Unlocking the Power of Inline Floating-Point Operations on Programmable Switches

Yifan Yuan
UIUC

Omar Alama
KAUST

Jiawei Fei
KAUST & NUDT

Jacob Nelson
Microsoft Research

Dan R. K. Ports
Microsoft Research

Amedeo Sapio
Intel

Marco Canini
KAUST

Nam Sung Kim
UIUC

Abstract

The advent of switches with programmable dataplanes has enabled the rapid development of new network functionality, as well as providing a platform for acceleration of a broad range of application-level functionality. However, existing switch hardware was not designed with application acceleration in mind, and thus applications requiring operations or datatypes not used in traditional network protocols must resort to expensive workarounds. Applications involving floating point data, including distributed training for machine learning and distributed query processing, are key examples.

In this paper, we propose FPISA, a floating point representation designed to work efficiently in programmable switches. We first implement FPISA on an Intel Tofino switch, but find that it has limitations that impact throughput and accuracy. We then propose hardware changes to address these limitations based on the open-source Banzai switch architecture, and synthesize them in a 15-nm standard-cell library to demonstrate their feasibility. Finally, we use FPISA to implement accelerators for training for machine learning as an example application, and evaluate its performance on a switch implementing our changes using emulation. We find that FPISA allows distributed training to use one to three fewer CPU cores and provide up to 85.9% better throughput than SwitchML in a CPU-constrained environment.

1 Introduction

The rise of programmable network devices has transformed distributed systems design. Instead of simply moving data between servers using standard routing protocols, network devices can be programmed using domain-specific languages like P4 [8] and NPL [10] to support new network functionality, such as congestion control [100], load balancing [55, 78], and packet scheduling [103]. Commodity Ethernet switch ASICs with programmable data planes [11, 42, 92] enable the execution of these programs at many terabits per second.

While these capabilities were originally targeted at increasing network functionality, much recent work has explored their utility in accelerating application-level functionality as well. Consensus protocols [17, 65, 93], concurrency control [45, 64], vector addition [75, 97, 98], query processing operators [34, 63], and key-value stores [49, 66, 112] have all

been shown to benefit from this in-network computation [94].

However, an important class of applications has struggled to take advantage of in-network computation: those using floating point (FP) values. These occur in two broadly-deployed datacenter applications: distributed training for machine learning, and distributed data processing systems. Since programmable switches were originally optimized for networking applications, their design includes basic support only for integer operations. Applications wanting to take advantage of in-network computation with floating point values have so far worked around this in one of three ways.

The first approach is to approximate floating point operations in software running on end-hosts. This is the approach taken by SwitchML [98] as it sums gradient vectors as part of training deep neural networks. For each chunk of gradient vector elements, SwitchML executes a protocol that requires running code to convert between floating point and integer values on end hosts, as well as performing two rounds of communication. This protocol overhead is costly (see Sec. 5.3.3).

The second approach is to build a switch ASIC that includes floating point hardware. This is the approach taken by the Mellanox Quantum switch [32, 76]. Dedicating chip resources for this purpose is expensive: we show (Sec. 4.2) that adding dedicated FPU hardware takes more than $5\times$ the die area and power of integer ALUs. As a result, this is not a general-purpose approach; it has only been taken for InfiniBand switches, which have simpler routing designs and buffer requirements than Ethernet switches, and hence have spare die area. It also lacks flexibility: it is tied to specific operations on specific floating-point formats. New ML-specific numeric representations (*e.g.*, FP16 [79, 106], bfloat16 [21, 30, 53], TF32 [86], and MSFP [18]) represent an area of ongoing innovation, and adding support for a new format requires developing and manufacturing a new ASIC – an expensive and time-consuming endeavor. For example, it took four years for Mellanox to release its second version of switches with floating point support [31, 32].

A related approach is to use FPGAs or other non-switch programmable devices to implement switch-like specialized accelerators [5, 20, 27, 70]. While this yields a functional solution, the fine-grained programmability of a FPGA comes at the cost of power [111] and area: for example, Xilinx's flagship FPGA

supports ~8 Tbps [114] of Ethernet I/O, while the Intel Tofino 2, a regular programmable switch, supports 12.8 Tbps [43].

In this paper, we argue for a different approach. We propose FPISA, which implements floating point computation as a P4 program running directly on a programmable switch. This is not straightforward: the multi-cycle nature of floating-point operations is at odds with the streaming-pipeline architecture common to P4-programmable switches today. To make it work, FPISA breaks apart each floating point value into *exponent* and *signed mantissa* and stores them separately in different pipeline stages, decomposing the corresponding sub-operations appropriately to ensure correct execution. Rather than requiring specialized floating-point hardware, FPISA repurposes network-oriented hardware elements in the switch pipeline to implement the sub-operations not supported by the switch's integer ALUs.

FPISA is a generic approach. We evaluate its feasibility on the Intel Tofino [42], a commercially-available PISA switch. We observe that constraints of the existing Tofino architecture present obstacles to a full FPISA implementation. We address this in two ways. First, we introduce an approximate FPISA design (FPISA-A) that is implementable on existing hardware, albeit with some precision and throughput limitations. Second, we propose some simple and cheap hardware modifications, based on the open-source Banzai [102] switch architecture, to enable high throughput and accuracy with FPISA. We show that such enhancements are feasible in a 15-nm standard-cell library with minimal power, area, and timing cost relative to a baseline switch chip.

Through an emulation-based study, we assess the performance benefits of our approach by implementing accelerators for the use case of distributed training for machine learning, based on the recent SwitchML [98] framework. Enhancing SwitchML with FPISA (based on both regular FP32 and ML-specific FP16) allows it to use 1-3 fewer CPU cores, giving up to an 85.9% improvement in training throughput on CPU-limited configurations, while still achieving the same training accuracy and convergence.

2 Background and Challenges

Conventional network switches are fixed-function, requiring redesign to add new features or support new protocols. However, in today's era of software-defined networking [58], rapidly evolving networking techniques and applications require new packet processing support. Programmable switches, which allow the data plane behavior to be reconfigured, provide the necessary flexibility. The RMT-based Protocol-Independent Switch Architecture (PISA) [9] has emerged as the de facto standard for programmable switch architecture.

2.1 PISA

We depict the basic protocol-independent switch architecture design in Fig. 1. The parser is a programmable state machine responsible for extracting user-specified fields of the inbound
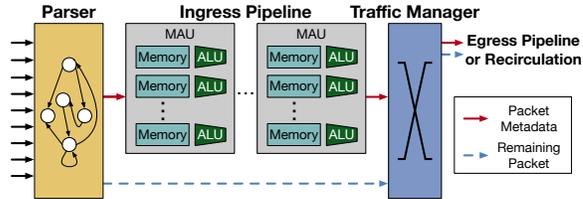

Figure 1: Basic PISA design.

packet to per-packet metadata.[1] The ingress pipeline consists of multiple cascaded match-action units (MAUs). Each MAU has some memory (SRAM and TCAM) and ALUs. It matches fields from the packet metadata against the memory to determine the corresponding action to be taken by the ALUs. The ALUs support basic integer arithmetic and logic operations, and can be used to modify fields in the packet metadata. They can also manipulate registers, which hold state that persists across different packets.

After going through the ingress pipeline, the packet is routed to an egress port and queued by the traffic manager. Before being output, it passes through an egress pipeline that has the same structure as the ingress pipeline, and the packet header and body are reassembled by the deparser.

