDF) of T_{lag} for different applications .	136

List of Abbreviations

ADI	Abstract Device Interface
AI	Artificial Intelligence
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuits
BSP	Bulk Synchronous Parallel
CDFG	Control Data Flow Graph
CFD	Computational Fluid Dynamics
CGRA	Coarse Grained Reconfigurable Array
CXL	Compute Express Link
DFG	Dataflow Graph
DMA	Direct Memory Access
DNN	Deep Neural Network
DSP	Digital Signal Processor
DSSP	Stale Synchronous Parallel
FF	Flip Flop
FPGA	Field-Programmable Gate Array
GCN	Graph Convolutional Network
GPU	Graphics Processing Unit
HLS	High Level Synthesis
HPC	High Performance Computing
IP	Internet Protocol
ISA	Instruction Set Architecture
LUT	Look-Up-Table
MAC	Multiply-Accumulate
MD	Molecular Dynamics
MGT	Multi Gigabit Transceiver
ML	Machine Learning
MPI	Message Passing Interface
NIC	Network Interface Card
NOS	Network Operating System
OS	Operating System
PDF	Probability Density Function
PE	Processing Element
PGEMM	Parallel Generic Matrix-Matrix Multiply
QoS	Quality of Service

RDMA	Remote Direct Memory Access
RTL	Register Transfer Level
SDN	Software-Defined Networking
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessors
SoC	System-on-a-Chip
SpMM	Sparse-dense Matrix Multiplication
SpMV	Sparse Matrix Vector multiplication
SSP	Stale Synchronous Parallel
UDP	Unreliable Datagram Protocol
VoQ	Virtual Output Queue

Chapter 1

Introduction

Ever since the 1990s, when Massively Parallel Processors (MPPs) took over from Vector Supercomputers as the dominant architecture in High End Computing (HEC), network performance has been vital to overall system performance. In parallel, High Performance Computing (HPC) peak performance has depended, first, on ever faster/denser CPUs, and, in the last twenty years, just on increasing density alone. As operating frequency, and also now feature size, have levelled off, two approaches are becoming central to achieving higher net performance: configurability and integration. Configurability enables hardware to map to the application, as well as vice versa. Integration enables system components that have generally been single function-e.g., a network to transport data—to have additional functionality, e.g., also to operate on that data during transport. More generally, integration enables compute-everywhere: not just in CPU and accelerator, but also in network and, more specifically, the communication switches.

1.1 Context and Thesis Statement

A tenet of HPC is that compute should go to the data and not the reverse: there are orders-of-magnitude advantages in power and latency with a local computation over an inter-node data transfer. We investigate a novel approach that challenges this conventional assumption. Once communication has been minimized algorithmically (Demmel et al., 2005; Demmel, 2013), the remaining data transfers typically are

unavoidable as they need to be combined with other remote data. We explore the freedom of choosing where this combining process takes place. For instance, since these data streams can be passing one another in the network on the way to being combined, it follows this combining could occur in the switches, rather than at an end point. In this thesis we show that this approach offers multiple advantages in terms of performance and resource utilization.

HPC systems continue to face crises in performance-portability and scalability (Jones et al., 2003; Balaji et al., 2009; Holmes et al., 2016; Ayala et al., 2021). Problems include increasing communication latency (De Sensi et al., 2020), networks that are both overloaded and underutilized (Schonbein et al., 2019), load balancing (Walshaw and Cross, 1999; Khaleghzadeh et al., 2021), and process skew (Ferreira et al., 2008; Widener et al., 2014; Bayatpour et al., 2020; Nikitenko et al., 2021). Our recent study (Haghi et al., 2021), shows that, on average, 45% of the total execution time (of a representative set of HPC applications) is wasted due to workload imbalance and other types of performance variability. This waste worsens with larger numbers of nodes (Bhatele et al., 2013; Nikitenko et al., 2021). Moreover, these problems are only aggravated with ever more powerful compute nodes and the increasing importance of applications dominated by sparse data and non-uniform communication. In addition, since these measurements do not include many other overheads that arise from other causes, a large amount of achievable performance is currently being left on the table: performance that we show can be obtained through judicious deployment of advanced computation and control in the switch fabric.

HPC applications often rely on collective communication for performing operations that require interaction among multiple processes; collectives comprise a large fraction of total HPC communication (Klenk and Fröning, 2017; Bernholdt et al., 2018). Simple examples of collectives are the broadcast of data from one process

to many, or the gathering of data from many processes into one, usually combined (reduced) with an operator such as *add* or *max*. Offload of collective processing into Network Interface Cards (NICs) (Arap and Swamy, 2016; Graham et al., 2010; Bayatpour et al., 2021) 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 enables better communication-computation overlap; and fourth, some network-host communication is removed as the NIC handles additional send/receive operations. But while NICs are valuable, this scheme still forces processing into endpoints. Another approach is to offload collective processing into switches (Peng et al., 2011a; Graham et al., 2016; De Sensi et al., 2021). This has three additional benefits: first, latency and bandwidth are improved as computation can be distributed, rather than performed in a single source or endpoint; second, switches support far more communication traffic than NICs so the potential benefit is proportionally increased; and third, communication volume may be drastically reduced as messages are quickly merged (reduction) or slowly replicated (broadcast).

In-switch computing brings additional advantages compared to processing in NICs. However, previous works on in-switch computing lack two limitations: (i) hardware switch support is fixed with lack of flexibility and reconfigurability; and (ii) there is no or at least little application support. In this thesis, we propose four novel methods of Advanced Computing in the Switch (ACiS). More specifically, we propose various flexible and application-aware accelerators that can be embedded into or attached to existing communication switches to improve the performance and scalability of HPC and Machine Learning (ML) applications. We follow a modular design discipline through introducing composable plugins to successively add ACiS capabilities. In our approach, we design ACiS with minimal additional redesign of either network,

NIC, or processing node, which means ACiS will be cost-effective and practical as well as transformative in terms of performance. **Our thesis is that flexible, complex, and integrated application-level processing in communication switches improves the performance and scalability of HPC and machine learning applications.**

1.2 ACiS Taxonomy

In this thesis we propose a taxonomy of ACiS Types. Type 0 includes well-established methods of compute in the network including transformations on streams, such as changes in data types or appending a CRC (Faber et al., 2021; Patel et al., 2022b), and supporting collectives (Almàsi et al., 2005; Graham et al., 2016; Graham et al., 2020), but on a limited number of primitive data types (e.g., int) and operations (e.g., add, max). Current implementations of simple collectives are integrated with MPI. Datatypes, operations, and communication contexts (MPI communicator) are fixed. An example is an MPI_Allreduce on a float with the aggregation logic accelerated by a fixed function switch (Graham et al., 2016).

Four additional ACiS Types comprise the primary contributions of this dissertation.

Type 1 ACiS: In the first method, we propose an inline accelerator to communication switches for user-definable collective operations. Message Passing Interface (MPI) collective operations can often be performance killers in HPC applications; we solve this bottleneck by offloading them to reconfigurable hardware within the switch itself. We also introduce a novel mechanism that enables the hardware to support a large number of MPI communicators of arbitrary shape, and that is scalable to very large systems. An example is an MPI_Allreduce on a float data type with multiplication as the reduction operator.

Type 2 ACiS: In the second method, we propose a look-aside accelerator for

communication switches that is capable of processing packets at line-rate. Functions requiring loops and states are addressed in this method. The proposed in-switch accelerator is based on mixing ISA (subset of RISC-V instructions) with dataflow graphs found in coarse-grained reconfigurable arrays (CGRAs). To facilitate usability, we develop a framework to compile user-provided C/C++ codes to appropriate back-end instructions for configuring the accelerator. Examples include accelerating compression algorithms (Krishnan et al., 2020) and communication-intensive parts of machine learning inference for large-scale datasets (Haghi et al., 2023a).

Type 3 ACiS: In the third method, we extend ACiS to support fused collectives and the combining of collectives with map operations. We observe that there is an opportunity of fusing communication (collectives) with computation. Since the computation can vary for different applications, ACiS support should be programmable in this method. An example is fusing Allgather with Allgatherv found in the NAS sort benchmark (Bailey et al., 1991). Another example is accelerating MapReduce type of operations, used in Deep Neural Network (DNN) inference for small ML models, (Swamy et al., 2022).

Type 4 ACiS: In the fourth method, we propose that switches with ACiS support controls and manages the execution of applications, the switch is an active device with decision-making capabilities. Switches have a central view of the network; they can collect telemetry information and monitor application behavior and then use this information for control, decision-making, and coordination of nodes. An example is workload rebalancing for certain applications (e.g., graph convolutional network training (Shi et al., 2021)). Another example is flexible synchronization as is used in Stale Synchronous Parallel (SSP) and Dynamic Stale Synchronous Parallel (DSSP) (Zhao et al., 2019) applications (e.g., DNN training).

These new capabilities, through ACiS, improves the performance and scalability

by improving the following: (i) latency of communication operations; (ii) overhead for collective processing in general; (iii) network load; (iv) process skew and load imbalance; and (v) application and middleware overhead.

These are the approaches we take for ACiS types:

- Investigate and address the problems (without ACiS)
- Investigate network/switch architecture/design and its support
- Investigate codesign with algorithms and applications
- Investigate codesign with programming models, run-times, and middlewares
- Performance measurement/evaluation and engineering

ACiS has certain requirements. First, to support line rate communication, packet processing should be done in hardware. 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 augmenting existing switches with reconfigurable logic (Swamy et al., 2022); by FPGA-augmented switches, e.g., from Arista (Arista, 2023); or by using the FPGA itself as a switch (e.g., New Wave (New Wave DV, 2023)).

1.3 Thesis Contributions

We summarize the main contributions of this thesis for each ACiS type as follows:

Type 1 ACiS:

- Design, implementation, and evaluation of a set of flexible in-switch MPI collectives in addition to efficient support of communicators of arbitrary shape with small memory footprint.

Type 2 ACiS:

- Design and implementation of an in-switch computing look-aside acceleration through a novel combining of CGRA architecture with RISC-V vector instruction support. To efficiently process machine learning workloads the ISA is extended with sparse vector instructions.
- A software toolchain to compile user-provided packet handlers to the instructions for configuring the accelerator at software speed (rather than HLS).
- Experimental results showing that a cluster with switches enhanced with Type 2 ACiS improves the performance and scalability of GCN applications. The performance advantage is on average $3.4\times$ (across five real-world datasets) on 24 node.

Type 3 ACiS:

- Proposing fusion of collectives and a systematic framework to accelerate them with reconfigurable switches transparent to MPI and experimental results showing that this approach improves the performance of applications.

Type 4 ACiS:

- A deadline-based Extra Work Method (EWM) is proposed to improve the predictability and the efficiency of system utilization for applications with an average of 12% improvement. This provides new insights on designing future collective offloads and motivation for ACiS Type 4.
- Proposing new capabilities in the switches to control and manage the execution of applications with different case studies: workload rebalancing, time synchronization, and new paradigms of parallel computing.

1.4 Organization

The remainder of this thesis is organized as follows. We review the background, state of the art on in-switch computing, and models in Chapter 2. Chapter 3 presents user-definable inline collectives as type 1 ACiS. In Chapter 4, we introduce our type 2 ACiS which is look-aside acceleration in the switch. Chapter 5 presents our work on fused collectives for type 3 ACiS. Chapter 6 presents introduces control in the switch which is our type 4 ACiS. In Chapter 7, we discuss the future directions and conclude this thesis.

Chapter 2

Fundamentals, Related Work, and Models

2.1 Fundamentals

In this section, we present background on Message Passing Interface (MPI), Collectives, and Field Programmable Gate Arrays (FPGAs).

2.1.1 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a standardized communication protocol and library for parallel computing (Gropp et al., 1996). It provides a programming model that allows multiple processes or threads to exchange data and synchronize their execution in a distributed computing environment, typically used in HPC systems.

MPI enables communication between processes by sending messages from one process to another. It provides a set of functions that allow processes to send and receive messages, perform collective operations, and manage the synchronization and coordination of computations across multiple processes.

The basic concept in MPI is that processes operate independently and exchange data through explicit message passing. Each process has a unique identifier known as a MPI process (rank), and messages are sent and received based on the ranks of the processes involved. The data exchanged between processes can be of various types, such as integer or floating-point numbers.

MPI supports both point-to-point communication and collective communication. Point-to-point communication involves sending messages between specific pairs of pro-

cesses, while collective communication involves coordinating communication among a group of processes, such as broadcasting data to all processes or performing reductions across multiple processes. MPI also supports other types of advanced communication to provide better scalability such as non-blocking communication, partitioned communication (Holmes et al., 2021), persistent operations (Morgan et al., 2017), neighborhood collectives, and one-sided communication.

MPI is designed to be portable and scalable, allowing parallel programs written with MPI to run on a wide range of hardware architectures, from multi-core machines to large clusters and supercomputers. It is widely used in scientific and engineering applications that require efficient parallel computing, such as weather modeling, Molecular Dynamics (MD) simulations, and Computational Fluid Dynamics (CFD).

There are different implementations of MPI, including Open MPI, MPICH, and Intel MPI, which provide libraries and tools for developing MPI applications. These implementations adhere to the MPI standard, ensuring compatibility across different systems and architectures.

2.1.2 Collectives

MPI collectives refer to a set of communication operations in MPI that involve the coordination and data exchange among a group of processes. These collective operations are executed by all participating processes together, and they often require synchronization and coordination among the processes. MPI provides a variety of collective operations to facilitate efficient communication patterns in parallel applications. There are numerous MPI collectives such as `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce`, `MPI_Scatter`, `MPI_Gather`, `MPI_Allgather`, and etc. For instance, `MPI_Bcast` broadcasts data from a single process (the root) to all other processes in the group. The root process provides the data, and all processes receive the same data. And, `MPI_Reduce`: Combines values from all processes in a group and stores

the result on a specified root process. The reduction operation can be a sum, product, minimum, maximum, etc. `MPI_Allreduce` is similar to `MPI_Reduce`, but the result is made available to all processes in the group. All processes perform the reduction operation, and the final result is stored in the output buffer of each process. `MPI_Allreduce` is used extensively in ML training to update weights from different workers. MPI collectives can be performance killers in many HPC applications (Klenk and Fröning, 2017; Bernholdt et al., 2018). One possible solution is to offload collective processing to NICs, and specially switches.

2.1.3 Field Programmable Gate Arrays (FPGAs)

FPGAs, or Field-Programmable Gate Arrays, are integrated circuits that can be customized and reprogrammed by users even after they have been manufactured. Unlike traditional Application-Specific Integrated Circuits (ASICs) that are designed for specific functions and cannot be altered once produced, FPGAs provide a programmable platform that allows users to implement and modify digital logic circuits according to their specific requirements. These devices are widely used due to their exceptional performance, low power consumption, support for reconfigurability, and their well-established integration in the communication stack, including routers (Bolaria and Byrne, 2009; Sheng et al., 2017a) and network facing components (Caulfield et al., 2016; Eran et al., 2019; Guo et al., 2022b).

FPGA normally consists of five types of hardware resources for efficient computation and communication (Hauck and DeHon, 2008).

1. Look-Up-Tables (LUTs) can be used to provide both flexible computation and high-concurrency data storage.
2. Flip-Flops (FFs) are used as registers.
3. DSP units are generally used to perform high-precision and high-performance addition, multiplication, and multiply-accumulation operations.

4. Block RAMs (BRAMs) are on-chip memories that provide tens of GByte on-chip storage.

5. Multi Gigabit Transceivers (MGTs) provide efficient inter-FPGA communication. Each FPGA chip normally has hundreds of high-bandwidth (over 20 Gb/s for each MGT) and low-latency MGTs as I/Os. They can be bundled together to support 100 Gb/s networking. These I/Os can be directed connected to other on-chip resources.

The hierarchical and programmable on-chip interconnect network embedded in FPGAs allows users to seamlessly integrate hardware resources according to their specific needs. By programming the interconnect network, users have the freedom to tailor the FPGA to address their target problems effectively. This inherent flexibility of FPGAs has positioned them as a compelling platform extensively utilized for HPC and the acceleration of ML tasks (Gokhale and Graham, 2005; Herbordt et al., 2007a; Herbordt et al., 2008; VanCourt and Herbordt, 2009; Benkrid and Vanderbauwhede, 2013).

FPGAs and GPUs, the widely used acceleration devices, are often compared, although they differ conceptually. However, in practical usage, FPGAs often exhibit similarities to GPUs. FPGAs incorporate hardware resources into numerous parallel computing units, similar to GPUs' Streaming Multiprocessors (SMs). Each FPGA computing unit comprises computation pipelines implemented with LUTs, DSPs, and FFs, as well as local memories utilizing BRAMs. BRAMs can serve as a shared global scratchpad memory, akin to caches in GPUs. Nevertheless, FPGAs also possess distinct advantages and diverge from GPUs in various aspects. The computing units of FPGAs can be customized precisely to match specific target problems, potentially achieving higher efficiency. Furthermore, FPGAs offer a flexible and customizable interconnect, enabling seamless connections between computing units without restrictions on inter-unit communication, except for physical interconnect limitations. This

flexibility makes FPGAs an attractive choice for addressing irregularity problems.

Due to the exceptional communication support provided by FPGAs, it is logical to configure them directly within FPGA-centric clusters. Extensive research has been devoted to modeling and constructing such clusters, which involves determining the types of interconnects, such as direct (FPGA-FPGA) or indirect (via a router) (Sheng et al., 2015; Sheng et al., 2016b; Sheng et al., 2017b; Sheng et al., 2016a; Putnam, 2014; Xiong et al., 2018; Plesl, 2018; Boku et al., 2019; Shahzad et al., 2021).

Given the advantages mentioned earlier, FPGAs play crucial roles in HPC and neural network acceleration (VanCourt and Herbordt, 2007; VanCourt and Herbordt, 2006c; VanCourt and Herbordt, 2005a; VanCourt and Herbordt, 2005b; VanCourt and Herbordt, 2004; Sanaullah et al., 2018c; Sanaullah et al., 2018a; Jamieson et al., 2018). Although FPGAs have not yet achieved the same level of prevalence as GPUs in HPC, they have displayed promising potential as a component of future HPC systems. Researchers have effectively showcased the efficiency and advantages of FPGAs in machine learning (Geng et al., 2018b; Geng et al., 2018a; Wang et al., 2020; Geng et al., 2021a; Geng et al., 2019b; Geng et al., 2019a; Geng et al., 2021b; Haghi et al., 2020d; Peng et al., 2022; Shi et al., 2020) and various critical scientific computing applications, including Molecular Dynamics (VanCourt et al., 2004; VanCourt and Herbordt, 2006b; Sukhwani and Herbordt, 2008; Sukhwani and Herbordt, 2009; Chiu et al., 2008; Chiu and Herbordt, 2009; Chiu and Herbordt, 2010; Chiu et al., 2011; Yang et al., 2019a; Yang et al., 2019b; Yang et al., 2017; Wu et al., 2020; Wu et al., 2021; Wu et al., 2022; Wu et al., 2023), Algebraic Multigrid (VanCourt and Herbordt, 2006a; Haghi et al., 2020a), Adaptive Mesh Refinement (Wang et al., 2019a; Wang et al., 2019b), bioinformatics (Herbordt et al., 2007b; Herbordt et al., 2006), as well as security (Wolfe et al., 2020; Patel et al., 2020; Patel et al., 2022a), and others (Peng et al., 2021). The programmability of FPGAs is often believed to be an ob-

stacle to the wide adoption of FPGAs in the real HPC system. However, researchers have demonstrated that this can be addressed in better design tools (Sanaullah and Herbordt, 2018b; Sanaullah et al., 2018b; Sanaullah and Herbordt, 2018a; Sanaullah and Herbordt, 2018c; Herbordt, 2019; Shahzad et al., 2022) and middleware (Haghi et al., 2020c; Haghi et al., 2020b; Xiong et al., 2020; Bandara et al., 2022).

2.2 Related Work

While the idea of "the network is the computer" (Perry, 2019) has been around for a long time, it has typically involved using the network as a passive and transparent channel. However, in recent years, there has been a shift in the data center model, moving from a compute-centric approach to a data-centric one (Yoshida, 2020; Morgan, 2021), in part, through the emergence of SmartNICs (Arap and Swamy, 2016; Graham et al., 2016; Bayatpour et al., 2021; Xilinx, a; Xilinx, b; Inventec, ; Silicom, ; Napatech, ; Intel, ; Mellanox,) and similar network-facing devices (Caulfield et al., 2016), and their use in offloaded application (Chung et al., 2018; Xiong et al., 2019) and system (Li et al., 2016; Firestone et al., 2018) processing.

There are two types of communication switches: fixed function (Graham et al., 2016) and programmable switches (Bosshart et al., 2014). The former, while can be configurable to some extent, cannot support new transport protocols or new packet processing capabilities. The latter, however, exposes programmability to the users, for example, by customizing packet header fields (leads to support different transport protocols), the type of packet processing (actions), and the matching rule (match) such as P4-based switches.

The IBM BlueGene systems (Almàsi et al., 2005) offload collectives into the fixed function network router. Recent work by Mellanox (Graham et al., 2016) offloads MPI collectives to *fixed logic* switches using reduction trees for short message size. It has

limitations, in particular, supporting only a small number of simple operations with no extensibility; also there are no published (or generally available) design details. Also, in general, they have offered only modest benefit in typical environments (Hoefler et al., 2009; Haghi et al., 2021). A hardware-based streaming aggregation capability was later added to provide high bandwidth for large messages (Graham et al., 2020). Another limitation of prior art is the lack of flexible MPI communicator support. For example, some prior works do not even support communicators (Peng et al., 2011b; Saldaña et al., 2010); others support only a limited number of communicators at a time (Gilge, 2013). We address this limitation by a flexible and scalable hardware design (Type 1 ACiS).

Increasing switch flexibility has long been a goal of the networking community, culminating, in part, in programmability using P4 (Bosshart et al., 2014). P4 is a high-level language used to control packet forwarding. While there has been some exploration of application-level processing (Sankaran et al., 2021) in P4 switches, including support of collectives (Sapio et al., 2021), these capabilities are currently limited and likely to fall short in certain applications, including machine learning (Swamy et al., 2022). P4 has aided in increasing switch flexibility (Bosshart et al., 2014) including accelerating applications that are beyond the basic switching function. Examples are consensus algorithms (Dang et al., 2016), database transaction processing (Jasny et al., 2022), caching (Liu et al., 2017), and key-value store processing (Jin et al., 2017). While there has also been some exploration of application-level processing (Sankaran et al., 2021) in P4 switches, including support of collectives in ML training (Sapio et al., 2021), these capabilities are currently limited (Swamy et al., 2022) and likely to fall short in certain applications, including machine learning. This includes limited set of operations (e.g., no multiply), data types (e.g., no sparse data types), and memory footprint. Another limitation is that packets can only access each

memory location once within a traversal (De Sensi et al., 2021) (limited data reuse); while it is possible to recirculate packets, this technique reduces throughput. To address these limitations, ACiS is designed to handle advanced calculations, such as fused multiply-accumulate (MAC) and sparse accumulation, off-chip memory access with controllable buffers, data reuse, and control.

In-switch computing is becoming an active area of research. The work in (Li et al., 2019) provides an in-switch computing paradigm, implemented on NetFPGA, to accelerate aggregation of gradients used in the training phase of reinforcement learning. The authors in (Sapio et al., 2021) accelerate distributed training of DNN models by designing a communication primitive that uses P4. The work (Sensi et al., 2021) designs a flexible programmable switch on top of PsPIN (Di Girolamo et al., 2021) building blocks to accelerate Allreduce with custom operators and data types; that is, sparse data (Type 1 on a single collective type). The work (Liu et al., 2020) presents a Remote Direct Memory Access (RDMA) compatible in-network reduction architecture to accelerate distributed DNN training in which the FPGAs are connected to the switch and the switch is configured to route the packets that need to be aggregated to the FPGA. The work in (Swamy et al., 2022) adds custom hardware based on a MapReduce pattern/abstraction (built upon a CGRA) to P4 devices to enable per-packet inference of machine learning (Type 3 - but with no fusion of collectives). To summarize, we are not aware of previous work that fully supports user-defined or complex collectives (Types 1 and 3) or in any way addresses look-aside or control capability (Types 2 and 4).

2.3 Models

In this section, we discuss methods of accelerator integration, switch configuration, methodologies and tools as well as switch models.

2.3.1 Accelerator Integration Methods

We consider two approaches for integrating our accelerators: Accelerator-integrated-switch (AiS) and Accelerator-attached-switch (AaS). In AiS, the accelerator is integrated into existing switch pipelines that implement a full switch functionality, including packet forwarding. In AaS, an accelerator (implemented on FPGA device) is attached to an existing switch: packets are directed to the accelerator for further processing and are redirected back to the switch for switching. In both cases, there is additional functionality to configure the accelerator.

2.3.2 Indirect/Direct Network Model

There are at least two plausible architectural targets for using reconfigurable logic for in-switch compute-in-the-network. One is to use reconfigurable logic in the switch (AiS (Arista Networks, Inc., 2013)) or to attach a reconfigurable device to the switch (AaS) for packet processing in an *indirect* network (e.g., fat-tree topology). A second is in FPGA-centric clusters with *direct* FPGA-FPGA interconnects (Putnam, 2014; George et al., 2016; Miyajima et al., 2018; Stewart et al., 2021).

2.3.3 Fixed Function and Programmable Switches

There are two types of communication switches: fixed function (Graham et al., 2016) and programmable switches (P4-based switches) (Bosshart et al., 2014). The former, configurable to some extent, cannot support new transport protocols or new packet processing capabilities. The latter, however, expose programmability to users, e.g., by customizing packet header fields (to support different transport protocols), the type of packet processing (actions), and the matching rule (match) through a match-action pipeline. ACiS plugins can be used to extend the functionality of both fixed function and P4-based switches.

2.3.4 Switch Configuration Methods

We classify switch configuration methods into three approaches: (1) configuration through host: Typically, there is a host attached to communication switches; sometimes these hosts run a specialized operating system called Network Operating System (NOS). After installing the switches it is possible to connect to them from leaf nodes through, for example, secure shell protocol (SSH) and load the configuration or change Quality of Service (QoS) of switch. (2) Configuration through network packets: another approach is to configure the switches on-the-fly from leaf nodes through a specialized packet called a network packet distinguished by a specific header. An example from an academic work is (Li et al., 2019). (3) Configuration through Software-Defined Networking (SDN): a mechanism to automate this process in data centers and clouds is to utilize SDN (Kreutz et al., 2015). SDN is an technology for network management that enables dynamic and programmatically efficient network configuration in order to improve network performance and monitoring, making it more like cloud computing. This can be accomplished through certain Application Programming Interface (APIs) and special server(s) are used for this purpose. In our research we use the first method.

2.3.5 Methodology and Tools

Experimental results in this thesis are generated either from a simulator or from a testbed. For the simulation, we use an emulation method inspired from (Li et al., 2019). To make a fair comparison with baseline CPU cluster, the emulation has (i) the same number of network hops, (ii) the same amount of traffic in the network links, and (iii) accurate accelerator overhead. We also capture process skew of the application and consider the arrival time of each process; these can affect the performance of in-switch offload (Graham et al., 2016). For the process skew, we run MPI appli-

cations on a CPU cluster and obtain time traces for all of the processes in the system by annotating the application code. For the accelerator overhead, we use simulation results from a cycle-accurate RTL simulation using a testbench to drive signals, generate traffic, and measure the ACiS accelerator’s performance and/or throughput. The testbench has emulation modules for off-chip memory and streaming ports that realize the handshaking. Xilinx Vivado is used for this purpose. Also, we consider the latency overhead of MPI with the same message size for sending/receiving packets used in our approach.

For the testbed, we use two clusters, one for direct network and one for indirect network. For the direct network, our testbed is a two-node Alveo FPGA boards connected together in a ring fashion with 100 Gb/s cables, where each FPGA is attached to a host CPU. We use this testbed for direct-network version of Type 2 in AiS configuration. For the indirect network, we are using CloudLab (Handagala et al., 2022) infrastructure where the accelerator is implemented on a Xilinx Alveo U280 FPGA attached to a Dell Z9100-ON switch and several leaf nodes connected to the Dell switch with NIC facing the network (the maximum number of leaf nodes is four). These leaf nodes send packets to the FPGA, the FPGA processes packets and sends back the processes packets to leaf nodes. We also call this prototype as headless NIC testbed. The accelerator has a socket table which binds Internet Protocol (IP) addresses to port numbers and makes it feasible to listen on incoming packets, process them, and send them back to leaf nodes. This is used for the indirect-network version (AaS) of Types 2 and 3. For the baseline CPU reference, applications/benchmarks were run on the TACC Stampede2 (Stanzione et al., 2017) 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.

For the direct network, ExaMPI (Skjellum et al., 2020) implementation is used for

making our approach transparent without the need for underlying hardware knowledge. ExaMPI is a light-weight MPI implementation that focuses on key blocks of functionality and new MPI concepts and it is designed for modularity and extensibility. In order for the host CPU to communicate with FPGA we use OpenCL API calls supported by Xilinx Vitis. Vitis is a unified framework that exposes APIs to enable host-FPGA communication; however, it is not designed for communication between FPGAs. Therefore, we modify existing open-source repositories that provide networking support for FPGAs (TCP and UDP) to tailor our need. For the indirect network, run-time and middleware are supported through Python functions with Unreliable Datagram Protocol (UDP) as the transport layer and with the help of PYNQ and socket libraries.

Finally, for Types 2 and 3, we use LLVM toolchain (Lattner and Adve, 2004) for the front-end compilation and an in-house back-end compiler.

Chapter 3

Type 1 ACiS: User-Definable Inline Collectives

This chapter introduces user-defined inline collectives. Collectives such as Reduce and Allgather are supported. Two important ACiS plugins are (1) a collective control unit, for flexible management of MPI communicator context, and (2) a programmable aggregation unit, for supporting aggregations with different data types and operations. This chapter is based on the work published in *Concurrency and Computation: Practice and Experience (CCPE)* ©2022 Wiley (Haghi et al., 2022) and *High Performance Extreme Computing Conference (HPEC)* ©2020 IEEE (Haghi et al., 2020c).

3.1 Introduction

Collectives are often a large fraction of total communication in HPC applications (Klenk and Fröning, 2017; Bernholdt et al., 2018) and have been shown to bottleneck performance (Pjesivac-Grbovic et al., 2005; Faraj et al., 2009; Graham et al., 2016). As collectives are integral to HPC, and since much communication in production HPC is based on MPI (see *e.g.*, (Bernholdt et al., 2018)), addressing the acceleration of collectives necessarily means dealing with them within an MPI framework. Collectives in MPI implementations (such as MPICH (Gropp et al., 1996)) generally consist of point-to-point messages with computations in between. Thus much support has been added at the software level (Pjesivac-Grbovic et al., 2005; Thakur et al., 2005; Chan et al., 2004); however, the addition of these algorithms has greatly complicated the

software stack (Rafenetti et al., 2017).

Compute-in-the-network has been studied since the early days of computing through structures such as adder trees and sorting networks (Knuth, 1973); it is also fundamental to the more powerful PRAM models explored in the 1980s (Eppstein and Galil, 1988). However, there appear to be just two modern commercial versions of in-switch computing that can support collectives: the IBM BlueGene family (Almàsi et al., 2005; Faraj et al., 2009) and, in current use, certain switches from Mellanox (Graham et al., 2016). Both of these have limitations that are, in part, the result of being ASIC-based and so having strictly bounded capabilities with limited flexibility. Moreover, being commercial products, few details are available about their implementation, which curtails their use as the basis of further research.

In Type 1 ACiS, we offload MPI collectives into reconfigurable FPGA hardware (*MPI-FPGA*) and, in particular, into logic appended to the communication switches. Implementations with reconfigurable logic have several inherent advantages over those with fixed logic. First, they are not limited to a small, fixed set of operations, *e.g.*, *SHArP* only supports `MPI_Allreduce` and `MPI_Barrier` operations (Graham et al., 2016) and a few primitives. Second, hardware resources can be configured to match application requirements, *e.g.*, by increasing the arithmetic support as needed. This can even be done at runtime for applications in which the communication data volume is unpredictable (Haghi et al., 2020a). Third, support can be extended beyond simple datatypes to more complex structures such as matrices, tensors, and user defined datatypes (Haghi et al., 2020a; Haghi et al., 2020c). Fourth, compute-in-the-network can be generalized still further to support *altruistic* or *opportunistic* computing (Munafò, 2018). And finally, since reconfiguration time is similar to program load time, only resources that will actually be used need to be configured; the remaining logic can be used for other purposes. All of these scenarios can be accomplished without

incurring the cost of designing and fabricating ASIC chips for each application, or devoting ASIC resources to functions that rarely occur.

An essential part of implementing MPI collectives is handling the critical MPI feature of the *communicator*; these are used to define a safe communication context for message passing within a specific group of processes. Communicators have significant scalability issues (Kamal et al., 2010), meaning we cannot implement them in hardware with the same methods used for managing communicators in software; for this reason in-switch implementations sometimes limit communicator geometry (Almàsi et al., 2005). In this chapter we introduce an in-switch design for general communicator support that consumes minimal memory resources. Moreover, as the resources are guaranteed to grow no faster than the log of the number of nodes, this solution is likely to remain relevant far beyond exascale.

The main contribution is the design, implementation, and evaluation of a set of FPGA-based in-switch MPI collectives. We believe this to be the first FPGA version to be fully integrated into a general router. To achieve this end, a novel 2-level in-switch design is proposed with full-pipeline capability. Essential to this design are the seamless integration to a general router, added hardware support for aggregating (reordering) packets for reduce-type (gather-type) operations, and the flexibility to add more computational resources as the switch bandwidth increases. *MPI-FPGA* is fully integrated into MPICH; *MPI-FPGA* is therefore transparently usable by any MPI application. It is also easily extended to support additional collectives or integrated into other MPI implementations. A second contribution is the finding that all collective routing decisions—including those with arbitrarily complex communicators—can be made using only a small amount local information. Finally, an efficient streaming interface is provided to ensure a fine-grained communication specially for long message sizes.

In this chapter we consider both indirect and direct networks. Our experiments show that *MPI-FPGA* in a direct network can achieve a significant improvement for MPI collectives over a CPU cluster for a large range of message sizes. For FPGA-based high-radix switches in an indirect network it is possible to make comparisons with *SHArP* technology: *MPI-FPGA* is competitive and improves the performance of Allreduce operation. We have also experimented with NAS parallel benchmarks and two MiniApps from the Mantevo project (miniFE and HPCCG). The most significant result, however, is that this performance is possible while simultaneously supporting reconfigurable in-switch computing in MPI collectives and many extensions (*e.g.*, (Haghi et al., 2020c)).

The rest of this chapter is organized as follows. Section 3.2 provides background information on some of the key concepts. Section 3.3 introduces in-network communicator support and it sheds light on some of the design choices. Section 3.4 describes in detail the proposed switch and the modules used in the design. Section 3.5 analyzes and compares the performance of our approach (in-network collective offload) with that of the existing approach. Section 6.2.4 presents experimental results. Section 6.3 concludes this chapter.

3.2 Concepts

We review the MPI software stack to identify opportunities for, and the benefits of, offloading collectives. Next, we discuss our hardware model. We then cover MPI communicators and the difficulties they create for hardware implementations. Finally, we discuss the proposed streaming interface needed to efficiently support long messages.

3.2.1 MPI Collectives

MPI collectives are generally implemented so that processes execute sequences of point-to-point messages and computations. Various algorithms are used, but priority is given to those that avoid congestion and minimize the total number of packets. However, these may translate into more work in software for deciding which chunks of data to send and receive, and the processes with which to communicate. For example, a trivial implementation of *MPI_Reduce* has every process send data directly to the root and leads to congestion in a large network. In contrast, with a binomial tree algorithm, as seen in Figure 3.1(a), each process could be either a leaf, intermediate, or root process. Leaf processes simply send data to their parent, but intermediate processes must determine the identities of their children, receive data from them, and perform the reduction operation on the received data. They then determine their parent and then send their intermediate result. In summary, this algorithm lessens the number of packets in the network and unclogs the root, but it forces much additional work in software. Other algorithms that are commonly used in collectives – such as recursive halving and recursive doubling – can similarly improve the performance of the collective, but also require that each process perform extra work.

