FLASH: FPGA-Accelerated Smart Switches with GCN Case Study

Pouya Haghi
Boston University
haghi@bu.edu

William Krska
Boston University
wkrska@bu.edu

Cheng Tan
Microsoft
chengtan@microsoft.com

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

Po Hao Chen
Boston University
bupochen@bu.edu

Connor Greenwood
Boston University
cgre@bu.edu

Anqi Guo
Boston University
anqiguo@bu.edu

Thomas Hines
University of Tennessee at Chattanooga
thomas-hines01@utc.edu

Chunshu Wu
Boston University
happycwu@bu.edu

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

Anthony Skjellum
University of Tennessee at Chattanooga
tony-skjellum@utc.edu

Martin Herbordt
Boston University
herbordt@bu.edu

ABSTRACT

Some communication switches, e.g., the Mellanox SHArP and those in the IBM BlueGene clusters, are augmented to process packets at the application level with fixed-function collectives. This approach, however, lacks flexibility, which limits their applicability in diverse and dynamic workloads. Recently, a new type of programmable packet processor, which uses high-level languages, *e.g.*, P4, has emerged as a possible candidate. P4-based switches, however, fall short in certain applications, including machine learning, where capabilities not currently supported by P4 are needed. These include more complex calculation, such as sparse computation and fused multiply-accumulate, data-intensive floating point operations, data reuse, and significant memory. The problem addressed here is that such a switch augmentation needs to support: a large amount of state, significant flexible compute capability, and ease of programming, all while maintaining full functionality, including ensuring high throughput, and demonstrating utility.

In this work, we propose a programmable look-aside-type accelerator that can be embedded into, or attached to, existing communication switch pipelines and that is capable of processing packets at line rate. The proposed in-switch accelerator is based on mixing an ISA (subset of RISC-V instructions) with dataflow graphs (found in CGRAs). To augment performance, vector instructions are also supported. To facilitate usability, we have developed a complete toolchain to compile user-provided C/C++ codes to appropriate back-end instructions for configuring the accelerator. While this approach is flexible enough to support various workloads, in this paper, we consider Graph Convolutional Networks (GCNs) as a case study. Experimental results show that this approach considerably improves the performance of distributed GCN applications.

ICS '23, June 21–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0056-9/23/06…$15.00
https://doi.org/10.1145/3577193.3593739

CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable computing**; **Neural networks**.

KEYWORDS

In-Switch Computing, FPGAs, High Performance Computing

ACM Reference Format:
Pouya Haghi, William Krska, Cheng Tan, Tong Geng, Po Hao Chen, Connor Greenwood, Anqi Guo, Thomas Hines, Chunshu Wu, Ang Li, Anthony Skjellum, and Martin Herbordt. 2023. FLASH: **F**PGA-Acce**l**erated Sm**a**rt **S**witc**h**es with GCN Case Study. In *2023 International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, Florida, FL, USA, 13 pages. https://doi.org/10.1145/3577193.3593739

1 INTRODUCTION

A growing trend in HPC is the importance of the network in application support. For example, offload of collective processing into SmartNICs [1, 4, 12, 15, 16, 42] is well-established and has several 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 and enables better communication-computation overlap; and fourth, some network-host communication is removed as the NIC handles additional send/receive operations. But while SmartNICs are invaluable, this scheme still forces processing into endpoints. Another approach is to offload collective processing into switches

[7, 13]. This has a number of additional benefits: latency and bandwidth are improved as computation can be both distributed and centralized, rather than performed in a single source or endpoint; switches support far more communication traffic than NICs so the potential benefit is proportionally increased; and communication volume may be drastically reduced as messages are quickly merged (reduction) or slowly replicated (broadcast).

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., [8, 13]). While academic work [19, 45] demonstrates support for user-defined extensions, the resulting switches are still confined to inline (aka streaming [27]) 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 [27]; 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 [5]. P4 is a high-level language used to control packet forwarding. While there has been some exploration of application-level processing [40] in P4 switches, including support of collectives [41], these capabilities are currently limited and likely to fall short in certain applications, including machine learning [47]. 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 [7]; 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-switch LA support has certain requirements. First, to support line rate communication, LA 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 [47]; by FPGA-augmented switches, e.g., from Arista [2]; or by using the FPGA itself as a switch (e.g., New Wave [33]). Because of their accessibility, we concentrate on the latter, but also consider FPGA-augmented switches.

To support different workloads and enable software-like programmability, while achieving near-ASIC performance, we use a coarse-grained reconfigurable array (CGRA) overlay architecture [48]. The proposed dataflow architecture is itself RISC-V compatible with a subset of scalar and vector instructions [39, 51]. 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 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, in this paper 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 (§3);
- A software toolchain to compile user-provided packet handlers to the instructions for configuring the accelerator at software speed (rather than HLS §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× (across five real-world datasets) on 24 nodes (§5).