Programmable switches following this architecture have become commercially available on commodity switches, thanks to reconfigurable switch silicon like the Intel (Barefoot) Tofino [42] and Marvell XPliant [88]. A long line of research has showed how to use PISA switches to implement new networking protocols, offload network functions, and accelerate application-level logic [36, 94].

2.2 Floating Point Overview

We describe the flow of the most common floating point operation in applications discussed in this paper – addition – here. Note that subtraction is performed using the same process, and comparisons are typically implemented using subtraction. Regardless of specific widths, floating point values are represented with three parts: 1-bit sign, n-bit exponent, and m-bit mantissa. Typically, a floating point number is represented in normalized form: the mantissa value is in the range of $[1,2)$, *i.e.*, it begins with a leading "1" bit (which can be omitted, *i.e.*, "implied 1"). A floating point addition $C = A + B$ is performed using a five-step process: (We assume here that $\mathrm{abs}(A) \leq \mathrm{abs}(B)$.)

Extract. The three parts of A and B are extracted from the packed data. The implied "1" in the packed mantissa is expressed explicitly.

Align. The two mantissas are aligned to represent values at the same magnitude. Specifically, $mantissa_A$ (the smaller one) is right-shifted by $exponent_A - exponent_B$ bits.

Add/subtract. Now that the two mantissas are aligned, they are added or subtracted, depending on sign: $mantissa_C = mantissa_B \pm mantissa_A$.

---

[1] The remainder of the packet is passed through the pipeline, but cannot be matched or manipulated.
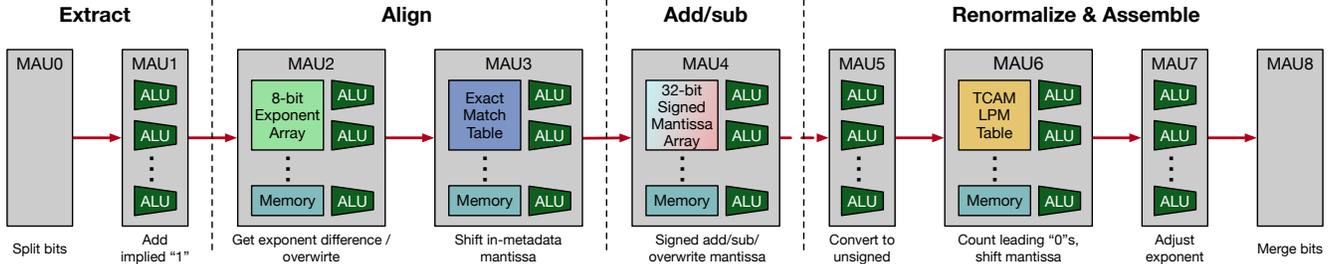
Figure 2: FPISA dataflow. Only hardware components relevant to FPISA are shown.

Renormalize. The result is scaled so that the mantissa is in the range of $[1,2)$. This is achieved by counting the leading "0" bits and left or right shifting $mantissa_C$ accordingly, then adjusting $exponent_C$ by the corresponding value.

Round and Assemble. Finally, the three parts of C are packed into a single value. The implied leading "1" of $mantissa_C$ is stripped. If more mantissa bits are available than can be represented in the packed format, the mantissa is rounded.

2.3 Challenges

Current PISA architectures do not natively support any floating point operations. This is no surprise, considering that they were designed for packet processing, and floating point support is expensive. FPUs have much larger power and area costs than integer ALUs [62, 68, 74], and the complex floating point addition procedure (Sec. 2.2) takes multiple cycles and thus introduces timing constraints.

This paper asks if we can build floating point addition operations on a commodity PISA architecture. Intuitively, it should be possible to decompose the canonical addition procedure and span it across multiple pipeline stages. However, we observe that this leads to two challenges.

First, registers are associated with specific pipeline stages, and can only be accessed from that stage. That is, each register can only be accessed once per packet, and data dependencies cannot "go backwards" to an earlier stage.[2] This poses a problem for applications, like in-network aggregation, that wish to maintain and update floating point state: it is not possible, for example, to perform the add-mantissa and renormalize steps in different pipeline stages.

Second, the available ALU operations may not be sufficient to implement all the operations necessary to implement floating point addition. For instance, on a CPU, the renormalization step might use a count-leading-zeros instruction (*e.g.*, lzcnt on x86), but we know of no PISA switch with such an instruction.

Hence, we must develop a PISA-friendly, decentralized (multi-stage) approach for floating point addition.

3 FPISA Design

How can we implement floating point operations on PISA architectures, given the challenges described above? We propose a design, FPISA, based on a new floating point

representation and a mapping of its operations to PISA pipelines, as shown in Fig. 2. In this section, we describe the basic FPISA approach in the context of an abstract PISA pipeline; Sec. 4 discusses additional challenges that occur when implementing it on existing PISA architectures.

FPISA has three key ideas:

Decoupled exponent and mantissa operations. FPISA processes operations on the exponent and (signed) mantissa components of floating point values separately, and internally stores them in separate registers. This decoupling allows them to be processed by different pipeline stages.

Delayed renormalization. Second, FPISA does not require intermediate values to be renormalized on every operation. That is, in a SwitchML-like [98] aggregation workflow, values from each client are added to an accumulator whose value is not renormalized until the final result is output. This is based on two observations about floating point renormalization. First, renormalization does not affect the correctness of floating point operations. Scaling the mantissa to place the leading "1" in its correct location is needed to produce an output value in canonical format, but a denormalized form can equally represent the same arithmetic value. Second, renormalization introduces data dependencies between the mantissa and exponent components, which makes it challenging to fit into a PISA pipeline. In particular, renormalization requires the exponent to be adjusted based on the computed mantissa, whose computation itself depends on the exponent – a circular data dependency that cannot be represented in a single pipeline traversal. To avoid this, when we read from the accumulator, we read the denormalized value, and normalize it just before sending out the final result. We do not store the normalized value back into the accumulator.

Extra bits in mantissa register. PISA architectures commonly have registers with limited bit widths: 8-, 16-, or 32-bit registers are common; on the other hand, floating point values commonly have mantissas with smaller bitwdith. We take advantage of this difference in two ways. First, we can use bits to the right of the mantissa as guard bits to aid in rounding, as is common in standard FPUs. Second, we can use bits to the left of the mantissa to avoid overflow when summing multiple values with similar exponents. When we add two values with mantissas that are all ones, the addition simply carries into the bits to the left of the mantissa.

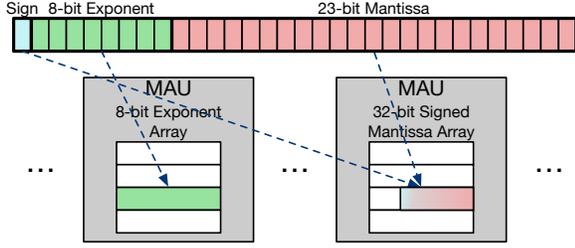In this section, we use IEEE 754 FP32 – which has a 1-bit

Figure 3: FPISA's representation of FP32 in the switch.

Figure 4: Example of FPISA addition: computing the sum of 3.0 ($0b1.1 \times 2^1$) and 1.0 ($0b1 \times 2^0$). Computation is done using a 32-bit mantissa; 21 trailing zero bits are elided.

sign, 23-bit mantissa, and 8-bit exponent – as an example to demonstrate FPISA design. Other FP formats with different widths can also be supported. Fig. 2 shows FPISA's dataflow.