3.2.2 MPI-FPGA - Overview

MPI-FPGA removes the need for this software and moves its functions into the network. Throughout this paper, we refer to the CPU (leaf nodes) as the *processor* and to the FPGA-augmented switches as the *network*. *MPI-FPGA* requires no changes to the MPI API and so is transparent to application; this makes it completely portable: it can be integrated into HPC applications without requiring the programmer to have any knowledge of the underlying hardware or make any changes to existing programs. Rather, *MPI-FPGA* constructs access capabilities automatically through enhanced

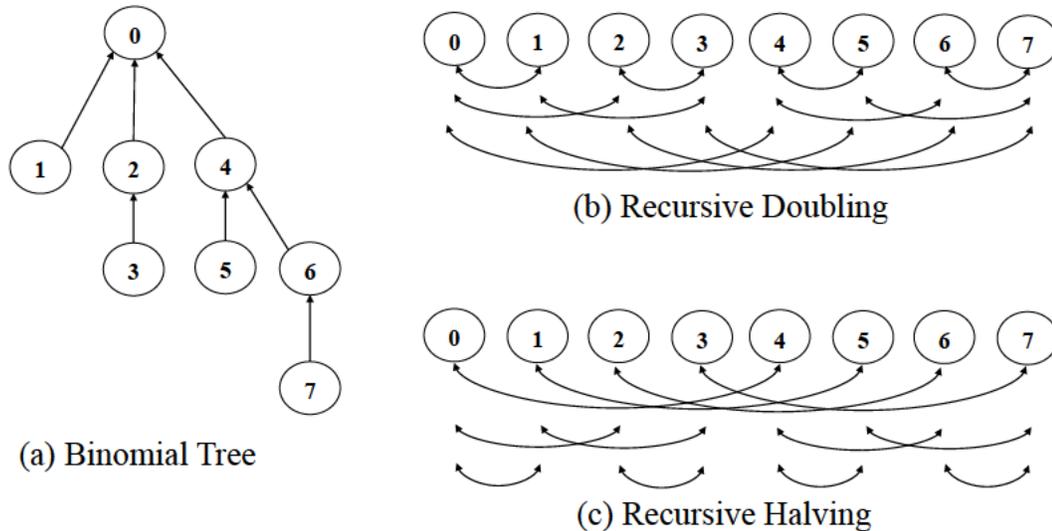


Figure 3-1: Subset of commonly used MPICH collective algorithms

middleware. The design makes no assumptions about the type of end-systems being used, as it only affects data as it is routed through the switches in the network. We create new functions for each offloaded collective (e.g., *MPI-FPGA_Reduce*), and place these underneath existing MPI collective functions (see Figure 3-2). If the hardware supports the offload of a particular collective, then the *MPI-FPGA* replacement function is used. If a collective does not have offload support, then it is performed by the software as usual.

The basic operation of *MPI-FPGA* is as follows. Upon receiving a valid packet header from the processor, the switch begins the collective operation and performs all of the necessary steps to complete it. If an MPI process is not required to receive the final data—such as the root process in a broadcast operation—then control returns to the user application as the network is responsible for progressing the message. If the calling process *does* need to receive the result—for instance, all processes in an *Allgather* operation—then the process waits until it is interrupted, which happens when the final collective results have been received and passed to the processor from

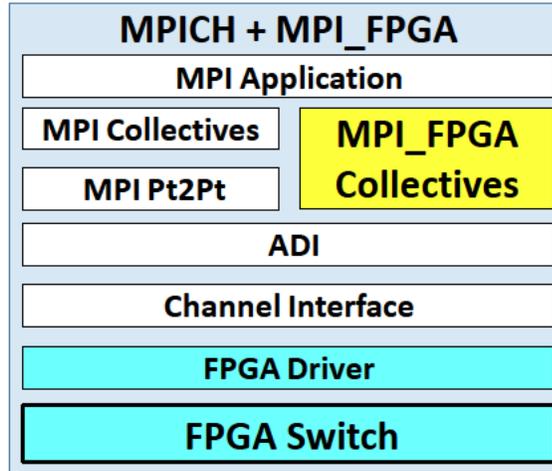


Figure 3-2: *MPI-FPGA* Software Stack

the network.

We focus on blocking MPI collectives; the extension to nonblocking operations is simple. In the MPICH implementation of MPI middleware (Gropp et al., 1996), all the functionality of the Abstract Device Interface (ADI) is maintained. This work currently uses MPICH-3.2 (Gropp et al., 2003); tasks such as packing and computing predefined reduction operations are performed identically in this design. At the MPICH channel interface, we add code to transfer data into the network, with the actual FPGA hardware sitting below the channel interface (see Figure 3-2).

3.2.3 Hardware Model

For integrating *MPI-FPGA*, we consider both direct and indirect networks. For simplicity, in both cases the network switches consist entirely of FPGAs. Since the configurable parts of the design are modular, the extension to systems that combine fixed and reconfigurable logic is straightforward. For other advantages of fully configurable switches see (Sheng et al., 2018). For direct networks, this is identical to clusters with direct FPGA-FPGA interconnects (e.g., (Putnam, 2014; George et al., 2016)). Each FPGA switch is attached to the processor through a PCIe interface

and FPGAs themselves are connected directly together through a secondary network (3D-torus). For indirect networks the FPGAs are proxies for high-radix switches with integrated configurable logic and are attached to multiple processors through NICs. We currently use a fat-tree network (spine-leaf architecture (Alizadeh and Edsall, 2013)). Switches in the direct (indirect) model are homogeneous (non-homogeneous) between leaf and spine switches.

Figure 3-3 shows the hardware model of the switch. To simplify the figure, transceivers and DMA engines are duplicated on the left and right. A Collective Control Module (CCM) manages the coordination between different MPI processes for a collective operation. Inside the 2-level switch are, first, Configurable Gather Reduction (CGR) units that perform the gather or reduction operation on incoming packets, and, then, hierarchical switches for reordering or redirecting packets. The CCM and CGR units are discussed in depth in Section 3.4. Processor-FPGA and FPGA-FPGA communication, as well as in-switch processing, are based on a streaming interface: data is pushed to the DMA engine, transceivers, and buffers, respectively, without having to wait for the entire packet to arrive. User logic units surround the switch for pre- and/or post-processing. Although this design is implemented on FPGAs, its portability ensures that it is independent of the type of hardware used. Units added in support of any typical network switch can be inserted or removed seamlessly.

3.2.4 Communicators

Communicators are an essential MPI capability and must be supported in any useful system; yet, little work on collective offload into the network addresses it. While the default communicator is *MPI_COMM_WORLD*, where the process group is the entire set of processes (Dongarra et al., 1995), many MPI programs use multiple communicators having subsets of processes. A typical example where sub-communicators

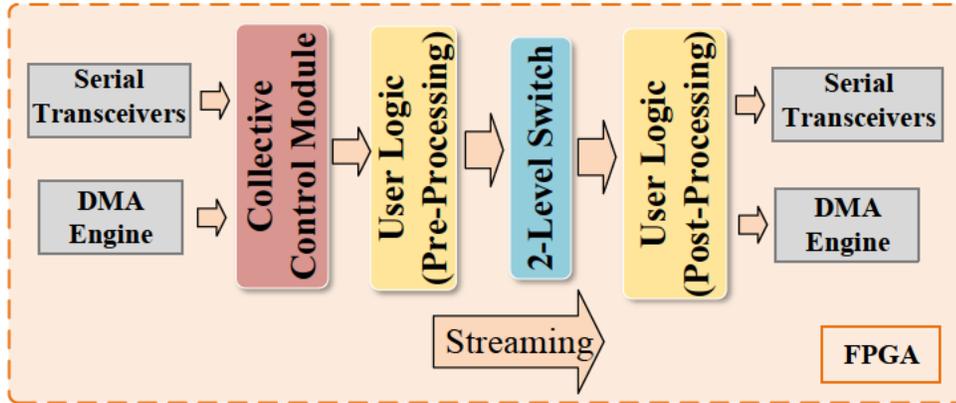


Figure 3.3: FPGA Switch Hardware Model

are used is when the workload is partitioned among an array of MPI processes and collectives are performed on rows or columns of processes. The most common way to create these partitions is to call a function like *MPI_Comm_split*.

All communicators have a *context ID*, identifying the communicator, and a *process group* containing the list of processes in that communicator. When a new communicator is created, a new process group is created and stored in memory. In large systems, with correspondingly large communicators, the memory consumption of these process groups leads to scaling issues (Kamal et al., 2010). To have an entire process group in FPGA memory would require storing the list of all MPI processes included in the communicator. The number of bits required would be the product of *COMM_SIZE* and *BITS_PER_PROCESS*, meaning that the resource utilization would grow linearly with the communicator size. For a system with thousands of nodes, it would require many thousands of bits in the switch for *each* of many communicators in a single application. Since high performance routing depends on having routing information close to the switching logic, replicating information about these entire process groups would quickly use up these resources, even for mid-sized clusters.

3.2.5 Streaming Interface

While current MPI implementations rely on bulk transfers involving large buffers (De Matteis et al., 2019), we adopt a streaming interface for both processor-FPGA and FPGA-FPGA communication. The advantage of providing this support is three-fold. First, the latency of collective operations is reduced as data is processed/transferred cycle by cycle. Second, the streaming interface facilitates fine-grained communication-computation overlap. And third, it avoids the congestion caused by messages with a large payload.

To ensure the finest granularity we convert (inside the switch) processor packets into multiple small-sized network packets. Decoupling these two kinds of packets is beneficial as processor packets may contain a large payload that could block other packets from progressing. Network packets, on the other hand, are small-sized with replicated headers. The size of the packet is itself one of the fields in the new header. The structure of processor and network packets is discussed in more detail in Section 3.3.1. We set the phit size to 256 bits; however, the DMA engine can process two phits on each cycle.

Flow control is implemented in a standard way using back-pressure and stalling without incurring costly handshakes. There is a margin m in the FIFOs; upon reaching this threshold the upstream node is notified to stop sending packets. When the buffer gets depleted past the margin the upstream node is notified to resume sends.

3.3 In-Network Communicator Support

In this section we introduce the Communicator Table (CT) design, which supports collectives across any intra-communicator. This design takes advantage of the existing MPI collective algorithms in order to minimize resource utilization and latency. For simplicity, in this section and the next section we assume a direct network. The design

Table 3.1: Algorithms commonly used by MPICH for processing collectives (Thakur et al., 2005)

MPI Collective Algorithms		
Reduce	Binomial Tree	Recursive Halving and Doubling
Allreduce	Recursive Doubling	Recursive Halving and Doubling
Broadcast	Binomial Tree	Binomial Tree and Ring
Scatter	Binomial Tree	Binomial Tree
Gather	Binomial Tree	Binomial Tree
Allgather	Recursive Doubling	Ring

is analogous for indirect networks; necessary modifications are given in Section IV.D.

3.3.1 Communicator Table (CT) Design

The purpose of the CT is to manage communicator information that is needed by the CCM to make packet forwarding decisions. To minimize the resources required, the table only holds the local data that is necessary to complete the implemented collectives. This means that each switch needs a way of obtaining this local data, which is a list of the other MPI processes with which it must communicate to perform each collective. The contents of this list, for a given communicator, can be determined immediately after its initialization.

Table 3.1 shows a subset of MPICH algorithms for widely used collectives (Thakur et al., 2005). The three most commonly used are binomial tree, recursive halving, and recursive doubling. The ring algorithm is also common, but since its implementation is trivial we focus on the others. By being able to implement these three algorithms, we can perform all of the collectives that use them. Returning to Figure 3-1, for each MPI process we can identify the subset of processes with which a given MPI process must communicate. For example, process 0 must communicate with the following process set in all three algorithms: 1, 2, 4. Storing this subset in switch memory is much more efficient than storing an entire process group as it is equal to the log of the communicator size (which can be proved directly from the properties of binomial trees).

The CT holds a row for each communicator of which the current switch is a member. In each row, we store a small amount of meta-information: the communicator size, the MPI process in the communicator with which the current switch is associated, and the subset of processes with which the switch will be communicating. If there are multiple processes per node, one of the processes (the parent) is designated as the local MPI process. Each communicator entry is indexed into the table using its context ID. For any incoming packet, it is thus easy to look up its communicator as the context ID is a field in the packet header. The current design supports 32 outstanding communicators with a table size of around 1 KB; this is sufficient for up to 64K MPI processes.

Once the switch has a table entry for a given communicator, it can use that data to perform any collective that uses a binomial tree, recursive halving, or recursive doubling. For any collective algorithm in a communicator, each MPI process will communicate with the same subset of MPI processes regardless of how many times the collective is called. Once a valid entry is loaded into the table, no updates on that entry are required until the communicator is freed.

3.3.2 Communicator Table (CT) Entry Creation

When a new communicator is created in software, the switch needs a way of obtaining the CT entry from the processor. A new CT entry creation function has been inserted at the end of each MPICH communicator creation function. This added function checks whether an MPI process is a member of a new communicator and, if so, calculates the subset of MPI processes with which it communicates. This requires that, for each communicator creation function call, the processor calculates the physical addresses of the subset of MPI processes that will be stored in the table entry. Once the new entry is created, the switch can handle new collectives occurring within this communicator. The addresses are then packaged alongside communicator

metadata to be sent to the switch. Although this operation does lead to a small amount of overhead in creating communicators, this overhead is only paid for once during communicator creation.

In addition to the CT, there is a forwarding table in each switch which stores a dictionary of global ranks (used by *MPI_COMM_WORLD*) as the keys with the processor physical addresses as the values. When there are multiple processes in a node, multiple global ranks associated with the processor appear in the forwarding table. This information is collected during `MPI_Init` (called only once at the start of the program). It is then successively used in the routing phase. Since each new communicator could assign a new rank for the same process, storing the information in the forwarding table for each new communicator has the potential to exhaust memory resources. Hence, the CT entry creation function sends the global ranks of the processes instead of the actual rank of that specific communicator. As a result, the switch deals with the global MPI process ranks for packet forwarding while the software on the processor considers the actual MPI process ranks (based on the communicator).

3.4 Implementation

3.4.1 Overview

The base design is a standard virtual channel router (see Figure 3-4 and (Sheng et al., 2016a; Sheng et al., 2018)) extended with the proposed streaming interface. It uses the classic four stage pipeline (Dally and Towles, 2004): route computation, virtual channel allocation, switch allocation, and switch traversal. In this example it is designed to be used in an FPGA cluster interconnected in a 3D torus and so has six input and six output ports, each connected to serial transceivers.

To minimize back-pressure from switch to processor, the size of input FIFOs should

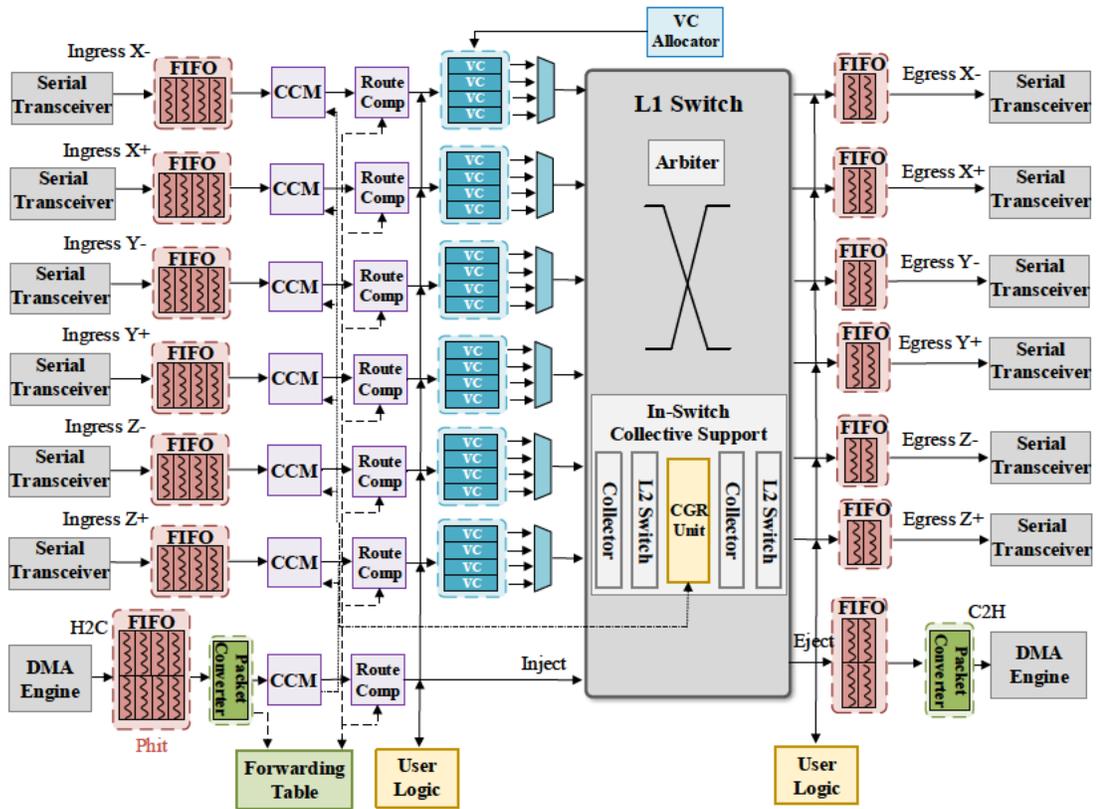


Figure 3-4: MPI-FPGA switch with collective support: Collective Control Module (CCM) and Configurable Gather/Reduction (CGR) Unit.

be sufficient to tolerate the latency of the collective tree. This happens if the tree is deep and the root has to wait a considerable amount of time for the leaf processes. We calculate the minimum buffer size as in Eq. 3.1:

$$min_buf_size = (m + (d + q) \times (\frac{L}{T_clk})) \times Ph \quad (3.1)$$

where m is the flow control margin, d is the maximum depth of the tree-based collective algorithm (accounting for the number of nodes used in the FPGA cluster), and q is a constant to provide slack for other operations to proceed. L , T_clk , and Ph are the link latency, clock period, and phit size, respectively.

The MPI offload support is designed to keep the overall design modular: the accelerator architecture is portable to any other standard router. The two key modules are the collective control module (CCM) and the in-switch collective support module. The former determines new forwarding and multicast destinations for collective packets; it also contains the communicator support. The module is placed before the router so that the packets' output ports can be calculated during the route computation stage after it has been assigned a new destination. The latter is integrated into the L1 switch (see Section 3.4.3) and is used for reordering, redirecting, and in-switch processing. Details follow in the next subsections.

3.4.2 Collective Control Module (CCM)

The CCM (Figure 3-5) performs all of the algorithmic work found in the software of *MPIReduce*, *MPIAllreduce*, *MPIBcast*, *MPIScatter*, *MPIGather*, *MPIAllgather*, and *MPIReduce_scatter*, as well as any other collectives implemented in the future. When packets enter the router, they first go through the CCM. If they are not part of a collective operation, or are not destined for the current process, then they simply pass through unchanged. If they are part of an offloaded collective

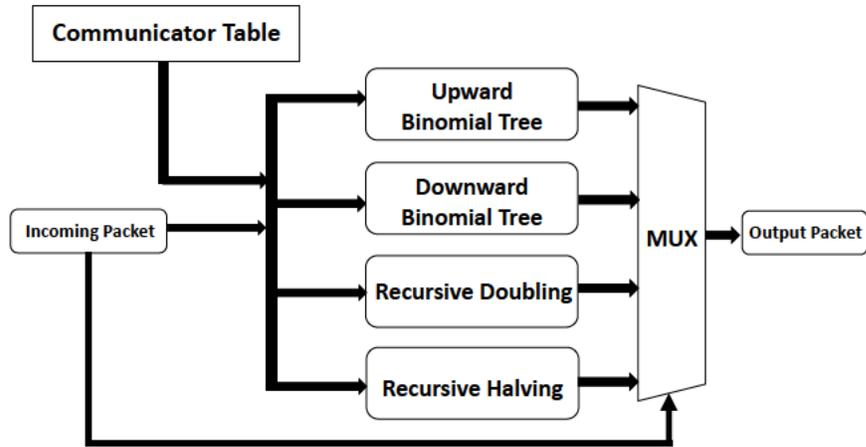


Figure 3-5: Collective Control Module

and the destination address in the packet header matches that of the current process, then the CCM uses the CT to determine new destinations for the packet.

In order for the CCM to determine which collective a packet is a part of a collective, an opcode field has been added to the packet header. With this *MPI-FPGA* can perform work for each collective algorithm in parallel and then use the opcode to decide which algorithmic results to use for the packet (see Figure 3-5). Within each of these algorithm blocks, *MPI-FPGA* performs computations using input from the packet header and the CT entry. For a reduction, the router calculates the parent node to send the packet to; or, for a broadcast, all of the child nodes to multicast the packet to.

The communicator table also eases the computation required to calculate these destinations. When a packet needs to be sent to multiple destinations, these are also adjacent in the table entry. For multicast a bit vector is used to keep track of these destinations; this results in much less work than if destinations were calculated repeatedly. Once a packet passes through the CCM, it is passed to the route computation stage (in the routing pipeline) where its output port is calculated.

As in MPICH-3.2, the implementation also supports multiple algorithms for the same collective operations. The choice of algorithm is often determined by packet size. In *MPI-FPGA* we support algorithm selection by adding bits to the packet header opcode field.

3.4.3 Two-Level Switch

As shown in Figure 3-4, the overall switch design consists of hierarchical switches (L1 and L2) for directing packets. The L1 switch directs packets to the correct output port and is followed by output FIFOs. The current design supports a round-robin arbitration policy. Note that the arbiter is disabled for the *Eject* port; once the communication between the switch and the processor is established, packets in other input ports do not contend for the *Eject* port as the network packets are again converted into processor packets. To support processing of the collectives there is an in-switch collective support module that is seamlessly integrated to the L1 switch. This module performs the gather-type (*Gather* and *Allgather*) or reduce-type (*Allreduce*, *Reduce*, and *Reduce_scatter*) operations. If network packets are identified as part of one of these collectives (by their opcode field), they are transferred to the in-switch collective support module. It has two collectors, two L2 switches, and CGR unit(s). The first collector is responsible for waiting for all packets corresponding to the child MPI processes. The first L2 switch and the last collector are only used for gather-type operations. The former reorders incoming packets according to their sender's rank, while the latter serializes the gathered data. Finally, the last L2 switch is responsible for directing the CGR outputs to the correct switch output port.

The CGR unit consists of configurable vectorized Aggregation Logic Units (AgLUs), as shown in Figure 3-6. Each vectorized AgLU is cascaded to match the number of switch ports to perform concurrent reductions with full pipelining. The aggregation is made up of two separate paths with vectorized AgLUs chained together

which are combined at the end. Each vectorized AgLU can accommodate multiple AgLUs based on the bitwidth of the datatype being used. The CGR unit also has gather and reduction tables that are indexed and capable of supporting multiple gathers and reductions, respectively, and with different communicators. There is a *counter* field inside the tables to track the number of processed phits; this is then compared with the *count* field (total number of phits according to the message size). When the threshold is reached, a completion notification is sent to the DMA engine. The CCM configures the value of the *count* field in the reduction (gather) table according to the processor packet header. *Count* and *counter* fields have the same bitwidths as the message size in the processor packet header. There are six columns for the reduction (gather) matching the switch radix for the current design (3d-torus), each with a bitwidth equal to the phit size.

To support multiple simultaneous reductions (gathers), there are SA (Simultaneous Aggregations) number of CGR units. Clearly, SA can be equal to the radix of the switch; but it also can be fewer depending on the available hardware resources (discussed in Sec. 3.6.1). There are $R \times 4$ AgLUs (64-bit) for each CGR unit (with a double data type), where R is the radix of the switch. This results in $SA \times R \times 4$ AgLUs in the L1 switch.

The arithmetic unit is constructed using standard methods including the use of vendor IP. The current design supports sum, min, max, XOR, AND, and OR but is trivially extendable for other operations, including user specified functions.

This approach is flexible enough to support multi-user processing. When the number of jobs is small – not larger than the number of AgLUs in a VAgLU – it is possible to divide phit size channels (each arrow in Figure 3.4), into 64 bit sub-channels and assign each of them uniquely to each job. In this case, there is no resource/time sharing involved in the switch and each AgLU is mapped to a single job.

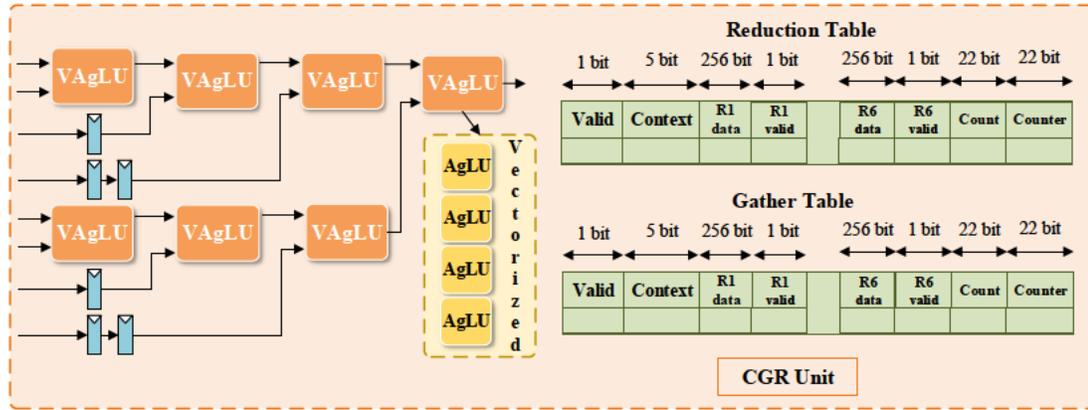


Figure 3-6: Configurable Gather/Reduction (CGR) Unit: aggregation has two paths with vectorized AgLUs chained together and combined at the end.

When there are more jobs, it is possible to employ time slotting (Byma et al., 2014; Khawaja et al., 2018). In either case, if the computation requirements of a certain application surpass the operations that are supported by the AgLUs, or there is a need to add user logic for inline pre/post-processing (see Figure 3-4), it is possible to take advantage of partial reconfiguration. For the former case, partial reconfiguration does not disturb other existing jobs but, for the latter, it would introduce minimal idle time (depending on the amount of new logic) as only the AgLUs and the user logic would be reconfigured, rather than the entire design.

3.4.4 Considerations for Indirect Networks

The *MPI-FPGA* switch design for indirect networks follows the same overall design as Figure 3-4 but with some differences:

- In the direct model, each FPGA card is attached to a processor through a PCIe interface. In the indirect network model, FPGA switches are attached to the processors through network ports.
- For the indirect model, storing information in the forwarding table is a two-step

process. During `MPI.Init`, a dictionary — where the keys are the global process IDs and the values are the port numbers — is sent from the processors to each of the corresponding leaf switches and then from leaf switches to spine switches. As a result, each spine switch maintains a dictionary of all the processors within the system inside its forwarding table;

- The *count* and *context ID* fields in the reduction/gather tables of the spine switches are configured by sending the information from the leaf switches to the spine switches based on the collective operation shown in the incoming packet header;
- For the leaf switches, the number of packet converter modules is equal to the number of downlink ports.

3.5 Performance Analysis of In-Network Collective Offload

In order to develop a general understanding of how this approach could improve the performance of MPI collectives and the conditions that need to exist for this approach to benefit application performance, we consider a LogGP model (Alexandrov et al., 1995). L , o , g , G , and P represent the latency, overhead, gap between messages, gap per byte, and number of processors, respectively. As is standard in reports on hardware implementations of communication operations, including collectives, we assume that all of the processes start at the same time with no skew. As it has been stated previously, substantial process skew necessarily diminishes the performance benefit of any aggregation algorithm (Graham et al., 2016). Skew is discussed further in (Haghi et al., 2021).

We compare two models: a baseline with no compute-in-the-switch and the new one where the switch is able to perform the computation, multicast, and reordering.

Since the total latency of an MPI operation is determined by the slowest process; i.e., the maximum latency of any process, processes having different roles during the collective need to be considered. For *MPI-FPGA*, we consider the latency of two processes in an MPI_Bcast for a binomial tree (see Figure 3.1 (a)): one receiving a number of messages (e.g., process 7) and one sending multiple messages consecutively (process 0).

We first derive expressions for the time spent on the second process type. Let $T_{Bl,i}$ and $T_{Is,i}$ denote the time needed for the i^{th} process to finish the MPI operation in baseline and in-switch models, respectively.

$$T_{Bl,P-1} = \lfloor \log(P) \rfloor \times (O_s + (k \times G) + L + O_r) \quad (3.2)$$

$$T_{Is,P-1} = O_s + (k \times G) + L + O_r \quad (3.3)$$

We distinguished the overhead time on the sender side (O_s) from that on the receiver side (O_r); these are the times it takes for the processor to transmit and receive a message, respectively. Note that $\lfloor \log(P) \rfloor$ is reduced to 1 in Eq. 3.3. These equations indicate that the *MPI-FPGA* approach effectively eliminates the overhead time (O_s and O_r) for intermediate MPI processes since communicator support is offloaded into the FPGA fabric. Also, the above model suggests that the in-switch approach unlocks higher performance for collectives involving many-to-many communication patterns (Allreduce, Allgather, Reduce_scatter, and Barrier) as the number of times that CPUs are bypassed is higher in these collectives. The in-switch approach also scales better with the number of nodes. Finally, note that the term G in the baseline is greater than for in-switch: the cost of the transport protocol is high compared to that of the light-weight transport engine.

We now analyze the time spent on the root process (process 0):

$$T_{Bl,0} = O_s + ((\lceil \log(P) \rceil - 1) \times \max\{g, O_s\}) + (\lceil \log(P) \rceil \times k \times G) \quad (3.4)$$

$$T_{Is,0} = O_s + (k \times G) \quad (3.5)$$

In the in-switch model the g term is eliminated by the pipelining communication. For example, for MPI_Bcast the message is sent from the processor to the switch only once where it is then multicast to the different MPI processes. For MPI_Scatter, different messages are lumped into a large message. Consequently, for the in-switch model $g=0$ and assume only G for both short and long messages. Clearly if the message size is small, then the G terms in all of the above equations are irrelevant. One observation is that, for the in-switch model, the coefficient $\lceil \log(P) \rceil$ is reduced to 1 in Eq. 3.5 as the switch benefits from in-network multicast. Another observation is that a multi-port switch attached to each node in the direct network reduces the *effective* G for intermediate MPI processes as multiple messages can be sent simultaneously.

One major benefit of the in-switch model is in-network reduction; *inline* high-throughput reduction is performed in the switch, which eliminates the need to perform the reduction in the CPU. This is especially beneficial for large messages. Assuming n MPI processes involved in the reduction collective, in the baseline model the operation could be performed $n-1$ times. In the in-switch model nearly all of these operations are eliminated owing to the pipelined, high-throughput AgLUs. Another advantage of the in-network reduction is reducing network traffic and the number of data copies in the processor. From the above discussion, we infer that the application characteristics that maximize *MPI-FPGA* benefit are, first, a large number of MPI collectives, especially Allreduce and Allgather, and, second, reductions with large message sizes.

Table 3.2: Inter-FPGA Latency and Bandwidth for Xilinx Alveo U280

Parameter (FPGA-to-FPGA)	Latency (ns)	Peak Bandwidth (Gb/s)
Value	440	95.9

3.6 Evaluation

We implemented and tested *MPI-FPGA* on a two-node FPGA system using Xilinx Alveo U280 boards and the standard Xilinx development tool suite. Each board exposes two 100 Gb/s QSFP128 interfaces and a PCIe Gen3 x16 interface. Vendor-provided IPs (Xilinx QDMA with AXI4-Streaming interface and Aurora) were used to implement the DMA engine and transceivers. The design is coded in *Verilog HDL*. We have provided the FPGA-to-FPGA latency and peak bandwidth of the Alveo U280 card in Table 3.2. The bandwidth starts to saturate at a message size of just under 1 KB. The information in this table is used during the simulation, which is discussed in the next paragraph.

Given the challenges of HDL coding, an efficient cycle-accurate simulator is essential for exploring a large number of nodes. The simulator used in this chapter is an version of that described in (Sheng et al., 2018; Yang, 2019) updated by changing the transceiver parameters collected from the two-node testbed (table 3.2). The simulator is implemented in C++; every hardware module in the RTL model has a corresponding class in the simulator. These classes are organized in the same hierarchical structure as the RTL model. To give the cycle-accurate simulator good extensibility, we define an interface standard for all hardware modules. We adopt a producer-consumer model with every module being both a producer and consumer. The simulator has been validated with respect to the RTL code for the two-node FPGA system; the behavior of RTL simulation matches the simulation.

For cluster tests, we target Xilinx XCVU13P FPGA devices. The performance

Table 3.3: Resource usage on Xilinx XCVU13P FPGA Devices

Design	LUT	FF	DSP	BRAM	URAM	#SA	#AgLU
Direct	401852 (23.3%)	448117 (13.0%)	1441 (11.7%)	454 (16.9%)	6 (0.4%)	6	144
Indirect-8	700409 (40.5%)	1062680 (30.7%)	2561 (20.8%)	523 (19.4%)	0 (0%)	8	256
Indirect-16	1408301 (81.5%)	2126184 (61.5%)	5121 (41.7%)	987 (36.7%)	0 (0%)	8	512
Indirect-28	1551853 (89.8%)	2349600 (68.0%)	4481 (36.5%)	1543 (57.4%)	0 (0%)	4	448

results for the FPGA cluster are obtained from a cycle-accurate simulator. Resource utilization is reported using *Vivado Design Suite 2019.2*. The operating frequency of the current implementations is 250 *MHz*. For the CPU reference, benchmarks were run on the Stampede2 (Stanzione et al., 2017) Skylake (SKX) compute cluster, accessed through XSEDE, 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 (fat tree topology). We used Intel MPI 18.0.2 as an Intel-compatible MPI compiler and launcher as recommended for the TACC Stampede2 cluster. We found that it usually gives a better performance compared to MPICH 3.2.

To assess the efficiency of *MPI-FPGA*, we compare the performance with respect to the CPU cluster for *Allgather*, *Allreduce*, *Broadcast*, *Gather*, *Reduce*, *Reduce_scatter*, and *Scatter* operations using the OSU micro-benchmarks (v5.6.2) (?). The study investigates a range of message sizes. For FPGA-based indirect networks, we considered three switch designs: indirect-8, indirect-16, and indirect-28, which denote switches with radix-8, -16, and -28, respectively. Radix-28 was the highest radix we could implement on the XCVU13P FPGA according to the number of available GTY transceivers.

3.6.1 Resource Utilization

FPGA resource consumption is shown in Table 3.3 for four different designs (direct and indirect networks). As discussed previously, each AgLU supports a different set

of operations. Depending on application requirements, the user can remove the support for unused operations or add user-defined operations; these actions decrease or increase resource utilization, respectively. The current result is based on using floating point add operation (double-precision). Overall, BRAM utilization is dominated by serial transceivers, input FIFO buffers, and the DMA engine, while the L1 switch accounts for a large fraction of DSP, LUT, and FF utilization.

According to Table 3.3, the overall utilization increases with the radix of the switch. For all designs except indirect-28, the number of SAs is equal to the number of ports. There the number of SAs needed to be reduced in order to fit the entire design onto the FPGA. This is because the number of AgLUs for each CGR unit increases with the radix of switch. The critical resource is LUTs with the CGR units constituting a large fraction of total LUT utilization.

3.6.2 Performance of MPI Collectives

We evaluated *MPI-FPGA* for seven different MPI collectives using the OSU Micro-benchmarks. For all collectives, double precision floating point was used. The results were averaged over 1000 iterations (with 200 warm-up iterations) and five different runs (different node allocations).

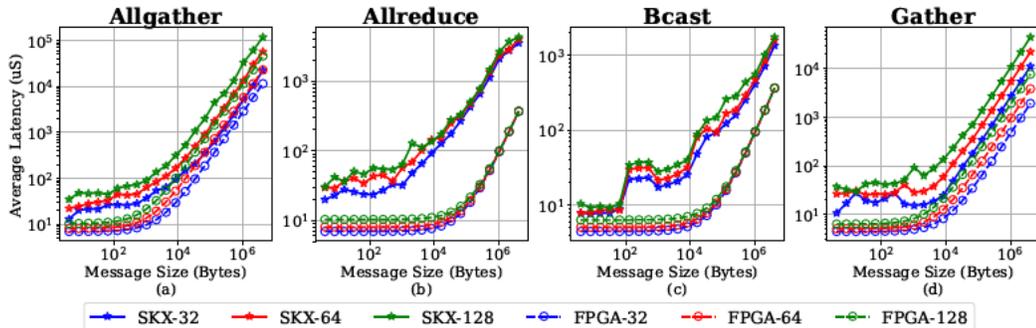


Figure 3.7: MPI CPU cluster (SKX) vs *MPI-FPGA* execution times for 32, 64, and 128 nodes: (a) `osu_allgather`, (b) `osu_allreduce`, (c) `osu_bcast`, and (d) `osu_gather`.