The organization of this paper is as follows. §2 gives preliminaries and motivation. §3 describes the switch look-aside hardware accelerator. §4 presents the software framework. §5 evaluates FLASH. Related work is discussed in §6.

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.

2.1 Graph Convolutional Network (GCN)

We provide GCN background and elaborate on distributed GCNs.

2.1.1 GCN Background. Equation 1 shows the layer-wise forward propagation in a multi-layer GCN.

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

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 [10, 11] 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 [10]. 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 [49], which leads to worse scalability. Second, computation is coupled with communication in distributed SpMMs: the same data are used for both computation and communication.

2.1.2 Distributed GCNs. GCNs typically operate on large and irregular input graphs with small models, i.e., with few layers. This is in contrast to Deep Neural Networks (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 [24]. 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 [24]. 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 [3, 9]. 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 1 (a), which depicts results from a CPU cluster (with Skylake processor) and that cluster enhanced with FLASH. Each node runs 24 processes; §5.1 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 (§2.1.3).

Challenge 2: Graphs are extremely sparse [56] 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 (§3.1).

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 [49]. 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 [34]. Figure 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 [49]. The Allgather is an all-to-all type communication with a large message size which can be a bottleneck, especially for large-scale datasets (§5.3 and §5.4). Instead of a large bursty communication followed by the computation, we use a pipelined fine-grained communication that overlaps local computation.

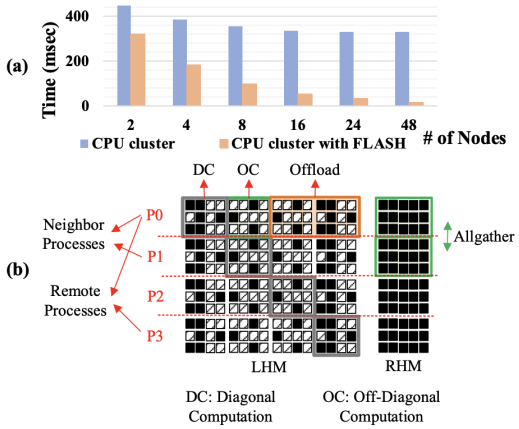

Figure 1: (a) GCN inference execution time on a CPU cluster (Skylake processors) without and with FLASH accelerators. (b) FLASH matrix partitioning for an SpMM kernel with tasks shown for process 0 (P0). Gray tiles in LHM belong to the diagonal part and the rest are off-diagonal.

2.1.3 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 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 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. However, in this work, we assume these processes reside on the same node.

2.2 Motivation

Figure 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 2 summarizes some of these benefits for a large-scale dataset (ogbn-products [23]) over a baseline without in-switch acceleration. Again 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.

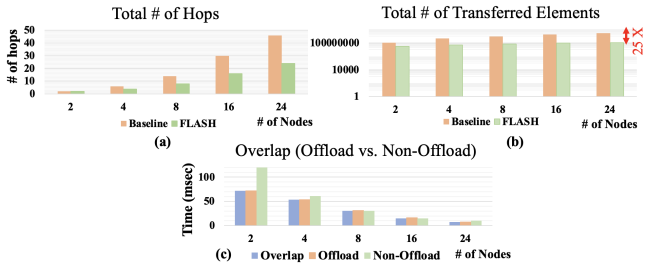

Figure 2: FLASH results for ogbn-products dataset: (a) number of transferred elements, (b) number of hops, and (c) overlap.

There are two types of communication switches: fixed function [13] and programmable switches [5]. 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).

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 [22], 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 [50].

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 [26]. 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 [41] (§3.2).
- P4 has limited set of datatypes and operations [7, 36]. 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 §3.3.
- 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 [7]. By tiling and reusing data it is also possible to decrease off-chip memory latency overhead (§3.3).
- P4 does not support loops [36], which is required to express LA behavior, including ML applications (§4.3).

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

2.3 FLASH Models

We consider two approaches for integrating LA acceleration: FLASH-integrated-switch (FiS) and FLASH-attached-switch (FaS). In FiS, LA is integrated into existing switch pipelines that implement a full switch functionality, including packet forwarding. In FaS, an FPGA device is attached to an existing switch: packets are directed to the FPGA for further processing and are redirected back to the switch for switching. In both cases, there is additional functionality, which we refer to as the *host*, to configure (through instructions) and initialize the accelerator.

3 HARDWARE DESIGN

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 (§3.2).
- We extend the ISA with sparse vector instructions, specializing it for GCN inference acceleration (§3.3).
- An important requirement of switch accelerator architectures is that it should not compromise the line rate. Thus, we apply several optimizations (§3.4).
- We address a number of questions for integrating the proposed accelerator into existing switches (§3.5).