3.1 Representing FP in PISA

To meet the constraints of PISA, FPISA splits the storage of floating point values using the representation shown in Fig. 3. The exponent field is stored in an 8-bit-wide register array. The 23-bit mantissa is stored, right-aligned, in a 32-bit register. To unify signs and addition/subtraction operations, we store the mantissa in two's-complement signed representation.

FPISA needs more memory space to store a floating point number (*e.g.*, 8+32=40 bits for a FP32 number). However, we argue that this will not significantly reduce the efficiency of FPISA since exponent and mantissa have to be stored in different MAUs anyway. Hence, the per-MAU parallelism of floating point operations will not be affected.

3.2 Performing FP operations in PISA

By delaying renormalization until the output phase and storing exponents and mantissas separately, FPISA makes it possible to adapt the standard extract-align-add-renormalize-assemble floating point addition flow to a PISA pipeline. Fig. 2 shows the mapping of functionality to MAUs. We use a running example (Fig. 4) where an input of 1.0 is added to a register containing the value 3.0.

Extract. The first stages extract the exponent and mantissa

Match (Man$_{metadata}$)	Action (Man$_{metadata}$)
64.0.0.0/2	Right-shift 7 bits
…	…
1.0.0.0/8	Right-shift 1 bit
0.128.0.0/9	Do nothing
0.64.0.0/10	Left-shift 1 bit
…	…
0.0.0.1/32	Left-shift 23 bits
Default	Do nothing

Figure 5: LPM match-action table (MAU6) in FPISA design.

from a FP32 value in the input packet into separate metadata registers (MAU0), then add the implied "1" to the extracted mantissa field (MAU1). The decoded values are shown in Fig. 4 step (1).

Align. FPISA then compares the provided exponent value with the one stored in memory in MAU2. This updates the exponent and determines which of the two operands's mantissa must be right-shifted and by how much. The right shift itself is performed for the metadata value by MAU3, and for the memory value by MAU4 (where the mantissa register is located). In Fig. 4 step (2), 1.0 is shifted right to be expressed as 0.1×2^1

Add. In addition to shifting the mantissa of the in-memory value, MAU4 performs the mantissa addition itself. Depending on the sign bit, it either adds or subtracts the shifted mantissa value generated in the previous stage from the stored mantissa value (step (3) in Fig. 4). The resulting mantissa value replaces the previous stored mantissa.

Note that MAU4 is used both to perform the right shift of the stored mantissa and its addition. This is a necessity because the PISA architecture can only update a given register from one stage. Existing implementations may not be able to perform both operations with a single stateful ALU; we discuss how to extend them or how to work around this limitation in Sec. 4.

At the end of this process, the exponent and mantissa registers contain the result of the addition, but may not be in normalized form. For example, in step (4) of Fig. 4, the registers store the value $0b10.0 \times 2^1$. This is indeed a valid representation of the result 4.0, but is not in normalized form because the mantissa has more than one digit to the left of the binary point.

Renormalize and Assemble. FPISA delays renormalization: it does not renormalize the intermediate value stored in registers, but only when the result is to be output. Thus, multiple additions can be performed before renormalization. This offers two benefits. As mentioned before, it eliminates the need to adjust the exponent stored in memory after calculating the mantissa, avoiding a data dependency. Second, since the renormalization and assembly steps are stateless, we can place them in the (normally underutilized) egress pipeline, making more efficient use of resources.

The renormalization process itself is performed in four steps. The aggregated mantissa is first converted from its two's complement signed representation to unsigned value and sign (MAU5). FPISA then counts the number of leading zeros and

shifts the mantissa value accordingly, in order to place the leading "1" bit in the right location (MAU6).

Because no PISA switches support a count-leading-zeros operation, FPISA exploits a TCAM-based longest prefix match (LPM) table – commonly used in IP routing – to implement this function. Specifically, we construct a LPM table where each entry has an IP address with only the ith bit set, and a netmask that matches the first i bits. A match indicates that the mantissa has $i-1$ leading zeros. This is used to select the right shift action that places the leading 1 in its canonical location (bit 24 for FP32). In the example, the leading "1" is located using a match, whose bitwise representation is shown in step (5), which corresponds to the CIDR address 0.128.0.0/9; the lookup table (Fig. 5) indicates that the mantissa should be shifted right by 1. The exponent is adjusted also according to the leading zeros' count (in MAU7) – here, incremented by 1. This gives a normalized result; all that remains is to merge the sign, exponent, and lower 23 bit of the 32-bit mantissa fields (in MAU8) to put it in FP32 format.

3.3 Additional Floating Point Features and Operations

Overflow. The denormalized representation has the potential to overflow if similar values are added many times. With a signed register size of 32 bits and a mantissa size of 24 bits, there are 7 bits to the left of the mantissa available for holding overflows. This is sufficient to represent 128 additions of values with the maximum mantissa with the same exponent – an extreme case – into a single register without overflow. However, for the use cases described later in the paper, the number of operations per register is equivalent to the number of nodes in the distributed system. If overflow occurs, it can be detected and signaled to the user, who can handle it in an application-specific way.

Other FP formats. FPISA can be trivially modified to support floating point formats with different exponent and mantissa width (*e.g.* FP16, which we evaluate in Sec. 5). Likewise, block floating point formats, where multiple values share one exponent [18], can be supported by replicating the exponent register.

Rounding. For simplicity, we have described FPISA without guard digits. The combination of no guard digits and two's-complement representation provide round-toward-negative-infinity semantics. An implementation with n guard digits would simply store the mantissa shifted left n bits from what is show in Fig. 3, and would use those to perform other types of rounding after renormalization.

Reproducibility. FPISA provides reproducibility in that the same sequence of operations and values will always produce the same result. However, since FPISA performs operations in a different order than that specified in the IEEE 754 standard, the same sequence of operations and values performed on an IEEE-754-compliant CPU may yield a different result than FPISA. For the use cases we describe in this paper, IEEE 754 compliance is not a requirement.

In this paper, we have covered the two commonly-used floating point operations – addition and comparison. They are sufficient for many distributed applications. However, other more complex and costly floating point operations may be needed in the future with emerging applications (*e.g.*, congestion control [26, 54] and network security [34]). To pave the way for future PISA implementations, we briefly discuss the possibility of supporting them.

Multiplication and division. The flow of floating point multiplication is similar to that of addition in Sec. 2.2. The two major differences are (1) the two exponents are added, and (2) the two mantissas are multiplied, all as integers. For small floating point types, the mantissa multiplication can be implemented as a table lookup, without hardware modifications. For larger floating point types, integer multiplers could be added to the hardware. We implement one based on Banzai and find its overhead is acceptable: approximately the same as an adder and a boolean module w.r.t. power and area.

Floating point division has a different flow and takes more clock cycles than other basic operations [104], which means it is unsuitable to have a direct hardware implementation in programmable switches. For some use cases, division can be implemented by converting the dividend to its reciprocal at the end-host and then multiplying in the switch.

Logarithms. The core operation of a floating point logarithm is the integer logarithm of the mantissa. As prior research [3, 99, 109] shows, this can be done by a lookup table of fewer than 2000 entries with low error (<1%).

Square roots. Square roots are even more expensive and time-consuming (*e.g.*, more than 20 clock cycles) than division [69, 87, 104]. As with logarithms, we suggest a lookup-table-based approximation for this algorithm.