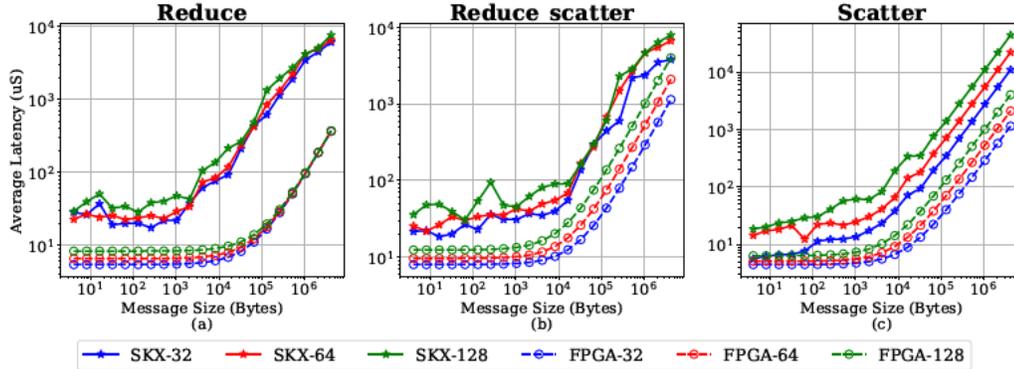


Figure 3-8: MPI CPU cluster (SKX) vs *MPI-FPGA* execution times for 32, 64, and 128 nodes: (a) `osu_reduce`, (b) `osu_reduce_scatter`, and (c) `osu_scatter`.

Overall Collective Latency Figure 3-7 and Figure 3-8 show the simulation results of MPI and *MPI-FPGA* collectives for small (4 Bytes to 4 KB) and medium-to-large message sizes (4 KB to 4 MB) for 32, 64, and 128 nodes using the OSU benchmarks. The reported average latency is the average time it takes for the processes to finish the operation. Processor-FPGA communication latency is included in the time. To isolate the impact of the design under study, e.g., from contention at the PCIe interface, we focused simulations with one process per node.

One of the advantages of *MPI-FPGA* is that utilization of the application layers in the network stack (such as MPI) can be bypassed for the nodes associated with root and intermediate processes because communicator support is offloaded, while reduction operations (if any) can be performed by a network switch. Although having a low-latency network topology (such as a fat tree) for the reference CPU cluster (as opposed to 3D torus for the FPGA cluster) can offset the aforementioned benefits, we observe that *MPI-FPGA* has a higher overall performance. As is evident from Figure 3-7 and Figure 3-8, *MPI-FPGA* speedup relative to the CPU cluster is higher for *Allreduce* operation, since a greater number of nodes corresponding to the intermediate processes are involved and data volume is reduced during transfer. For the `Reduce_scatter` operation, the switch corresponding to the root process is able to

perform the reduction on incoming data received by ingress ports and then scatter the result directly. This effectively bypasses the processor as it eliminates the need to transfer data to the processor to perform reduction and scattering.

To view the results from a different perspective, Figure 3-9 shows the average speedups for each of these collectives on 32, 64, and 128 nodes. The geometric mean is used to summarize *MPI-FPGA* speedup ratios with respect to different message sizes. The speedup ratio ranges from $2.0\times$ to $32.6\times$.

MPI-FPGA achieves higher performance than the CPU cluster for both small and medium to large message sizes. The central reason is that *MPI-FPGA* does compute-in-the-network. This bypasses the processor in the nodes corresponding to the root and intermediate processes, which in turn reduces the latency, especially for short messages. Also, *MPI-FPGA* utilizes a streaming interface with fine-grained communication as opposed to processor-based bulk transfers involving large buffers; this helps for long messages.

With respect to problem size, of note is that the *MPI-FPGA* speedup is maintained as the number of processes grows; this indicates the expected benefit for larger systems through reduced network traffic. Interestingly, the speedup usually increases for larger systems, especially for medium-to-long message sizes, showing that the advantage of *MPI-FPGA* scales. According to the results in Figure 3-9, the speedup in medium-to-large messages is higher than for small messages (except Allgather) due to the efficient streaming-based interface. An exception is Allgather where there is more wait time and traffic, the latter since the volume of data expands as it traverses the network.

In order to characterize the latency variation of MPI collectives across different iterations and different runs, we have tabulated the standard deviation (std), maximum, and minimum latency for messages of size 1 KB and 1 MB on 32, 64, and 128

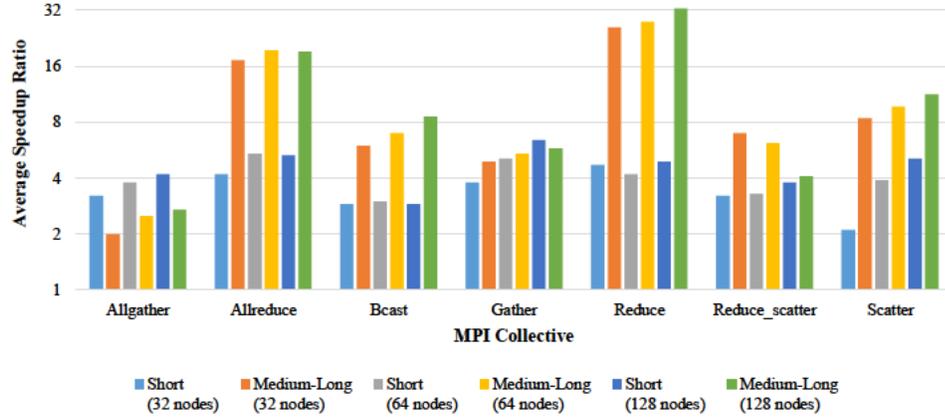


Figure 3-9: Average *MPI-FPGA* speedup ratios for OSU Benchmarks running on 32, 64, and 128 nodes of Stampede2 for short and medium to long messages.

Table 3.4: MPI collective latency variation on the Stampede2 compute cluster for message sizes 1 KB and 1 MB on 32, 64, and 128 nodes.

MPI Collective	32 nodes						64 nodes						128 nodes					
	1 KB			1 MB			1 KB			1 MB			1 KB			1 MB		
	std	max	min	std	max	min	std	max	min	std	max	min	std	max	min	std	max	min
Allgather	6.7	48.8	27.1	250.3	5466.9	4645.1	12.2	87	44.8	808.5	14164.9	11327	17.2	130.2	66	990.7	26439	23233.2
Allreduce	6.3	46	21	81.4	2224.9	1896.1	10.7	74.1	36.9	182.9	2614	1940	11.6	86.1	33.2	291.4	2985	1997.9
Bcast	0.6	18.1	16	39.5	493	349	2.1	26.2	17.8	35.2	546.2	412	3.4	36	22.8	116	816.8	402.9
Gather	4.2	26.9	9.8	5	2739	2712	4.4	41	21	7.4	5465	5425.9	25.9	151.1	44.1	35.3	11007	10847
Reduce	6.8	39.1	11.9	151.6	3720.1	3089.9	11.2	57	9.7	217.3	4524.9	3695.1	12.2	72	21.9	255.8	4709.9	3807
Reduce_scatter	6.6	38.2	16.9	113.4	2577.9	2033.9	10.1	62	25	187.2	4908.9	4150.1	8.4	61	31	203.6	4904.2	4195.7
Scatter	3.1	24.8	11	12.1	2781.8	2729.1	3.9	34.1	21.9	16.6	5549.2	5483.8	10	95.1	45.8	20.8	11018	10937

nodes (SKX cluster), see Table 3.4. As expected MPI collectives for large messages (1 MB) have larger variation than those of small messages (1 KB) and the deviation usually increases as we scale out.

Indirect Network Study Figure 3-10 shows the average latency of two MPI collectives, Allgather and Allreduce, for small message sizes (4 Bytes to 4 KB). Four different kinds of networks are considered: direct, indirect-8, indirect-16, and indirect-28. These are a direct network with 3D torus topology and indirect networks with radix-8, -16, and -28 switches, respectively. The switches in the indirect network have a default over-subscription of 3:1. Overall, use of an indirect network reduces the average latency compared with the direct network. As expected, switches with higher radix result in more performance benefit. There are cases, however, in which

this does not hold, e.g., in (Figure 3-10 (f)). There, having radix-16 switches could not reduce the number of hops compared to the radix-8 switches. This slightly affects the performance as higher radix switches are more complex and have more AgLUs. It should be noted that, for Allgather, average latency increases more rapidly compared to Allreduce as packets are combined throughout the network.

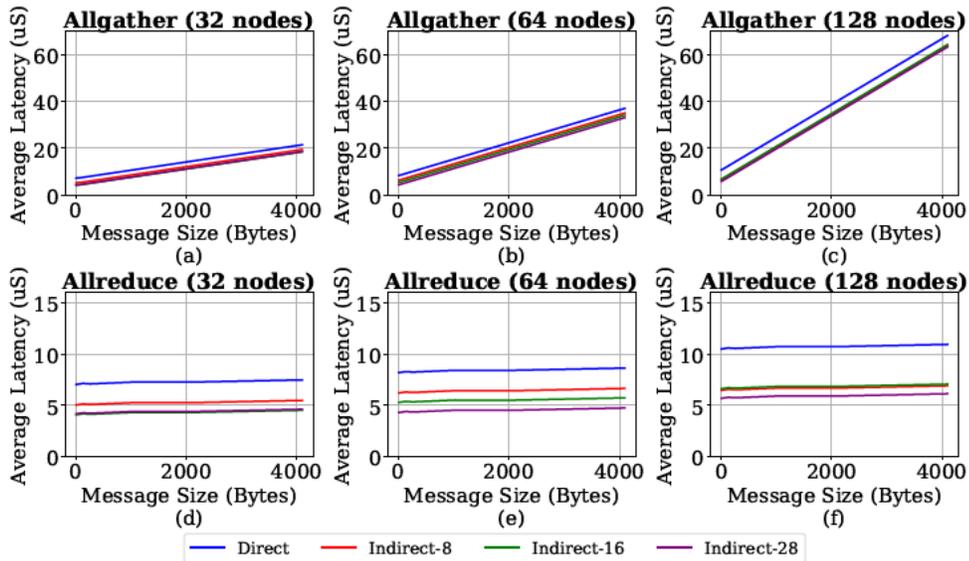


Figure 3-10: Performance comparison for short messages of *MPI-FPGA* with direct and indirect networks on 32-, 64-, and 128-node systems for Allgather and Allreduce operations.

Of interest is the comparison of *MPI-FPGA* with the commercial ASIC-based version of in-switch offload of collectives from Mellanox. Unfortunately gaining access to SHArP-based systems is still extremely challenging. SHArP only supports Allreduce and Barrier collectives (Graham et al., 2016) while the reconfigurable approach embraces a diverse and extensible set of collectives. Although results published in SHArP may not be directly comparable to those described here, as they are independent sets of experiments, it appears that the *MPI-FPGA* approach is competitive with SHArP and could outperform SHArP beyond a certain message size (Graham et al., 2016). One likely reason is that reduction of messages that are larger than

SHArP hardware maximum message size are performed by posting multiple reductions. In contrast, in *MPI-FPGA* only one request is sent. Also, *MPI-FPGA* has an efficient streaming interface where large processor packets are converted internally to multiple small network packets. There is a newer version of the SHArP protocol which is based on streaming aggregation interface, but it is not optimized for small message size (Graham et al., 2020) and the latency is higher than that of the original work (Graham et al., 2016).

Figure 3-11 summarizes the average speedup of indirect network switches over the baseline direct network switch (from Figure 3-10) for short messages. The geometric mean is used to summarize speedup ratios over different message sizes. The overall speedup ratio is between $1.2\times$ to $1.9\times$.

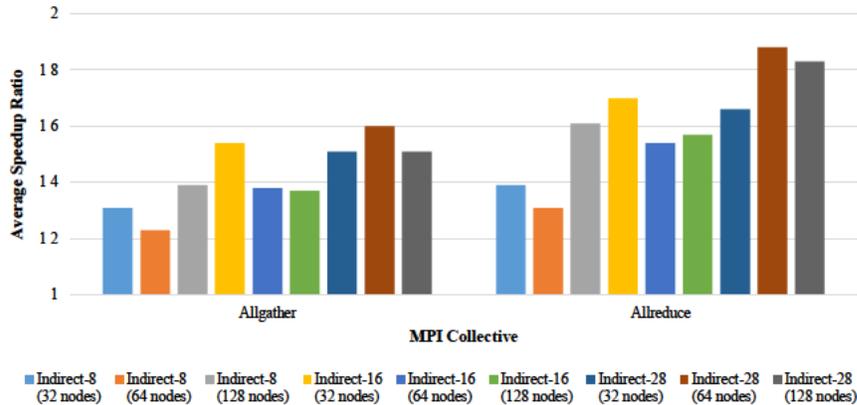


Figure 3-11: Average speedup ratios of indirect network switches over direct network switches for short messages.

3.6.3 MiniApp-Level Benchmarking

To show the efficacy of *MPI-FPGA* in application performance, we have benchmarked various HPC kernels and pseudo applications using NAS parallel benchmarks (NPB) as well as two MiniApps from the Mantevo Project (miniFE and HPCCG).

When running the benchmark codes on the FPGA cluster we found that the addition of barriers simplified the instrumentation. To ensure that we compared the

FPGA cluster results with the best possible baseline, we also created versions of the baseline codes with barriers. For the baseline cases, original and “barrier,” we found that the performance of the two versions was indistinguishable.

For NPB, class A is used for CG and SP, the others (IS and MG) are benchmarked with class C. The results are averaged over five runs. Table 3.5 compares NPB, miniFE, and HPCCG for the CPU cluster (SKX) and *MPI-FPGA* for 64 and 128 nodes. Analyzing the NPB results, it can be inferred that *MPI-FPGA* improvement increases scales with the number of nodes. This aligns with the discussion of the LogGP model in Section 3.5. For MG, *MPI-FPGA* achieves considerable improvement (about 2% on 128 nodes) as there are a large number of MPI collective calls in this benchmark. For IS there is an MPI_Allreduce with a large message size in which *MPI-FPGA* benefits from in-network computing (about 3% improvement on 128 nodes).

For miniFE and HPCCG, we used 48 OpenMP threads on SKX as this number of threads yielded the highest performance according. The problem size used for miniFE application on 64 (128) nodes is $512 \times 256 \times 64$ ($512 \times 512 \times 64$). When scaling up, the problem size per processor is kept fixed (weak scaling). Our approach provides satisfactory speedup as one of the Allreduce collectives is called for 400 times. Weak scaling is measured for the HPCCG benchmark as well (the grid dimension on each process is $100 \times 100 \times 100$). Similar to miniFE, one of the Allreduce collectives is called for a large number of times (298).

Figure 3-12 shows the CPU cluster (SKX) execution time and *MPI-FPGA* speedup for Allreduce (Reduce) collectives used in (a) NPB and (b) miniFE and HPCCG. Because each application may have a large number of collectives, we only used Allreduce: these are common and account for most of the collective execution time. Each instance is evaluated separately. Also, for those collective instances with multiple executions, we take the arithmetic mean of their execution times (maximum time

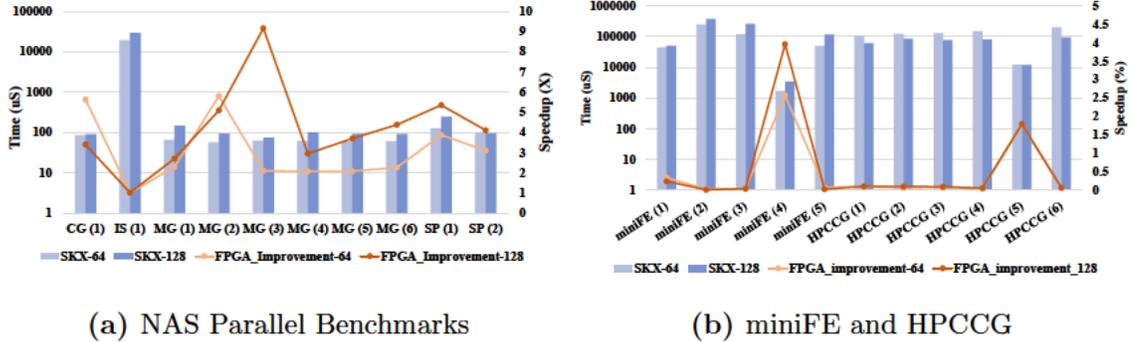


Figure 3.12: CPU cluster (SKX) execution time and *MPI-FPGA* speedup of Allreduce (Reduce) collectives used in the applications under study. Each collective instance is evaluated separately, e.g., with MG having six instances of Allreduce.

Table 3.5: Performance comparison of NAS parallel benchmark, miniFE, and HPCCG, for MPI CPU cluster (SKX) vs MPI-FPGA for 64 and 128 nodes. *For the SP benchmark the number of processes must be a perfect square.

Benchmark/ Application	64 nodes			128 (121*) nodes		
	SKX (ms)	FPGA (ms)	Improvement (%)	SKX (ms)	FPGA (ms)	Improvement (%)
CG	40.3	39.7	1.45	69.9	68.5	2.06
IS	151.7	150.3	0.89	88.3	85.7	2.96
MG	536.7	532.4	0.80	303.3	294.5	2.92
SP*	708.3	707.7	0.08	626.7	625.5	0.18
miniFE	32886.9	32868.2	0.06	71052.9	71037.8	0.02
HPCCG	24890.9	24858.9	0.13	58268.6	58247.4	0.04

among all processes). The exception is CG which does not use Allreduce: Reduce was used instead.

3.7 Conclusion

In this chapter we present a comprehensive solution to processing collectives in network switches with reconfigurable logic. This has the advantage over fixed logic alternatives of being able to support capabilities as needed. This includes varying the supported operation types and numbers of simultaneous operations, but also enabling more general user-defined processing in the network.

As part of this solution we present a new method for supporting MPI communicators and accelerating collectives in the network switch. We begin by considering the movement towards exascale computing and the need for offloading collectives and

communicator support into hardware, in particular, for collectives occurring over irregular communicators. We find that storing entire process groups in the network is not a scalable solution. We then introduce the Communicator Table (CT), which takes advantage of the properties and patterns of collective communication in order to provide the accelerator hardware with the minimum amount of communicator information needed to perform collectives.

A novel 2-level switch design is introduced to efficiently process in-network collectives; yet it remains flexible enough to embrace user-defined collectives. We provide a streaming interface to improve the performance for long messages. By supporting a full offload of seven popular collectives, we remove nearly all of the collective operation software from MPI and implement the functionality in the switch. The hardware support has been integrated into a reconfigurable router which remains portable enough that it is independent of the type of router. We evaluate *MPI-FPGA* with respect to a CPU cluster and find that the in-switch accelerator achieves significant and scalable speedups.

Chapter 4

Type 2 ACiS: Look-Aside Acceleration

This chapter introduces look-aside (stateful) capabilities. In this type, communication is processed not just through transforming and merging streams but also by recycling, looping, and saving states. For example, many compression algorithms require look-aside processing. In the first part of this chapter, we consider a direct network with a distributed dense matrix multiply as a case study. However, it suffers from two limitations: First, while direct networks are a good testbed for in-switch computing their use cases in real scenarios are limited; current supercomputers and data centers rely on indirect networks. Second, while the target platform is reconfigurable the approach taken for the acceleration is fixed meaning that it takes considerable effort to design and implement a new kernel or application. In the second part, we focus on an indirect network with a graph convolutional network (GCN) case study. This time, we take another approach for the acceleration: we design a programmable yet high performance in-switch accelerator where users can simply write a packet handler in high-level language to define the functionality of the switch with software-speed compilation time. The key ACiS plugin (for the second part) is a programmable CGRA. This chapter is based on the work published in International Conference on Supercomputing (ICS) ©2023 ACM (Haghi et al., 2023a) and International Conference on Field-Programmable Technology (ICFPT) ©2020 IEEE (Haghi et al., 2020b).

4.1 Introduction

A growing trend in HPC is the importance of the network in application support. For example, offload of collective processing into SmartNICs (Arap and Swamy, 2016; Guo et al., 2022a; Guo et al., 2023; Sheng et al., 2016a) is well-established and has several benefits. But while SmartNICs are invaluable, this scheme still forces processing into endpoints. Another approach is to offload collective processing into switches (Graham et al., 2016; De Sensi et al., 2021), which has additional advantages. Current in-switch processing, however, is limited in a number of ways. Commercial implementations only support collectives, and this support covers only a small set of scalar and fixed function operations (e.g., (Faraj et al., 2009; Graham et al., 2016)). While academic work (Stern et al., 2017; Haghi et al., 2022) demonstrates support for user-defined extensions, the resulting switches are still confined to inline (aka streaming (Krishnan et al., 2020)) processing.

We posit that beyond collectives there are additional acceleration opportunities that are not feasible in the current streaming-only paradigm. To tackle these challenges, we propose a programmable *look-aside* (LA) accelerator that can be integrated into, or attached to, existing communication switch pipelines. Recently, look-aside accelerators have been proposed for SmartNICs (Krishnan et al., 2020); their efficacy has been demonstrated in applications including machine learning at the edge, data compression, and storage disaggregation. Extending LA capabilities from NIC to switch yields benefits analogous to those just enumerated for extending collective support.

Increasing switch flexibility has long been a goal of the networking community, culminating, in part, in programmability using P4 (Bosshart et al., 2014). P4 is a high-level language used to control packet forwarding. While there has been some exploration of application-level processing (Sankaran et al., 2021) in P4 switches,

including support of collectives (Sapio et al., 2021), these capabilities are currently limited and likely to fall short in certain applications, including machine learning (Swamy et al., 2022). More specifically, P4-based switches suffer from having a limited set of operations (e.g., no multiply), data types (e.g., no sparse data types), and memory footprint. Perhaps most significantly, packets can only access each memory location once within a traversal (De Sensi et al., 2021); while it is possible to recirculate packets, this technique reduces throughput. To address the above limitations, the proposed programmable in-switch accelerator is designed to handle more complex calculations, such as fused multiply-accumulate (MAC) and sparse accumulation, off-chip memory access, and data reuse.

Adding LA switch support confers several additional advantages to in-switch computing. Most importantly, it improves application scalability. For instance, during scale-out, inference communication time for graph convolutional networks (GCNs) with real-world graphs can quickly outweigh the total execution time. LA switch support can improve scalability by reducing communication data volume as the switch can aggregate data and act as a storage device.

In the first part of this chapter, we propose a novel in-switch hardware accelerator design, called FPin, that integrates kernel-level optimizations. We focus on distributed matrix multiplication. There are several problems that must be solved. (1) Existing support of collective offload only exploits the optimization opportunities at a per-operation level; kernel-level global optimization has been neglected. (2) Since the computation may be more complex than naive reduction, providing full overlap of computation and communication should still maintain line-rate packet processing. (3) In-network processing usually leverages streaming data with limited local state (Hoeffler et al., 2017). In contrast, complex kernels often require a large amount of data reuse to achieve higher performance which is not supported in current hard-

ware collective offloads. To tackle the problems mentioned above, first, we propose a novel collective offload design which integrates kernel-level optimizations for matrix multiplication. Second, we propose a smart offload engine to efficiently handle communication-computation overlap and matrix multiplication scheduling. In other words, *scheduling* is offloaded to the accelerator hardware; by pushing most of the logic from software to hardware our approach is capable of improving the performance of applications.

The first work has some limitations; first, it is only for direct networks. Second, while the target platform is reconfigurable the approach taken for the acceleration is fixed meaning that it only works for matrix multiplication.

Thus, in the second part of this chapter, to support different workloads and enable software-like programmability, while achieving near-ASIC performance, we use a coarse-grained reconfigurable array (CGRA) overlay architecture (Taras and Anderson, 2019). The proposed dataflow architecture is itself RISC-V compatible with a subset of scalar and vector instructions (Waterman and Asanovic, 2017). The CGRA is composed of multiple vector processing elements (VPEs) pipelined together based on the applications’ needs. Both on-chip (vector register files) and off-chip memory (HBM banks) are accessible through the datapath. To efficiently process machine learning workloads the ISA is extended with sparse vector instructions.

Two critical challenges are addressed. First, to facilitate usability, a software framework has been created to compile user-provided C/C++ codes (packet handlers) into back-end instructions for configuring the switch. Compile time is negligible, especially with respect to that of high-level synthesis (HLS) tools. And second, despite adding complexity, the accelerator does not compromise line rate. A number of optimizations are implemented to improve throughput, including vector instructions with large vector length and a technique called *idle tile skipping* to avoid stalls.

We propose an in-switch computing framework, *FLASH*, to support generic application-level acceleration. Our approach is orthogonal to programmable switches; we do not invent a new programmable switch or a language. The accelerator can be integrated or attached to existing switches to accelerate parts that are communication-intensive with coupled computation. While FLASH can accelerate a variety of workloads, we consider GCN inference as a case study. We summarize the contributions of this work:

- Extending look-aside capabilities from NICs to switches for enhancing support of application-level processing;
- Addressing the major challenges of extending LA capabilities to switches – i.e., maintaining full switch functionality and usability – through a novel combining of CGRA architecture with RISC-V instruction support (§4.3);
- A software toolchain to compile user-provided packet handlers to the instructions for configuring the accelerator at software speed (rather than HLS §4.4);
- The first in-switch GCN inference accelerator; and
- Experimental results showing that FLASH improves the performance and scalability of GCN applications. The performance advantage is on average $3.4\times$ (across five real-world datasets) on 24 nodes (§5.7).

4.2 Background, Motivation, Basics

We describe the GCN algorithm used to showcase LA in-switch computing, then discuss motivation, enumerate limitations of current programmable switches, and present the FLASH models.

4.2.1 Graph Convolutional Network (GCN)

We provide GCN background and elaborate on distributed GCNs.

GCN Background

Equation 4.1 shows the layer-wise forward propagation in a multi-layer GCN.

$$X^{(l+1)} = \sigma(AX^{(l)}W^{(l)}) \quad (4.1)$$

We denote A as an adjacency matrix such that $A_{u,v} = 1$ if and only if vertex u and v are connected by an edge, otherwise, $A_{u,v} = 0$. $X^{(l)}$ is a matrix denoting the features at layer- l . $W^{(l)}$ is the weight matrix at layer- l . Finally, $\sigma(\cdot)$ denotes a non-linear activation function.

Multiplying $(A \times X)W$ first results in a sparse-sparse matrix multiplication that produces a large dense matrix. Previous work (Geng et al., 2020; Geng et al., 2021c) found that the order of computation, $A \times (X \times W)$, greatly reduces the scale of computation since both are sparse-dense matrix-multiplications (SpMM). We therefore first compute the product of feature and weight matrices, which is the called combination phase. Subsequently, matrix A is multiplied with the result of combination phase (updated feature matrix); this is called aggregation. Similar to the prior art, we follow a 2-layer vanilla GCN model (Geng et al., 2020). From now on, for a SpMM computation, we refer to the first (sparse) matrix as LHM (left-hand matrix) with the size $m \times k$ and the second (dense) matrix as RHM (right-hand matrix) with size $k \times n$.

GCNs are good candidates to benefit from in-switch LA. First, their sparse connectivity leads to a higher communication-to-computation ratio compared with, say, dense matrix operations (Tripathy et al., 2020), which leads to worse scalability. Second, computation is coupled with communication in distributed SpMMs: the same

data are used for both computation and communication.

Distributed GCNs

GCNs typically operate on large and irregular input graphs with small models, i.e., with few layers. This is in contrast to DNNs where the model and collection of input samples are large, but the size of each sample (e.g., image) is small. In some cases, these graphs are so large that they cannot be stored in the memory of a single node (Jia et al., 2020). Thus, for training, it is common to employ sampling techniques so that the data can fit in the memory of a single device, but at the cost of reduced accuracy (Jia et al., 2020). In contrast, for inference, the whole graph is typically processed (in one batch). More than 90% of infrastructure cost is due to inference and less than 10% is due to training on AWS (AWS, 2019; Gasteiger et al., 2022). And inference is also necessary during training.

The interest here as a case study is that scalability is essential for high performance and accurate GCN inference and can be achieved more easily by enhancing a cluster with FLASH. This is shown in Figure 4.1 (a), which depicts results from a CPU cluster (with Skylake processor) and that cluster enhanced with FLASH. Each node runs 24 processes; §5.7.1 has details.

Accelerating distributed GCNs with FLASH poses challenges:

Challenge 1: In addition to local computation, an efficient partitioning method and communication scheme is needed.

Proposed solution: We provide such a partitioning method among nodes and switches and a novel communication scheme with overlapping between offloaded and non-offloaded parts (§4.2.1).

Challenge 2: Graphs are extremely sparse (Zhang et al., 2021) leading to irregular communication and local computation.

Proposed solution: FLASH streams data (instead of sending several messages destined

to different processes) in a fine-grained manner with pipelining of communication and computation in the switch.

Challenge 3: Graph sparsity can also cause large idle time in the switch pipeline; switches are not aware of the sparsity distribution in advance.

Proposed solution: we propose a runtime technique called *idle tile skipping* to advance the control flow for idle tiles (§4.3.2).

As discussed previously, there are two SpMMs in each layer. The combination phase is usually done locally since the weight matrix is so small it can be replicated among processes (Tripathy et al., 2020). The aggregation phase, however, is performed distributively. Thus, we only accelerate the aggregation phase. In distributed matrix multiply algorithms, it is customary to divide LHM into two parts: diagonal and off-diagonal (Park et al., 2015a). Figure 4.1 (b) depicts the partitioning of both LHM and RHM matrices into four processes (row-wise).

It is evident that the diagonal part contributes to local computation since the associated RHM part is already stored in the same process. On the other hand, the off-diagonal part requires communication with other processes since the associated RHM parts are not stored in the process locally. Typically to accelerate distributed SpMMs, each process obtains RHMs from other processes with an Allgather after which each process performs the calculation locally (Tripathy et al., 2020). The Allgather is an all-to-all type communication with a large message size which can be a bottleneck, especially for large-scale datasets (§4.6.2 and §4.6.2). Instead of a large bursty communication followed by the computation, we use a pipelined fine-grained communication that overlaps local computation.

Partitioning and Overlapping in FLASH

In this subsection, we describe how FLASH distributes GCN inference and how the workloads are partitioned and overlapped. There are two approaches for offloading

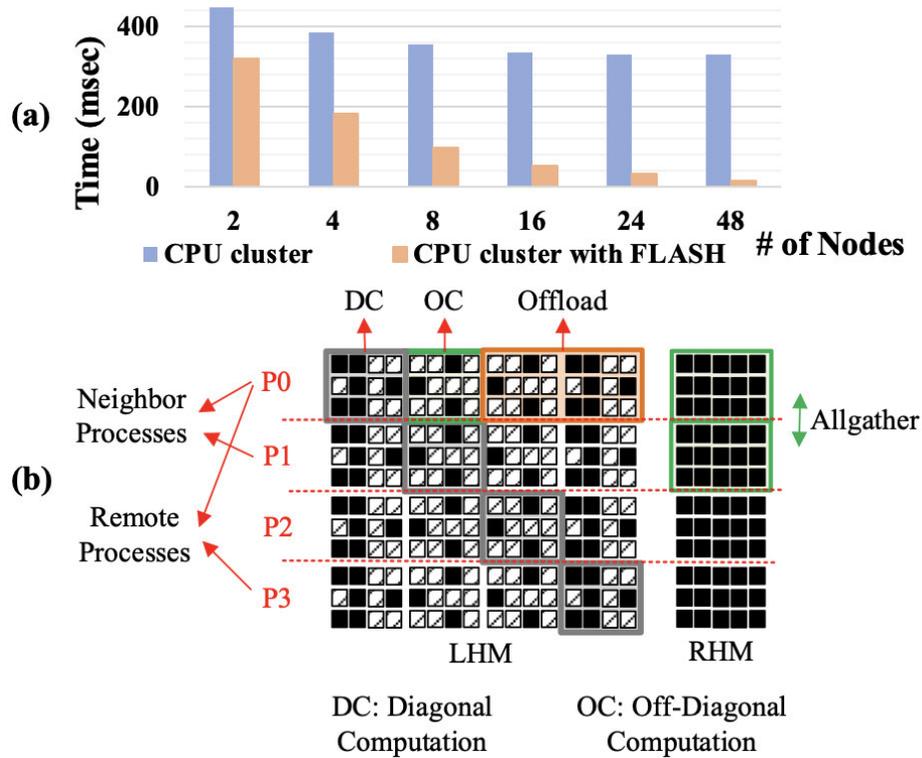


Figure 4.1: (a) GCN inference execution time on a CPU cluster (Skylake processors) without and with FLASH accelerators. (b) FLASH matrix partitioning for an SpMM kernel with tasks shown for process 0 (P0). Gray tiles in LHM belong to the diagonal part and the rest are off-diagonal.

a workload to smart switches: sending part of LHM from nodes to switches while storing RHMs in the switches (RHMs are reused) and vice versa. We follow the former approach as the latter potentially comes with a higher hardware cost and worse workload imbalance among processing elements (i.e., LHM is sparse and it is harder to distribute evenly in the off-chip memory of the switch accelerator).

A naive method is to offload all of the off-diagonal parts to switches and perform the diagonal part locally. There are two downsides, however. First, the off-diagonal part can constitute a large fraction of the total execution time, especially when there are a large number of processes. Second, this approach is not efficient for intra-node processes (i.e., sending LHM data to be processed in switches); this is avoided with an intra-node Allgather.

Therefore, in our approach, (1) diagonal parts are computed locally; this is called the diagonal computation task. (2) An Allgather on parts of the RHM is performed on a set of processes (called neighbor processes); afterward computations are performed locally (off-diagonal computation task). And (3) the rest of the off-diagonal part is offloaded to the switches for processing where the RHM is stored (offload task). We further optimize performance by overlapping the non-offloaded (1 & 2) and the offloaded part (3). Referring to Figure 4.1 (b), processes P0 and P1 are neighbors as there is an Allgather on corresponding parts of their RHM, but P0 with P2 and P3 are remote processes. We note that neighbor processes are not necessarily constrained to one node for large systems.

4.2.2 Motivation

Figure 4.1 (a) shows that while parallelizing GCNs improves performance, the scalability is limited. The FLASH-aware implementation improves the scalability by restructuring the application and using in-network computing. Figure 4.2 summarizes some of these benefits for a large-scale dataset (ogbn-products (Hu et al., 2020))

over a baseline without in-switch acceleration. 24 processes per node is used. The benefits include reducing the number of elements transferred among nodes, the total number of hops, and enabling overlap between the switch offloaded task and the rest of the application. FLASH achieves a tremendous reduction in the number of transferred elements since (i) some processing happens directly inside switches instead of moving data and subsequently processing them and (ii) only transferring needed data elements and doing so in a fine-grained pipeline fashion. Similarly the application benefits from the overlap of offloaded parts and the parts executed in the CPUs. These become more pronounced during scale-out.

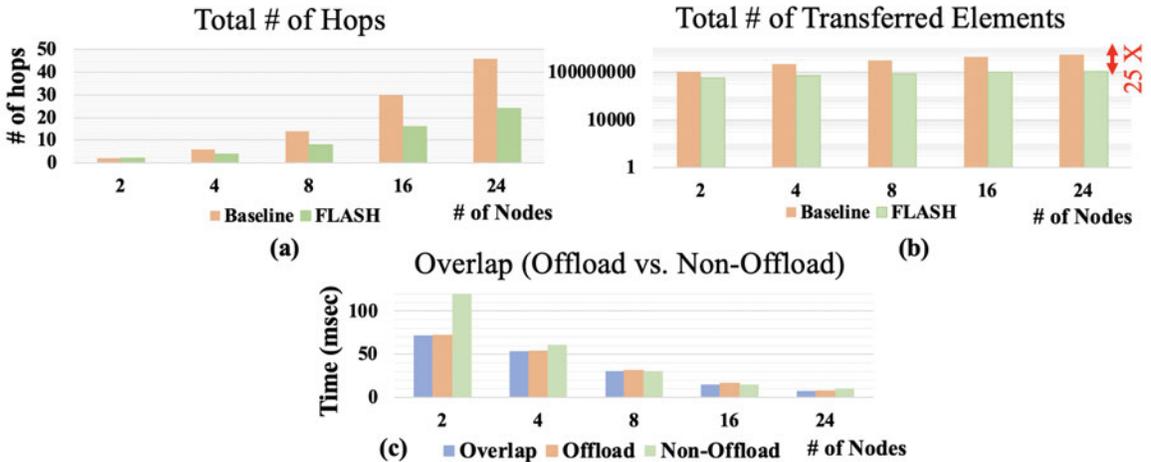


Figure 4.2: FLASH results for ogbn-products dataset: (a) number of transferred elements, (b) number of hops, and (c) overlap.

P4 is a high-level language used to control packet forwarding in protocol-independent *programmable* network devices. It raises the abstraction level of programming networks and is target-independent (FPGA, ASIC, software switches, *etc*). Since its introduction, different P4 architectures have emerged (Hauser et al., 2021), including the SimpleSum Architecture, the portable switch architecture (PSA), and the Tofino native architecture (TNA). Parsers, match-action tables, and deparsers are the most important pipeline elements (Wang et al., 2017).