3.1 Overview

Dataflow: FLASH is ideal for the execution of (nested) loop-intensive applications due to optimizations (§3.4) and massive parallelism of the LA architecture. LA consists of several VPEs pipelined together to process incoming packets. Figure 3 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 [52] 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 §3.5.

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 3). 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 (§2.3) from off-chip memory (HBM in this work), 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 (see §3.2). **Assembler** re-assembles the packet and calculates the checksum.

3.2 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 back-pressure logic implemented in case 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.

3.3 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 §4.2). In the context of GCN acceleration, the first (second) instruction resembles a weight (output) stationary approach [6]. We note that utilizing the second instruction

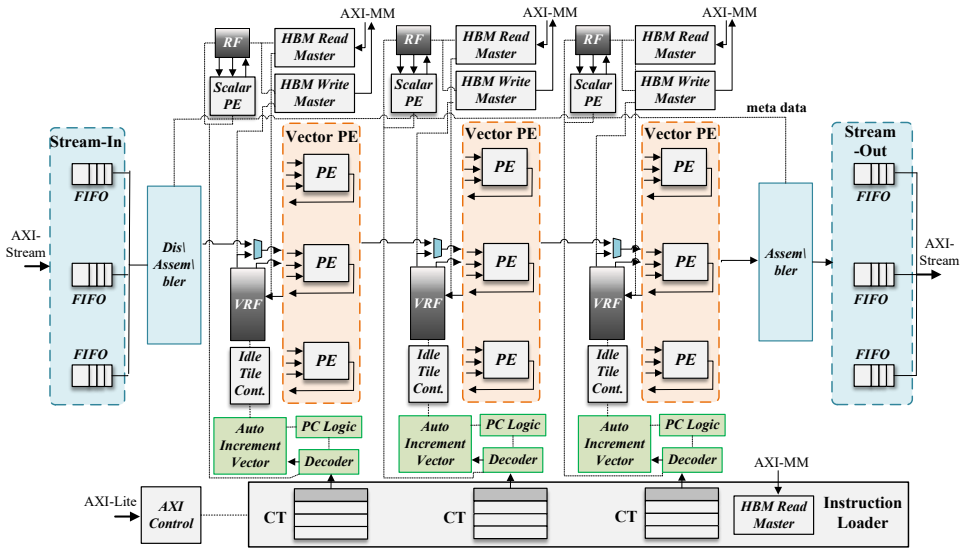

Figure 3: The proposed switch accelerator with three pipelined vector PEs. It is packaged with an AXI interface to facilitate integration with switch pipelines.

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 (discussed in §3.4). 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 §5.2), but the LHM is tiled both vertically and horizontally.

3.4 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 §5.4, 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*) [39]. 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) as there are 32 HBM banks and one bank is used for storing instructions.

3.5 Integration with Existing Switches

Methods: For FaS, 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 FiS, 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 [32]. There are two methods of acceleration: off-the-pipeline [30] and in-pipeline [13]. 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 FaS and FiS. We use a Xilinx-compliant AXI interface to package the accelerator. Figure 4 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, 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.

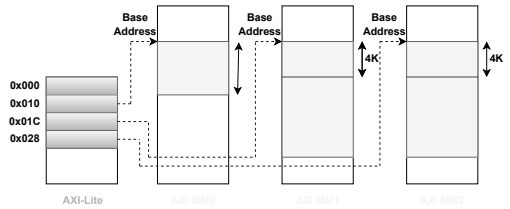

Figure 4: Memory map for AXI interfaces with three AXI MMs; the first is used for application instruction and two others for application data.

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 [37]. 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 SOFTWARE DESIGN

4.1 Compiler

To support a packet handler that is described with a high-level language we use a compiler, based on LLVM [28], to generate back-end RISC-V instructions. RISC-V is adopted in this work 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 5 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 [53] to program and interact with the FPGA.

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 [39]. Others are proposed here based on RISC-V specifications [38]. Sparse accumulation is discussed in §3.3. Fused MAC is similar to that used in [39] 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.

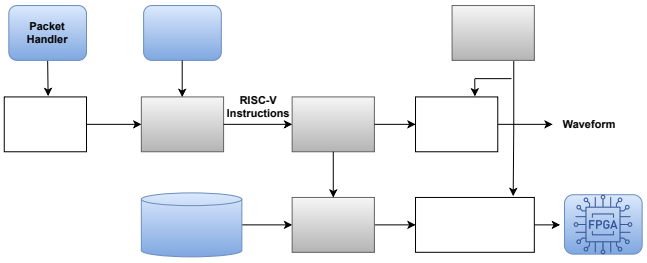

Figure 5: Framework to map a machine learning model with a user-provide packet handler to the reconfigurable switch accelerator. Gray components are proposed in this work. White components are powered by existing tools.