4 Realizing FPISA on PISA Architectures

The previous section shows how FPISA can map floating point operations to an abstract PISA architecture. Actual PISA implementations may have restrictions on MAU operations. We have implemented FPISA in P4 for the Tofino architecture. In doing so, we encountered several architectural limitations (Sec. 4.1). We show that simple architectural extensions, which can be implemented with minimal power and chip area cost, can resolve these limitations and enable a full FPISA implementation (Sec. 4.2). Alternatively, we describe an approximate approach, FPISA-A, which works around these limitations to implement a variant of FPISA for the existing Tofino architecture, albeit with tradeoffs in accuracy and resource utilization (Sec. 4.3).

4.1 Challenges

We implement FPISA addition in the P4 language [8] ($\sim$580 LoC) in a modularized manner (*i.e.*, one floating point addition per module) and compile it to the Tofino ASIC [42]. Tab. 1 shows the resource utilization of the FPISA module out of a single Tofino pipeline. Most of these resources cannot be

Table 1: FPISA resource utilization. Nine pipeline stages (out of 12 in total) are used.

Resource	Total usage	Max usage in a MAU
SRAM	1.15%	5.00%
TCAM	0.03%	4.17%
Stateful ALU	8.33%	50.00%
VLIW instruction slots	19.01%	96.88%
Input crossbar	0.09%	4.38%
Result bus	2.34%	12.50%
Hash bit	1.06%	7.93%

shared across multiple FPISA instances.

Using this implementation, we identify three limitations of the the current Tofino hardware that impact the functionality and efficiency of our FP operations.

Resource utilization of shift operations. In general, multiple FPISA modules can be deployed in parallel, sharing the same pipeline stages and overlapping with each other. For many applications, performing as many operations per packet as possible is essential to achieve high performance [98]. Unfortunately, the current Tofino architecture can only accommodate one FPISA module in its ingress pipeline, *i.e.*, only one floating point addition can be performed per packet.

After analyzing the resource utilization, we observe that the main source of overhead is performing shift operations. Specifically, FPISA needs to shift fields by a variable number of bits, in order to implement the alignment and renormalization stages. However, the Tofino ALUs can only perform shift operations with a fixed shift distance, specified as an immediate. While it is possible to emulate a variable-length shift operation with the current functionality, doing so is resource intensive. In particular, per-stage VLIW instruction utilization prevents multiple FPISA instances from sharing pipeline stages.

Lack of atomic shift-and-add. One of the pipeline stages in the abstract design (MAU4 in Fig. 2) must perform two operations: right-shifting the stored mantissa to align it with the value being added, and performing the mantissa addition. Both are stateful operations on the mantissa register, so they must be performed by the same stage's ALU. However, the Tofino's ALUs cannot perform *both* a shift and an add operation. In Sec. 4.3, we show how to work around this limitation by left-shifting the other mantissa value (from the packet metadata) instead; this allows the FPISA design to be implemented on the existing Tofino architecture, but can lead to numerical error for some workloads.

Endianness conversion. While hardly unique to FPISA, endianness conversion is a non-trivial source of overhead for FPISA applications. Network devices interpret values in network byte order (big-endian), whereas most general-purpose CPUs are little-endian. To identify and process the data correctly in the switch, endianness conversion is necessary. Traditional networking applications only need to convert byte order for headers, which are relatively small. For data-intensive in-switch applications, byte order conversion for the full payload can have high overhead. While the Tofino has functional units that can
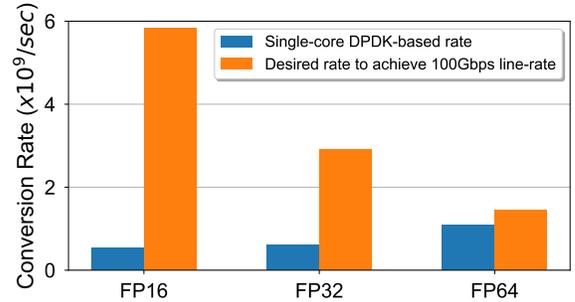

Figure 6: Endianness conversion rate that a core can achieve and that is desired to achieve 100 Gbps line-rate.

Table 2: Stateless ALU and stateful RAW/RSAW unit areas and minimum critical-path delays in FreePDK15 library. Each of the compiler targets contains 300 instances of one of the ALUs. Power and area are evaluated at 1 GHz frequency target.

	Default ALU	FPISA ALU	Default RAW	FPISA RSAW	ALU+ FPU
Dynamic power (μW)	594.2	669.4	637.6	721.1	3590.6
Leakage power (μW)	18.6	22.8	16.8	22.1	109.8
Area (μm^2)	505.4	618.6	468.8	633.0	3837.7
Min Delay (ps)	133	135	133	151	136

do this conversion, they are not plentiful enough to convert full payloads, and thus the conversion must be done on end hosts.

To quantify the overhead, we test how rapidly a single x86 core (running at 2.3 GHz) can perform endianness conversion for different floating point formats, using DPDK's highly-optimized APIs with "O3" optimization. Fig. 6 compares the measured results with the rate needed to achieve line-rate conversion at 100 Gbps. The gap is large, particularly for lower-precision values. In particular, to reach 100 Gbps for FP16, one will need at least 11 (*i.e.*, $\lceil$desired rate/single-core rate$\rceil$) cores. Hence, the high overhead of endianness conversion will lead to either low network throughput or extra CPU or GPU utilization. In many applications, these resources are not free; for instance, in DNN training, CPUs are often busy with data preprocessing.

4.2 PISA Architectural Extensions

To avoid these problems, we propose to extend the PISA architecture with some additional support. We show that the cost of these additions is low by extending the Banzai switch architecture model [102] and demonstrating that the increase in chip area, power, and timing budget is not significant.

2-operand shift instruction. We propose to enhance the existing shifter by allowing the shift distance operand to come from metadata instead of an immediate. The proposed instruction format is `shl/shr reg.distance, reg.value`. This little-effort enhancement will significantly improve the resource efficiency of FPISA, since the shifter can directly take the table match result as operand, and two instructions (left- and right-shift) can handle all the cases.

Combined shift+add operation in one stage. If the switch can support an atomic "shift+add" operation on a register in

a single stage, we will be able to swap the mantissa, with no compromise of potential error.

In-parser hardware-based endianness conversion. Endianness conversion in the hardware is straightforward and cheap – pure combinational logic shuffling the wires. We propose a simple enhancement to the switch's parser and deparser to implement this. Specifically, we propose a P4 type annotation `@convert_endianness`, applied to entire headers, that indicates to the compiler that the parser and deparser should convert the header fields' endianness as they enter and leave the pipeline. The parser will store the corresponding result to the metadata along with a implicit tag bit adjacent to the header's valid bit. When the packet is re-assembled, the deparser will check this tag bit to determine the byte order to be emitted.