P4 is an established and effective approach for programming network devices.

However, it falls short for many workloads, especially for application-level processing. We summarize them (and give the section that describes how it is addressed with FLASH):

- The SRAM capacity of the P4-based switches is small, which precludes storage of large machine learning models (Kfoury et al., 2021). FLASH supports off-chip memory so stores the entire ML model (as limited by the switch’s off-chip memory size). This avoids the latency overhead of streaming many small messages to a pool-based switch memory (Sapio et al., 2021) (§4.3.2).
- P4 has limited set of datatypes and operations (Qiao et al., 2020; De Sensi et al., 2021). For example, current P4-based switches do not support sparse data. However, some machine learning applications (*i.e.* GCNs) operate on sparse data calculation. Further, there is a limited multiplication capability in P4, e.g., to powers of two. Hence, the scope of P4 is limited to simple calculations; we discuss how FLASH supports more complex calculations in §4.3.2.
- Data reuse is crucial to many applications, but in P4-based switches, each packet can only access each memory location once within a traversal of the pipe (De Sensi et al., 2021). By tiling and reusing data it is also possible to decrease off-chip memory latency overhead (§4.3.2).
- P4 does not support loops (Qiao et al., 2020), which is required to express LA behavior, including ML applications (§4.5.2).

We address P4 limitations not through extending the language, but rather through a programmable accelerator that can be integrated into existing switch pipelines.

4.3 Hardware Design

In this section, we first describe FPin architecture in detail and then we introduce the FLASH hardware design.

4.3.1 FPin

Fig. 4-3 shows the FPin overall architecture for an FPGA with four transceivers. Transceivers and DMA engines are duplicated on the left and right to simplify the figure. In FPin, we consider a systolic array architecture as a part of the offload; this is an efficient solution for convolution operations and any computation patterns similar to a matrix multiplication. After the input FIFOs are packet parsers which then get directed to progress shapers and then virtual channels (VCs). The data path for host-to-card (H2C) streaming packets is slightly different as there is a support for having both stream (stateless) and stateful (with DRAM/HBM) interfaces with the systolic array.

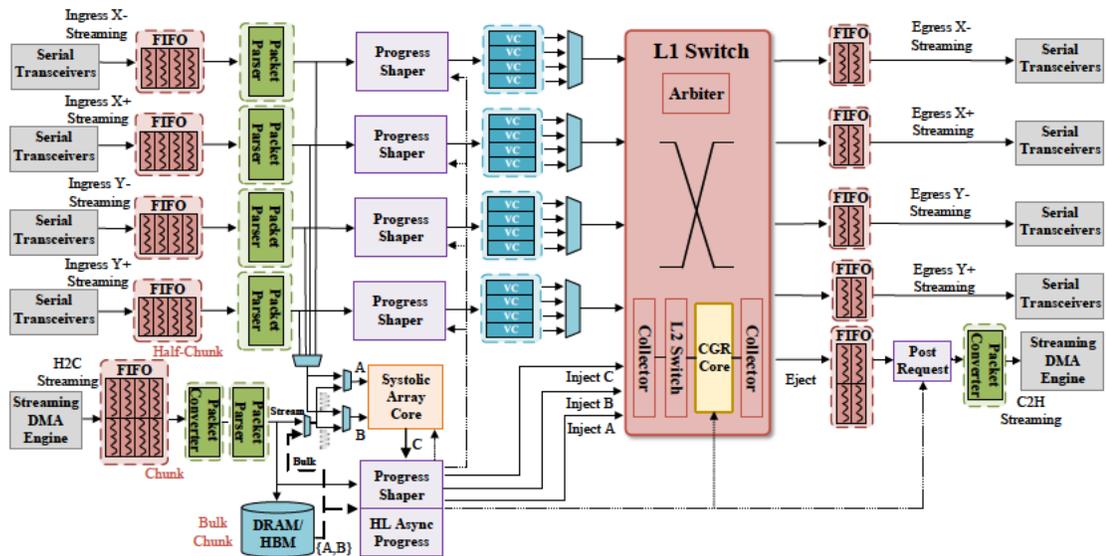


Figure 4-3: FPin Overall Architecture

An L1 switch directs packets to the correct output port and is followed by the

output FIFOs. To support in-switch processing for collective operations there are two collectors, an L2 switch and a configurable gather/reduction (CGR) core. The first collector is responsible for waiting for all child nodes to arrive. The L2 switch and the last collector are only for gather-type operations. The former reorders incoming packets according to their ranks while the latter serializes the gathered data. In the case of receive operations, there is a PR unit in the card-to-host (C2H) path which performs message matching and asserts completion requests. Asserting completion requests for collectives is performed by the CGR core.

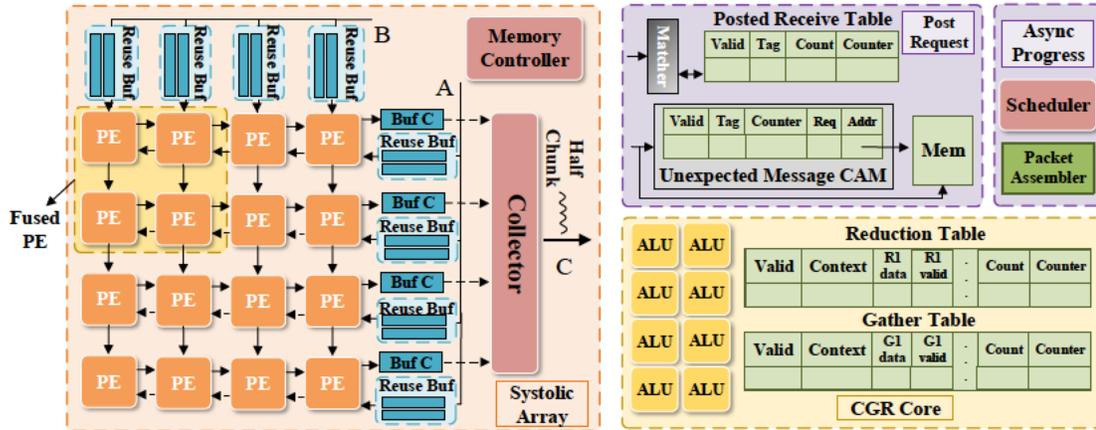


Figure 4.4: Systolic array, CGR core, PR, and HL progress unit Architecture

Fig. 4-4 shows the systolic array, CGR core, PR, and HL progress architecture in more details. In the systolic array, double buffering is used to tolerate DRAM latency. Each PE has a register file (RF) which stores intermediate results of buffer tilings. To support complex float data type, four surrounding PEs supporting float can be merged to form a fused PE. Inside the progress engine are a packet assembler and a scheduler. The former assembles headers for incoming FPGA packets to communicate with remote FPGAs, while the latter schedules the systolic array.

The PR unit accepts posted receive requests and unexpected messages (PRQ and UMQ in the MPI standard). PRQ and UMQ are used for storing unresolved requests

and messages received but not requested. A Content Addressable Memory (CAM) and a separate memory are used to store unexpected messages. Given a header, the CAM finds a matching address with the stored tag. The PRQ table also keeps track of the number of received elements which are then compared with the count field. A completion request is asserted when this threshold is reached.

The CGR core consists of configurable vectorized Aggregation Logic Units (ALUs). Each ALU is cascaded to match the number of switch ports to perform concurrent reductions with full pipelining.

4.3.2 FLASH

After an overview, we address the challenges of designing a programmable switch accelerator:

- We summarize a number of challenges for designing a switch accelerator since LA has different characteristics than stream packet processing.
- We extend the ISA with sparse vector instructions, specializing it for GCN inference acceleration.
- An important requirement of switch accelerator architectures is that it should not compromise the line rate. Thus, we apply several optimizations.
- We address a number of questions for integrating the proposed accelerator into existing switches.

Overview

Dataflow: FLASH is ideal for the execution of (nested) loop-intensive applications due to optimizations and massive parallelism of the LA architecture. LA consists of several VPEs pipelined together to process incoming packets. Figure 5.7 shows

the architecture with three VPEs. Each VPE consists of a number of floating-point Processing Elements (PEs) arranged in a SIMD fashion. The application’s loop body forms a dataflow graph (DFG) running on VPEs while the loop’s control flow is mapped to scalar processing elements that control the termination of the loop body. We elaborate on VPE and scalar PE below. To better understand how FLASH supports dataflow, consider that each VPE performs independent operations, e.g., multiplication. These VPEs are then pipelined together to enable supporting simple DFGs with SIMD parallelism. When all of the inputs to a PE become valid, data is consumed and processed accordingly. It is possible to have more complex DFGs with inter-PE reduction; this is future work.

Architecture: The accelerator has two AXI streams for input and output interfaces, referred to as *stream_in* and *stream_out*. In addition, there is one AXI-Lite to control the accelerator (starting and finishing the kernel, etc.), one read-only AXI memory-mapped (MM) for application instructions, and several AXI MMs (both read/write) for application data (RHM) and immediate floating-point data. More information is provided in.

FLASH has two main parts, data plane and control plane, with the former responsible for the actual packet processing while the latter controls *how* packets are processed (see Figure 5-7). The control plane has AXI-control and instruction loader modules. **AXI-control** allows the switch accelerator host to start the accelerator kernel and interrupts the host when the kernel is done. The **Instruction loader** module reads stored instructions used to configure the switch from off-chip memory (HBM), into on-chip configuration tables (CTs) from which instructions are sent to the data plane. The data plane consists of all the other modules. We divide it further into front-end and back-end. Program counter (PC) logic, decoder, and auto-increment vector modules belong to the front-end, while the back-end includes other modules.

We now describe the modules in the data plane. **PC logic** governs *-enable*, *-increment*, and *-load* signals for CTs in the control plane. **Decoder** decodes incoming instructions and asserts corresponding control, register file (RF) addresses, vector length (VLEN), and immediate signals. **Auto-increment vector** increments VLEN times on a base read and/or write RF address from the decoder for vector instructions, and asserts a done signal upon completion. Stalls from memory read/write and invalid data during arithmetic vector operations are considered in this module. **HBM read/write masters** handle the AXI-MM handshake and data transfer with an HBM bank. **RF** stores scalar values, while vector register file (**VRF**) stores vectors of floating-point (FP) values. **Scalar PE** performs scalar instructions. **Vector PE** performs arithmetic vector instructions on data from incoming packet and VRF which will be then written back to VRF again. **Idle tile control** monitors writes to a special register for keeping track of Tk (Tm) and increments this register with an offset equal to the difference of a tile required by incoming packet and the current tile for output stationary (weight stationary) dataflow. Tk (Tm) is tiling in the k (m) direction. **Disassembler** detects the packet header of the incoming stream and asserts a corresponding metadata signal for it. It also shifts parts of the packet from one beat to another. **Assembler** re-assembles the packet and calculates the checksum.

Coupling Accelerators and Streaming Packet Processing

Accelerators often load blocks of data from off-chip memory, do the processing in batch, and store them again in the memory. This differs from stream processing where data is processed cycle-by-cycle and directly forwarded to the output. We summarize some considerations here.

Vector load/store instructions dealing with memory and data path should be detached from streaming ports with stall and backpressure logic implemented in case

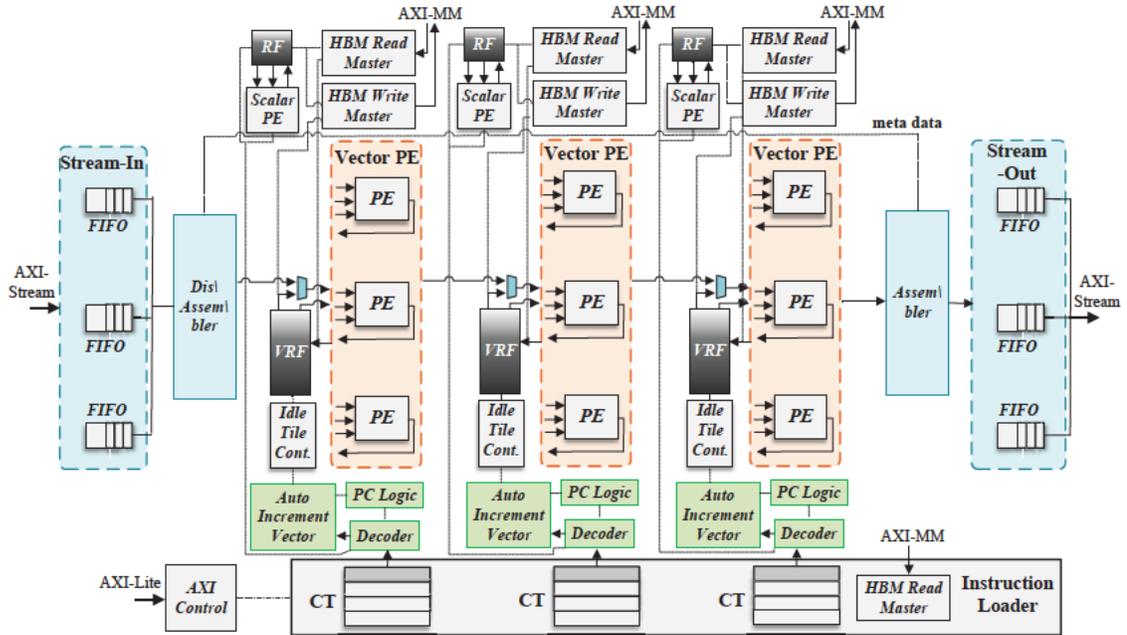


Figure 4.5: The proposed switch accelerator with three pipelined vector PEs. It is packaged with an AXI interface to facilitate integration with switch pipelines.

memory is not ready. Vector arithmetic instructions are used where packets from the *stream_in* interface meet data loaded from memory to VRFs. To enable such a processing we hardwire one of the PE inputs to *stream_in* (the first input source operand in the instruction). Furthermore, to move data from VRFs to the *stream_out* interface after packet processing we introduce a streamout instruction (`streamout.v`) which steers data from VRFs to the *stream_out* port serially across VPEs. Also, the packet header length (IP packets on top of Ethernet frames which yields 34 Bytes header) is not a multiple of PE bitwidth (32 bits) and because part of the payload comes with the header in the same beat (512 bits); this creates a misalignment which prevents packets from being processed at a granularity of PE bitwidth. To deal with it, we shift the packets for 34 Bytes across beats within a disassembler module. Lastly, since the accelerator can potentially modify the packet length header we calculate the checksum for the new header.

Sparse Vector Instructions

Instructions: Sparse matrix multiplication is critical to GCN. We implement two types of sparse vector accumulation instructions to accelerate SpMMs:

```
spvacc.vx v2, v1, x4
# (v2[v1[i]])+=(v1[i]>=VLMAX)?0:v0[4]
spvacc.xv x4, v2, v1
# v0[4]+=(v1[i]>=VLMAX)?0:v2[v1[i]]
```

where VLMAX is the maximum vector length for VRF groups; we divide VRFs into groups based on `vsetivli` instruction (supported instructions are summarized in §4.4.2). In the context of GCN acceleration, the first (second) instruction resembles a weight (output) stationary approach (Chen et al., 2017). We note that utilizing the second instruction yields fewer vector load instructions for the GCN packet handler and is more efficient.

Accelerating SpMM with sparse vector instructions: We elaborate on how SpMM computation in GCN is accelerated with FLASH. First, RHM is stored in FPGA switch memory and matrix LHM is streamed. Since LHM in GCN represents an adjacency matrix and is either 0 or 1, we only stream the column indices that are non-zeros. We use the *tiling* technique to improve GCN performance. A tile of RHM is loaded into VRFs, which are then accumulated according to indices that the matrix LHM is providing. The results are then streamed out from the VRFs to the output interface. One optimization is to unroll the sparse vector accumulation by reusing data in the VRFs. Since n is small, the RHM is tiled vertically (except for datasets in which the number of classes is larger than the maximum available VPEs, see §4.6.2), but the LHM is tiled both vertically and horizontally.

Optimizations

A number of optimizations improve line rate and latency:

(1) We employ the *idle tile skipping* technique to automatically advance the control flow of the program in the event of tiles without nonzero data elements; otherwise the pipeline is stalled. In §4.6.2, we show that this technique saves a large number of idle tiles.

(2) Currently, the RISC-V has only 32 vector registers. However, this is not sufficient for accelerators targeting compute-intensive applications. It can support grouping vector register files with a factor of up to 8 (*LMUL*). To enable efficient tiling, each VRF in FLASH can contain a large number of vector registers (4K). We extend the grouping factor by up to 2K. This means, e.g., that a vector load with a *LMUL* as 2K (which is set by `vsetivli`) can load up to 2K values to the VRF automatically with a single instruction.

(3) One approach to move the processed packets to *stream_out* is to store them in off-chip memory and then move them to the output interface. However, this hampers line rate as the packet processing pipeline could be stalled when storing and then loading data. Alternatively, FLASH directly moves the processed packets from VRFs to the *stream_out* port bypassing memory.

(4) Unrolling vector arithmetic operations improves the reuse of data stored in VRFs. This reduces the number of vector loads.

(5) FLASH provides massive parallelism: SIMD parallelism allows concurrent processing with the same instruction (there are 16 SIMD lanes). And processing elements across different VPEs add another level of parallelism. There can be up to 31 VPEs in the FPGA (§5.7) as there are 32 HBM banks and one bank is used for storing instructions.

Integration with Existing Switches

Methods: For AaS, Ethernet media access control (MAC) and networking capabilities are incorporated into the FPGA. On the FPGA, LA performs packet processing

and swaps the source and destination of headers (in the assembler module) after network packets are passed through the Ethernet MAC and networking kernels. Traffic is then sent back to the networking kernel. For AiS, LA is added to the switch pipeline. For example, for a NetFPGA design, the accelerator is placed between the input arbiter and output port lookup modules (Naous et al., 2008). There are two methods of acceleration: off-the-pipeline (Li et al., 2019) and in-pipeline (Graham et al., 2016). Off-the-pipeline has the benefit of reduced latency when the accelerator is not needed. FLASH has a pass-through mode that enables the traffic to be forwarded directly from input to output, making this approach off-the-pipeline.

Look-aside Interface: A standard interface is vital for both AaS and AiS. We use a Xilinx-compliant AXI interface to package the accelerator. Figure 4-6 shows the memory map for AXI interfaces used here with an example with three AXI MMs. The first is used for application instructions, the others for application data. *ap_start* and *ap_done* signals are accessed through the control register (address 0x000). As shown in Figure 4-6, the base address of the AXI MMs is accessed through AXI-Lite. One challenge with RTL kernels is that *ap_done* logic should be implemented in the accelerator itself. To support any number of packets of any length we assert this signal when the application is finished. This is done by placing a *wfi* instruction at the very end, which indicates kernel finish.

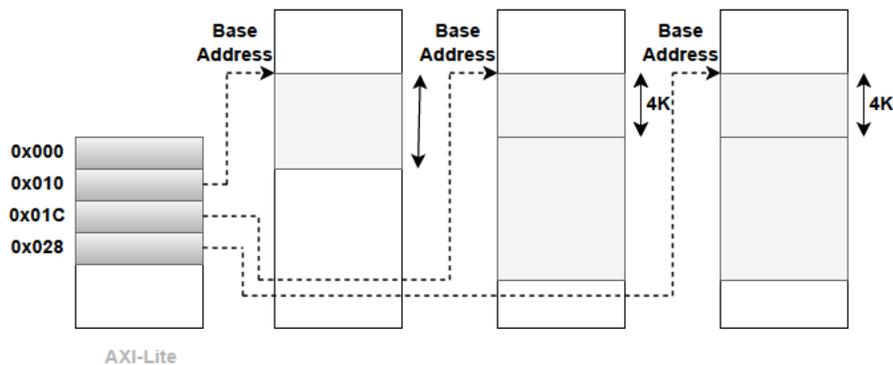


Figure 4-6: Memory map for AXI interfaces with three AXI MMs; the first is used for application instruction and two others for application data.

Hierarchical network switches: Finally, scaling out the acceleration and supporting a rack-scale cluster or data center is trivial as only leaf (access) switches in a spine-leaf (three-tier) architecture are accelerated with FLASH (Rios et al., 2021). Each leaf switch stores the whole RHM. While in this work we can store the whole RHM in the switches, one solution to handle very large graph datasets (more than the HBM limit) is to distribute RHM among switches.

4.4 Software Design

The following is the software design and framework for FLASH. For FPin, we use ExaMPI implementation. It is a light-weight MPI implementation which focuses on key blocks of functionality and new MPI concepts. As it is designed for modularity and extensibility we use it for transparent integration of the FPGA in FPin.

4.4.1 Compiler

To support a packet handler that is described with a high-level language we use a compiler, based on LLVM (Lattner and Adve, 2004), to generate back-end RISC-V instructions. RISC-V is adopted since it is an open-source ISA that lends itself to hardware specializations and ISA extensions. The Clang front-end generates LLVM intermediate representation (IR). We chose this step to introduce target-dependent parameters to support high-level languages other than C (e.g. Python), and to apply several optimizations. These parameters come from a configuration file, which includes CGRA dimension, number of SIMD lanes, unroll factor, register file (RF) size, *etc.* We built our back-end, which involves parsing the IR, generating the DFG, code generation, scheduling, and register allocation. Figure 4.7 shows the proposed framework to map an ML model with a user-provided packet handler to the reconfigurable switch accelerator. The compiled host code, along with the binary file generated from the CGRA overlay kernel, is used by the Xilinx runtime (XRT) library to program

and interact with the FPGA.

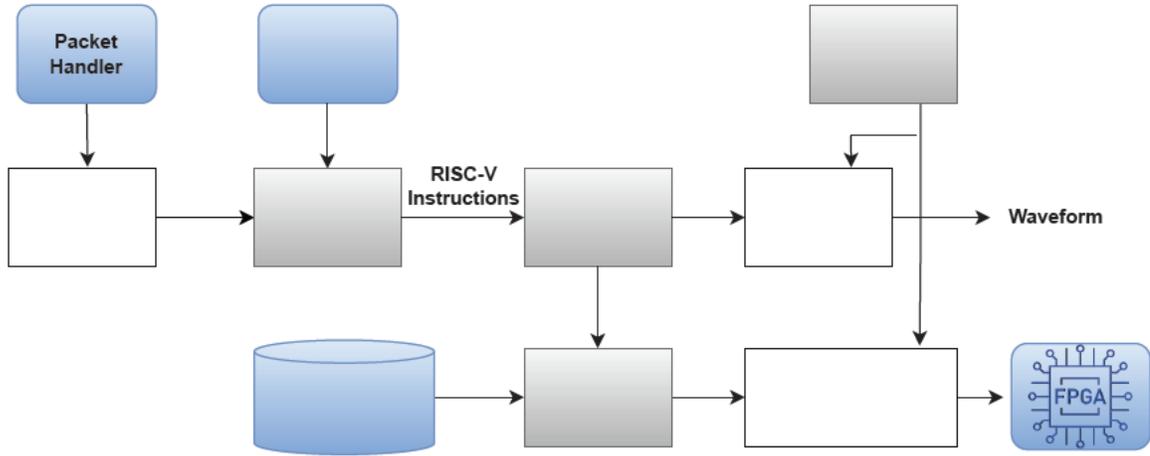


Figure 4-7: Framework to map a machine learning model with a user-provide packet handler to the reconfigurable switch accelerator. Gray components are proposed. White components are powered by existing tools.

4.4.2 Supported Instructions

Three types of instructions are supported: vector, scalar, and configuration. Vector instructions include load (`vle32.v`), store, FP addition & multiplication, sparse FP accumulation, FP fused MAC, and streamout (`streamout.v`). The first two are based on the current RISC-V 'V' vector extension specifications. Others are proposed here based on RISC-V specifications. Sparse accumulation is discussed in §4.3.2. Fused MAC is similar to that used in with the exception that here one operand is fixed (scalar RF) and is not being auto-incremented. This is useful for ML applications to exploit data reuse.

For scalar instructions, we support `lui`, `addi`, `add`, and `bne` all with the same specifications as (RISC-V, 2023). Also, `csrrw`, `vsetivli`, and `wfi` instructions are used to read (write) specialized control/status registers (CSRs), configure VLEN, and finish the execution of the kernel. The two currently supported CSRs are *cycle* and *packet length*, which are used to monitor the performance and set the packet

length, respectively. In summary, vector instructions perform arithmetic operations, load/store, and steer packets to the output port; scalar instructions govern the control flow of the program; and configuration instructions monitor the status (read cycle count CSR) of the accelerator, configure (writing to packet length CSR), and trigger (`wfi`) the accelerator.

4.5 Application Acceleration

We first describe workload decomposition and scheduling of distributed matrix multiplication kernel in FPin and then discuss how to accelerate GCN using FLASH.

4.5.1 Distributed Matrix Multiplication

Workload Decomposition and Scheduling

In the most general case, we decompose FPGAs into a 3D grid as shown in Fig. 4-8; each FPGA owns a portion of the cube. Since it is feasible to trade off memory for communication, we define two modes: large memory (LM) and small memory (SM). In LM, we have enough memory for each FPGA so that each can store a redundant copy of matrix A and B from all other FPGAs in the same grid row and column, respectively. In this case, there is no communication between FPGAs in the m and n directions, and only outer product matrices are reduced in the k direction. On the other hand, in SM mode there is no data copy. Computation is overlapped by (1) sending streams of matrix A bulk chunks (blue) to a neighbor in the same row, (2) sending streams of matrix B bulk chunks (red) to a neighbor in the same column in a ring fashion, and (3) reducing outer product matrices in the k dimension as shown in Fig. 4-8 (b). These correspond to *comm_A*, *comm_B*, and *comm_C* operand communicators in MPI_GEMM, respectively. We define context switching as a change of phase from halting local computation and starting on processing the

packets received from remote FPGAs, or vice versa. As shown in Fig. 4-8, horizontal (vertical) context switching corresponds to the matrix A (B) of each FPGA which can be programmed by sch_A and sch_B in MPI_GEMM.

Let $\hat{A}[i][z]$, $\hat{B}[z][j]$, $\hat{C}[i][j]$, Sm , and Sn denote, respectively, the i^{th} bulk chunk of matrix A in column z , the j^{th} bulk chunk of matrix B in row z , the outer product matrix on the first node corresponding to $\hat{A}[i][z]$ and $\hat{B}[z][j]$ vectors, and the number of rows and columns in the systolic array. Algorithm 1 describes the proposed scheduling for distributed matrix multiply kernel on the first node for SM mode.

Algorithm 1: Proposed distributed matrix multiply kernel scheduling on the first node

1 **Input:** $\hat{A}, \hat{B}, M, N, K, Pm, Pn, Pk, Tm, Tn, Tk, Sm, Sn$

2 **Output:** \hat{C}

3 $Rm = \frac{M}{Pn \times Pm}$, $Rn = \frac{N}{Pn \times Pm}$, $Qm = \frac{M}{Pn \times Pm \times Tm}$, $Qn = \frac{N}{Pn \times Pm \times Tn}$ for $m = 0 : Qm - 1$ do

4 $n = 0 : Qn - 1$ outer product reduction for $k = 0 : \frac{K}{Pk \times Tk} - 1$ do

5 $pn = 0 : Pn - 1$ horizontal context switching for $pm = 0 : Pm - 1$ do

6 vertical context switching

7 for $tm = 0 : \frac{Tm}{Sm} - 1$ do

8 $tn = 0 : \frac{Tn}{Sn} - 1$ for $tk = 0 : Tk - 1$ do

9 $\hat{C}[m + pn \times Rm + tm][n + pm \times Rn + tn] +=$

$\hat{A}[m + pn \times Rm + tm][Tk \times k + tk] \otimes \hat{B}[Tk \times k + tk][n + pm \times Rn + tn]$

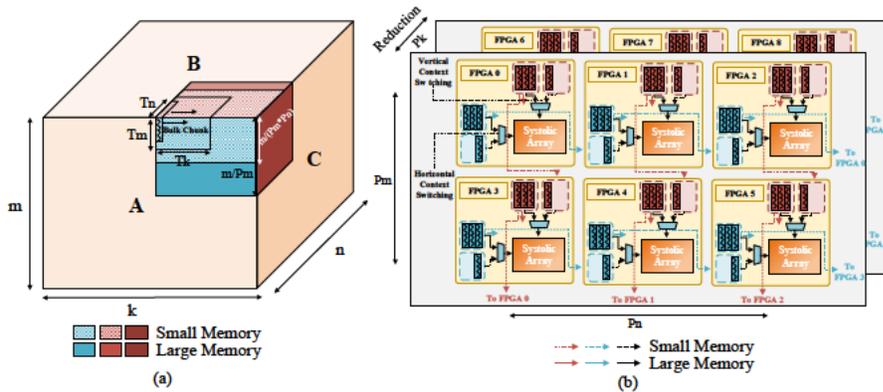


Figure 4-8: (a) Workload decomposition and (b) scheduling for small and large memory modes.

Communication-Computation Overlap Tuning

We now derive expressions for computation and communication time and buffer sizes for one node before vertical context switching (the three inner-most loop in Algorithm 1).

$$comp_time = Tk \times \frac{Tn}{Sn} \times \frac{Tm}{Sm} \times T_clk \quad (4.2)$$

$$comm_time = \frac{\max\{Tm, Tn\} \times Tk \times SZ}{link_BW} + L \quad (4.3)$$

$$buf_size = \max\{Tm \times Tk, Tn \times Tk, Tm \times Tn\} \times SZ \quad (4.4)$$

$$RF_size = \frac{Tm}{Sm} \times \frac{Tn}{Sn} \times Pm \times Pn \times SZ \quad (4.5)$$

where $link_BW$, and SZ are the bandwidth of serial link for FPGA-to-FPGA interconnect and size of one data element, respectively. buf_size in Eq. 4.4 represents sizes of both reuse buffers and input FIFOs.

From these equations, we can infer that if Tm is large enough then computation time outweighs communication time (Eq. 4.2, 4.3), but the size of buffers (register files) increase and the output matrix would lose its streaming nature.

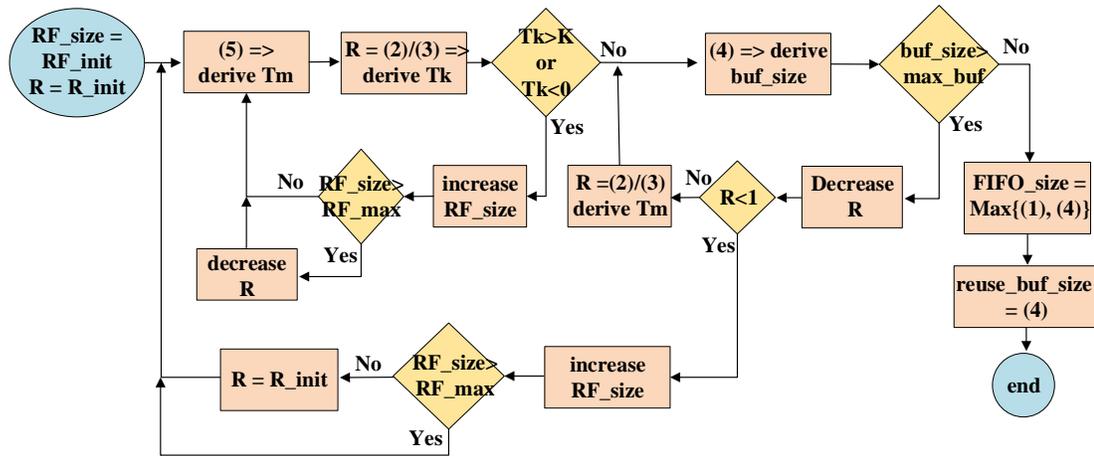


Figure 4-9: Proposed tuning methodology to obtain the size of tiles.

Assuming R is the computation to communication time ratio, Fig. 4-9 shows

the proposed methodology to keep the amount of buffering below a threshold while ensuring enough overlap between computation and communication. We start by initializing a small value for register file size and a number greater than one for R (full overlap). We can then derive Tm from Eq. 4.5 and Tk from Eqs. 4.2 and 4.3. In the simplest case, when Tk (buffer size) exceeds K (*max_buf*) then register file size (ratio R) is increased (decreased).

4.5.2 Graph Convolutional Network

Overview: The program flow of distributed ML inference acceleration is as follows: at the worker nodes, data is streamed over the switch by means of a communication library; at the switch accelerator, packets are processed according to a packet handler written by the user. We use socket APIs and MPI middleware as the communication library. MPI remains the *de facto* standard, but other methods, e.g., socket programming, are also possible.

On the switch accelerator, application data (RHM) and instructions are loaded from the host switch accelerator. Upon starting the kernel (through the *ap_start* signal), packet handler instructions are fetched from off-chip memory into distributed on-chip configuration tables. When loading is finished, the switch accelerator asserts a ready signal to its input interface and begins accepting streamed data. Packets start being processed and valid data is streamed from the output port (using `streamout.v` instruction). Once the `wfi` instruction is read from the configuration tables, an *ap_done* signal is asserted by the accelerator, which interrupts the host.

Requirements: Some requirements for the packet handler are as follows. (i) It should be encapsulated within a function with *stream_in*, *stream_out*, off-chip memory pointer, and packet length (new length after packet processing). The latter is essential because in some applications (e.g. GCN) data received from the output port might have a different packet length than from the input port. (ii) The packet handler should

be agnostic to switch port and worker node ID (*rank* in MPI terminology). Packet processing in the switch is finished as soon as the program flow reaches the end of function; it is the responsibility of the user to send/receive packets from worker nodes to the switch. (iii) The order of streamed data from worker nodes should correspond to that of processing within the switch as dictated by the packet handler.

4.6 Experimental Results

In this section, first we evaluate FPin using distributed matrix multiplication and then we assess the efficacy of FLASH using GCN.

4.6.1 Distributed Matrix Multiplication

Experimental Setup

To evaluate FPin, we compare the performance of our approach to that of the COSMA algorithm (Kwasniewski et al., 2019). We considered three sets of matrices: square ($m=n=k=16K$), tall and skinny ($m=n=1K, k=128K$), and flat ($m=n=64K, k=1K$).

For the FPGA cluster, we target Xilinx VCU128 boards. To study the effect of topology for the FPGA cluster, we consider 2D mesh and 3D torus topologies with up to 16×8 and $8\times 4\times 4$ nodes, respectively. The performance results for the FPGA cluster are obtained from a cycle-accurate simulator. Host-FPGA latency and the time to distribute initial partitioned matrices are considered within the total latency. The operating frequency of the current design is 250MHz. For the CPU reference, benchmarks were run on the Stampede2 Skylake (SKX) cluster at TACC with 48 cores per node (2 sockets), Intel Xeon Platinum 8160 CPUs, and a 100 Gb/s network.

Performance

In order to find Parallel General Matrix Multiply (PGEMM) grid parallelism we first increase Pm and Pn until we exceed device memory capacity to store input matrices.

Table 4.1: PGEMM Grid Parallelism for Float Datatype

# of node	Square			Tall and Flat			# of node	Square			Tall and Flat		
	Pm	Pn	Pk	Pm	Pn	Pk		Pm	Pn	Pk	Pm	Pn	Pk
1	1	1	1	1	1	1	16	4	4	1	4	4	1
2	2	1	1	2	1	1	32	4	4	2	8	4	1
4	2	2	1	2	2	1	64	4	4	4	8	4	2
8	4	2	1	4	2	1	128	4	4	8	8	4	4

We then increase Pk . To stress the amount of allowable memory on each device we perform strong scaling in conjunction with memory scaling (Kwasniewski et al., 2019); as the number of nodes is increased, the maximum available device memory is decreased. Table 4.1 shows PGEMM grid parallelism for both LM and SM modes on the base float datatype. We obtained 192, 160, and 64 for Tm , Tn , and Tk , respectively, in the base float after employing the proposed tuning methodology.

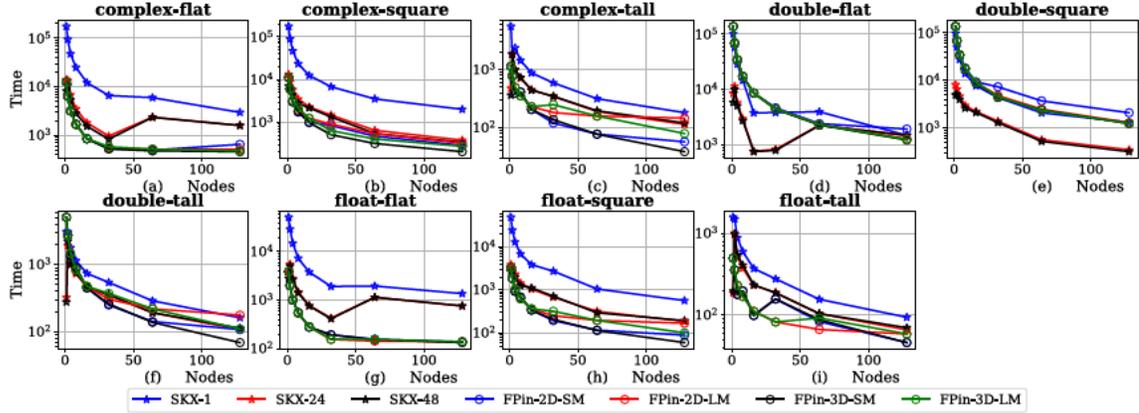


Figure 4-10: Strong scaling performance comparison of COSMA implementation on SKX cluster and PGEMM kernel accelerated with MPI_Gemm on FPGA-based cluster for square, tall, and flat matrices with complex float (a)-(c), double (d)-(f), and float (g)-(i) data types.