For scalar instructions, we support `lui`, `addi`, `add`, and `bne` all with the same specifications as [38]. 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 [39]. The two currently supported CSRs in this work 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.

4.3 Program Flow

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.

5 EXPERIMENTAL RESULTS

5.1 Experimental Setup

For the CPU reference, benchmarks were run on the TACC Stampede2 [44] 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 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 (§2.1.2).

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 (FiS model) or the capability of attaching FLASH to TACC switches (FaS 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 FiS mode, or kernel execution time at the accelerator's host CPU, in FaS mode.

We now describe the emulation. We use the same method as [30]. 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 [13, 17].

As mentioned, for the FiS 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 FaS model, the testbed is a two-node system on CloudLab [20, 21] 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 [54] 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 FiS and FaS 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 FiS-OS, FiS-WS, FaS-OS, and FaS-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 [30]). Below we show that this limit is not because of the FLASH's resource requirements (§5.7); rather, we found that this gives the best FLASH scalability in GCN applications with datasets of interest.

Table 1: 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

Table 2: 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

5.2 Datasets and Configuration Parameters

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 1 provides the details). The hidden dimension is 16. Adjacency matrices are evenly distributed among nodes (row-wise).

Configuration parameters: Table 3 summarizes the tiling and configurations (unroll factor, CGRA dimension, and VLEN for `spvacc.vx`/`spvacc.xv`) for the datasets in both dataflow modes. We note that the Tm reported in this table is per rank; to get the actual Tm for the packet handler one should multiply the numbers in this table by the number of ranks. As pointed out in §3.3, having a larger unroll factor and VLEN is preferred to improve performance (the maximum value for unroll factor and VLEN is 32 and $\frac{\#vector-registers}{2}$). Hence we reduce Tm with scaling if possible. Otherwise, VLEN is decreased with scaling.

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 2 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 3: Configuration parameters for different datasets and dataflow modes (OS: output stationary, WS: weight stationary)

Dataset	Mode	Tk	SIMD Lanes	Vector PE Pipelines	Tm						VLEN						Unroll Factor					
					Number of Nodes						Number of Nodes						Number of Nodes					
					2	4	8	16	24	48	2	4	8	16	24	48	2	4	8	16	24	48
PPI	OS	1	16	31	8	4	2	1	1	-	1767	1767	1767	1767	1767	-	7	7	7	7	4	-
	WS	53	16	31	1	1	1	1	1	-	56	28	14	7	4	-	32	32	32	32	32	-
Citeseer	OS	3	16	16	8	4	2	1	1	-	1109	1109	1109	1109	1109	-	13	13	13	13	7	-
	WS	104	16	16	1	1	1	1	1	-	104	52	26	13	7	-	32	32	32	32	32	-
Pubmed	OS	10	16	16	20	10	5	3	2	-	2048	2048	2048	2048	2048	-	32	32	32	32	32	-
	WS	617	16	16	1	1	1	1	1	-	617	309	155	78	39	-	32	32	32	32	32	-
Ogbn-products	OS	1196	16	31	100	50	25	13	7	4	2048	2048	2048	2048	2048	2048	32	32	32	32	32	32
	WS	76533	16	31	13	7	4	2	1	1	256	256	256	256	256	128	32	32	32	32	32	32
Ogbn-mag	OS	360	16	31	30	15	8	4	2	1	2048	2048	2048	2048	2048	2048	32	32	32	32	32	32
	WS	23013	16	31	4	2	1	1	1	1	256	256	239	120	60	30	32	32	32	32	32	32

5.3 Communication performance

Figure 6 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 §3.3. 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 (§3.1). We therefore consider only OS for the rest of this subsection. As discussed in §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.

5.4 Application scalability

Figure 7 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 1). 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×, 2×, 1.1×, 1.4×, and 10.1× for PPI, Citeseer, Pubmed, ogbn-mag, and ogbn-products on 24 nodes with an average of 3.4× 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.

5.5 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 8 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 Tm for OS) for small datasets. Finally, throughput values for WS mode are higher than for OS mode. This is because the number of times that `spvacc.vx` is called is larger (due to a larger Tk).