To evaluate the cost of the first two changes (the last change has near-zero cost), we modify the open-source Banzai [102] switch architecture, a PISA-like design. We modify the Verilog code for Banzai's ALU to support our proposed shift instruction and synthesize it using Synopsys Design Compiler [107] with the FreePDK 15nm FinFET standard-cell library [73], a technology node similar to that used by the Tofino. We first check whether the design can operate at 1 GHz, evaluate its power and area, and then search the minimum critical-path delay of each design to find the impact of our modifcation on timing. As the results in Tab. 2 show, an enhanced ALU may use 13.0% more power and 22.4% more area than the original ALU, while slightly increasing the minimum delay. The overhead mainly comes from connecting and storing the second operand in the shifter. We implement a stateful read-shift-add-write (RSAW) unit based on Banzai's atomic predicated read-add-write (RAW) unit. The synthesis results in Tab. 2 demonstrate that the RSAW unit uses 13.6% more power and 35.0% more area than the regular RAW unit. In terms of minimum delay, RASW is 13.5% longer than RAW, but still far from the 1ns bound at 1 GHz. Banzai provides implementations only for the functional units, not for the entire switch chip, so we are unable to directly evaluate the impact of our modifications on the full chip design. However, prior work suggests that ALUs take up only a small portion (*i.e.*, ∼ 10%) of the power/area budget for the entire chip [9]; from this we infer that our modifications would have negligible impact. In other words, this hardware enhancement is feasible today, and is unlikely to become a bottleneck in future hardware generations.

Finally, to compare our approach with one that includes specialized floating-point units (like the Mellanox Quantum switch [32, 76]), we synthesize an ALU that includes a hard floating point unit. The ALU+FPU column in Tab. 2 shows the result: the hard FPU is more than five times larger and more power hungry than either the default ALU or the FPISA ALU. Its high area and leakage power are costs that must be paid even when the FPU is not in use, making it challenging for a switch chip including these features to be competitive with ordinary switches in terms of efficiency, and forcing vendors to maintain separate specialized switch designs for different applications.

Conversely, the FPISA approach allows the same ALUs to support both floating-point and non-floating-point computations, enabling a single switch chip design to support both floating-point and non-floating-point workloads efficiently.

4.3 FPISA-A: FPISA on Existing Architectures

The architectural changes described above allow us to implement the full FPISA approach. We additionally want a solution that allows FPISA to run on existing Tofino switches. Achieving this requires addressing the shift-and-add limitation. (The other two, while important, impact only resource utilization.) We provide a way to approximate FPISA on existing switches by avoiding the problematic shift. This approximation, which we call FPISA-A, can lead to inaccuracies for certain patterns of inputs, though we show later that it is not a problem for some applications, including in-network aggregation for ML training workloads (Sec. 5).

Recall that the problem arises because the alignment phase may require shifting the in-memory mantissa value to align it with the value to be added, which conflicts with the need to perform addition on the same value. Note that this is not a problem when the in-memory value has a larger exponent than the in-metadata value, as only the smaller of the two is right shifted. Taking advantage of FPISA's tolerance for denormalized representations, FPISA-A *always shifts the in-metadata mantissa* rather than the in-memory value. That is, if the in-metadata value is larger than the in-memory value, we keep the exponent unchanged and left-shift the in-metadata mantissa.

This approach works, within a certain range, because FPISA internally uses wider registers for the mantissa than the basic floating point representation. For FP32, IEEE 754 uses a 23-bit mantissa, while FPISA stores it in a 32-bit register. This gives 7 bits of *headroom*, after accounting for the implicit 1-bit and the sign bit. If the value being added is *much* larger than the in-memory value, i.e., its magnitude is greater by a ratio of more than $2^7 = 128$, the headroom would be exceeded. Instead, we detect this case during the exponent comparison (MAU2 in Fig. 2) and replace the in-memory value entirely with the in-metadata one. Doing so introduces numeric error in the low-order bits.

The FPISA-A variant is supported by the current commodity Tofino switch. As described above, it can introduce numeric error (which we call "overwrite" error). However, the error only occurs when input values vary widely in magnitude, and is bounded by the difference between headroom and mantissa width. For some applications, this approximation poses little difficulty: as we demonstrate in Sec. 5, ML model training gradients generally have a relatively narrow exponent range, and the workload is in any event resilient to small inaccuracies. For others, it may be more problematic; in those cases, the architectural modifications of Sec. 4.2 will be needed.

5 Case Study: Distributed ML Training

As the model and dataset sizes have increased for ML training jobs, large-scale distributed training has become increasingly important [1, 12, 13, 21, 33, 38, 40, 41, 47, 67, 81, 91, 113]. In this paper, we focus specifically on data-parallel training, a common approach to distributed training.[3] In data-parallel training, the dataset is partitioned to multiple worker machines, each with a replica of the model. In a training iteration, each machine performs learning on its local dataset and model, generating gradient vectors. These gradient vectors are then used to update the weights that make up the model. Modern supervised ML typically employs stochastic gradient descent (SGD) [82, 83, 95] or its variants as the optimizer for iterative training. In general, the core operation of SGD is as follows:

$$weight_{(next)} = weight_{(current)} - learning\_rate \cdot gradient_{(current)},$$

where $gradient_{(current)}$ is the element-wise mean of all the local gradient vectors produced by each worker. Computing this mean requires summing (or aggregating) gradient vectors from all workers.

Prior work has observed that, as the number of workers and the size of the model grows, communication costs – specifically, the gradient aggregation procedure – increasingly become a bottleneck in distributed training [70, 71, 98, 118]. Gradient aggregation can be viewed as an "all-reduce" collective operation, a familiar concept from the HPC world – the gradient vectors are **gathered** from all worker machines, **reduced** to one vector, and **sent back** to all worker machines. It is traditionally implemented either using a parameter server [67] or a distributed reduction protocol like ring all-reduce [6, 90].

In-network aggregation has been proposed as a promising way to accelerate this collective operation, and thus distributed training [2, 27, 31, 32, 57, 61, 70, 98]. In-network aggregation performs the "reduce" (*i.e.*, sum) step of all-reduce in a network switch on the fly. This offers higher throughput and lower latency than a parameter server approach, where both the network link and host-side network stack can become bottlenecks. Compared to ring-based and other distributed all-reduce algorithms, in-network aggregation requires exchanging fewer messages, again reducing latency and network usage.

PISA switches are well suited for, and have been used for, implementing in-network aggregation without specialized hardware. A major challenge, however, is the lack of floating point support. The recent state-of-the-art in-network aggregation work, SwitchML [98], works around this by quantizing floating point values at end hosts so that the PISA switch only operates on fixed-point values. While this quantization approach has been shown not to impact accuracy [98], we show that it harms performance. In particular, quantization and format conversion requires significant CPU overhead on the worker hosts. Computing the scaling factor to use for each block also requires

---

[3]Other parallel modes, like model-parallel, may also benefit from what is discussed in this work, but we do not explore them here.

an additional network round trip. Both costs could be avoided if the switch could operate on floating point values directly.

5.1 Setup

Environments. Given the hardware constraints of the current Tofino ASIC described in Sec. 4.1, we are not able to evaluate FPISA's applicability/benefit on the distributed ML training scenario entirely on a real system. Hence, we employ different evaluation approaches for different aspects of the process.

Specifically, to measure training accuracy and the impact of error, we write a C library that simulates gradient aggregation using a faithful implementation of the FPISA-A addition algorithm and integrate this C library into PyTorch [89] to train the models. We use the apex [85] PyTorch extension to evaluate both FP32 and FP16 floating point formats. Experiments and plots with this approach are labeled with "**[SIMULATION]**".

To analyze the numerical characteristics of the trained models' gradients and measure training throughput, we use an 8-machine cluster where each node is equipped with one NVIDIA P100 16 GB GPU, two 10-core Intel Xeon E5-2630v4 2.2 GHz CPUs, and 128 GB of DRAM with data served from local SSD storage. The cluster is networked at 100 Gbps and includes one Tofino-based Wedge100BF-65X programmable switch. This cluster deploys in-network aggregation through SwitchML [98].