Fig. 4-10 shows strong scaling (with memory scaling) performance of the PGEMM kernel accelerated with MPI_Gemm and COSMA implementation for square, tall, and flat matrices with complex float, double, and float data types. For the CPU reference, we performed three experiments: SKX-1 (1 thread), SKX-24 (24 threads), and SKX-48 (48 threads). For each experiment, the number of nodes was varied from 1 to 128 nodes. In general the performance of our approach becomes more pronounced

as the number of nodes is increased since computation and communication are *fully* overlapped. The CPU implementation outperforms our approach for double data type since there are only a limited number of PEs supporting double data types in our design. On the other hand, for complex and float data types, we could achieve the highest speedup for flat matrices. This is because, in this case, both m and n are very large which makes the outer product matrix very large. This in turn, makes the reduction too costly both within and between nodes. Our approach avoids the aforementioned problem by having a large number of streamed small outer products.

4.6.2 Graph Convolutional Network

Experimental Setup

For the CPU reference, benchmarks were run on SKX cluster. We used Intel MPI 18.0.2 as an Intel-compatible MPI as recommended for this cluster; we also found it usually gives better performance than other MPI implementations. We experiment with up to 24 (48) nodes for small (medium/large) datasets. For small and medium/large datasets, we ran the experiments with 1 and 24 process(es) per node, respectively. We ran each experiment in SKX 10 times and used the median result. We wrote MPI code for the distributed GCN application following the Allgather-based approach (§4.2.1).

Performance benefits shown are by comparing the HPC cluster (TACC) with a FLASH-enhanced HPC cluster. Since we do not have direct access to either the TACC switch internals (AiS model) or the capability of attaching FLASH to TACC switches (AaS model) we created a proxy testbed. In this proxy testbed, parts that are executed in the nodes (non-offloaded tasks) are run in the actual testbed (TACC), and parts that pertain to the offloaded tasks are run on an FPGA. The recorded times for each process, in addition to the accelerator’s runtime, are passed to an emulator (described below). The accelerator’s runtime was measured either from

RTL simulation (*ap_start* to *ap_done* interval), in AiS mode, or kernel execution time at the accelerator’s host CPU, in AaS mode.

We now describe the emulation. We use the same method as (Li et al., 2019). That is, the emulation must have (i) the same number of network hops, (ii) the same amount of traffic in the network links, and (iii) accurate accelerator overhead. We also capture the workload imbalance among processes (process skew) on TACC; this can affect the performance of any in-switch offload (Graham et al., 2016; Haghi et al., 2021).

As mentioned, for the AiS model, we use simulation results from a cycle-accurate RTL simulation using a testbench to drive signals, generate traffic, and measure the LA performance (accelerator overhead) and throughput. The testbench has emulation modules for HBM and streaming ports that realize the handshaking with LA. For the AaS model, the testbed is a two-node system on CloudLab (Handagala et al., 2021; Handagala et al., 2022) 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. The LA accelerator is packaged as an RTL kernel. It is coupled with a modified version of (Xilinx, 2023) 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.

For both AiS and AaS models, we use two dataflow modes: one accelerated with `spvacc.xv` (*OS* for *output stationary*) and the other with `spvacc.vx` (*WS* for *weight stationary*). The four combinations are AiS-OS, AiS-WS, AaS-OS, and AaS-WS. We also note that while this approach works with high-radix switches, we consider switches with up to eight ports as a proxy for larger scale systems (using the same method as (Li et al., 2019)). Below we show that this limit is not because of the

Table 4.2: Dataset Sizes

Dataset	#nodes	#edges	#features	#classes
PPI	2372	34113	50	121
Citeseer	3327	9464	3703	6
Pubmed	19717	88676	500	3
Ogbn-mag	736389	5416271	128	349
Ogbn-products	2449029	61859140	100	47

FLASH’s resource requirements (§4.6.2); rather, we found that this gives the best FLASH scalability in GCN applications with datasets of interest.

Datasets

Datasets: We consider two types of datasets. Small-scale datasets are protein-protein interactions (PPI), Citeseer, and Pubmed; Ogbn-mag and Ogbn-products are the medium/large datasets (Table 4.2 provides the details). The hidden dimension is 16. Adjacency matrices are evenly distributed among nodes (row-wise).

Application-level packetizing: At the application level, we set the maximum packet size to 64 KBytes (in case the UDP transport protocol is used). This happens for ogbn-mag and ogbn-products. Similarly, the minimum packet length should be set by the user as otherwise there is incorrect GCN program flow. This is because the packet handler performs partial sums on tiling with k dimensions. The packets may therefore only be decomposed in the m direction.

Vector Instruction Incidence: Table 4.3 shows the number of vector instructions for different datasets and dataflow modes for 24 nodes. It is evident that vmacc.vx is the most widely used instruction across all datasets, which means that this operation should be considered the key operation to be optimized. Also, the number of vector instructions in WS dataflow is higher than that of OS but WS has a smaller vector length.

Table 4.3: The number of vector instructions for different datasets and dataflow modes (OS: output stationary, WS: weight stationary)

Dataset	Mode	vle32.v	vmacc.vx/ vmacc.xv	streamout.v
PPI	OS	2	4	1
	WS	54	1696	1
Citeseer	OS	4	21	1
	WS	105	3328	1
Pubmed	OS	22	640	2
	WS	618	19744	1
Ogbn-products	OS	8379	267904	7
	WS	76534	2449056	1
Ogbn-mag	OS	722	23040	2
	WS	23014	736416	1

Communication performance

Figure 4.11 shows the GCN communication performance of the five datasets as they are scaled from 2 nodes to 24 (48) nodes for both the baseline SKX cluster and FLASH with different configurations. Comparing OS with WS: the latter typically provides better communication performance on a small number of nodes for small-scale datasets (PPI, Citeseer, and Pubmed), but its scalability is worse than OS due to the inefficiencies discussed in §4.3.2. For larger datasets (ogbn-products and ogbn-mag), the OS always provides better performance. One reason is that idle tile skipping technique in WS is not as effective as OS due to a small Tm (§4.3.2). We therefore consider only OS for the rest of this subsection. As discussed in §4.2.2, RHM is communicated among the nodes in GCN. If RHM is large enough compared to LHM, the application is communication-heavy (giving room for FLASH to improve performance). This implies a large n and a small m . Of the datasets, PPI has the largest n/m ratio and FLASH provides good communication performance improvement. For ogbn-products and ogbn-mag, FLASH provides considerable communication performance improvement. One reason is that the total number of transferred elements in FLASH is greatly reduced.

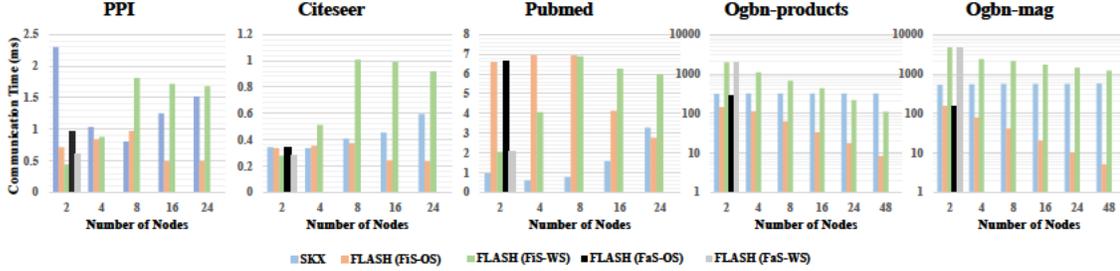


Figure 4.11: Communication performance and scalability comparison of GCN for a baseline CPU cluster (SKX) vs. FLASH with different configurations.

Application scalability

Figure 4.12 shows the application performance and scalability of GCN with and without FLASH acceleration across all datasets. For Pubmed, FLASH does not provide good performance for small numbers of nodes. This is because the number of classes (n) is small in Pubmed (Table 4.2). This leaves little data reuse for the data transferred to the switch (each class is mapped to each VPE and VPEs are pipelined). The larger the number of classes (the upper limit is the maximum available VPEs) the better the data reuse, and FLASH achieves more efficient computation with less data movement. Nevertheless, FLASH outperforms the baseline at 24 nodes for Pubmed. This demonstrates its superior scalability.

For larger datasets (ogbn-products and ogbn-mag), FLASH provides excellent scalability as idle tile skipping is more efficient and the overhead of sending/receiving packets to/from the accelerator becomes negligible. Ogbn-products performs better than ogbn-mag in FLASH since LHM is streamed multiple times, as the number of classes (n) in this dataset is much larger than the maximum number of vector PE pipelines (31). On average, FLASH (OS mode) improves application performance compared to a baseline SKX cluster by a factor of $2.2\times$, $2\times$, $1.1\times$, $1.4\times$, and $10.1\times$ for PPI, Citeseer, Pubmed, ogbn-mag, and ogbn-products on 24 nodes with an average of $3.4\times$ across all datasets.

We note that while the idle tile skipping technique is less efficient for small-scale

datasets, it saves about 27% and 77% of total tiles in OS mode for ogbn-products and ogbn-mag, respectively.

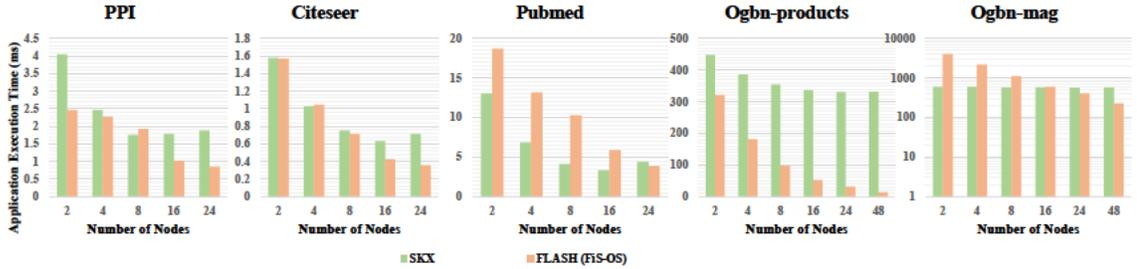


Figure 4.12: Application performance and scalability comparison of GCN on a baseline CPU cluster (SKX) vs. FLASH.

Overall application throughput

The overall finding is that FLASH communication output matches communication input (streaming rate) all the way up to the port bandwidth. The exceptions are when there is a reduction in data so that less data is output than input, or when there is an algorithmic dependency that prevents data from being transmitted. Our results show that LA itself can saturate network bandwidth at around 95.7 Gbps for message sizes beyond 1.5 KB. Limitations do occur, however, if the application has some characteristics that prevent input packets being processed by LA (e.g., control flow instructions).

We show the overall application throughput measured at the input interface in Figure 4.13 with the scaling of nodes across all datasets. For OS mode, throughput values typically increase as the application is scaled out. This is because application time decreases but the number of times that the `spvacc.xv` instruction is called remains fixed. On the other hand, throughput more or less remains fixed for WS during scaling since the execution time at the switch changes much more slowly (this time VLEN is changed during scale-out instead of decreasing T_m for OS) for small datasets. Finally, throughput values for WS mode are higher than for OS mode. This

Table 4.4: Compilation time and other parameters of the back-end compiler for GCN packet handler

Avg. Time (ms)	STD	SLOC	Initial/Optimized DFG (#node)	Initial/Optimized DFG (#edge)
91.36	5.7	64	65/11	138/18

is because the number of times that `spvacc.vx` is called is larger (due to a larger Tk).

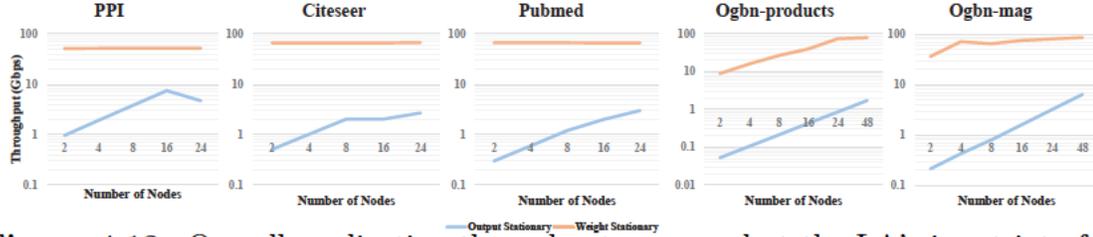


Figure 4-13: Overall application throughput measured at the LA’s input interface for different datasets.

FLASH compilation time

We measured the compilation time of FLASH’s back-end compiler for the GCN packet handler. The average time (across 20 runs) along with the standard deviation (STD) are reported in Table 4.4. We used LLVM 11.0.0 running on an Intel Xeon Gold 6242 @2.80GHz. The compilation time does not differ across datasets since we are using the same packet handler code with different configurations. Source lines of code (SLOC) is also shown in Table 4.4, as well as the number of nodes and edges of both initial and optimized DFG (after optimization passes) of the packet handler code. Nodes in the initial DFG represent LLVM instructions while they represent rolled RISC-V instructions. Of note is that compilation times are at the “software” scale of milliseconds rather than the hours typical for HLS tools.

Resource requirements

Figure 4-14 shows hardware resource utilization on the Xilinx Alveo U280 FPGA board for a AiS configuration with 16 pipeline vector PEs. The switch is implemented using a NetFPGA design (Naous et al., 2008). AaS utilization results can also be

inferred as there is only the accelerator itself in this configuration. As it is evident from the figure, LA consumes DSP blocks and Ultra RAMs (URAMs), while the switch logic takes up other resources. It is also clear that, as the number of ports increased, the overall utilization increases. We note that VRFs and CTs are mapped mostly to URAMs.

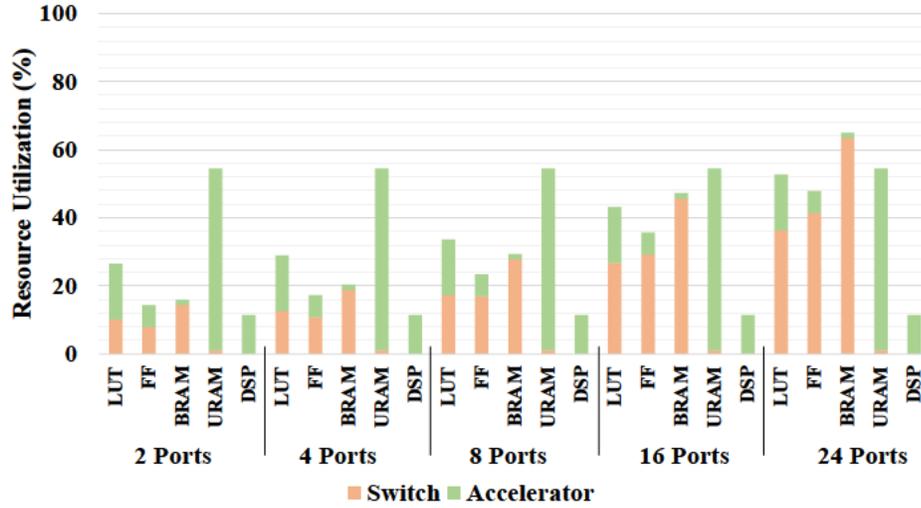


Figure 4-14: Resource utilization for AiS configuration with 16 pipeline vector PEs on Alveo U280.

FLASH Overheads

We measured the time to send RHM data from the switch host CPU to the accelerator (i.e., moving data from a CPU’s off-chip memory to HBM banks in the FPGA board). We repeat this experiment ten times. On average, it takes about 0.2, 0.1, 0.1, 32.2, and 48.3 milliseconds for PPI, Citeseer, Pubmed, Ognb-products, and Ognb-mag, respectively. These overheads are negligible compared to total execution time (Figure 4-12) for most datasets Ognb-products. The overhead is similar to that of setting up non-FLASH versions in distributed computing systems (since data is not always available in the corresponding nodes).

Discussion

We anticipate that scaling GCN applications to a larger number of nodes will bring increasing performance advantages (for large datasets) due to the FLASH benefits (reducing the number of transferred elements and hops, overlap, etc). We also expect FLASH to improve the performance and scalability of other communication-intensive applications as it is generic enough to support different workloads and it directly improves the communication time through in-switch computing.

4.7 Conclusion

In this chapter, we introduce look-aside accelerator to enable stateful packet processing for HPC and machine learning applications. In the first part of this chapter, we present a compute-in-the-network FPGA assistant to accelerate distributed matrix multiplication. The assistant utilizes vectorized ALUs supporting a variety of data types as well as a smart offload engine to handle communication-computation overlap. Experimental results show that our approach achieves, on average, $2.4\times$ and $1.8\times$ speedups compared to the state-of-the-art COSMA algorithm for float and complex float, respectively.

In the second part, we design, implement, and evaluate a programmable look-aside accelerator that can be embedded into, or attached to, existing communication switches. To facilitate usability, we develop a software toolchain to compile user-provided code for configuring the switch. While our approach is generic and supports a variety of workloads, we consider graph convolutional network (GCN) inference as a case study. Experimental results show that this approach improves both performance and scalability. The performance advantage is on average $3.4\times$ (across five real-world datasets) on 24 nodes.

Chapter 5

Type 3 ACiS: Fused Collectives

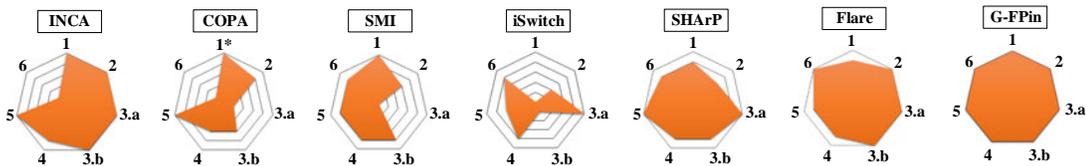
This chapter introduces ACiS fused collective capabilities. This type combines multiple collectives with map operations from user code automatically. The advantage is that the number of communication operations and, generally, the amount of overhead in node/network processing are reduced. In this type, the plugins from Types 1 and 2 are combined together with the aid of intermediate buffers.

5.1 Introduction

In-switch computing for MPI collective offload has recently gained considerable attention (Peng et al., 2011a). Currently, however, switch hardware support for collectives is limited to a small set of scalar operations and data types (e.g., (Faraj et al., 2009; Graham et al., 2016)). Moreover, beyond collectives there are additional acceleration opportunities. Our hypothesis is that it would be beneficial to further augment switches to accelerate additional and more complex functions that integrate communication with computation; we refer to these as complex communication-computation functions (CCCFs). We propose in-switch hardware support of three types of CCCFs. The first, is *fused collectives*, which are built by fusing multiple existing collectives such as Allreduce with Alltoall (Haghi et al., 2023b). The second is *semi-fused collectives*, which are built by combining a collective with a map computation. The third is *distributed kernels*, which are built by combining multiple communications and computations. An example of a distributed kernel 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 distributed kernels to switches, much of the computation can occur in transit (Van De Geijn and Watts, 1997; Agarwal et al., 1995). Applications that benefit from the specification and acceleration of distributed kernels include finite element methods (Massing et al., 2013), iterative solvers (Falgout, 2006), and graph algorithms (Page et al., 1999).

Adding switch support for CCCFs confers additional advantages to in-switch computing. First, additional processing can be removed from the host as the switch manages complex function operations *autonomously*. Second, performance can be improved through handling hardware-accelerated irregular computation and data caching. Third, two-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.



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 5.1: Characterizing *G-FPIn* and related work with respect to benefits of collective offload to network devices; *COPA* (Krishnan et al., 2020), *INCA* (Schonbein et al., 2019), and *SMI* (De Matteis et al., 2019) are in-NIC approaches while *iSwitch* (Li et al., 2019), *SHArP* (Graham et al., 2016), *Flare* (De Sensi et al., 2021), and *G-FPIn* are in-switch methods. *Denotes a headless configuration in *COPA*.

Multiple challenges need to be addressed. First, while CCCFs can be programmed

directly, much 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 (Gropp et al., 1996)). And third, CCCF support in the switch must remain general-purpose while handling high-bandwidth communication.

Our solution, which we call *G-FPin*, 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 *G-FPin* switch hardware. The second is the *G-FPin* switch hardware itself. This has multiple components for the different types of CCCFs. To support fused and semi-fused collectives *G-FPin* uses a coarse-grained reconfigurable array (CGRA) (Weng et al., 2020) overlay architecture for packet processing. A second, more specialized unit (we refer to as *kernel logic*) supports distributed kernels. A feature of *G-FPin* 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 *G-FPin* switch hardware. In this chapter, we focus on AiS model and a direct network.

Fig. 5.1 compares *G-FPin* and prior art (in-NIC and in-switch) with respect to various benefits of offloading computation into the network. In-switch approaches, if designed efficiently, can attain substantial gains for the aforementioned benefits.

We summarize the contributions:

- Demonstrate the advantages of accelerating CCCFs with reconfigurable switches (Sec. 5.3).
- A systematic framework to accelerate CCCFs with reconfigurable switches transparent to MPI (Sec. 5.4).
- A network-optimized CGRA with Single Instruction Multiple Data (SIMD)

and asynchronous execution features to fuse communications and computations (Sec. 5.5.2).

- A new compute-in-the-switch model to abstract communication and computation and to predict the performance of in-switch computing approaches (Sec. 5.6).
- Experimental results showing that this approach improves performance of a variety of HPC applications (Sec. 5.7).

5.2 Preliminaries

5.2.1 Complex Communication-Computation Functions

CCCFs 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. 5.3). That useful CCCFs 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. 5.7). The current study examines the following CCCFs:

- **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 (*op*) and data generated during the computation is in the send buffer of the second collective.
- **Semi-fused collective:** A collective followed by computation (*op*) (*collective_op*), where the receive buffer from the first collective is used during the computation.

- **Distributed kernel:** A kernel with multiple computations and communications following a well-defined and repeatable communication pattern. A simple example is a dot-product; more complex are matrix-multiply and FFT.

5.2.2 *G-FPin* Basic Building Blocks

To support general-purpose computation in the switch in conjunction to collective offload, *G-FPin* is built around three fundamental building blocks: (1) basic collective operation units, (2) a network-optimized CGRA (NGRA), 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)-fused collectives and distributed kernels, respectively. The second block is inspired by CGRAs with the distinction that it processes packets (CDFGs are mapped to this block). This building block enables *G-FPin* 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.

5.3 Motivation

We now give an example to illustrate the idea of CCCFs. Fig. 5-2 (a) shows an example of a fused collective for NAS parallel benchmark (IS). 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 this approach, 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. 5-2 (b) shows the C++ binding for a distributed kernel, Sparse Matrix Vector multiplication (*SpMV*), in which all of the communications and

computations are handled by the switch.

We now define terminology necessary to characterize (semi)-fused collectives. We refer to *stream-in*, *inbound*, *stream-out*, and *outbound* as, respectively: the send buffer of the first collective; a collection of all the array(s) consumed in the computation (but are not produced by the first collective); the receive buffer 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 *G-FPin* compiler. In addition, computation operations (*op*) are compiled into instructions (stored in *op* buffer in the host, see Fig. 5-2 (a)), which are transferred to the switch in order to program the NGRA at runtime (Sec. 5.5.2).

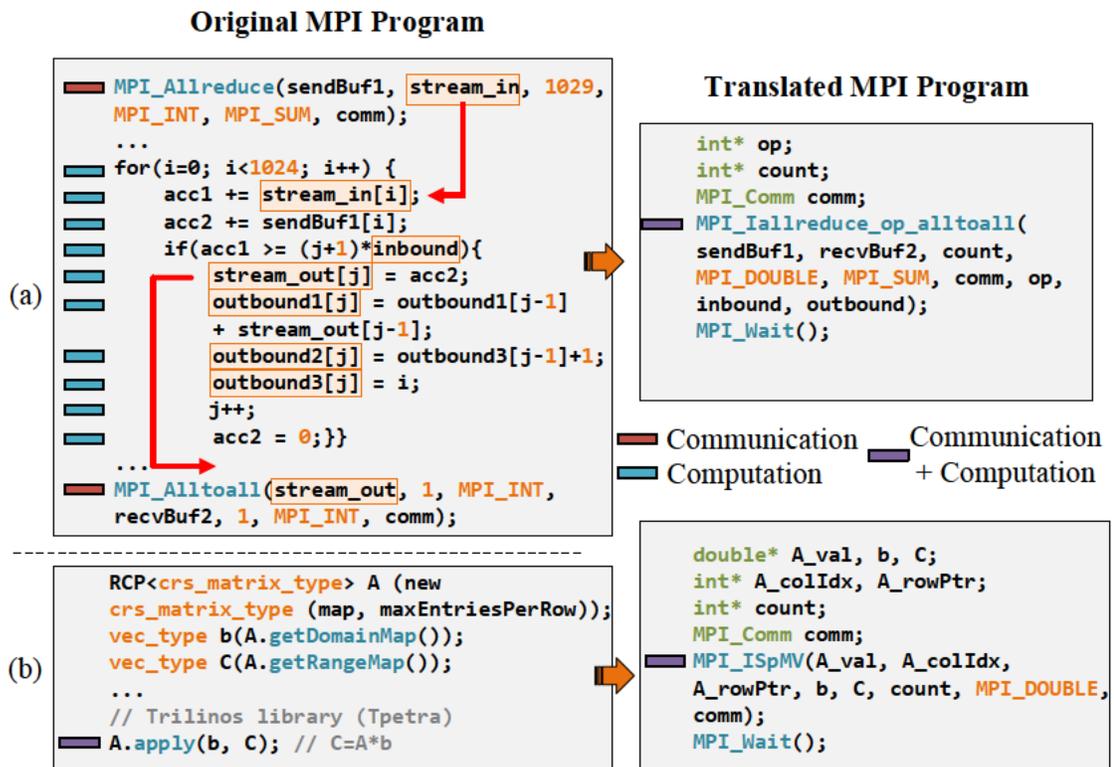


Figure 5-2: Examples of (a) a fused collective (`Allreduce_op_alltoall` in the IS application from the NAS parallel benchmark) and (b) a distributed kernel (SPMV).

To justify this project we conducted some preliminary experiments where we com-

pared *G-FPin* on an FPGA cluster with the original CPU implementation on a CPU cluster with 128 nodes. Fig. 5-3 summarizes *G-FPin* improvements over the original CPU implementation with respect to some of the aforementioned in-switch computing benefits for the two CCCFs shown in Fig. 5-2. It is evident that *G-FPin* 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)-fused collectives, we evaluated *G-FPin* with respect to a set of synthetic benchmarks. Although these synthetic benchmarks are not real HPC applications, they are representative of many communication-computation patterns in applications, including graph algorithms and iterative PDE solvers. Table 5.1 shows the specifications of the synthetic benchmarks and Fig. 5-4 depicts the average latency (among all ranks) for these benchmarks on a CPU cluster and *G-FPin* (128 nodes). As is apparent, *G-FPin* provides considerable improvement. For the CPU cluster specification and simulation setup used for FPGA cluster, refer to Sec. 5.7.1.

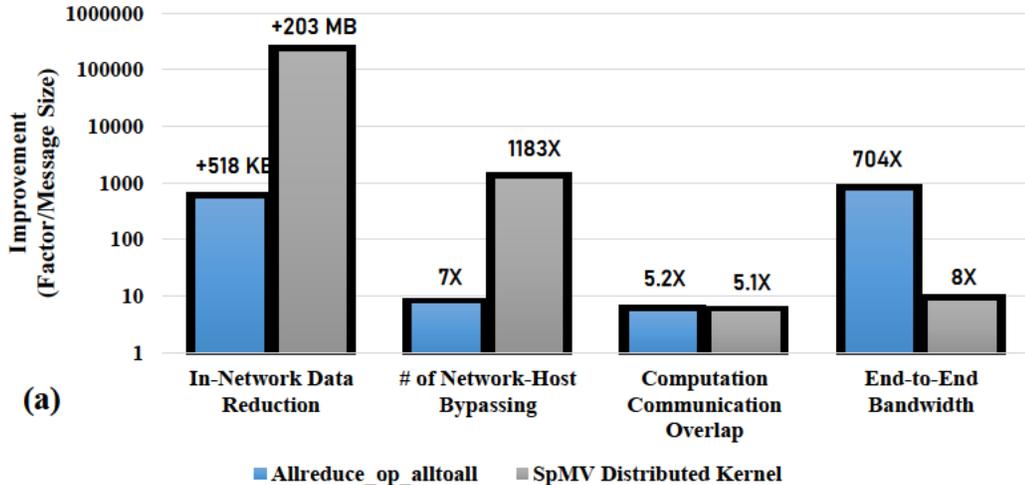


Figure 5-3: Evaluating our approach vs. the original CPU implementation on 128 nodes with respect to (a) some in-switch computing benefits for the two CCCFs introduced in Fig. 5-2.

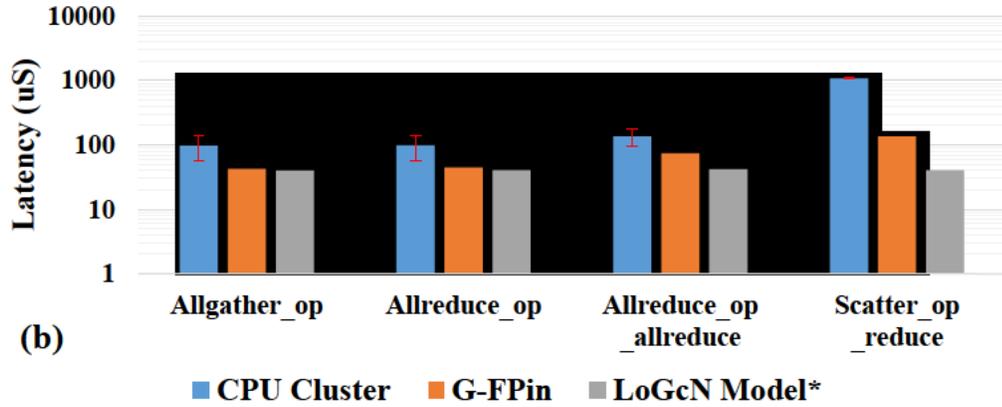


Figure 5-4: Latency of the synthetic benchmarks discussed in Table 5.1. * LoGcN model is discussed in Sec. 5.6.

Table 5.1: Specifications of the synthetic benchmarks.

Synthetic Benchmark	Op	Sample Application	Stream-In Size	Inbound Size
Allgather_op	MAC (Multiply-Accumulate)	PAGERANK Algorithm (Page et al., 1999)	128	128
Allreduce_op	Accumulation	Sorting Algorithms	512	0
Allreduce_op_allreduce	DOT-PRODUCT + SCALING	Norm Computation in GMRES (Yamazaki et al., 2017)	1	16K
Scatter_op_reduce	Accumulation	Master-Slave (Workload Distribution)	512	0

5.4 *G-FPin* Software Support

In this Section, we first present an overview of the software model and then we describe the proposed framework which can fuse communications and computations from the input program and construct their offload to the reconfigurable switches.

5.4.1 Software Model

There are two ways to support reconfigurable network devices for MPI specifications. The first is *offline* (De Matteis et al., 2019) 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) (Saldaña et al., 2010) where the reconfigurable network device (*i.e.* the FPGA) is transparent to MPI calls. We focus on the latter as it avoids regenerating the bitstream for each MPI program while addressing portability across HPC applications (Gropp et al., 1996) 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 (Skjellum et al., 2020). 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. CCCFs, however, bypass intermediate sends/receives; in this model, each rank has only up to one non-blocking send and receive routines. Other MPI functionality, including MPI rank management for the collective algorithms, is handled by the switch.

5.4.2 The Proposed Framework

Overview

The proposed framework translates an input MPI program into a new one enhanced with CCCF APIs and compiles the relevant parts of the program to the reconfigurable switch. Fig. 5-5 shows the *G-FPin* 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 distributed kernel APIs. If it is a distributed kernel, then it is replaced with a new API (similar to Fig. 5-2 (b)). Otherwise, in case of a (semi)-fused collective, an evaluator first evaluates whether fusing the collectives and operations is worth offload to the network. In case of passing that criterion, a control data flow graph is generated by *G-FPin* compiler for the *op* extracted via the parser. We used LLVM infrastructure (Lattner and Adve, 2004) for this purpose.

The next step is CDFG mapping. There, CDFG operations are mapped into NGRA processing elements (Sec. 5.5.2) with the aid of a configuration file, which embeds architecture-specific information (number of processing elements, etc). We use modulo routing resource graphs (MRRG) (Mei et al., 2003) 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) CCCF 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)-fused collective. In order to provide overlap between the CCCF 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.

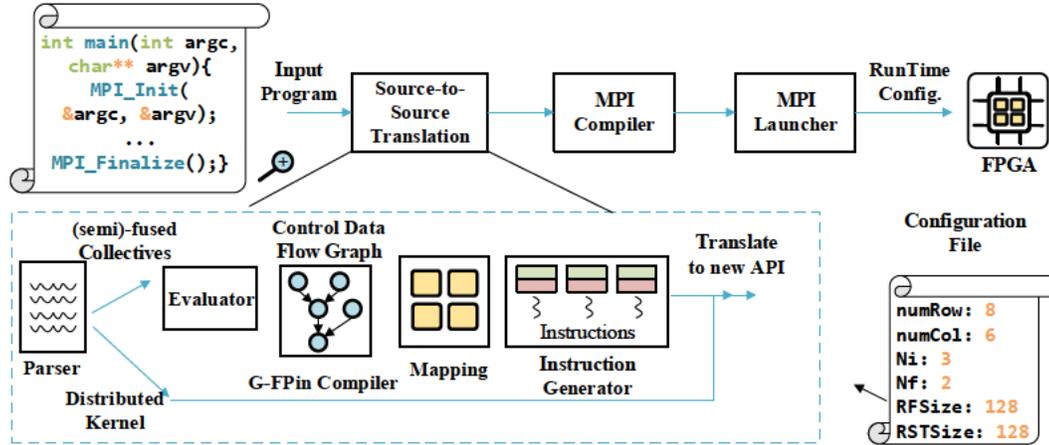


Figure 5-5: Proposed Framework

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

Evaluator

There are two criteria to determine whether a CCCF 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 NGRA. 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.

G-FPin 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 NGRA supports SIMD and reduction between SIMD lanes; this reduction is considered to be one of the operations for the CDFG. Also, indexed *if/else* statements (branches that depend on the loop iteration) should be han-

dled through predication (Hamzeh et al., 2014) when SIMD is involved.

Mapping

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

5.5 *G-FPin* Hardware Support

To support CCCFs in a generic switch, *G-FPin* is built around three main components: (1) basic collective support, (2) the NGRA, and (3) kernel logic. Fig. 5-6 shows the proposed smart switch design: it is based on a Virtual Output Queue (VOQ) design (Kim et al., 2005) and shown with four input and output transceivers. We now describe each *G-FPin* building block.

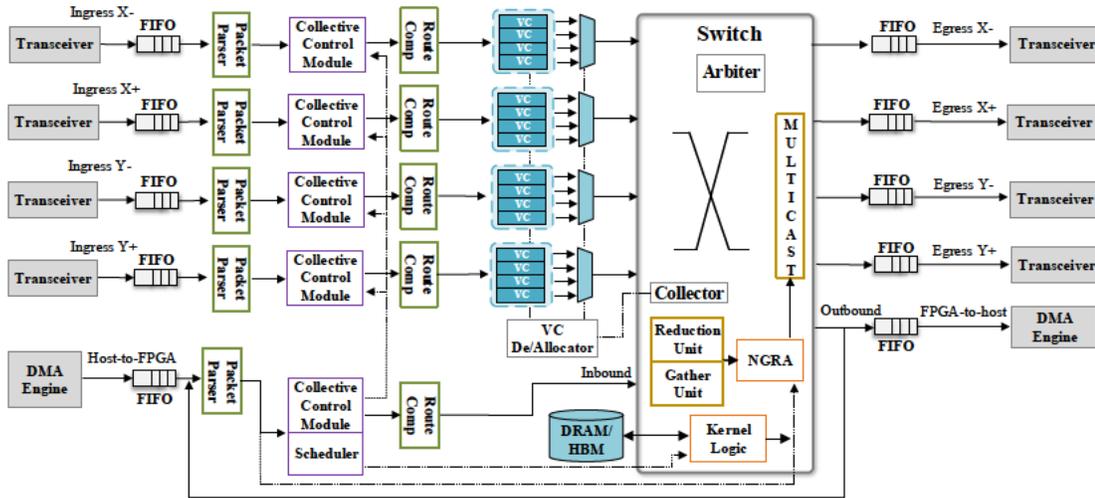


Figure 5-6: The proposed smart switch design with CCCF support.