5.6 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. 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, 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

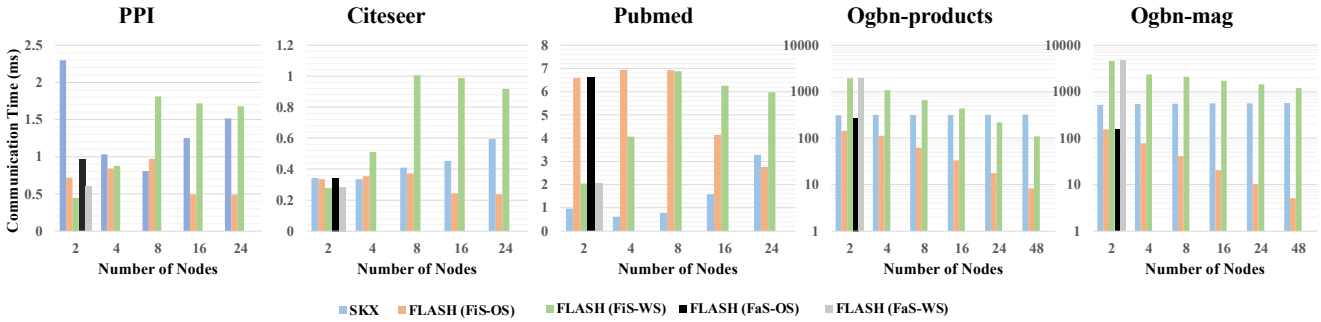

Figure 6: Communication performance and scalability comparison of GCN for a baseline CPU cluster (SKX) vs. FLASH with different configurations.

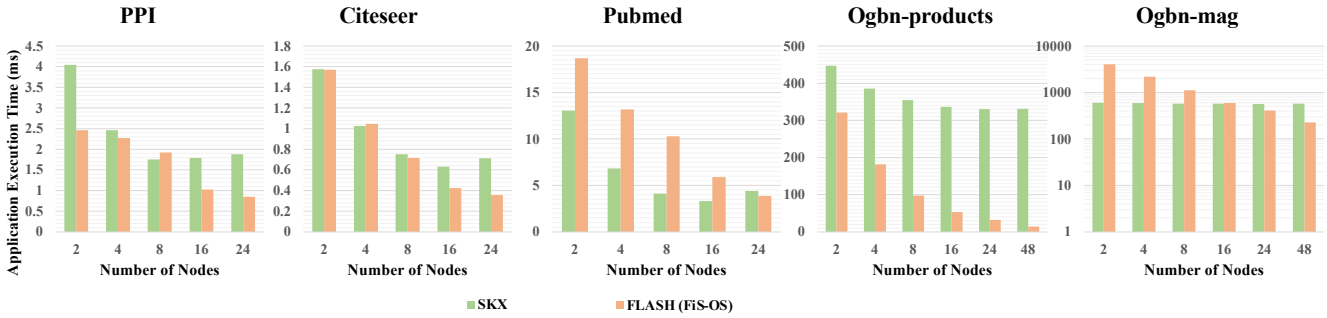

Figure 7: Application performance and scalability comparison of GCN on a baseline CPU cluster (SKX) vs. FLASH.

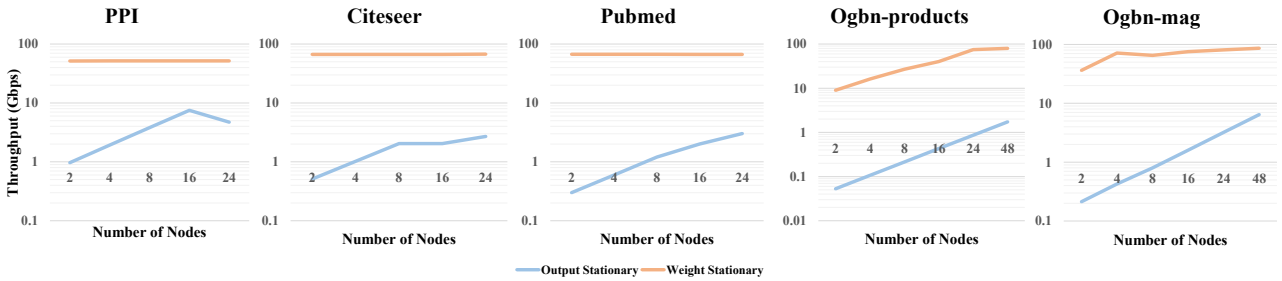

Figure 8: Overall application throughput measured at the LA's input interface for different datsets.

Table 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

the "software" scale of milliseconds rather than the hours typical for HLS tools.

5.7 Resource requirements

Figure 9 shows hardware resource utilization on the Xilinx Alveo U280 FPGA board for a FiS configuration with 16 pipeline vector PEs. The switch is implemented using a NetFPGA design [32]. FaS 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.

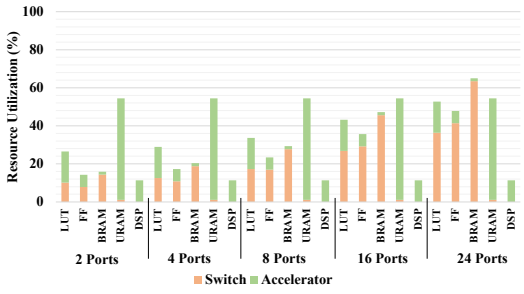

Figure 9: Resource utilization for FiS configuration with 16 pipeline vector PEs on Alveo U280.