For gradient numerical analysis, we directly dump the gradient vectors during the training processes. In these experiments, the workers compute gradients in the FP32 floating point format. Experiments and plots with this approach are labeled with "**[TRACE]**".

For performance (speedup) evaluation, we seek to measure the performance that FPISA-A can achieve with our variable-length shift extension, which allows multiple parallel FPISA-A instances per pipeline. Because current switch hardware does not support this, we emulate FPISA-A-enabled performance by removing the end-host format conversion/quantization at the workers and performing integer computations in place of floating point computations on the switch. While this emulation setup gives nonsensical output, it provides a realistic expectation of FPISA-A performance because: (1) under Tofino, data plane programs experience a switch processing latency that depends only on the number of stages and not on the computation intensity of their specific operations, without any effect on throughput (data plane programs operate at line rate) as confirmed experimentally in previous work (e.g., [17]); (2) SwitchML uses the full set of stages on the ingress pipelines of Tofino and any potential increase of in-switch latency can be mitigated by increasing the number of aggregation slots. Note that we use this approach only for performance evaluation, and it runs on the testbed configuration described above. Experiments and plots with this approach are labeled with "**[EMULATION]**".

Benchmarks. We select seven popular state-of-the-art ML models. These models are MobileNetV2 [96], VGG19 [101],
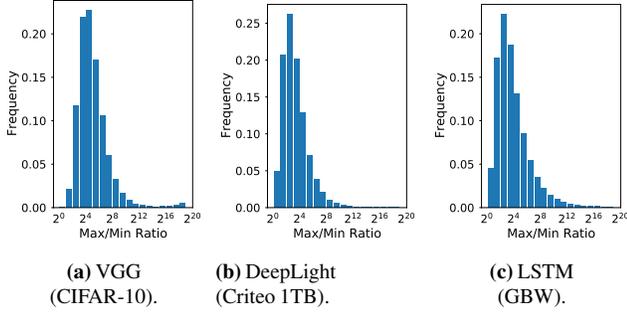
(a) VGG (CIFAR-10). **(b)** DeepLight (Criteo 1TB). **(c)** LSTM (GBW).

Figure 7: [TRACE] Element-wise max/min ratio distribution of different models (datasets).

(a) Epoch #1. **(b)** Epoch #20. **(c)** Epoch #40.

Figure 8: [SIMULATION] FPISA-A's error distribution of VGG19 gradient aggregation at early, middle, and final training stages.

ResNet-50 [37], GoogleNet [108], LSTM [52], DeepLight [22], and BERT [24]. We use all of these to evaluate training throughput, but evaluate accuracy only for the first four, since emulating FPISA-A in software is costly and those four CNNs train much faster than the other models. For CNN models, we use the CIFAR-10 dataset [59] with a learning rate of 0.1, momentum of 0.9, and weight decay of 0.0005. For other models, we use the same setting as in the SwitchML evaluation [98]. Regarding the batch size, for the accuracy experiments, we use a batch size of 16 because small batches represent a worst-case configuration from an accuracy standpoint; for the performance experiments, we use the standard batch sizes of each model listed in the MLPerf benchmark [80] and the SwitchML work [98] (*i.e.*, 2^{13} for DeepLight, 4 for BERT and 64 for others).

5.2 Characteristics of Training Gradients

The gradient aggregation workload has some common numerical characteristics that make it well suited for in-network aggregation with FPISA. In particular, FPISA can be used with existing Tofino switches using the FPISA-A approximation (Sec. 4.3); the resulting numerical error is rare and (as we demonstrate) has no impact on training accuracy.

High aggregation parallelism. In general, for each training iteration, the entire gradient vector corresponding to the training model needs to be aggregated from all worker machines. These vectors can range from several MBs to GBs. Aggregation is just vector addition; this element-wise summation provides ample parallelism.

Vector-wise distribution. As studied in INCEPTIONN [71], gradient values in each vector largely fall in the narrow range of $[-1,1]$, and most are close to "0".

Element-wise distribution. We find that for the same element from different workers' gradient vectors at the same iteration, the relative range is also narrow. To demonstrate this, we analyze the distribution of element-wise *max/min* ratio among eight workers' gradient vectors of the training of three models and datasets (see Sec. 5.3 for detailed setup and configuration), and plot the results at the early training phase (*i.e.*, the first epoch) in Fig. 7 (we have observed similar distributions
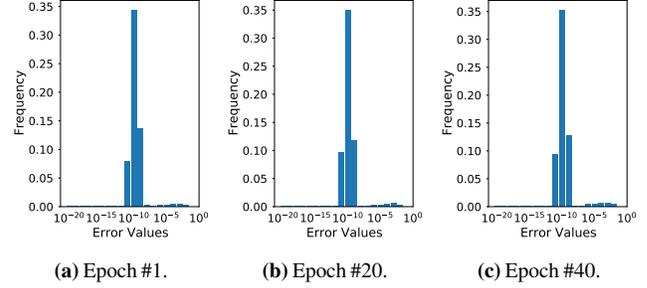
through the mid/final phases of the training). We find that, regardless of the model and dataset, most ($\sim 83\%$) elements' *max/min* ratio is smaller than 2^7.

Precision loss/error tolerance. It is well known that small floating point error does not dramatically affect the convergence and final accuracy of ML models [15, 19, 23, 71]. This observation has motivated extensive prior research about training with low or mixed-precision floating point operations [19, 25, 46, 50, 79, 115] and compression or quantization [35, 39, 44, 71].

Thanks to these numerical characteristics, FPISA-A addition can be directly applied to the in-network aggregation scenario on current Tofino switches. As discussed in Sec. 4.3, the lack of a shift-and-add operation introduces error only when adding values that differ by more than a 2^7 ratio – which Fig. 7 shows is rare – and the workload can tolerate such error. We show later that it has no impact on model accuracy or convergence. However, as discussed in Sec. 4.1, the cost of shift operations does mean the current Tofino only accommodates one FPISA-A module per pipeline. Hence, in-network aggregation performance will benefit from the variable-length shift enhancement we propose.

5.3 Evaluation

We take a two-step approach to our evaluation. (1) We first show that FPISA-A addition will not affect the training convergence (*i.e.*, FPISA-A will not incur more training iterations), and do not consider time-wise performance. (2) We demonstrate that FPISA-A can reduce the time of each training iteration and do not consider the convergence (because it is agnostic to per-iteration time). Taken together, we conclude that FPISA-A reduces end-to-end training time.

5.3.1 FPISA-A Error Analysis

To investigate the errors to which FPISA-A addition may lead, we record the gradient vectors from eight workers during a training job. We use the C library to compare the results of FPISA-A vs. standard floating point addition for aggregating the same gradient vectors. Fig. 8 shows the (absolute) error distribution of VGG19 during different training phases.

Similar to the gradient distribution [71], the error distribu-

tion remains similar among early, middle, and final phases of training, showing Fpisa-a's wide applicability. Most errors (>95%) are in the range of $[10^{-10}, 10^{-8}]$, enough to be tolerated by ML training, which we demonstrate in the next section. We further investigate the sources of the errors and find that most errors come from rounding, while the errors caused by the overwrite and left-shift mechanisms happen rarely (less than 0.9% and 0.1%, respectively, among all the addition operations in the aggregation procedure). These errors arise because, in some cases, a gradient vector's element-wise distribution is larger than Fpisa-a's left-shift headroom. As a result, the smaller values may be ignored in the aggregation procedure, leading to small errors (*i.e.*, smaller than 10^{-8}).