5.5.1 Basic Collective Support

The basic collectives that *G-FPin* 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` and `MPI_Allgather`); 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.

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, recursive doubling, and a tree-based algorithm optimized for a 3D-torus topology.

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 comprises parallel double-precision floating point ALUs since the phit size (data bitwidth of interface, 512 bits) is wider than that bitwidth.

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 (Gropp et al., 1996) and the rank ID of the child processes in the collective algorithm.

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.

5.5.2 Network-Optimized CGRA

In order to support a variety of workloads for in-switch computing we propose a network-optimized CGRA (NGRA). CGRAs are typically used to accelerate computation-intensive loop kernels. As far as we know, they have not previously been used for packet processing in network devices. 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 (Weng et al., 2020) through a runtime state table (RST) and an FSM to efficiently keep track of the loops and operations. To take advantage of the wide phit size, the proposed NGRA benefits from SIMD features with a reduction tree that performs the reduction among SIMD lanes.

As shown in Fig. 5.7, the proposed NGRA consists of input interfaces (stream-in and inbound); output interfaces (stream-out and outbound); an array of PEs grouped into different types, which are accompanied by RFs; runtime configuration tables (RCTs); and an RST. RCTs store instructions (configurations) designated for PEs (generated by the instruction generator, Sec. 5.4). These instructions specify the cor-

rect 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.

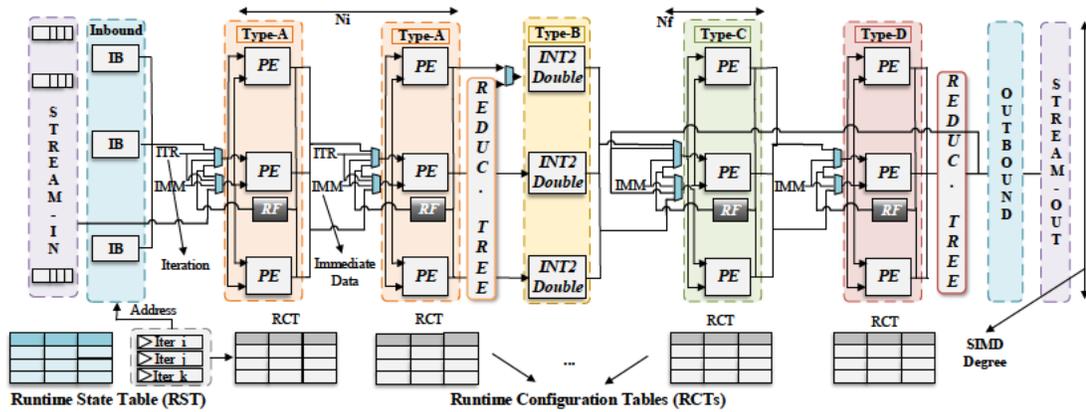


Figure 5.7: The proposed NGRA architecture; it is based on an array of SIMD-based processing elements.

To maximize end-to-end network bandwidth we applied a number of optimizations. (1) Since stream-in data can arrive at the NGRA 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 NGRA design:

Data Streaming

As the NGRA 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 access 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*.

Supporting Conditionals

To facilitate conditionals, we support partial predication (Hamzeh et al., 2014), in which both the *if*-path and *else*-path are executed in parallel and the correct outcome is selected by evaluating the condition.

Supporting a variety of workloads

To support different types of workloads, PEs inside the NGRA 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.

5.5.3 Kernel Logic

Kernel logic is responsible for handling kernel-level operations. 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. 5-8 shows the proposed architecture. It is divided into two concurrent parts: computation and communication engines. 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. There are a number of registers that are responsible for tuning communication computation overlap. 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, switch is the *master*.

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.

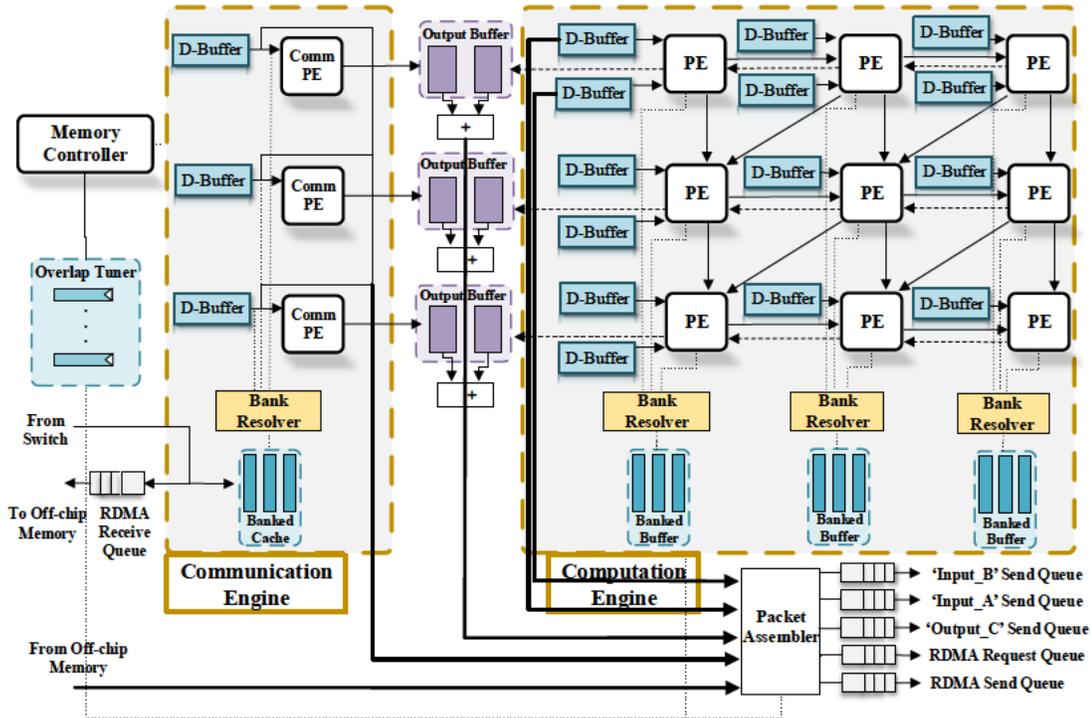


Figure 5-8: The proposed kernel logic architecture; it is divided into two concurrent parts: computation and communication engines.

SpMV

We consider a row-wise partitioning (Park et al., 2015b) for the input matrix and input/output vectors across the switch devices and then apply a 2D partitioning scheme among PEs for the local computation. Diagonal (off-diagonal) matrix elements (Park

et al., 2015b) 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. 5.8). To illustrate the process, PEs in the computation engine calculate the partial sums which get written into output buffers (Fig. 5.8). Meanwhile, column indices in the off-diagonal matrix provide the addresses for issuing RDMA requests to remote switches. 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 switches instantly; rather, they are lumped together (according to a threshold given by overlap tuner registers) and sent later.

PGEMM

We apply a pipelined version of the SUMMA algorithm (Van De Geijn and Watts, 1997). For this kernel, the computation engine is used as a systolic array for local computation. Computation is overlapped by sending input matrices to the neighbor switches. It is possible to tune communication and computation by varying the size of memory tiling between matrix dimensions.

5.6 Model for In-Switch Computing

In order 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 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 CCCFs. Moreover, 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 CCCFs 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:

$$T = (2 \times o) + L(N) + \frac{m}{G} + c(N, G) \quad (5.1)$$

$$c = (N \times \alpha) + \frac{(\gamma - 1) \times m}{G} \quad (5.2)$$

$$G = \beta \times BW \quad (5.3)$$

where T , L , N , m , and BW are the time it takes to complete a CCCF 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 case of fused collectives (distributed kernels). α , β , and γ represent the latency of the computational units (*G-FPin* 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. 5.1 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 NGRA 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. α , β , and γ 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)-fused collectives (Sec. 5.3).

5.7 Experimental Evaluation

In this section, we evaluate *G-FPin*. First, we provide the experimental setup used in this chapter. Afterwards, resource utilization of the smart switch is presented. Finally, we shed light on the performance and scalability of *G-FPin* for HPC kernels and applications.

5.7.1 Experimental Setup

For proof-of-concept we have implemented and tested *G-FPin* on a two-node FPGA-based system using the Xilinx *Vitis* unified software platform. In this 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. We used *Vitis* 2019.2 and Xilinx Runtime Library (XRT) 2.6.655. In this experiment, a sender process sends 1408 bytes worth of data to a receiver process using TCP/IP network logic handled by FPGAs. Fig. 5-9 gives a breakdown of the total execution time for different types of OpenCL APIs in both processes. It is repeated for five runs and the standard deviation (*std*) is shown.

As can be seen from the figure, FPGA programming (configuration) takes the bulk of the time. Moreover, OpenCL overhead associated with kernel setup, and with creating a context and a command queue, takes substantial time, more than an order-of-magnitude compared with the kernel execution.

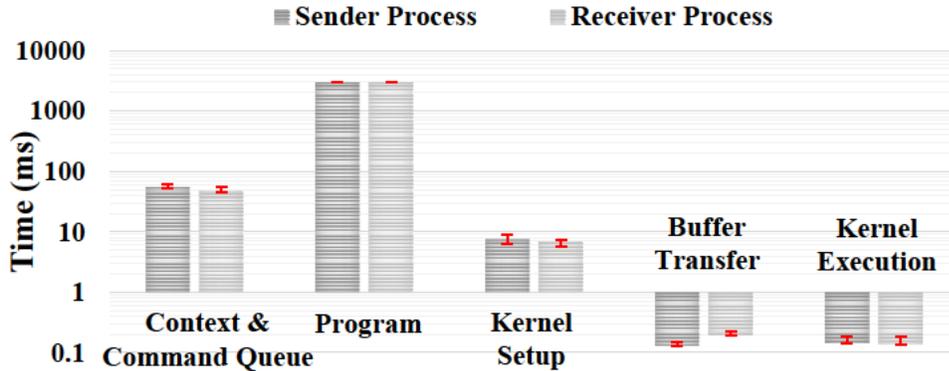


Figure 5-9: Time execution breakdown of a sender and a receiver process for 1408 bytes worth of data using *Vitis* framework and TCP/IP as the transport protocol. Error bars indicate *std* for five runs.

Since using *Vitis* has considerable overhead, we instead used Vivado flow and a simulator to compare *G-FPin* with the original MPI implementation. For simulation we used the SST simulator. We have written a new SST element for the router model. Router components are connected together through SST links and each component has an event handler associated with each link, in addition to a clock generator to also generate the events.

Table 5.2: Parameters used in the LoGcN model and our simulation.

* Denotes the Aurora IP latency.

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

For the rest of this section, the $G\text{-FPin}$ results shown are generated using the simulator and the LoGcN model. We consider an FPGA cluster (Xilinx VCU128) with up to 128 nodes with a direct network (3D-torus topology). The parameters used (derived from our system setup) are shown in Table 5.2. The last three parameters are used to evaluate the term L in the LogcN model. The latency of the $G\text{-FPin}$ components is not included in the reported port-to-port latency.

For the CPU reference, benchmarks were run on the TACC Stampede2 (Stanzione et al., 2017) 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.

For applications, first are two HPC kernels: SPMV and PGEMM; they are the backbone of many scientific, graph analytic, and machine learning applications. We also use a variety of applications including (1) NAS parallel benchmark (NPB) to cover various communication patterns; (2) MINIFE as an example of unstructured finite element methods; and (3) Algebraic Multigrid (AMG) (Park et al., 2015b) which is a widely used iterative solver with irregular communication.

5.7.2 Performance and Scalability

In this subsection, we evaluate the performance and scalability of $G\text{-FPin}$ for two kernels and a variety of applications.

Distributed Kernels

We consider two distributed kernels with different sizes: SPMV and PGEMM. For SPMV, we study two sparse matrices from *SuiteSparse*: `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 (Heroux et al., 2005) (Tpetra package) as the CPU reference. For PGEMM, we consider a square (16K×16K) and a tall-skinny (1K×128K) dense matrix and compare *G-FPin* with the state-of-the-art COSMA (Kwasniewski et al., 2019) CPU implementation. We considered strong scaling for both PGEMM and SPMV. Also, the former is based on single-precision floating point while the latter is double-precision.

Fig. 5-10 shows a performance comparison of *G-FPin* 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, *G-FPin* provides considerable improvement as it is able to almost *fully* overlap communication and computation in the switch. According to the results, *G-FPin* provides 3.2× and 1.5× performance improvements compared with an optimized CPU implementation (Kwasniewski et al., 2019) for square and tall-skinny matrices, respectively. Similarly, for the SPMV kernel, *G-FPin* 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 *G-FPin* is able to maintain the speedup with scale.

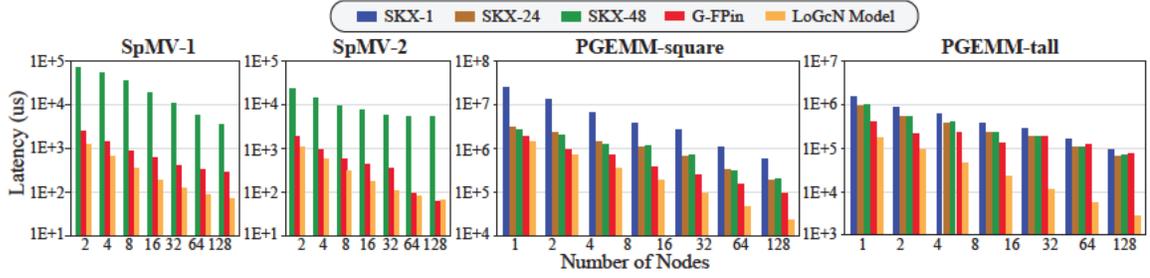


Figure 5-10: Scaling comparison of distributed kernels on SKX vs. *G-FPin*. The two left-most plots are the SPMV kernel (parabolic_fem and sme3Dc matrices) and the two right-most are the *PGEMM* kernel (square and tall matrices). 1, 24, and 48 OpenMP threads were used for the CPU SKX cluster.

Applications

Fig. 5-11 shows the performance benefit of *G-FPin* 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. 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. 5-11). One reason is that this model is not able to predict the performance of distributed kernels 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. 5-11, 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 *G-FPin* can take advantage of communication-computation overlap and in-network data reduction. For MG, this is partly because

there are a larger number of CCCF calls. In contrast, SP manifests the smallest performance gain simply because the number of CCCF calls is the smallest. Also, note that it is possible to utilize non-blocking CCCFs, taking advantage of CPU idle time during the CCCF offload. This happens for one of the CCCFs in LU benchmark.

For MINIFE, the performance improvement percentage is typically higher than that of NPB. One reason is that distributed kernels (DOT-PRODUCT and 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 CCCF.

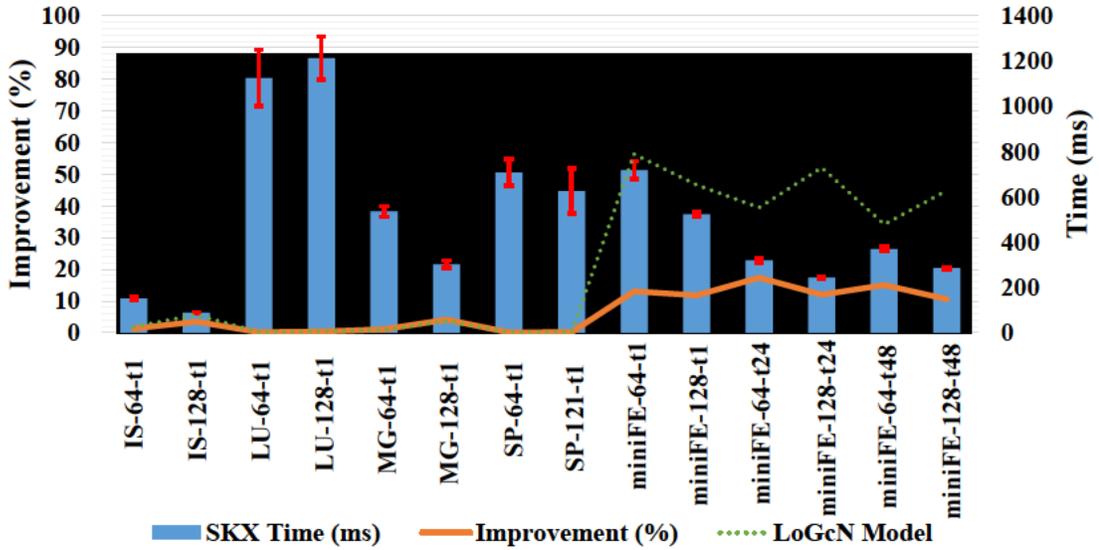


Figure 5-11: Performance improvement of *G-FPIn* over original MPI implementation on SKX cluster (64 and 128 nodes) for the NAS parallel benchmarks and MINIFE. SKX time and error bars represent the time it takes on SKX cluster and *std* for five runs, respectively.

Fig. 5-12 shows the performance and scalability of *G-FPIn* vs. the optimized CPU implementation (Hypre) and SHaRP (Graham et al., 2016) for AMG. We used the *lap3d* matrix (Park et al., 2015b) with size 1M rows. The numbers in front of SKX and *G-FPIn* 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 *G-FPin*. 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 *G-FPin* 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), *G-FPin* 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. Our approach provides considerable improvement compared with SHArP.

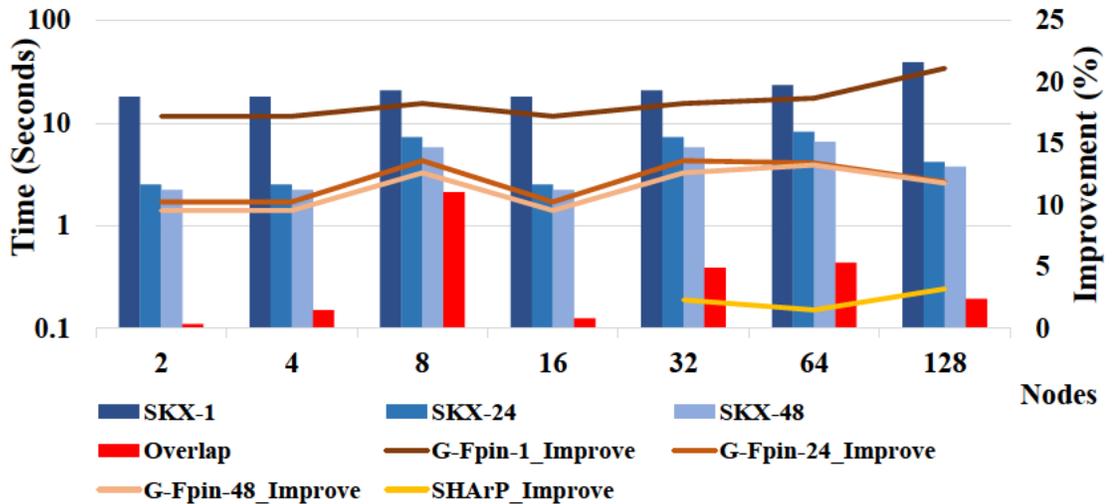


Figure 5.12: AMG performance and scalability comparison between *G-FPin*, the optimized CPU implementation, and SHArP (with data adopted from (Graham et al., 2016)) for 2 up to 128 nodes with different number of threads (1, 24, and 48).

5.8 Conclusion

In this chapter, we introduce fused collective support in the switches (Type 3 ACiS). We propose a general-purpose, MPI-transparent, framework for in-switch computing in reconfigurable devices to accelerate complex communication-computation functions (CCCFs). 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 CCCFs. 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.

Chapter 6

Type 4 ACiS: Control

This chapter introduces ACiS control capabilities. This time the switch accelerator controls and manages the execution of applications as an active device (rather than a passive device). The previous plugins can be used for this type. The case studies are: (1) implementing time synchronization algorithms in the switch accelerator, (2) new paradigm of parallel computing for machine learning applications (e.g., dynamic stale synchronous parallel), and (3) workload rebalancing. We focus on the first use case and we propose a predictive mechanism for time synchronization to improve the workload imbalance and process skew among processes in collectives. The implementation of our proposed time synchronization mechanism is left as a future work. This chapter is based on the work published in High Performance Extreme Computing (HPEC) ©2021 IEEE (Haghi et al., 2021) and High Performance Extreme Computing (HPEC) ©2022 IEEE (Chen et al., 2022).

6.1 Introduction

One problem with compute-in-the-network is that it is limited by inefficiency in computation, specially process skew that we will discuss in this section. It is common for MPI programs to designate a specific process to act as a coordinator or manager. This process may handle tasks such as distributing work among other processes, aggregating results, or orchestrating the overall execution. Similar to this approach, we can use the switch to control the execution of applications as switches have a global

view of leaf nodes. For example, leaf nodes can send telemetry information to the switch and the switch can synchronize or rebalance the workload depending on the use case. Figure 6-1 shows the general model for ACiS Type 4 (control in the switch) where telemetry information and other signals are sent to the switch and the switch controls leaf nodes, for example, by synchronizing them.

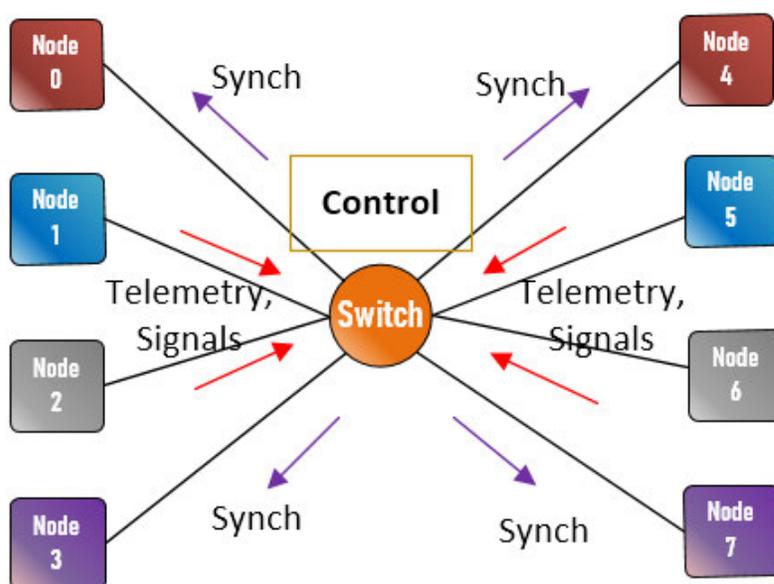


Figure 6-1: General model of control-in-the-switch. Telemetry information and other signals are sent to the switch and the switch controls leaf nodes, for example, by synchronizing them.

The first use case is application synchronization. In our previous work, we showed that process skew can be detrimental to performance of HPC applications (Haghi et al., 2021). To address it, we introduced a new method called extra work by considering a deadline (Chen et al., 2022). In the first use case, we posit that the switch can handle the management of deadline by sending signals to leaf nodes using a counter. This is a lightweight overhead for switches. This scenario is depicted in Figure 6-2 where the leaf nodes send data for collective communication as before and the switch keeps track of deadline and it sends a signal to leaf nodes when the counter

is triggered.

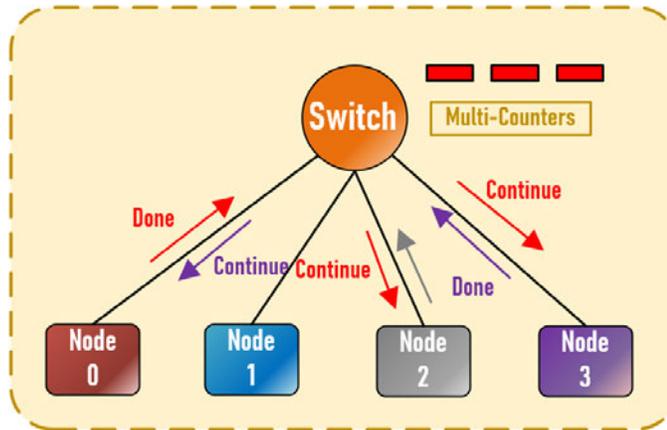


Figure 6-2: Control-in-the-switch model for the first use case: global support for time synchronization algorithms. Leaf nodes send data for collective communication as before and the switch keeps track of deadline and it sends a signal to leaf nodes when the counter is triggered.

The second use case is the emerging models for DNN training such as Stale Synchronous Parallel (SSP) where instead of synchronizing all processes, some processes can be a number of iterations ahead of others bounded by a limit. This way, accuracy is sacrificed for performance. To realize it, the switch keeps track of iterations for each process by having multiple counters. In this model, leaf nodes send a done signal to the switch and switch sends back a continue signal if at least the done signal of P processes are received. This is shown in Figure 6-3.

The last use case is workload rebalancing by managing task queues which has applications in graph partitioning algorithms and temporal graph networks. For example, in graph applications, evolving graphs have serious workload imbalance problems. Switches can monitor the workload and manage the task queues at runtime. For the rest of this section, we will focus on the first use case, global support for time synchronization algorithms.

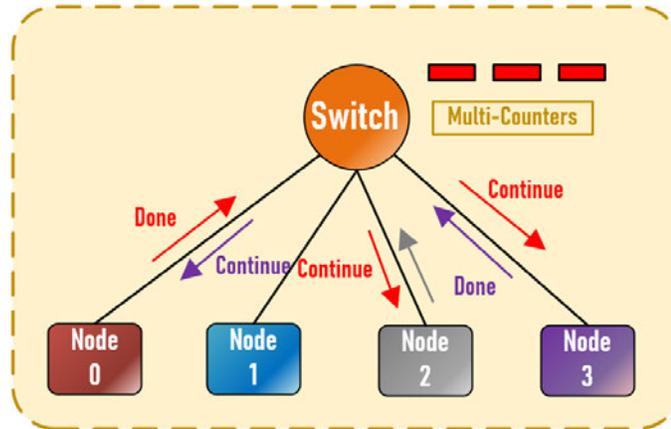


Figure 6.3: Control-in-the-switch model for the second use case: stale synchronous parallel for DNN training. Leaf nodes send a done signal to the switch and switch sends back a continue signal if at least the done signal of P processes are received.

6.2 Sample Use Case for Type 4 ACiS: Global Support for Application Synchronization

The fundamental problem of parallel processing is determining which process should do what work. The most basic method of partitioning (i.e., decomposition into tasks and assignment of tasks to processes (Culler et al., 1999)) is the set of methods known as load balancing. Depending on the application, partitioning may be obvious, as in, say, many BSP applications; or it may require some run-time updates (semi-static and dynamic methods). In all cases, however, there are limits to the quality of partitioning with the result that, frequently, much of a parallel program’s execution time is spent idle or performing overhead operations (Haghi et al., 2021; Haghi et al., 2020c; Haghi et al., 2022). Workload imbalance can originate from different sources. One type is inherent to the application itself (*i.e.*, some processes have more work to do). Another type is due to external noise coming, e.g., from the operating system, network, checkpointing, or mapping (Ferreira et al., 2008; Petrini et al., 2003; Widener et al., 2014).

One possible solution is to spend some of this idle time performing ever more

frequent and complex dynamic load balancing; the gain, however, is limited by new overhead for extra work, synchronization, communication, and scheduling. An alternative is to reduce the cost of the idle compute resources—the slack time of the leader processes waiting for laggards—e.g., by slowing down the leader processes (Cesarini et al., 2020). Another alternative is for the leader processes to do extra work while waiting for the laggards. The most general form here is simply time sharing: the leader processes block while waiting and turn over their resources to the operating system. This is generally not useful in HPC, however, due both to the immense amount of OS overhead and the way that processes are scheduled. A different way to execute extra work, and the focus of our study here, is to allocate the excess computing resources to predefined, user-space tasks. For now we leave the *extra work* undefined, but postulate either work that is useful for the current program, or some generic or fungible other work (e.g., SETI at home or bitcoin mining). The value of the extra work is likely to be less than that of the *real work*; the ratio of their values can be a parameter used in optimization.

Determining whether the Extra Work Method (EWM) is viable is the focus of this study and requires several research questions to be answered. First, is there sufficient slack time to make EWM worth the overhead? Second, is it predictable that certain processes will be leaders or laggards? If so, then traditional semi-static load balancing might be preferable. Third, which method should be used to implement EWM? There are at least two possibilities. In *asynchronous* EWM, extra work is performed until the final laggard process completes (reaches the soft barrier) and messages sent to the other processes. In *synchronous* EWM, the execution time of the laggard process is predicted. Then, when any process completes its real work, it checks whether it has exceeded the predicted time and, if not, executes extra work until then. Questions arising in synchronous EWM include: how variable is the execution time (is prediction

viable)? How can it be predicted? How accurate are the predictions? What is the cost/benefit of missing a prediction? Should the prediction be guaranteed? How often should the prediction be updated? For simplicity, we concentrate on Bulk Synchronous Parallel (BSP) applications, which often consist of a series of similar iterations.

To obtain results with respect to several of these questions we use Stampede 2 cluster (Stanzione et al., 2017) with up to 1536 processes (32 nodes) and a set of five BSP proxy applications from Exascale Computing Project (ECP). We find that all of these applications benefit from EWM with an average improvement of 12% improvement and that this benefit is likely to increase with production applications on realistic problem sizes. We summarize our contributions:

- We propose a new method of improving the efficiency of BSP programs in HPC through the use of extra work;
- We demonstrate an optimization method for EWM based on predicting the time interval during which the extra work should be executed;
- We evaluate the workload imbalance of time intervals for BSP supersteps on Stampede2 compute cluster in different HPC applications;
- We present experimental results showing predictability, the characteristics of the extra work deadline, and the overall benefit of the approach for a preliminary set of BSP proxy HPC applications.

6.2.1 Application Space

The Bulk Synchronous Parallel Model

The Bulk Synchronous Parallel (BSP) model (Cheatham et al., 1996) was developed as a theoretical augmentation of the PRAM model, but is now commonly used to

refer to parallel programs that execute in well-defined iterations (supersteps), each terminated by a global barrier. In each superstep, processes perform local computation, communication, and, finally, synchronization. A large fraction of HPC applications fall generally into this model. For instance, in stencil computations, processes communicate with neighbors and perform computation; this happens for a number of iterations (Krishnamoorthy et al., 2007). Iterative solvers operate by having each process perform part of the computation; processes are synchronized and convergence checked at the end of each superstep (Haghi et al., 2020a).

Applications

Our study adopted five selected applications from the ECP Proxy Applications Suite. The *proxies* are designed to be lightweight and simplified while encapsulating the essence of large applications. The simplicity offers the ideal subjects for our evaluation. We described the following applications that implement BSP algorithms.

MiniVite is an iterative graph proxy implementation based on MPI+OpenMP. It performs the first phase of Louvain’s method, a community detection algorithm (Ghosh et al., 2018). In real-world large networks, the vertices exhibit densely connected clusters (communities) with considerably more edges than the connections outside. The notion of *modularity* serves as a metric to reflect the density of the edges within the community relative to the ones outside, and it is optimized as the iterations progress. The algorithm converges in a non-deterministic number of iterations.

MiniFE is a proxy application that approximates unstructured implicit finite element by solving a sparse linear system of equations. The kernels exhibit patterns that are similar to many HPC applications. Specifically, the element-operator computations (diffusion matrix and vector); communications via scattering to sparse matrix and

vector; sparse linear algebra operations (conjugate gradient solver); and scalar-vector operations.

HPCCG is a conjugate gradient solver for a 3D chimney domain. Similar to MiniFE, the main kernels involve generating finite difference matrix, sparse linear algebra operation (matrix-vector multiplication) and scalar-vector operations. In addition, the implementation allows for configurable sub-block size for each processor.

CoMD is a proxy application for molecular dynamics simulations. The performance depends on the efficiency of evaluating of all forces between atom pairs within some short distance. The program can be broken down into three main stages: inter-node computation, each node computes forces and evolving positions of the atoms within its domain; intra-node computation, assignment of atoms to link cells and checking for interactions; inter-node communication, exchanging information of kinetic and potential energy.

LULESH is a proxy application that approximates hydrodynamics equations by partitioning the Sedov blasts problem into a collection of volumetric elements simulated by a mesh. In addition to following BSP model, the program allows for control over load balance in each set of elements.

6.2.2 The Synchronous Extra Work Method

Overview of the EW method

The purpose of EWM is to increase the utilization of compute resources during execution of HPC applications. The basic idea is that, rather than idling, waiting processes perform extra work (EW). We make certain assumptions:

Assumption 1: There is a global distributed clock sufficiently accurate so that the skew is small in comparison to the time scales critical to EWM, say, on the order of microseconds. Although current leadership class systems do not have such a capability

(Jones et al., 2018; Stunkel et al., 2020), there are no technological hurdles to their introduction (see, e.g., (Geng et al., 2018c)).

Assumption 2: EW exists that is useful and whose execution is predictable, i.e., it can be *tokenized*. Ideally the EW is useful to the running application, but we do not want to restrict EW to just that. There are many fungible applications whose executions have non-zero value.

Assumption 3: EW is not as useful as the application work. Further, the user (or system administrator) can specify the utility of the EW as some fraction of the utility of the application work.

Assumption 4: There does *not* exist an efficient mechanism to preempt a process. While these mechanisms can be implemented, they are not a standard feature of current large scale systems. We therefore concentrate on *synchronous* EWM and leave *asynchronous* EWM to a further study.

Assumption 5: The cost of starting up and tearing down EW is small. Since we are constraining EW this is reasonable for at least some types of EW, but will be investigated further in another study.

We now sketch EWM for a generic BSP application.

- * There is a start-up phase of some number of iterations during which data is collected so that a prediction can be made as to the iteration time and also to determine the distribution, over the processes executing the application, of the amount of application work (and idling) performed by the processes.

- * This prediction is used to create a deadline, which is used to optimize EWM. Note that this deadline is unrelated to the use of this term in various types of real time systems. Also, that the prediction is completely empirical (*a posteriori*) and not related to any “hard deadline” of the type found in real-time operating system (RTOS) and often determined *a priori*. In particular, violation is only an inconvenience and

not comparable to a violation in a hard RTOS.

* Using this deadline we have the following pseudocode for each process in EWM.

Algorithm 2: Extra Work Method (EWM)

```

1 do
2   DO ApplicationWork    // for this iteration
3   if  $T < T_{DL}$  then
4     // beat deadline?
5     do
6       ExtraWork
7       while  $T \neq T_{DL}$ ;
8   BARRIER()
9 while Termination Condition = False;

```

6.2.3 Definitions

Before continuing with details of EWM we summarize the definitions in use.

Iteration := a.k.a. superstep. This is the fundamental unit of BSP and the unit of prediction.

AW and *EW* are the application and extra work, respectively.

T_{global} := or simply T , is the current time on the (postulated) accurate distributed clock.

T_{lag} := is the time it takes the last (*laggard*) process to finish its work during an iteration and defines the iteration time in non-EWM.

T_{AW} := for each process, the amount of time during an iteration spent working on the application.

T_{slack} := $T_{lag} - T_{AW}$. Without EW, this is the amount of *slack* time the process spends idling while waiting for the laggard process.

T_{DL} := A timestamp (roughly, a soft deadline) from the beginning of the iteration.

It is used to optimize EWM. Like T , this is a global value.

$T_{EW} := \max(0, T_{DL} - T_{AW})$. EW is executed in the interval between the time when the application work is completed and the deadline. If the process misses the deadline, then there is no EW.

$T_{idle} :=$ Without EW, this is T_{slack} . With EW, there can also be idle time, but only if the laggard process misses the deadline ($T_{lag} > T_{DL}$). In that case, for processes that meet the deadline ($T_{AW} < T_{DL}$), $T_{idle} = T_{lag} - T_{DL}$. For non-laggard processes that *also* miss the deadline, $T_{idle} = T_{lag} - T_{AW}$.

$T_{iter} := \max(T_{lag}, T_{DL})$ is the execution time for a particular iteration (superstep). See Figure 6.4: Since in EWM T_{DL} can be longer than T_{lag} , T_{iter} is the greater of the two.

$U :=$ The coefficient of utility of the EW. This is the assigned fraction of utility of EW with respect to AW.

Scenarios

We assume a timestamp mechanism based on the predicted time per iteration (T_{lag}). The timestamp, or *deadline* (as we call it here), can be either *safe* or not. By safe we mean that all processes are guaranteed (with high probability) to complete by the deadline T_{DL} . As we see in the next section, there is substantial variation in iteration time T_{iter} which would make the safe very inefficient. We assume that T_{DL} is set to maximize overall performance. In this version, the laggard process, which sets the non-EWM iteration time, can finish its AW either before the deadline ($T_{lag} < T_{DL}$) or after ($T_{lag} > T_{DL}$); the implications of both are discussed in Section IV.