5.8 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, Ogbn-products, and Ogbn-mag, respectively. These overheads are negligible compared to total execution time (Figure 7) for most datasets Ogbn-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).

5.9 Comparison with Prior Work

To demonstrate the applicability of the approach to other applications, and compare it with other in-switch computing approaches, we consider DNN training on FLASH and Mellanox SHArP [14]. We compare the time it takes to update the (last-layer) weights of the AlexNet model during DNN training using FLASH to that of doing so on the same Stampede2 cluster using Mellanox switches. Our simulation results show that FLASH achieves 1.7× speedup on 64 nodes for the last layer update. Since we do not have access to switch internals in HPC clusters, we superimpose the result from a Mellanox paper [14] on the Stampede2 cluster for the Allreduce collective based on the message size. We only accelerate the last layer as we find that this could lead to better coupling of the computation part (other layers) with the communication part (last layer).

Our approach improves communication time by restructuring the application as follows: each node processes forward propagation for all layers except the last; the switches perform matrix/vector multiplications for the last layer (the last-layer weights are stored in switches); then the new weights stored in switches are updated by aggregating the local weights from each node. This happens for each iteration. Weight updates for all layers except the last are performed in the same way as in the baseline (synchronous Allreduce-based training [29]). Instead of communicating and transferring weights back and forth to the nodes (Allreduce for the current iteration), performing computation on them (forward propagation for the next iteration), and then another communication (Allreduce for the next iteration), weights are processed in the switches, resulting in reduced communication time. We note that it is not possible to take advantage of Mellanox offload support for GCN inference as these switches do not support Allgather collectives.

6 RELATED WORK

In-switch collective processing: Previous work has shown significant benefits of optimizing collectives and offloading them to the switch. Mellanox [13] has offloaded MPI collectives to ASIC-based switches using reduction trees. Their approach supports fixed functions and data types with no extensibility; also, few design details are provided. The authors in [30] propose an FPGA-based in-switch acceleration scheme for distributed reinforcement learning to move gradient aggregation from server nodes to the network switches. In [18, 19, 46] a new method for supporting MPI communicators and accelerating collectives in the reconfigurable switches is presented. Finally, the authors in [7] design a flexible programmable switch architecture for in-network data reduction. Although it is possible to process custom operations through packet handlers, their evaluation is only limited to dense/sparse `MPI_Allreduce`. These are all inline acceleration methods. We note that while the latter work is based on RISC-V cores the largest memory footprint is 4 MBytes.

In-switch application processing: Taurus [47] adds a custom MapReduce block to programmable switch devices to enable per-packet ML inference. N2Net [43] demonstrates implementations of binary neural networks within network devices. IIsy [55] introduces a software and hardware-based prototype for mapping trained non-neural network ML models to switch match action pipelines. However, they are only applicable to traditional neural network algorithms with small memory models due to their limited on-chip memory.

CGRA: Many CGRA architectures have been proposed. Some prior art utilized a CGRA closely coupled with a CPU. For instance, ADRES [31] proposed a novel compiler-friendly architecture that is tightly coupled with a very long instruction word (VLIW) processor with reduced communication overhead. [25] introduces a CGRA architecture with reconfigurable interconnect with single cycle communication with distant PEs. Prabhakar et al. [35] proposed a new architecture as a collection of compute and memory units to efficiently execute applications composed of parallel patterns. The distinction of all the above work from ours is that our CGRA accelerator itself is composed of multiple RISC-V compatible cores pipelined together.

7 DISCUSSION AND WORK IN PROGRESS

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. Some extensions are in progress. Vector PEs are pipelined together and are independent from each other except that incoming streaming packets are the same. Other types of dependencies and more complex types are not yet supported. Finally, certain parallel patterns (e.g., breadth first search) may not map efficiently to the current FLASH architecture; in future work we seek to make FLASH more general purpose.

8 CONCLUSION

In this work, we designed, implemented, and evaluated a programmable look-aside accelerator that can be embedded into, or attached to, existing communication switches. To facilitate usability, we developed 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× (across five real-world datasets) on 24 nodes. As part of future work, we will demonstrate our approach for GCN training and other workloads with a larger number of nodes.

ACKNOWLEDGMENTS

This work was supported, in part, by the NSF through awards CCF-1919130, CNS-1925504, and CCF-2151021; by a grant from Red Hat; and by AMD and Intel both through donated FPGAs, tools, and IP.

REFERENCES

[1] O. Arap and M. Swany. 2016. Offloading Collective Operations to Programmable Logic on a Zynq Cluster. In *2016 IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI)*. 76–83.

[2] Arista. 2023. 7130 FPGA-enabled Network Switches - Quick Look. www.arista.com/en/products/7130-fpga-enabled-network-switches-quick-look.