Note that a switch implementing the full Fpisa proposal, rather than just Fpisa-a, would not experience these overwrite errors. Note also that no overflow occurs in this experiment, since the number of workers, and thus the number of operations per vector element, is less than the headroom available in the mantissa register.

5.3.2 Fpisa-a's Impact on Training Convergence

We investigate whether Fpisa-a will lead to training accuracy loss, due to the errors it imposes. We train four ML models for 40 epochs with both default and Fpisa-a addition in gradient aggregation. To show Fpisa-a's adaptability on different floating point formats, we train using both standard single-precision FP32 and half-precision FP16 for each model.

We plot the accuracy value during the training procedures of each model in Fig. 9 to observe Fpisa-a's impact on convergence. Note that the jitters in the curves are due to the small batch size we are choosing; these are normal and do not affect the training procedure. First, we find that floating point precision does affect the training convergence. That is, in all four models, we observe slower convergence of FP16-based training compared to regular FP32-based training, as well as the final accuracy. However, Fpisa-a's addition errors will not amplify such gaps. In most cases, the curve of Fpisa-a addition is closely aligned with the curve of default addition. After 40 epochs, the accuracy differs by less than 0.1%. The results also demonstrate that regardless of the floating point format, Fpisa-a addition will not degrade each model's accuracy. Hence, we argue that Fpisa-a will not prolong the training by adding necessary epochs to converge.

5.3.3 Training Speedup with Fpisa-a

In the next experiments, we evaluate the potential speedup of Fpisa-a in an end-to-end training setting as well as the resulting reduction of host-based quantization overheads. SwitchML uses CPU cores at workers to scale and transform the numeric representation of gradient vectors, including both floating-point/integer conversion and byte order conversion. In contrast, Fpisa-a does not have these overheads as it sends gradient vectors as floating point values directly. As described in Sec. 5.1, from an end-to-end perspective, this is the sole source of expected performance variation between SwitchML and Fpisa-a. Thus, we vary the number of CPU cores and measure the throughput differences between these approaches through a microbenchmark.

In this microbenchmark, two workers reduce a 1 GB gradient vector;[4] we measure the time to complete the operation across the workers. We use 256 element packets which is the largest that SwitchML supports. After 50 warm-up invocations, we perform 250 reductions and report median and 10th-90th percentiles as the error bars.

SwitchML baselines. We use SwitchML's RDMA transport since it is more efficient that the DPDK one, and we run two versions to explore the performance implications of scaling and transforming gradient vectors on either the CPU or the GPU (where gradients are initially computed and stored). The base SwitchML version – denoted SwitchML/CPU – uses CPU cores. This benchmark assumes that the gradient vectors are already in host memory. Further, we create a new version of SwitchML – denoted SwitchML/GPU – that uses the GPU to scale and transform gradient vectors to the integer representation before copying them to pinned host memory.

Recall that SwitchML scales the gradient vectors in chunks, using a scaling factor adapted to each chunk based on a maximum exponent calculation that involves a round trip over the network. SwitchML saves the maximum exponent calculation's network overhead by overlapping the aggregation of the current chunk with the exponent calculation of the next chunk.

For SwitchML/CPU, we keep the original SwitchML logic where one chunk is equivalent to the RDMA message size. For SwitchML/GPU, we use a separate CUDA stream for each CPU core to allow parallel kernel execution. We also introduce a performance optimization where we asynchronously de-quantize aggregated messages from integer into floating point values on a separate CUDA stream thus having two CUDA streams for each CPU core. Despite these optimizations, there is an inherent overhead with launching one GPU kernel for each chunk. One potential way to avoid this could be to execute the per-chunk maximum exponent calculation as a pre-processing operation before the in-network aggregation phase; we leave this to future work.

Fpisa-a approaches. We run our Fpisa-a emulation in three settings. (1) Fpisa-a/CPU directly adopts the RDMA implementation of SwitchML and disables host-based type conversions. SwitchML's RDMA implementation, however, involves a CPU memory copy operation into a staging area. This memory copy is not necessary in the case of Fpisa-a since it can operate entirely on memory-resident native FP vectors without quantization; thus, we include a further optimization – (2) Fpisa-a/CPU(Opt) – that omits this extra memory copy. Lastly, (3) Fpisa-a/GPU (for comparison

---

[4] We use two workers to exclude the synchronization variability among a larger number of workers. This is to better quantify the performance differences due to the scaling and transformation overheads. We also tried 100 MB with similar results.
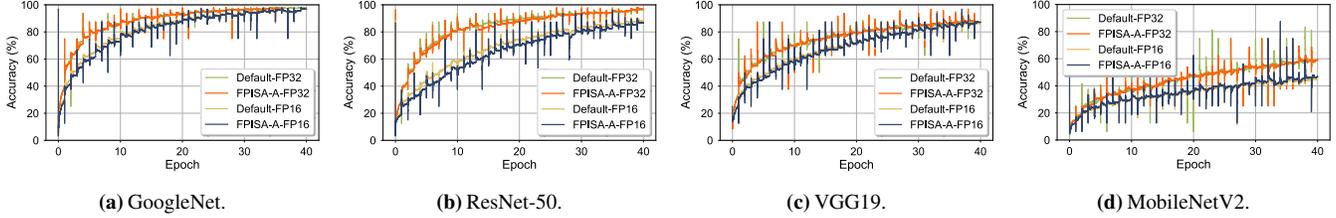
(a) GoogleNet.　　　　　**(b)** ResNet-50.　　　　　**(c)** VGG19.　　　　　**(d)** MobileNetV2.

Figure 9: [SIMULATION] Accuracy curves of different ML models with default addition and FPISA-A addition.

Cores vs Goodput (CPU) (16KB Msgs)　　　Message Size vs Goodput (CPU) (4 Cores)　　　Message Size vs Goodput (GPU) (4 Cores)
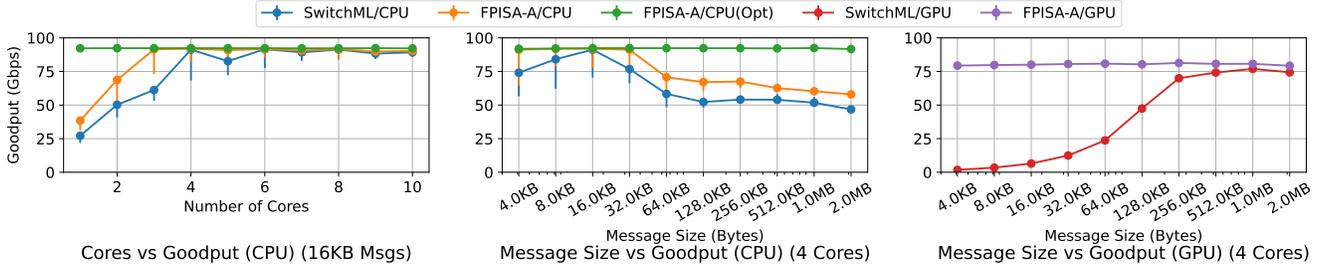
Figure 10: [EMULATION] Goodput of different floating point approaches on microbenchmark. The maximum theoretical goodput with framing overhead is 92 Gbps.
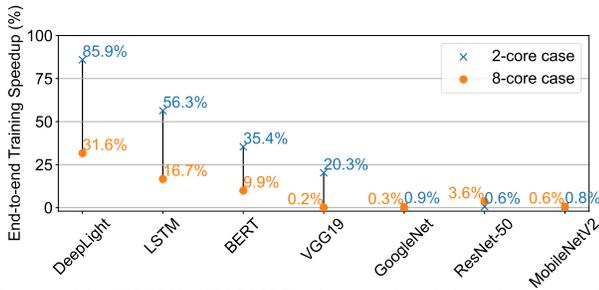

Figure 11: [EMULATION] End-to-end training time speedup of FPISA-A compared to the default SwitchML.

against SwitchML/GPU) includes a copy from GPU memory to pinned host memory and back.[5]

Because FPISA-A operates directly on FP vectors, we introduce two performance optimizations for FPISA-A/GPU that are not applicable to SwitchML/GPU (due to the need for chunk-based quantization). First, we use batching to amortize the cost of launching one copy operation for each chunk. Second, we asynchronously copy from GPU to host memory as a pipeline of one batch ahead of what needs to be consumed. Further, similar to the SwitchML/GPU case, we asynchronously copy back from host to GPU memory on a separate CUDA stream.