For non-laggard process execution there are three scenarios as shown in Figure 6.4; execution of the laggard process follows immediately. In (a), $T_{lag} < T_{DL}$. All processes finish their AW before the deadline and execute EW until the deadline. In (b) and

Table 6.1: Experimental Configuration for Proxy Applications

Application	Small Input	Medium Input	Large Input	# Processes
miniVite	-p 3 -l -n 128000	-p 3 -l -n 256000	-p 3 -l -n 512000	256
miniFE	-nx 20 -ny 20 -nz 20	-nx 40 -ny 40 -nz 40	-nx 60 -ny 60 -nz 60	1536
HPCCG	64 64 64	100 100 100	200 200 200	1536
CoMD	-x 128 -y 128 -z 128	-x 256 -y 256 -z 256	-x 512 -y 512 -z 512	1024
LULESH	-s 30 -p -i 500	-s 40 -p -i 500	-s 50 -p -i 500	512

(c), $T_{lag} > T_{DL}$. In (b), the process finishes its AW before the deadline, executes EW until the deadline, but then is idle until T_{lag} . In (c), the process executes AW beyond the deadline and then is (again) idle until T_{lag} .

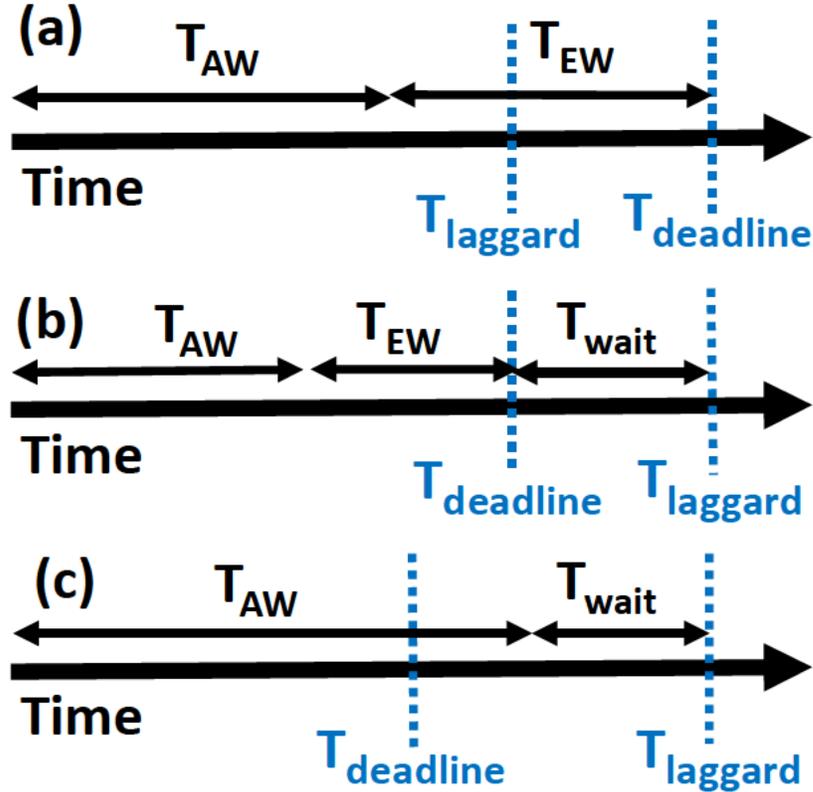


Figure 6.4: The three cases for non-laggard process execution.

6.2.4 Evaluation

Experimental Setup

For these preliminary experimental results, we executed the applications described in Section 6.2.1 on the Stampede 2 cluster (Stanzione et al., 2017). We picked five

BSP-based proxy applications inspired from the work (Guo, 2020) with the configuration shown in Table 6.1. We used up to 32 nodes each with 48 processes (1536 processes in total) but we used fewer processes for some of the applications due to their requirements. Figure 6-5 shows an example (HPCG) of our measurement. The figure shows T_{AW} for laggard and leader processes in addition to their deviation across the iterations. As evident from the figure, the deviation is comparable to T_{AW} itself. Due to limited space, we show a sample of results selected from configurations per application (bolded in Table 6.1), which covers small, medium, and large inputs. This sample is largest possible inputs for which runs completed, which is also the most commonly run in HPC environments.

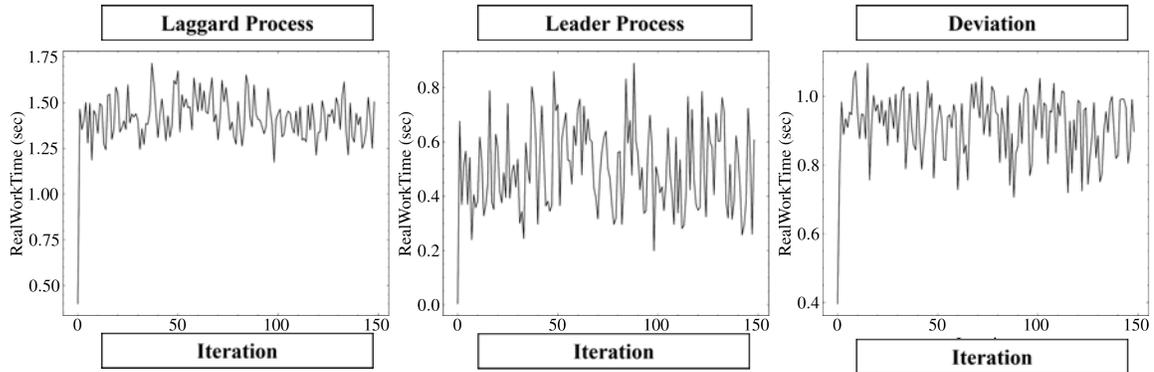


Figure 6-5: time measurement (application work) of HPCG per iteration for laggard and leader processes and the deviation.

Are Processes Systematically Laggard?

If we are able to identify processes as being *systematically laggard*, then the issue may be originated from the underlying hardware (e.g., slow clock), mapping (e.g., remote node), or the application itself (some processes have more work). In that case, non-EWM methods are preferred.

Figure 6-6 is a series of histograms, one for each application. The bins are the processes sorted by frequency of being laggard. The counts are normalized with the

expected frequency of laggardness ($\#$ -of-iterations/ $\#$ -of-processes) being set to 1.

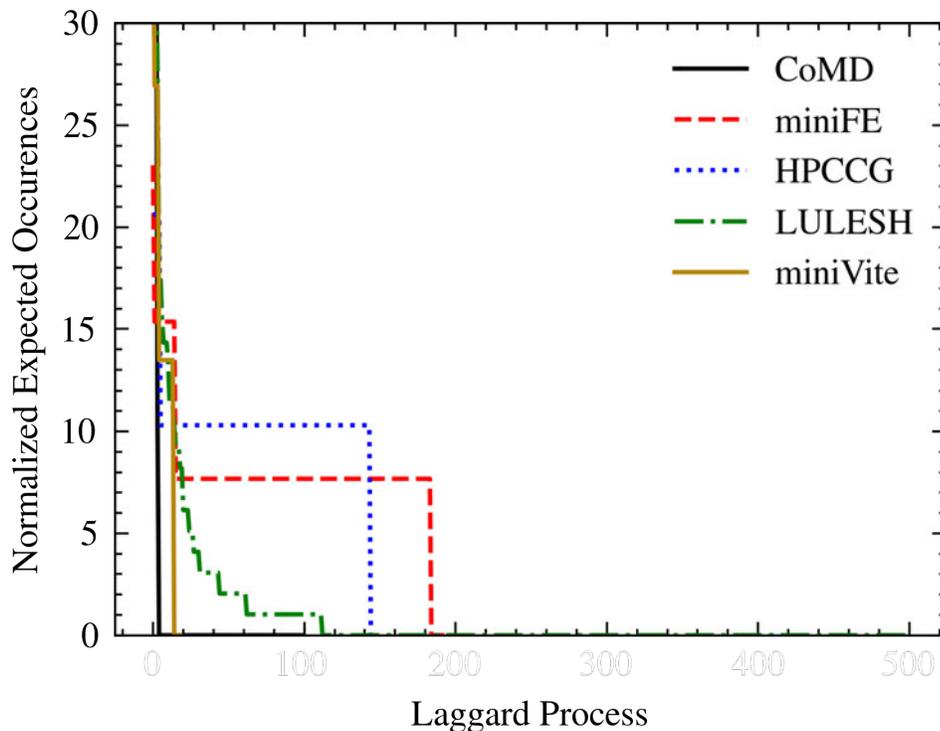


Figure 6-6: Laggard process occurrences sorted by the frequency.

If the laggard processes were chosen by chance we would expect a Poisson distribution and a maximum value of roughly the log of the $\#$ of processes. This is what we observe in Figure 6-6 indicating that—for these runs—the selection of the process that is laggard is indistinguishable from random selection.

What is the potential benefit of EWM?

We find here whether EWM is sufficiently promising to warrant further investigation. The maximum potential benefit of EWM is shown by the behavior of applications under normal execution (without executing EW), i.e., through the proportion of slack time with respect to total time (T_{slack} / T_{iter}) for all processes over all iterations. Table 6.2 shows, for each application, the proportion of idle time. Also shown is the average wall-clock idle time per process per iteration.

Table 6.2: Fraction of execution time that a process is idle and average idle time per iteration in seconds.

Application	Avg Idle Ratio	Avg Idle Time
miniVite	2.7%	0.0001
HPCCG	32%	0.32
CoMD	8.1%	0.46
LULESH	2.6%	0.003
miniFE	89%	0.001

We observe that the potential benefit for all of the applications with an average of 12% improvement. We note that miniFE and HPCCG have the highest benefits. We observe further that overhead time, e.g., to initiate EW, is likely to be small in comparison to the average idle time per process per iteration especially for HPCCG and CoMD.

Is optimizing T_{DL} beneficial?

We begin this subsection by continuing our examination of the various EWM scenarios (e.g., Figure 6-4) with the goal of finding whether it is beneficial to predict a deadline with the potential to be violated. If so, then the further question is how T_{DL} should be optimized.

We note that EW is not as valuable as AW. Without knowing the details of EW we postulate that, for any run, it is worth some fixed fraction of AW which we denote as U for *coefficient of utility*. We also note that idle/waiting time has $U = 0$. We therefore create as our metric *Average Work Rate* AWR:

$$AWR = \frac{T_{aw} + U * T_{ew}}{T_{iter}} \quad (6.1)$$

The iteration time is divided into three parts: AW, EW, and idling. AW is weighted at 1, EW at U , and idling at 0. Returning to Figure 6-4 we see that there are a number of cases.

Baseline: In the baseline case, with no EW, AWR is simply T_{aw}/T_{iter}

EWM: In this case, there are three possibilities:

a): In the case where $T_{lag} < T_{DL}$, all of the idle time has been replaced EW. We observe that the interval T_{EW} during which EW executes has two parts, before and after T_{lag} . The EW during the first interval is entirely positive ($U > 0$), while the *excess* T_{EW} , which continues execution after the normal termination of the iteration, is negative ($U < 1$).

b): Where $T_{DL} < T_{lag}$, in the first case the particular process itself has “beat” the deadline, although some other processes have not. In this case its EW is purely beneficial in that it replaces idling ($U > 0$), but its idling is not because it replaces EW ($0 < U$).

c): Where $T_{DL} \leq T_{lag}$, in the second case the particular process itself has *not* beat the deadline. For these processes there is no EW and the AWR is the same as that of the baseline.

To summarize, setting the deadline either too safely or too aggressively both have their detriments. If the deadline is too safe, then T_{iter} is extended unnecessarily and some AW is replaced with EW. If the deadline is too aggressive, then some EW is unnecessarily replaced with idling.

We predict an optimal deadline based on some number of iterations during which T_{AW} (for each process) and T_{lag} (for the entire iteration) are recorded (see Figures 6-5 and 6-8). Since the T_{lag} probability distribution function (Figure 6-8) may not lend themselves to a simple expression we estimate the optimal T_{DL} as the following.

Note that, despite the several cases, the computation reduces to simply maximizing AWR, which is itself a simple calculation. Figure 6-7 shows the optimal T_{DL} for various U from 0.1 to 1 in tenth increments. We note the variety of ranges and shapes of the curves: these are being investigated further.

Algorithm 3: Parameter Search to optimize T_{DL}

Input: $U \leftarrow$ Utility Coefficient

- 1 $MaxIterTime = \max(T_{iter})$
- 2 **for** $T \in [0, MaxIterTime]$ **do**
- 3 $T_{EW} = \max(0, T - T_{AW})$
- 4 **foreach** iteration i **do**
- 5 **foreach** process p **do**
- 6 $WorkRate_{i,p} = (T_{AW} + U * T_{EW}) / T_{iter}$
- 7 **end**
- 8 **end**
- 9 $AWR_T = \text{AVG}(WorkRate)$
- 10 **end**
- 11 **return** $T_{DL} = \underset{T}{\text{argmax}} AWR$
- 12

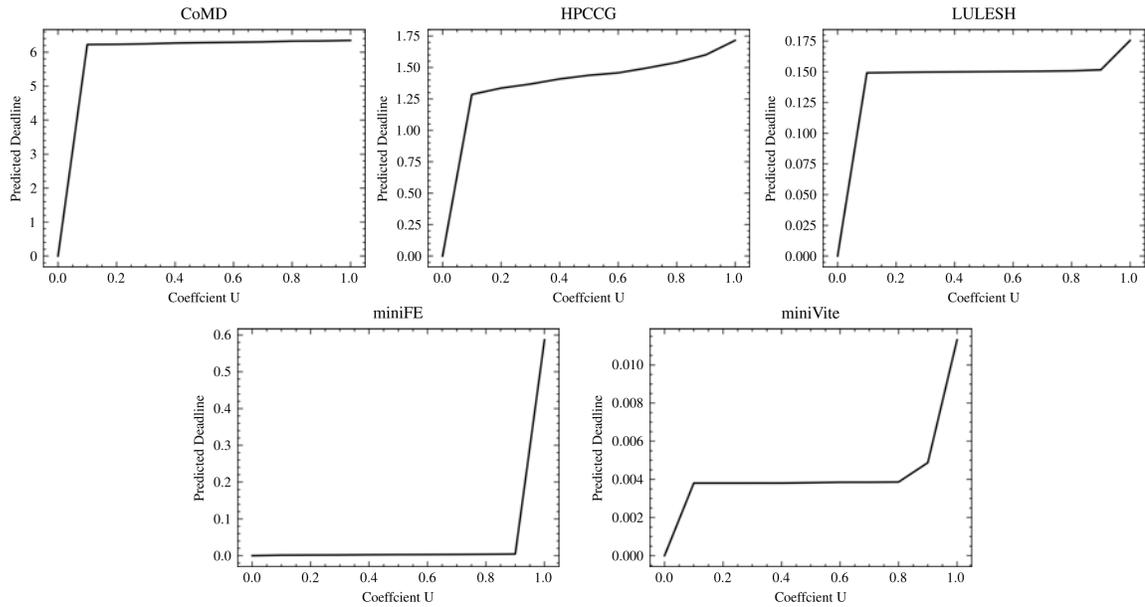


Figure 6-7: Deadline Prediction (in seconds) with different Coefficient U.

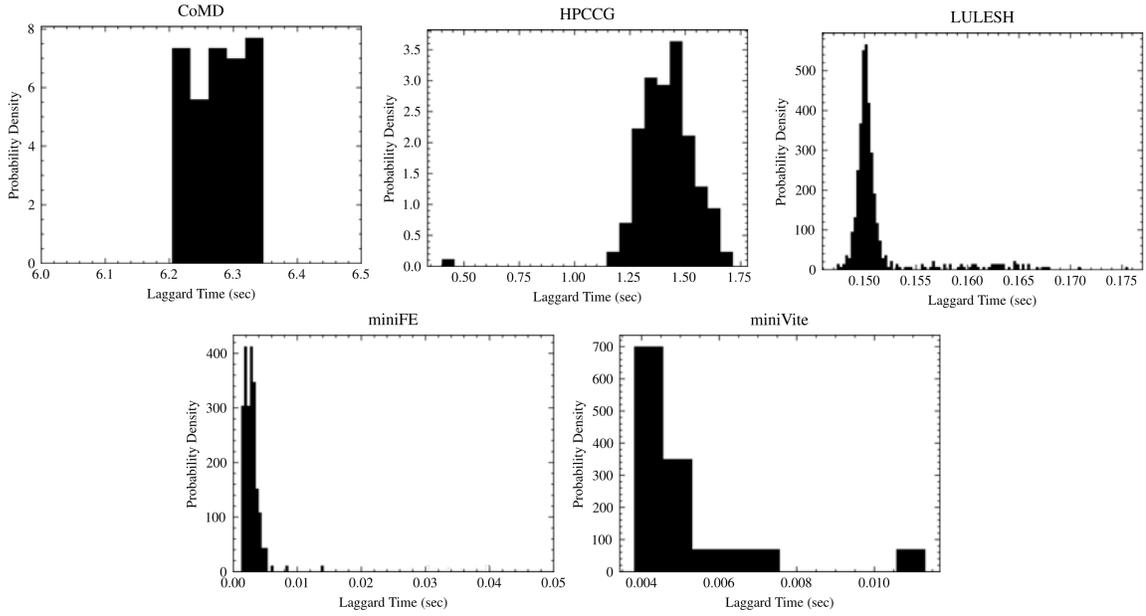


Figure 6-8: Probability Density Function (PDF) of T_{lag} for different applications

The benefit of EWM

Now we examine the benefit of synchronous EWM with optimal deadlines under the current workloads. To do so we find the AWRs for selected U s (0.5 and 0.9) and compare with the AWRs of the baseline (average of T_{aw}/T_{iter} for all processes and all iterations with no deadline or EW). The results are shown in Table 6.3.

Table 6.3: Average work rates AWR for baseline and EWM for two extra work utility coefficients U .

Application	AWR		
	baseline	$U = 0.5$	$U = 0.9$
miniVite	97.5%	97.6%	97.9%
HPCCG	76.64%	85.66%	96.64%
CoMD	92.56%	96.01%	99.16%
LULESH	97.42%	98.25%	99.17%
miniFE	62.47%	72.8%	93.22%

Discussion

With the current preliminary results we have found that, for most of the applications, EWM gives sufficient benefit to make this an approach worth examining further while

having little benefit for certain applications. Part of the promise is that having larger problem sizes and higher number of processes leads to higher imbalance. Moreover, we expect that using OpenMP parallelism in addition to MPI yields more imbalance as well.

A further question is how often T_{DL} should be recomputed. The calculation is simple, which indicates frequent computation; however, the limited range of optimal T_{DL} (in Figure 6.7) indicates that this may not be necessary.

6.3 Conclusion

One problem with compute-in-the-network is that it is limited by inefficiencies in computation, especially process skew. This motivates us to propose that switches take control of application execution. This is useful because switches have a central view of the network and they can collect telemetry information and monitor application behavior. In this chapter, we introduce Type 4 ACiS: control. This time the switch accelerator controls and manages the execution of applications as an active device (rather than a passive device). We discuss three use cases: deadline based time synchronization, deferred synchronization models used in machine learning applications, and workload rebalancing. We explore the possibility of the first use case in depth. In the first use case, we present a new deadline-based method, called extra work method (EWM), to exploit the time wasted by processes idling while running bulk synchronous parallel (BSP) applications. This idling is a result of many factors including workload imbalance and various forms of system noise. EWM is based on, first, using the idle time for extra work, and, second, by optimizing the method by predicting a deadline for supersteps of BSP. We run proxy applications on a production supercomputer and find that there is benefit for all five of the applications with an average of about 12%. We posit that since switches have a global view of the

network, they are good candidates for this deadline-based method; sending a signal to leaf nodes upon reaching the deadline and thereby controlling the execution of applications. ACiS implementation of use cases will remain as our future work.

Chapter 7

Summary and Future Work

7.1 Conclusion

A tenet of HPC is that compute should go to the data and not the reverse: there are orders-of-magnitude advantages in power and latency with a local computation over an inter-node data transfer. Through an approach we denote by advanced computing in the switch (ACiS), we seek to explore deviations from this premise. We note that once communication has been minimized algorithmically, it follows that the remaining data transfers are usually essential because those data need to be combined with other, remote, data. Varying where this combining occurs presents a degree of freedom we seek to exploit.

In this thesis, we propose four different methods of compute-in-the-switch. We present flexible and application-aware accelerators for these types that can be embedded into or attached to existing communication switches. Experimental results show that these approaches improve the performance and scalability of applications. Our thesis is that **flexible, complex, and integrated application-level processing in communication switches improves the performance and scalability of HPC and machine learning applications.**

For the first ACiS type, we present a comprehensive solution to processing collectives in network switches with reconfigurable logic. This has the advantage over fixed logic alternatives of being able to support capabilities as needed. This includes varying the supported operation types and numbers of simultaneous operations, but

also enabling more general user-defined processing in the network. As part of this solution we present a new method for supporting MPI communicators and accelerating collectives in the network switch. We begin by considering the movement towards exascale computing and the need for offloading collectives and communicator support into hardware, in particular, for collectives occurring over irregular communicators. We find that storing entire process groups in the network is not a scalable solution. We then introduce the Communicator Table (CT), which takes advantage of the properties and patterns of collective communication in order to provide the accelerator hardware with the minimum amount of communicator information needed to perform collectives.

For the second ACiS type, we design, implement, and evaluate a programmable look-aside accelerator that can be embedded into, or attached to, existing communication switches. To facilitate usability, we develop a software toolchain to compile user-provided code for configuring the switch. While our approach is generic and supports a variety of workloads, we consider graph convolutional network (GCN) inference as a case study. Experimental results show that this approach improves both performance and scalability. The performance advantage is on average $3.4\times$ (across five real-world datasets) on 24 nodes.

For the third ACiS type, we propose a general-purpose, MPI-transparent, framework for in-switch computing in reconfigurable devices to accelerate complex communication-computation functions (CCCFs). 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 CCCFs. 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.

For the fourth ACiS type, we present a new method, called Extra Work Method (EWM), to exploit the time wasted by processes idling while running Bulk Synchronous Parallel (BSP) applications. This idling is a result of many factors including workload imbalance and various forms of system noise. EWM is based on, first, using the idle time for extra work, and, second, by optimizing the method by predicting a deadline for supersteps of BSP. In this preliminary study we run proxy applications on a production supercomputer and find that there is benefit for all five of the applications with an average of about 12%.

To provide insights, we summarize some of the important observations and conclusions of this thesis:

- 1) Switches have the potential to accelerate the applications by optimizing the communication and performing computation on that data on-the-fly. There is a degree of freedom on how and where this packet processing takes place which leads to different ACiS types.

- 2) We need system-level optimizations and application awareness, in addition to hardware (architectural) design and software support.

- 3) Process skew can be detrimental in the performance of real applications and leaves negligible performance benefits for simple collective communication offload if it is not efficiently addressed. This motivates new ACiS types with complex packet processing (Types 1 to 3) and control (Type 4).

- 4) For complex packet processing, there should be a deadline as otherwise packet loss happens. We handled this situation in ACiS.

- 5) Typically, on-chip buffers (memories) are the most valuable resources in our

ACiS accelerators. Designers should judiciously use up these resources and apply architectural and algorithmic optimizations to minimize their usage.

6) Switches cannot solve all of the inefficiencies of applications. While they benefit from a tremendous bandwidth they have limited hardware resources.

7) Packet reliability becomes very important compared to other communication devices such as NICs. This is because all leaf nodes can send packets to them simultaneously (becomes more important for larger message sizes). While we addressed some of the reliability issues in ACiS, there are still other considerations for a fully reliable transport. As part of our future work, we would like to add support for a reliable UDP transport to ACiS.

7.2 Future Work

We briefly discuss future work. As part of future work, we would like to continue our research in the following directions:

1) Implementing our proposed time synchronization algorithm (EWM) in ACiS switch accelerator and evaluating the performance and predictability of real applications.

2) Large Language Models (LLMs) are becoming increasingly important in our daily life. However, they suffer from communication and scalability inefficiencies. We would like to explore the usability of ACiS in this application as it has its own unique features in terms of communication patterns and parallelism.

3) We would like to explore the usability of ACiS in other novel communication libraries such as Portals4 (Barrett et al., 2017) that are closer to hardware and more suitable for Exascale computing.

4) We would like to explore the usability of ACiS in other emerging systems such as Compute Express Link (CXL) based and disaggregated memory systems as they

are considered as a potential option for future data centers.

References

- Agarwal, R. C., Balle, S. M., Gustavson, F. G., Joshi, M., and Palkar, P. (1995). A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582.
- Alexandrov, A., Ionescu, M. F., Schauser, K. E., and Scheiman, C. (1995). LogGP: Incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, page 95–105, New York, NY, USA. Association for Computing Machinery.
- Alizadeh, M. and Edsall, T. (2013). On the data path performance of leaf-spine datacenter fabrics. In *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pages 71–74.
- Almàsi, G., Heidelberger, P., Archer, C. J., Martorell, X., Erway, C. C., Moreira, J. E., Steinmacher-Burow, B., and Zheng, Y. (2005). Optimization of MPI collective communication on BlueGene/L systems. In *Proceedings of the 19th annual international conference on Supercomputing (ICS'05)*, pages 253–262.
- Arap, O. and Swamy, M. (2016). Offloading collective operations to programmable logic on a zynq cluster. In *2016 IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI)*, pages 76–83.
- Arista (2023). 7130 FPGA-enabled Network Switches - Quick Look. www.arista.com/en/products/7130-fpga-enabled-network-switches-quick-look.
- Arista Networks, Inc. (2013). <http://www.aristanetworks.com/en/products/7100series/7124fx/>, accessed 10/2013.
- AWS (2019). Deliver high performance ML inference with AWS Inferentia. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Deliver_high_performance_ML_inference_with_AWS_Inferentia_CMP324-R1.pdf.
- Ayala, A., Tomov, S., Stoyanov, M., and Dongarra, J. (2021). Scalability Issues in FFT Computation. In Malyshekin, V., editor, *Parallel Computing Technologies*, pages 279–287. Springer International Publishing.

- Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., et al. (1991). The NAS parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73.
- Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E., Thakur, R., and Träff, J. L. (2009). MPI on a million processors. In Ropo, M., Westerholm, J., and Dongarra, J., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 20–30, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Bandara, S., Sanauallah, A., Tahir, Z., Drepper, U., and Herbordt, M. (2022). Enabling VirtIO Driver Support on FPGAs. In *2022 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pages 1–8. doi: 10.1109/H2RC56700.2022.00006.
- Barrett, B., Brightwell, R. B., Grant, R., Pedretti, K., Wheeler, K., Underwood, K. D., Riesen, R., Maccabe, A. B., Hudson, T., and Hemmert, S. (2017). The portals 4.1 network programming interface. Albuquerque, N.M., Sandia National Laboratories, doi: 10.2172/1365498.
- Bayatpour, M., Maqbool, J. H., Chakraborty, S., Suresh, K. K., Ghazimirsaeed, S., Ramesh, B., Subramoni, H., and Panda, D. (2020). Communication-Aware Hardware-Assisted MPI Overlap Engine. In Sadayappan, P., Chamberlain, B. L., Juckeland, G., and Ltaief, H., editors, *High Performance Computing*, pages 517–535. Springer International Publishing.
- Bayatpour, M., Sarkauskas, N., Subramoni, H., Hashmi, J. M., and Panda, D. K. (2021). BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs. In *High Performance Computing*, pages 18–37. Springer International Publishing.
- Benkrid, K. and Vanderbauwhede, W., editors (2013). *High Performance Computing Using FPGAs*. Springer Verlag. doi: 10.1007/978-1-4614-1791-0_4.
- Bernholdt, D., Boehm, S., Bosilca, G., Venkata, M., Grant, R., Naughton, T., Pritchard, H., Schulz, M., and Vally, G. (2018). A Survey of MPI Usage in the US Exascale Computing Project. *Concurrency and Computation: Practice and Experience*, Special Issue:1 – 16.
- Bhatele, A., Mohror, K., Langer, S. H., and Isaacs, K. E. (2013). There goes the neighborhood: Performance degradation due to nearby jobs. In *SC’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12.

- Boku, T., Kobayashi, R., Fujita, N., Amano, H., Sano, K., Hanawa, T., and Yamaguchi, Y. (2019). Cygnus: GPU meets FPGA for HPC. In *ISC*. https://www.rccs.riken.jp/labs/lpnctr/assets/img/lspanc2020jan_boku_light.pdf.
- Bolaria, J. and Byrne, J. (2009). *A Guide to FPGAs for Communications*. The Linley Group.
- Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communication Review*, 44(3):87–95.
- Byma, S., Steffan, J. G., Bannazadeh, H., Leon-Garcia, A., and Chow, P. (2014). FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116.
- Caulfield, A., Chung, E., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.-Y., Lo, D., Massengill, T., Ovtcharov, K., Papamichael, M., Woods, L., Lanka, S., Chiou, D., and Burger, D. (2016). A cloud-scale acceleration architecture. In *49th IEEE/ACM Int. Symp. Microarchitecture*, pages 1–13.
- Cesarini, D., Bartolini, A., Borghesi, A., Cavazzoni, C., Luisier, M., and Benini, L. (2020). Countdown Slack: A Run-Time Library to Reduce Energy Footprint in Large-Scale MPI Applications. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2696–2709.
- Chan, E. W., Heimlich, M. F., Purkayastha, A., and van de Geijn, R. A. (2004). On optimizing collective communication. In *2004 IEEE International Conference on Cluster Computing*, pages 145–155.
- Cheatham, T., Fahmy, A., Stefanescu, D., and Valiant, L. (1996). *Bulk Synchronous Parallel Computing — A Paradigm for Transportable Software*, pages 61–76. Springer US, Boston, MA.
- Chen, P. H., Haghi, P., Chung, J. Y., Geng, T., West, R., Skjellum, A., and Herbordt, M. C. (2022). The Viability of Using Online Prediction to Perform Extra Work while Executing BSP Applications. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7.
- Chen, Y., Emer, J., and Sze, V. (2017). Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators. *IEEE Micro*, 37(3):12–21.

- Chiu, M. and Herbordt, M. (2009). Efficient filtering for molecular dynamics simulations. In *2009 International Conference on Field Programmable Logic and Applications*. doi: 10.1109/ FPL15426.2009.
- Chiu, M. and Herbordt, M. (2010). Molecular dynamics simulations on high performance reconfigurable computing systems. *ACM Transactions on Reconfigurable Technology and Systems*, 3(4):1–37. doi: 10.1145/1862648.1862653.
- Chiu, M., Herbordt, M., and Langhammer, M. (2008). Performance potential of molecular dynamics simulations on high performance reconfigurable computing systems. In *2008 Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications*. doi: 10.1109/ HPRCTA.2008.4745685.
- Chiu, M., Khan, M., and Herbordt, M. (2011). Efficient calculation of pairwise nonbonded forces. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. doi: 10.1109/ FCCM.2011.34.
- Chung, E., Fowers, J., Ovtcharov, K., Papamichael, M., Caulfield, A., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., Abeydeera, M., Adams, L., Angepat, H., Boehn, C., Chiou, D., Firestein, O., Forin, A., Gatlin, K. S., Ghandi, M., Heil, S., Holohan, K., El Hussein, A., Juhasz, T., Kagi, K., Kovvuri, R. K., Lanka, S., van Megen, F., Mukhortov, D., Patel, P., Perez, B., Rapsang, A., Reinhardt, S., Rouhani, B., Sapek, A., Seera, R., Shekar, S., Sridharan, B., Weisz, G., Woods, L., Yi Xiao, P., Zhang, D., Zhao, R., and Burger, D. (2018). Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38(2):8–20.
- Culler, D., Singh, J., and Gupta, A. (1999). *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan-Kaufmann, San Francisco, CA.
- Dally, W. and Towles, B. (2004). *Principles and Practices of Interconnection Networks*. Elsevier.
- Dang, H. T., Canini, M., Pedone, F., and Soulé, R. (2016). Paxos made switch-y. *SIGCOMM Computer Communication Review*, 46(2):18–24.
- De Matteis, T., de Fine Licht, J., Beránek, J., and Hoefler, T. (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*, pages 1–33.
- De Sensi, D., Di Girolamo, S., Ashkboos, S., Li, S., and Hoefler, T. (2021). Flare: Flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA. Association for Computing Machinery.

- De Sensi, D., Di Girolamo, S., McMahon, K. H., Roweth, D., and Hoefler, T. (2020). An in-depth analysis of the slingshot interconnect. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20.
- Demmel, J. (2013). Communication-Avoiding Algorithms for Linear Algebra and Beyond. Keynote Talk, 27th IEEE International Parallel and Distributed Processing Symposium.
- Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, R., Petitet, A., Vuduc, R., Whaley, R., and Yelick, K. (2005). Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 92(2):293–312.
- Di Girolamo, S., Kurth, A., Calotoiu, A., Benz, T., Schneider, T., Beránek, J., Benini, L., and Hoefler, T. (2021). A RISC-V in-network accelerator for flexible high-performance low-power packet processing. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 958–971.
- Dongarra, J., Otto, S., Snir, M., and Walker, D. (1995). An introduction to the MPI standard. <https://netlib.org/utk/papers/intro-mpi/intro-mpi.html>.
- Eppstein, D. and Galil, Z. (1988). Parallel algorithmic techniques for combinatorial computing. *Annual Review of Computer Science*, 3:233–283.
- Eran, H., Zeno, L., Tork, M., Malka, G., and Silberstein, M. (2019). NICA: An Infrastructure for Inline Acceleration of Network Applications. In *USENIX Annual Technical Conference*.
- Faber, C. J., Plano, T., Kodali, S., Xiao, Z., Dwaraki, A., Buhler, J. D., Chamberlain, R. D., and Cabrera, A. M. (2021). Platform agnostic streaming data application performance models. In *2021 IEEE/ACM Redefining Scalability for Diversely Heterogeneous Architectures Workshop (RSDHA)*, pages 17–26.
- Falgout, R. D. (2006). An introduction to algebraic multigrid. *Computing in Science and Engineering*, 8(6):24–33.
- Faraj, A., Kumar, S., Smith, B., Mamidala, A., and Gunnels, J. (2009). MPI collective communications on the blue gene/P supercomputer: Algorithms and optimizations. *Proceedings - Symposium on the High Performance Interconnects, Hot Interconnects*, pages 63–72.
- Ferreira, K. B., Bridges, P., and Brightwell, R. (2008). Characterizing application sensitivity to os interference using kernel-level noise injection. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12.

- Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., Chandrappa, H. K., Chaturmohita, S., Humphrey, M., Lavier, J., Lam, N., Liu, F., Ovtcharov, K., Padhye, J., Popuri, G., Raindel, S., Sapre, T., Shaw, M., Silva, G., Sivakumar, M., Srivastava, N., Verma, A., Zuhair, Q., Bansal, D., Burger, D., Vaid, K., Maltz, D. A., and Greenberg, A. (2018). Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA. USENIX Association.
- Gasteiger, J., Qian, C., and Günnemann, S. (2022). Influence-based mini-batching for graph neural networks. *arXiv preprint arXiv:2212.09083*.
- Geng, T., Li, A., Shi, R., Wu, C., Wang, T., Li, Y., Haghi, P., Tumeo, A., Che, S., Reinhardt, S., and Herbordt, M. C. (2020). Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936.
- Geng, T., Li, A., Wang, T., Wu, C., Li, Y., Shi, R., Wu, W., and Herbordt, M. (2021a). O3BNN-R: An Out-of-Order Architecture for High-Performance and Regularized BNN Inference. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):199–213. doi: 10.1109/TPDS.2020.3013637.
- Geng, T., Wang, T., Sanaullah, A., Yang, C., Patel, R., and Herbordt, M. (2018a). A Framework for Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters with Work and Weight Load Balancing. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 394–394.
- Geng, T., Wang, T., Sanaullah, A., Yang, C., Xu, R., Patel, R., and Herbordt, M. (2018b). FPDeep: Acceleration and Load Balancing of CNN Training on FPGA Clusters. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 81–84. doi: 10.1109/FCCM.2018.00021.
- Geng, T., Wang, T., Wu, C., Yang, C., Li, A., Song, S., and Herbordt, M. (2019a). LP-BNN: Ultra-low-Latency BNN Inference with Layer Parallelism. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, volume 2160, pages 9–16. doi: 10.1109/ASAP.2019.00-43.
- Geng, T., Wang, T., Wu, C., Yang, C., Wu, W., Li, A., and Herbordt, M. C. (2019b). O3BNN: An out-of-Order Architecture for High-Performance Binarized Neural Network Inference with Fine-Grained Pruning. In *Proceedings of the ACM International Conference on Supercomputing, ICS '19*, page 461–472, New York, NY, USA. Association for Computing Machinery.