[3] 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.

[4] M. Bayatpour, N. Sarkauskas, H. Subramoni, J. Maqbool Hashmi, and D. K. Panda. 2021. BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs. In *High Performance Computing: 36th International Conference, ISC High Performance 2021*. Springer, 18–37.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (jul 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[6] Y. Chen, J. Emer, and V. Sze. 2017. Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators. *IEEE Micro* 37, 3 (2017), 12–21. https://doi.org/10.1109/MM.2017.54

[7] D. De Sensi, S. Di Girolamo, S. Ashkboos, S. Li, and T. Hoefler. 2021. Flare: Flexible In-Network Allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[8] A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels. 2009. MPI Collective Communications on the Blue Gene/P Supercomputer: Algorithms and Optimizations. *2009 17th IEEE Symposium on High Performance Interconnects* (2009), 63–72.

[9] J. Gasteiger, C. Qian, and S. Günnemann. 2022. Influence-Based Mini-Batching for Graph Neural Networks. *arXiv preprint arXiv:2212.09083* (2022).

[10] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M.C. Herbordt. 2020. AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing. In *53rd IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[11] T. Geng, C. Wu, Y. Zhang, C. Tan, C. Xie, H. You, M.C. Herbordt, Y. Lin, and A. Li. 2021. 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.

[12] R. L. Graham et al. 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)*. 1–8.

[13] R. L. Graham et al. 2016. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. 1–10.

[14] Richard L. Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, Ami Marelli, Valentin Petrov, Evyatar Romlet, Yong Qin, and Ido Zemah. 2020. Scalable Hierarchical Aggregation and Reduction Protocol (SHARP)TM Streaming-Aggregation Hardware Design and Evaluation. In *High Performance Computing*, Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief (Eds.). Springer International Publishing, Cham, 41–59.

[15] A. Guo, T. Geng, Y. Zhang, P. Haghi, C. Wu, C. Tan, Y. Lin, A. Li, and M.C. Herbordt. 2022. A Framework for Neural Network Inference on FPGA-Centric SmartNICs. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

[16] A. Guo, Y. Hao, C. Wu, P. Haghi, Z. Pan, M. Si, D. Tao, A. Li, M.C. Herbordt, and T. Geng. 2023. Software-Hardware Co-design of Heterogeneous SmartNIC System for Recommendation Models Inference and Training. In *ICS 2023: International Conference on Supercomputing*.

[17] P. Haghi, A. Guo, T. Geng, A. Skjellum, and M.C. Herbordt. 2021. Workload Imbalance in HPC Applications: Effect on Performance of In-Network Processing. In *IEEE High Performance Extreme Computing Conference*. doi: 10.1109/HPEC49654.2021.9622847.

[18] P. Haghi, A. Guo, Q. Xiong, R. Patel, C. Yang, T. Geng, J.T. Broaddus, R. Marshall, A. Skjellum, and M.C. Herbordt. 2020. FPGAs in the Network and Novel Communicator Support Accelerate MPI Collectives. In *IEEE High Performance Extreme Computing Conference*.

[19] P. Haghi, A. Guo, Q. Xiong, C. Yang, T. Geng, J.T. Broaddus, R. Marshall, D. Schafer, A. Skjellum, and M.C. Herbordt. 2022. Reconfigurable switches for high performance and flexible MPI collectives. *Concurrency and Computation: Practice and Experience* 34, 2 (2022). doi: 10.1002/cpe.6769.

[20] S. Handagala, M.C. Herbordt, and M. Leeser. 2021. OCT: The Open Cloud FPGA Testbed. In *31st International Conference on Field Programmable Logic and Applications (FPL)*.

[21] S. Handagala, M. Leeser, K. Patle, and M. Zink. 2022. Network Attached FPGAs in the Open Cloud Testbed (OCT). In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 1–6.

[22] F. Hauser et al. 2021. A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research. *arXiv preprint arXiv:2101.10632* (2021).

[23] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[24] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, I.S. Dhillon, D.S. Papailiopoulos, and V. Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/300.pdf

[25] M. Karunaratne, A. K. Mohite, T. Mitra, and L. Peh. 2017. HyCUBE: A CGRA with Reconfigurable Single-Cycle Multi-hop Interconnect. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/3061639.3062262

[26] E. F. Kfoury, J. Crichigno, and E. Bou-Harb. 2021. An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends. *IEEE Access* 9 (2021), 87094–87155.