- Geng, T., Wu, C., Tan, C., Xie, C., Guo, A., Haghi, P., He, S. Y., Li, J., Herbordt, M., and Li, A. (2021b). A Survey: Handling Irregularities in Neural Network Acceleration with FPGAs. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. doi: 10.1109/HPEC49654.2021.9622877.
- Geng, T., Wu, C., Zhang, Y., Tan, C., Xie, C., You, H., Herbordt, M., Lin, Y., and Li, A. (2021c). I-GCN: A Graph Convolutional Network Accelerator with Runtime Locality Enhancement Through Islandization. In *54th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. doi:10.1145/3466752.3480113.
- Geng, Y., Liu, S., Yin, Z., Naik, A., Prabhakar, B., Rosenblum, M., and Vahdat, A. (2018c). Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, Renton, WA. USENIX Association.
- George, A., Herbordt, M., Lam, H., Lawande, A., Sheng, J., and Yang, C. (2016). Novo-G#: A Community Resource for Exploring Large-Scale Reconfigurable Computing Through Direct and Programmable Interconnects. In *2016 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA*, pages 1–7. doi: 10.1109/HPEC.2016.7761639.
- Ghosh, S., Halappanavar, M., Tumeo, A., Kalyanaraman, A., Lu, H., Chavarrià-Miranda, D., Khan, A., and Gebremedhin, A. (2018). Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 885–895.
- Gilge, M. (2013). *IBM System Blue Gene Solution Blue Gene/Q Application Development*. An IBM Redbooks publication.
- Gokhale, M. and Graham, P. (2005). *Reconfigurable Computing: Accelerating Computation with Field Programmable Gate Arrays*. Springer.
- Graham, R. L., Bureddy, D., Lui, P., Rosenstock, H., Shainer, G., Bloch, G., Goldenberg, D., Dubman, M., Kotchubievsky, S., Koushnir, V., Levi, L., Margolin, A., Ronen, T., Shpiner, A., Wertheim, O., and Zahavi, E. (2016). Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *First International Workshop on Communication Optimizations in HPC (COMHPC)*.
- Graham, R. L., Levi, L., Bureddy, D., Bloch, G., Shainer, G., Cho, D., Elias, G., Klein, D., Ladd, J., Maor, O., Marelli, A., Petrov, V., Romlet, E., Qin, Y., and Zemah, I. (2020). Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) Streaming-Aggregation Hardware Design and Evaluation. In Sadayappan, P., Chamberlain, B. L., Juckeland, G., and Ltaief, H., editors, *High Performance Computing*, pages 41–59, Cham. Springer International Publishing.

- Graham, R. L., Poole, S., Shamis, P., Bloch, G., Bloch, N., Chapman, H., Kagan, M., Shahar, A., Rabinovitz, I., and Shainer, G. (2010). Overlapping computation and communication: Barrier algorithms and ConnectX-2 core-direct capabilities. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8.
- Gropp, W., Lusk, E., Ashton, D., Ross, R., Thakur, R., and Toonen, B. (2003). MPICH Abstract Device Interface Version 3.3 Reference Manual. Technical report, Draft MCS-TM-00, Argonne National Laboratory, 2002. <http://www-unix.mcs.anl.gov/mpi/mpich/adi3>.
- Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789 – 828. doi: 10.1016/0167-8191(96)00024-5.
- Guo, A. (2020). Mapping Applications Onto FPGA-Centric Clusters. Master’s thesis, Department of Electrical and Computer Engineering, Boston University. ProQuest Number: 27963166, Also available from <https://open.bu.edu/handle/2144/40943>.
- Guo, A., Geng, T., Zhang, Y., Haghi, P., Wu, C., Tan, C., Lin, Y., Li, A., and Herbordt, M. (2022a). A Framework for Neural Network Inference on FPGA-Centric SmartNICs. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 01–08.
- Guo, A., Geng, T., Zhang, Y., Haghi, P., Wu, C., Tan, C., Lin, Y., Li, A., and Herbordt, M. (2022b). FCsN: A FPGA-Centric SmartNIC Framework for Neural Networks. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–2.
- Guo, A., Hao, Y., Wu, C., Haghi, P., Pan, Z., Si, M., Tao, D., Li, A., Herbordt, M., and Geng, T. (2023). Software-Hardware Co-Design of Heterogeneous SmartNIC System for Recommendation Models Inference and Training. In *Proceedings of the 37th International Conference on Supercomputing, ICS ’23*, page 336–347, New York, NY, USA. Association for Computing Machinery.
- Haghi, P., Geng, T., Guo, A., Wang, T., and Herbordt, M. (2020a). FP-AMG: FPGA-Based Acceleration Framework for Algebraic Multigrid Solvers. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 148–156. doi: 10.1109/FCCM48280.2020.00028.
- Haghi, P., Guo, A., Geng, T., Broaddus, J., Schafer, D., Skjellum, A., and Herbordt, M. (2020b). 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)*, pages 159–164. doi: 10.1109/ICFPT51103.2020.00030.
- Haghi, P., Guo, A., Geng, T., Skjellum, A., and Herbordt, M. C. (2021). Workload Imbalance in HPC Applications: Effect on Performance of In-Network Processing. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. doi: 10.1109/HPEC49654.2021.9622847.
- Haghi, P., Guo, A., Xiong, Q., Patel, R., Yang, C., Geng, T., Broaddus, J. T., Marshall, R., Skjellum, A., and Herbordt, M. C. (2020c). FPGAs in the Network and Novel Communicator Support Accelerate MPI Collectives. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–10. doi: 10.1109/HPEC43674.2020.9286200.
- Haghi, P., Guo, A., Xiong, Q., Yang, C., Geng, T., Broaddus, J. T., Marshall, R., Schafer, D., Skjellum, A., and Herbordt, M. C. (2022). Reconfigurable switches for high performance and flexible MPI collectives. *Concurrency and Computation: Practice and Experience*, 34(6):e6769. doi: <https://doi.org/10.1002/cpe.6769>.
- Haghi, P., Kamal, M., Afzali-Kusha, A., and Pedram, M. (2020d). O4-DNN: A Hybrid DSP-LUT-Based Processing Unit With Operation Packing and Out-of-Order Execution for Efficient Realization of Convolutional Neural Networks on FPGA Devices. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(9):3056–3069. doi: 10.1109/TCSI.2020.2986350.
- Haghi, P., Krska, W., Tan, C., Geng, T., Chen, P. H., Greenwood, C., Guo, A., Hines, T., Wu, C., Li, A., Skjellum, A., and Herbordt, M. (2023a). FLASH: FPGA-Accelerated Smart Switches with GCN Case Study. In *Proceedings of the 37th International Conference on Supercomputing, ICS '23*, page 450–462, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3577193.3593739.
- Haghi, P., Marshall, R., Chen, P. H., Skjellum, A., and Herbordt, M. (2023b). A Survey of Potential MPI Complex Collectives: Large-Scale Mining and Analysis of HPC Applications.
- Hamzeh, M., Shrivastava, A., and Vrudhula, S. (2014). Branch-Aware Loop Mapping on CGRAs. In *2014 51st ACM/EDAC/IEEE Design Automation Conference, DAC '14*, page 1–6, San Francisco, CA, USA.
- Handagala, S., Herbordt, M., and Leeser, M. (2021). OCT: The Open Cloud FPGA Testbed. In *31st International Conference on Field Programmable Logic and Applications (FPL)*.
- Handagala, S., Leeser, M., Patle, K., and Zink, M. (2022). Network Attached FPGAs in the Open Cloud Testbed (OCT). In *IEEE INFOCOM*, pages 1–6.

- Hauck, S. and DeHon, A. (2008). *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing*. Morgan Kaufmann.
- Hauser, F., Häberle, M., Merling, D., Lindner, S., Gurevich, V., Zeiger, F., Frank, R., and Menth, M. (2021). A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research. *arXiv preprint arXiv:2101.10632*.
- Herbordt, M. (2019). Advancing OpenCL for FPGAs: Boosting Performance with Intel FPGA SDK for OpenCL Technology. In *The Parallel Universe*, pages 17–32.
- Herbordt, M., Gu, Y., VanCourt, T., Model, J., Sukhwani, B., and Chiu, M. (2008). Computing models for FPGA-based accelerators with case studies in molecular modeling. *Computing in Science and Engineering*, 10(6):35–45. doi: 10.1109/MCSE.2008.143.
- Herbordt, M., VanCourt, T., Gu, Y., Sukhwani, B., Conti, A., Model, J., and DiS-abello, D. (2007a). Achieving high performance with FPGA-based computing. *IEEE Computer*, 40(3):42–49.
- Herbordt, M. C., Model, J., Gu, Y., Sukhwani, B., and VanCourt, T. (2006). Single Pass, BLAST-Like, Approximate String Matching on FPGAs. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 217–226. doi: 10.1109/FCCM.2006.64.
- Herbordt, M. C., VanCourt, T., Gu, Y., Sukhwani, B., Conti, A., Model, J., and DiS-abello, D. (2007b). Achieving High Performance with FPGA-Based Computing. *Computer*, 40(3):50–57. doi: 10.1109/MC.2007.79.
- Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A., and Stanley, K. S. (2005). An Overview of the Trilinos Project. *ACM Transactions on Mathematical Software*, 31(3):397–423.
- Hoefler, T., Di Girolamo, S., Taranov, K., Grant, R. E., and Brightwell, R. (2017). Spin: High-performance streaming processing in the network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17*, New York, NY, USA. Association for Computing Machinery.
- Hoefler, T., Schneider, T., and Lumsdaine, A. (2009). The effect of network noise on large-scale collective communications. *Parallel Processing Letters*, 19:573–593.
- Holmes, D., Mohror, K., Grant, R. E., Skjellum, A., Schulz, M., Bland, W., and Squyres, J. M. (2016). MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale. In *Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI 2016*, page 121–129.

- Holmes, D. J., Skjellum, A., Jaeger, J., Grant, R. E., Bangalore, P. V., Dosanjh, M. G., Bienz, A., and Schafer, D. (2021). Partitioned collective communication. In *2021 Workshop on Exascale MPI (ExaMPI)*, pages 9–17.
- Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. (2020). Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133.
- Intel. FPGA Programmable Acceleration Card D5005. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/intel-fpga-pac-d5005/overview.html [Last accessed: April 29, 2021].
- Inventec. FPGA SmartNIC C5020X. <https://ebg.inventec.com/en/product/Accessories/Smart%20NIC%20Card/Inventec%20FPGA%20SmartNIC%20C5020X> [Last accessed: April 29, 2021].
- Jamieson, P., Sanaullah, A., and Herbordt, M. (2018). Benchmarking Heterogeneous HPC Systems Including Reconfigurable Fabrics: Community Aspirations for Ideal Comparisons. In *IEEE High Performance Extreme Computing Conference*.
- Jasny, M., Thostrup, L., Ziegler, T., and Binnig, C. (2022). P4db - the case for in-network oltp. In *Proceedings of the 2022 International Conference on Management of Data, SIGMOD '22*, page 1375–1389, New York, NY, USA. Association for Computing Machinery.
- Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. (2020). Improving the accuracy, scalability, and performance of graph neural networks with roc. In Dhillon, I., Papailiopoulos, D., and Sze, V., editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org.
- Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. (2017). Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, page 121–136, New York, NY, USA. Association for Computing Machinery.
- Jones, T., Dawson, S., Neely, R., Tuel, W., Brenner, L., Fier, J., Blackmore, R., Caffrey, P., Maskell, B., Tomlinson, P., and Roberts, M. (2003). Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 10–10.
- Jones, T., Ostrouchov, G., Koenig, G. A., Mondragon, O. H., and Bridges, P. G. (2018). An evaluation of the state of time synchronization on leadership class supercomputers. *Concurrency and Computation: Practice and Experience*, 30(4):e4341. e4341 cpe.4341.

- Kamal, H., Mirtaheri, S. M., and Wagner, A. (2010). Scalability of communicators and groups in MPI. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 264–275. ACM.
- Kfoury, E. F., Crichigno, J., and Bou-Harb, E. (2021). An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends. *IEEE Access*, 9:87094–87155.
- Khaleghzadeh, H., Shahid, M. F. A., Manumachu, R., and Lastovetsky, A. (2021). Bi-Objective Optimization of Data-Parallel Applications on Heterogeneous HPC Platforms for Performance and Energy Through Workload Distribution. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):543–560.
- Khawaja, A., Landgraf, J., Prakash, R., Wei, M., Schkufza, E., and Rossbach, C. J. (2018). Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127, Carlsbad, CA. USENIX Association.
- Kim, J., Dally, W. J., Towles, B., and Gupta, A. K. (2005). Microarchitecture of a high-radix router. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture, ISCA '05*, page 420–431, USA. IEEE Computer Society. doi: 10.1109/ISCA.2005.35.
- Klenk, B. and Fröning, H. (2017). An overview of MPI characteristics of exascale proxy applications. In *High Performance Computing: 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017, Proceedings*, page 217–236, Berlin, Heidelberg. Springer-Verlag.
- Knuth, D. (1973). *The Art of Computer Programming: Volume III Sorting and Searching*. Addison-Wesley, Reading, MA.
- Kreutz, D., Ramos, F. M. V., Veríssimo, P. E., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. (2015). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76.
- Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., and Sadayappan, P. (2007). Effective automatic parallelization of stencil computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07*, page 235–244, New York, NY, USA. Association for Computing Machinery.
- Krishnan, V., Serres, O., and Blocksome, M. (2020). Configurable network protocol accelerator (copa): An integrated networking/accelerator hardware/software framework. In *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 17–24.

- Kwasniewski, G., Kabić, M., Besta, M., VandeVondele, J., Solcà, R., and Hoefer, T. (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, SC '19*, New York, NY, USA. Association for Computing Machinery.
- Lattner, C. and Adve, V. (2004). LLVM: a compilation framework for life-long program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. doi: 10.1109/CGO.2004.1281665.
- Li, B., Tan, K., Luo, L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. (2016). Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*.
- Li, Y., Liu, I.-J., Yuan, Y., Chen, D., Schwing, A., and Huang, J. (2019). Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, page 279–291.
- Liu, M., Luo, L., Nelson, J., Ceze, L., Krishnamurthy, A., and Atreya, K. (2017). Incbricks: Toward in-network computation with an in-network cache. *SIGARCH Computer Architecture News*, 45(1):795–809.
- Liu, S., Wang, Q., Zhang, J., Lin, Q., Liu, Y., Xu, M., Chueng, R. C. C., and He, J. (2020). Netreduce: RDMA-compatible in-network reduction for distributed DNN training acceleration. *CoRR*, abs/2009.09736.
- Massing, A., Larson, M., and Logg, A. (2013). Efficient implementation of finite element methods on non-matching and overlapping meshes in 3d. *SIAM Journal on Scientific Computing*, 35:C23–C47.
- Mei, B., Vernalde, S., Verkest, D., De Man, H., and Lauwereins, R. (2003). Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 296–301. doi: 10.1109/DATE.2003.1253623.
- Mellanox. Innova-2 Flex Open Programmable SmartNIC. <https://www.mellanox.com/files/doc-2020/pb-innova-2-flex.pdf> [Last accessed: April 29, 2021].
- Miyajima, T., Ueno, T., Koshiba, A., Huthmann, J., Sano, K., and Sato, M. (2018). High-Performance Custom Computing with FPGA Cluster as an Off-loading Engine. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. https://sc19.supercomputing.org/proceedings/tech-poster/poster_files/rpost174s2-file3.pdf.

- Morgan, B., Holmes, D. J., Skjellum, A., Bangalore, P., and Sridharan, S. (2017). Planning for Performance: Persistent Collective Operations for MPI. In *Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI '17*, New York, NY, USA. Association for Computing Machinery.
- Morgan, T. (2021). Intel's best DPU will be commercially available – someday. *The Next Platform*, August 31, 2021.
- Munafo, R. (2018). Cooperative High-Performance Computing with FPGAs: Matrix Multiply Case Study. Master's thesis, Department of Electrical and Computer Engineering, Boston University. <https://open.bu.edu/handle/2144/30740>.
- Naous, J., Gibb, G., Bolouki, S., and McKeown, N. (2008). NetFPGA: Reusable Router Architecture for Experimental Research. In *Association for Computing Machinery PRESTO*, page 1–7, New York, NY, USA.
- Napatech. FPGA acceleration cards. <https://www.napatech.com/products/> [Last accessed: April 29, 2021].
- New Wave DV (2023). 32-Port Programmable Switch. <https://newwavedv.com/products/appliances/32-port-programmable-switch/>.
- Nikitenko, D., Wolf, F., Mohr, B., Hoeffler, T., Stefanov, K., Voevodin, V., Antonov, A., and Calotoiu, A. (2021). Influence of Noisy Environments on Behavior of HPC Applications. *Lobachevskii Journal of Mathematics*, 42:1560–1570.
- Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The PageRank citation ranking: Bringing order to the web. TR 1999-66, Stanford InfoLab.
- Park, J., Smelyanskiy, M., Yang, U. M., Mudigere, D., and Dubey, P. (2015a). High-performance algebraic multigrid solver optimized for multi-core based distributed parallel systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15*, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/2807591.2807603.
- Park, J., Smelyanskiy, M., Yang, U. M., Mudigere, D., and Dubey, P. (2015b). High-performance algebraic multigrid solver optimized for multi-core based distributed parallel systems. *SC*, 15-20-Nov:1–12.
- Patel, R., Haghi, P., Jain, S., Kot, A., Krishnan, V., Varia, M., and Herbord, M. (2022a). Distributed hardware accelerated secure joint computation on the COPA framework. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. doi: 10.1109/HPEC55821.2022.9926388.

- Patel, R., Haghi, P., Jain, S., Kot, A., Krishnan, V., Varia, M., and Herbordt, M. (2022b). Copa use case: Distributed secure joint computation. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–2.
- Patel, R., Wolfe, P.-F., Munafo, R., Varia, M., and Herbordt, M. (2020). Arithmetic and Boolean Secret Sharing MPC on FPGAs in the Data Center. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. doi: 10.1109/HPEC43674.2020.9286159.
- Peng, H., Chen, S., Wang, Z., Yang, J., Weitze, S. A., Geng, T., Li, A., Bi, J., Song, M., Jiang, W., Liu, H., and Ding, C. (2021). Optimizing FPGA-based Accelerator Design for Large-Scale Molecular Similarity Search (Special Session Paper). In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–7. doi: 10.1109/ICCAD51958.2021.9643528.
- Peng, H., Huang, S., Chen, S., Li, B., Geng, T., Li, A., Jiang, W., Wen, W., Bi, J., Liu, H., and Ding, C. (2022). A Length Adaptive Algorithm-Hardware Co-Design of Transformer on FPGA through Sparse Attention and Dynamic Pipelining. In *Proceedings of the 59th ACM/IEEE Design Automation Conference, DAC '22*, page 1135–1140, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3489517.3530585.
- Peng, Y., Saldana, M., and Chow, P. (2011a). Hardware support for broadcast and reduce in mpsoc. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 144–150. doi: 10.1109/FPL.2011.34.
- Peng, Y., Saldana, M., and Chow, P. (2011b). Hardware support for broadcast and reduce in MPSOC. In *International Conference on Field Programmable Logic and Applications*, pages 144–150.
- Perry, T. (2019). Does the Repurposing of Sun Microsystems’ Slogan Honor History, or Step on It? *IEEE Spectrum*, (30 July).
- Petrini, F., Kerbyson, D., and Pakin, S. (2003). The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asc q. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 55–55.
- Pjesivac-Grbovic, J., Angskun, T., Bosilca, G., Fagg, G. E., Gabriel, E., and Dongarra, J. J. (2005). Performance analysis of MPI collective operations. In *19th IEEE International Parallel and Distributed Processing Symposium*.
- Plessl, C. (2018). Bringing FPGAs to HPC Production Systems and Codes. In *H2RC'18 workshop at Supercomputing (SC'18)*. doi: 10.13140/RG.2.2.34327.42407.

- Putnam, A. (2014). A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *International Symposium on Computer Architecture*, pages 13–24. doi: 10.1109/ISCA.2014.6853195.
- Qiao, S., Hu, C., Brebner, G., Zou, J., and Guan, X. (2020). Adaptable switch: A heterogeneous switch architecture for network-centric computing. *IEEE Communications Magazine*, 58(12):64–69.
- Rafenetti, K., Oden, L., Archer, C., Bland, W., Blocksome, M., Si, M., Cofman, P., Jose, J., Sannikov, A., Chuvelev, M., Fischer, P., Otten, M., and Min, M. (2017). Why Is MPI So Slow ? Analyzing the Fundamental Limits in Implementing MPI-3 . 1. *Sc*.
- Rios, A. L. G., Bekshentayeva, K., Singh, M., Haeri, S., and Trajkovic, L. (2021). Virtual Network Embedding for Switch-Centric Data Center Networks. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5.
- RISC-V (2023). RISC-V Specifications. <https://riscv.org/technical/specifications/>.
- Saldaña, M., Patel, A., Madill, C., Nunes, D., Wang, D., Chow, P., Wittig, R., Styles, H., and Putnam, A. (2010). MPI as a programming model for high-performance reconfigurable computers. *ACM Transactions on Reconfigurable Technology and Systems (TRETTS)*, 3(4):22.
- Sanaullah, A., C.Yang, Alexeev, Y., Yoshii, K., and Herbordt, M. (2018a). Application Aware Tuning of Reconfigurable Multi-Layer Perceptron Architectures. In *IEEE High Performance Extreme Computing Conference*.
- Sanaullah, A. and Herbordt, M. (2018a). An Empirically Guided Optimization Framework for FPGA OpenCL. In *2018 International Conference on Field Programmable Technology (FPT)*, pages 46–53. doi: 10.1109/FPT.2018.00018.
- Sanaullah, A. and Herbordt, M. (2018b). FPGA HPC using OpenCL: Case Study in 3D FFT. In *9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, page 1–6. doi: 10.1145/3241793.3241800.
- Sanaullah, A. and Herbordt, M. (2018c). Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolflow. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. doi: 10.1109/HPEC.2018.8547646.
- Sanaullah, A., Sachdeva, V., and Herbordt, M. (2018b). SimBSP: Enabling RTL Simulation for Intel FPGA OpenCL Kernels. In *IEEE 4th Annual Heterogeneous High Performance Reconfigurable Computing*. doi: 10.1186/s12859-018-2505-7.

- Sanaullah, A., Yang, C., Alexeev, Y., Yoshii, K., and Herbordt, M. (2018c). Real-Time Data Analysis for Medical Diagnosis using FPGA Accelerated Neural Networks. *BMC Bioinformatics*, 19 Supplement 18. doi: 10.1186/s12859-018-2505-7.
- Sankaran, G., Chung, J., and Kettimuthu, R. (2021). Leveraging In-Network Computing and Programmable Switches for Streaming Analysis of Scientific Data. In *IEEE NetSoft*, pages 293–297.
- Sapio, A., Canini, M., Ho, C.-Y., Nelson, J., Kalnis, P., Kim, C., Krishnamurthy, A., Moshref, M., Ports, D., and Richtarik, P. (2021). Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association.
- Schonbein, W., Grant, R. E., Dosanjh, M. G. F., and Arnold, D. (2019). Inca: In-network compute assistance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19*, New York, NY, USA. Association for Computing Machinery.
- Sensi, D. D., Girolamo, S. D., Ashkboos, S., Li, S., and Hoeffler, T. (2021). Flare: Flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21*.
- Shahzad, H., Sanaullah, A., Arora, S., Munafo, R., Yao, X., Drepper, U., and Herbordt, M. (2022). Reinforcement Learning Strategies for Compiler Optimization in High Level Synthesis. In *The Eighth Workshop on the LLVM Compiler Infrastructure in HPC*. DOI: 10.1109/LLVM-HPC56686.2022.00007.
- Shahzad, H., Sanaullah, A., and Herbordt, M. (2021). Survey and Future Trends for FPGA Cloud Architectures. In *IEEE High Performance Extreme Computing Conference*. DOI: 10.1109/HPEC49654.2021.9622807.
- Sheng, J., Xiong, Q., Yang, C., and Herbordt, M. (2016a). Collective Communication on FPGA Clusters with Static Scheduling. *ACM SIGARCH Computer Architecture News*, 44(4).
- Sheng, J., Xiong, Q., Yang, C., and Herbordt, M. C. (2017a). Collective Communication on FPGA Clusters with Static Scheduling. *SIGARCH Computer Architecture News*, 44(4):2–7. doi: 10.1145/3039902.3039904.
- Sheng, J., Yang, C., Caulfield, A., Papamichael, M., and Herbordt, M. (2017b). HPC on FPGA Clouds: 3D FFTs and Implications for Molecular Dynamics. In *27th International Conference on Field Programmable Logic and Applications*. doi: 10.23919/FPL.2017.8056853.

- Sheng, J., Yang, C., and Herbordt, M. (2015). Towards Low-Latency Communication on FPGA Clusters with 3D FFT Case Study. In *International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*. <https://pdfs.semanticscholar.org/832d/c69145f5ba0ed6a951583201b1b20dd2096e.pdf>.
- Sheng, J., Yang, C., and Herbordt, M. (2016b). Application-Aware Collective Communication on FPGA Clusters. In *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. doi: 10.1109/FCCM.2016.55.
- Sheng, J., Yang, C., and Herbordt, M. (2018). High Performance Dynamic Communication on Reconfigurable Clusters. In *28th International Conference on Field Programmable Logic and Applications*. doi: 10.1109/FPL.2018.00044.
- Shi, M., Tang, Y., Zhu, X., Wilson, D., and Liu, J. (2021). Multi-class imbalanced graph convolutional network learning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI'20*.
- Shi, R., Dong, P., Geng, T., Ding, Y., Ma, X., So, H. K.-H., Herbordt, M., Li, A., and Wang, Y. (2020). CSB-RNN: A Faster-than-Realtime RNN Acceleration Framework with Compressed Structured Blocks. In *Proceedings of the 34th ACM International Conference on Supercomputing, ICS '20*, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3392717.3392749.
- Silicom. FPGA SmartNIC N5010 Series. https://www.silicom-usa.com/pr/fpga-based-cards/fpga-intel-based/fpga-intel-stratix-based/silicom-fpga-smartnic-n5010_series/ [Last accessed: April 29, 2021].
- Skjellum, A., Rüfenacht, M., Sultana, N., Schafer, D., Laguna, I., and Mohror, K. (2020). ExaMPI: A Modern Design and Implementation to Accelerate Message Passing Interface Innovation. *Communications in Computer and Information Science*, 1087 CCIS:153–169.
- Stanzione, D., Barth, B., Gaffney, N., Gaither, K., Hempel, C., Minyard, T., Mehringer, S., Wernert, E., Tufo, H., Panda, D., and Teller, P. (2017). Stampede 2: The Evolution of an XSEDE Supercomputer. In *Practice and Experience in Advanced Research Computing on Sustainability, Success and Impact*.
- Stern, J., Xiong, Q., Sheng, J., Skjellum, A., and Herbordt, M. (2017). Accelerating MPI.Reduce with FPGAs in the Network. In *Workshop on Exascale MPI*. <https://www.bu.edu/caadlab/exampi17.pdf>.
- Stewart, L., Pascoe, C., Davis, E., Sherman, B., Herbordt, M., and Sachdeva, V. (2021). Particle Mesh Ewald for Molecular Dynamics in OpenCL on an FPGA Cluster. In *IEEE Symposium on Field Programmable Custom Computing Machines*.

- Stunkel, C. B., Graham, R. L., Shainer, G., Kagan, M., Sharkawi, S. S., Rosenburg, B., and Chochia, G. A. (2020). The high-speed networks of the summit and sierra supercomputers. *IBM Journal of Research and Development*, 64(3/4):3:1–3:10.
- Sukhwani, B. and Herbordt, M. (2008). Acceleration of a Production Rigid Molecule Docking Code. In *2008 International Conference on Field Programmable Logic and Applications*, pages 341–346. doi: 10.1109/ FPL.2008.4629955.
- Sukhwani, B. and Herbordt, M. (2009). GPU acceleration of a production molecular docking code. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*.
- Swamy, T., Rucker, A., Shahbaz, M., Gaur, I., and Olukotun, K. (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*, ASPLOS 2022, page 1099–1114.
- Taras, I. and Anderson, J. H. (2019). Impact of FPGA Architecture on Area and Performance of CGRA Overlays. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 87–95.
- Thakur, R., Rabenseifner, R., and Gropp, W. (2005). Optimization of Collective Communication Operations in MPICH. *Int. J. of High Performance Computing Applications*, 19(1):49–66.
- Tripathy, A., Yelick, K., and Buluç, A. (2020). Reducing Communication in Graph Neural Network Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*, pages 1–14.
- Van De Geijn, R. A. and Watts, J. (1997). SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274.
- VanCourt, T., Gu, Y., and Herbordt, M. (2004). FPGA acceleration of rigid molecule interactions. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 300–301. doi: 10.1109/ FCCM.2004.33.
- VanCourt, T. and Herbordt, M. (2004). Families of FPGA-based algorithms for approximate string matching. In *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004.*, pages 354–364. doi: 10.1109/ ASAP.2004.1342484.
- VanCourt, T. and Herbordt, M. (2005a). LAMP: A tool suite for families of FPGA-based application accelerators. In *International Conference on Field Programmable Logic and Applications, 2005*. doi: 10.1109/ FPL.2005.1515797.

- VanCourt, T. and Herbordt, M. (2005b). Three dimensional template correlation: Object recognition in 3D voxel data. In *Seventh International Workshop on Computer Architecture for Machine Perception (CAMP'05)*, pages 153–158. doi: 10.1109/ CAMP.2005.52.
- VanCourt, T. and Herbordt, M. (2006a). Application-dependent memory interleaving enables high performance in FPGA-based grid computations. In *IEEE Conference on Field Programmable Logic and Applications*, pages 395–401. doi: 10.1109/ FCCM.2006.25.
- VanCourt, T. and Herbordt, M. (2006b). Rigid molecule docking: FPGA reconfiguration for alternative force laws. *Journal on Applied Signal Processing*, v2006:1–10. doi: 10.1155/ ASP/2006/97950.
- VanCourt, T. and Herbordt, M. (2006c). Sizing of processing arrays for FPGA-based computation. In *2006 International Conference on Field Programmable Logic and Applications*, pages 755–760. doi: 10.1109/ FPL.2006.311307.
- VanCourt, T. and Herbordt, M. (2007). Families of FPGA-based accelerators for approximate string matching. *Microprocessors and Microsystems*, 31(2):135–145. doi: 10.1016/ j.micpro.2006.04.001.
- VanCourt, T. and Herbordt, M. (2009). Elements of high performance reconfigurable computing. In Zelkowitz, M., editor, *Advances in Computers*, volume v75, pages 113–157. Elsevier. doi: 10.1016/ S0065-2458(08)00802-4.
- Walshaw, C. and Cross, M. (1999). Dynamic mesh partitioning and load-balancing for parallel computational mechanics codes. In *Parallel and Distributed Processing for Computational Mechanics*, pages 79–94.
- Wang, T., Geng, T., Jin, X., and Herbordt, M. (2019a). Accelerating AP3M-Based Computational Astrophysics Simulations with Reconfigurable Clusters. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 181–184. doi: 10.1109/ ASAP.2019.000-5.
- Wang, T., Geng, T., Jin, X., and Herbordt, M. (2019b). FP-AMR: A Reconfigurable Fabric Framework for Block-Structured Adaptive Mesh Refinement Applications. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 245–253. doi: 10.1109/ FCCM.2019.00040.
- Wang, T., Geng, T., Li, A., Jin, X., and Herbordt, M. (2020). FPDeep: Scalable Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters. *IEEE Transactions on Computers*, 69(8):1143–1158.

- Wang, Z., Paul, J., He, B., and Zhang, W. (2017). Multikernel Data Partitioning With Channel on OpenCL-Based FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–13.
- Waterman, A. and Asanovic, K. (2017). The RISC-V Instruction Set Manual Volume I: User-Level ISA, Document Version 2.2. <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>.
- Weng, J., Liu, S., Wang, Z., Dadu, V., and Nowatzki, T. (2020). A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 703–716.
- Widener, P., Ferreira, K. B., Levy, S., and Hoefler, T. (2014). Exploring the effect of noise on the performance benefit of nonblocking allreduce. In *Proceedings of the 21st European MPI Users' Group Meeting, EuroMPI/ASIA '14*, page 77–82, New York, NY, USA. Association for Computing Machinery.
- Wolfe, P.-F., Patel, R., Munafo, R., Varia, M., and Herbordt, M. (2020). Secret Sharing MPC on FPGAs in the Datacenter. In *IEEE Conference on Field Programmable Logic and Applications*.
- Wu, C., Bandara, S., Geng, T., Guo, A., Haghi, P., Sachdeva, V., Sherman, W., and Herbordt, M. (2022). Optimized Mappings for Symmetric Range-Limited Molecular Force Calculations on FPGAs. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 101–108. doi: 10.1109/FPL57034.2022.00026.
- Wu, C., Bandara, S., Geng, T., Sachdeva, V., Sherman, B., and Herbordt, M. (2021). System-Level Modeling of GPU/FPGA Clusters for Molecular Dynamics Simulations. In *IEEE High Performance Extreme Computing Conference*. doi: 10.1109/HPEC49654.2021.9622838.
- Wu, C., Geng, T., Guo, A., Bandara, S., Haghi, P., Liu, C., Li, A., and Herbordt, M. (2023). FASDA: An FPGA-Aided, Scalable and Distributed Accelerator for Range-Limited Molecular Dynamics. In *International Conference for High Performance Computing, Networking, Storage and Analysis*.
- Wu, C., Geng, T., Sachdeva, V., Sherman, W., and Herbordt, M. (2020). A Communication-Efficient Multi-Chip Design for Range-Limited Molecular Dynamics. In *2020 IEEE High Performance extreme Computing Conference (HPEC)*.
- Xilinx. Alveo SmartNIC Accelerator Card. <https://www.xilinx.com/products/boards-and-kits/alveo.html> [Last accessed: April 29, 2021].

- Xilinx. Alveo SN1000 Accelerator Card. <https://www.xilinx.com/applications/data-center/network-acceleration/alveo-sn1000.html> [Last accessed: April 29, 2021].
- Xilinx (2023). XUP Vitis Network Example (VNx). https://github.com/Xilinx/xup_vitis_network_example.
- Xiong, Q., Ates, E., Herbordt, M., and Coskun, A. (2018). Tangram: Colocating HPC Applications with Oversubscription. In *IEEE High Performance Extreme Computing Conference*.
- Xiong, Q., Yang, C., Haghi, P., Skjellum, A., and Herbordt, M. (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)*, pages 215–215.
- Xiong, Q., Yang, C., Patel, R., Geng, T., Skjellum, A., and Herbordt, M. (2019). GhostSZ: A Transparent SZ Lossy Compression Framework with FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 258–266. doi: 10.1109/FCCM.2019.00042.
- Yamazaki, I., Hoemmen, M., Luszczek, P., and Dongarra, J. (2017). Improving Performance of GMRES by Reducing Communication and Pipelining Global Collectives. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1118–1127. doi: 10.1109/IPDPSW.2017.65.
- Yang, C. (2019). *High-Performance Communication Infrastructure Design on FPGA-Centric Clusters*. PhD thesis, Department of Electrical and Computer Engineering, Boston University. <https://open.bu.edu/handle/2144/38207>.
- Yang, C., Geng, T., Wang, T., Patel, R., Xiong, Q., Sanaullah, A., Lin, C., Sachdeva, V., Sherman, W., and Herbordt, M. (2019a). Fully Integrated FPGA Molecular Dynamics Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–31. doi: 10.1145/3295500.3356179.
- Yang, C., Geng, T., Wang, T., Sheng, J., Lin, C., Sachdeva, V., Sherman, W., and Herbordt, M. (2019b). Molecular Dynamics Range-Limited Force Evaluation Optimized for FPGA. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 263–271. doi: 10.1109/ASAP.2019.00016.
- Yang, C., Sheng, J., Patel, R., Sanaullah, A., Sachdeva, V., and Herbordt, M. (2017). OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. doi: 10.1109/HPEC.2017.8091078.

- Yoshida, H. (2020). How is Data Ops Related to Data Centric Computing? Hitachi Blog, <https://community.hitachivantara.com/blogs/hubert-yoshida/2020/10/14/how-is-data-ops-related-to-data-centric-computing>.
- Zhang, B., Kannan, R., and Prasanna, V. (2021). BoostGCN: A Framework for Optimizing GCN Inference on FPGA. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 29–39.
- Zhao, X., An, A., Liu, J., and Chen, B. X. (2019). Dynamic stale synchronous parallel distributed training for deep learning. *CoRR*, abs/1908.11848.

CURRICULUM VITAE