[27] V. Krishnan, O. Serres, and M. Blocksome. 2020. COnfigurable Network Protocol Accelerator (COPA): An Integrated Networking/Accelerator Hardware/Software Framework. In *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*. 17–24. https://doi.org/10.1109/HOTI51249.2020.00018

[28] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *International Symposium on Code Generation and Optimization, CGO*. 75–86. https://doi.org/10.1109/CGO.2004.1281665

[29] A. Li, T. Geng, T. Wang, M.C. Herbordt, S. Song, and K. Barker. 2019. BSTC: A Novel Binarized-Soft-Tensor-Core Design for Accelerating Bit-Based Approximated Neural Nets. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. doi: 10.1145/ 3295500.3356169.

[30] Youjie Li and et al. 2019. Accelerating Distributed Reinforcement learning with In-Switch Computing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 279–291.

[31] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2003. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Field Programmable Logic and Application (FPL)*. 61–70.

[32] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. 2008. NetFPGA: Reusable Router Architecture for Experimental Research. In *Association for Computing Machinery PRESTO* (Seattle, WA, USA). New York, NY, USA, 1–7. https://doi.org/10.1145/1397718.1397720

[33] New Wave DV. 2023. 32-Port Programmable Switch. https://newwavedv.com/products/appliances/32-port-programmable-switch/.

[34] J. Park, M. Smelyanskiy, U. M. Yang, D. Mudigere, and P. Dubey. 2015. High-performance algebraic multigrid solver optimized for multi-core based distributed parallel systems. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[35] R. Prabhakar et al. 2017. Plasticine: A Reconfigurable Architecture for Parallel Patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 389–402. https://doi.org/10.1145/3079856.3080256

[36] S. Qiao, C. Hu, G. Brebner, J. Zou, and X. Guan. 2020. Adaptable Switch: A Heterogeneous Switch Architecture for Network-Centric Computing. *IEEE Communications Magazine* 58, 12 (2020), 64–69. https://doi.org/10.1109/MCOM.001.2000399

[37] A. L. G. Rios, K. Bekshentayeva, M. Singh, S. Haeri, and L. Trajkovic. 2021. Virtual Network Embedding for Switch-Centric Data Center Networks. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5. https://doi.org/10.1109/ISCAS51556.2021.9401784

[38] RISC-V. 2023. RISC-V Specifications. https://riscv.org/technical/specifications/.

[39] RISC-V. 2023. RISC-V 'V' Vector Specifications. https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc.

[40] G. Sankaran, J. Chung, and R. Kettimuthu. 2021. Leveraging In-Network Computing and Programmable Switches for Streaming Analysis of Scientific Data. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. 293–297. https://doi.org/10.1109/NetSoft51509.2021.9492726

[41] A. Sapio et al. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio

[42] J. Sheng, Q. Xiong, C. Yang, and M.C. Herbordt. 2017. Collective Communication on FPGA Clusters with Static Scheduling. *ACM SIGARCH Computer Architecture News* 44, 4 (2017). doi: 10.1145/ 3039902.3039904.

[43] G. Siracusano and R. Bifulco. 2018. In-Network Neural Networks. *arXiv preprint arXiv:1801.05731* (2018).

[44] D. Stanzione et al. 2017. Stampede 2: The Evolution of an XSEDE Supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing on Sustainability, Success and Impact (PEARC17)*. Article 15, 8 pages. https://doi.org/10.1145/3093338.3093385

[45] J. Stern, Q. Xiong, J. Sheng, A. Skjellum, and M.C. Herbordt. 2017. Accelerating MPI_Reduce with FPGAs in the Network. In *Workshop on Exascale MPI*.

[46] J. Stern, Q. Xiong, A. Skjellum, and M.C. Herbordt. 2018. A Novel Approach to Supporting Communicators for In-Switch Processing of MPI Collectives. In *Workshop on Exascale MPI*.

[47] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun. 2022. Taurus: a Data Plane Architecture for Per-Packet ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. 1099–1114.

[48] I. Taras and J. H. Anderson. 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)*. 87–95. https://doi.org/10.1109/FCCM.2019.00022

[49] A. Tripathy, K. Yelick, and A. Buluç. 2020. Reducing Communication in Graph Neural Network Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. 1–14. https://doi.org/10.1109/SC41405.2020.00074

[50] H. Wang et al. 2017. P4FPGA: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. 122–135.

[51] Andrew Waterman and Krste Asanovic. 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.

[52] Xilinx. 2023. AXI Reference Guide, Vivado Design Suite. https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide.

[53] Xilinx. 2023. Xilinx Runtime Library (XRT). https://www.xilinx.com/products/design-tools/vitis/xrt.html.

[54] Xilinx. 2023. XUP Vitis Network Example (VNx). https://github.com/Xilinx/xup_vitis_network_example.

[55] Z. Xiong and N. Zilberman. 2019. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. 25–33.

[56] B. Zhang, R. Kannan, and V. Prasanna. 2021. BoostGCN: A Framework for Optimizing GCN Inference on FPGA. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 29–39.