In-network aggregation goodput. Fig. 10 (left) shows that FPISA-A/CPU requires three CPU cores to achieve the 92 Gbps maximum goodput, as opposed to SwitchML/CPU, which needs four cores.[6] FPISA-A/CPU(Opt) achieves the maximum goodput with just a single core. This leaves more CPU cycles for data I/O, potentially avoiding training job stalls while waiting for input data to be preprocessed.
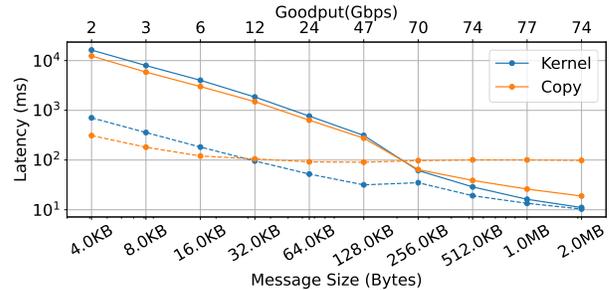

Figure 12: [EMULATION] SwitchML/GPU overheads at each iteration of the microbenchmark. To achieve high goodput, a message size of 256 KB or beyond is necessary. At smaller message sizes, the kernel and copy launches (solid lines) introduce a substantial latency compared to the actual kernel execution or copy latency (dashed lines).

The message size for this benchmark is 16 KB, which allows SwitchML/CPU to reach peak performance, according to the SwitchML paper [98]. Fig. 10 (middle) illustrates that FPISA-A achieves maximum goodput for a wide range of message sizes.

For the GPU variants, we find that the message/chunk size is the most important factor. Fig. 10 (right) shows that SwitchML/GPU is inefficient with message sizes below 256 KB. This is due to overheads of GPU kernel launches and copies at small message sizes (cf. Fig. 12). Increasing the number of cores does not help because CUDA implicitly synchronizes all kernel launch calls (kernel execution can be parallelized whereas kernel launches cannot). In contrast, using just a single CPU core, FPISA-A/GPU achieves the best possible performance – limited to 80 Gbps only by the bidirectional copy bandwidth of the GPU copy engines – since it can copy chunks in larger batches.[7] We expect that without

---

[5] Our testbed does not support GPU Direct, which would enable FPISA-A to use RDMA transfers out of and into GPU memory.

[6] SwitchML/CPU with 5 cores has a small performance dip due to work imbalance across cores in this particular configuration.

[7] We copy memory using 1 MB chunks as it gives the best results irrespective of the RDMA message size.

this bidirectional copy bandwidth limit (a constraint of our environment), FPISA-A/GPU would match the performance of FPISA-A/CPU(Opt) since it completely overlaps the memory copying with CPU and network operations.

SwitchML/GPU with a chunk size of 1 MB reaches a performance comparable (but still below) to FPISA-A/GPU. However, this requires an equally large RDMA message size whereas FPISA-A/GPU performs well even with 4 KB messages. Using large message sizes has several negative implications. First, it can introduce larger errors in SwitchML's quantization scheme since it chooses the scaling factor from a larger chunk. Second, it hurts the performance of loss recovery because the loss of a single packet entails resending the entire 1 MB message (1024 packets). Third, the performance degrades past a certain message size. This is due to limited network capacity and the reduction of pipelining, which in turn reduces the performance benefits of SwitchML's streaming aggregation. Thus, we conclude that, although performing quantization on the GPU might still be an interesting possibility for SwitchML, more work is necessary to devise an efficient implementation without increasing quantization errors and without affecting the GPU's availability for training.

Training throughput. We now confirm that FPISA-A's benefits translate into higher end-to-end training throughput. Fig. 11 reports the training throughput for seven real-world DNN benchmarks. For these experiments, we restrict the comparison to the DPDK implementation because SwitchML/RDMA is not currently integrated into the ML frameworks [98]. We focus on two scenarios – using either two or eight cores – and we measure the speedup in terms of training throughput (samples/s). We observe that FPISA-A speeds up training by up to 85.9% and 31.6% for the 2-core case and the 8-core case, respectively. Importantly, the higher speedup factors are obtained when using just two cores for communication, which frees up six cores for data I/O in this setting. The speedup is particularly significant in communication-bottlenecked models (*e.g.*, DeepLight, LSTM, BERT, VGG19), where FPISA-A is up to 85.9% faster compared to SwitchML when using the same number of cores. On the other hand, we do not see significant benefits of FPISA-A on models like GoogleNet, MobileNetV2, and ResNet-50, which are compute-bottlenecked.

By combining the accuracy results and the per-iteration end-to-end results, we can conclude that FPISA-A is able to reduce the end-to-end training time of a wide range of ML models.

6 Related Work

Accelerating distributed/networking applications with programmable switches. Recently, programmable switches have been used to accelerate a broad range of applications, including distributed key-value stores [49, 66, 112], distributed transactions [48, 64, 117], distributed storage [72, 120], packet queuing/scheduling [100, 103], network functions [56, 78], and network telemetry [7, 34, 105, 119]. While most of them

deal with packet header processing with few arithmetic operations, some perform computation on the packet's payload. SwitchML [98] and ATP [61] leverage switches for gradient aggregation but are constrained to fixed-point aggregation, which may lead to costly format conversion on the end-host and additional network round trips for exponent communication.

FPISA's approach is also applicable to other applications involving floating point operations and in-switch computing. For example, NETACCEL [63] and Cheetah [110] propose to use programmable switches to accelerate database queries by data pruning or query offloading. With the proposed architecture enhancements, FPISA can accelerate such queries with floating point as datatype. Also, other more complex floating point operations may be needed for future applications (*e.g.*, congestion control [26, 54] and network security [34]). Sec. 3.3 briefly discusses the possibility of supporting them.

Resource allocation. Much research has studied how to use in-network rate computations to support congestion control (e.g., XCP [54] and RCP [26]), queue management (e.g., CoDel [84] and AIFO [116]), or load balancing (e.g., CONGA [4]). P4QCN [29], P4-CoDel [60], and P4-ABC [77] are P4 implementations of specific protocols that require floating point support – currently unavailable in switch hardware. Sharma *et al*. proposed a library that applies approximation to work around this limitation [99]. InREC [51] and NetFC [16] proposed to use table-lookup for floating point operation emulation in programmable switches. However, they are constrained to stateless operations and need extra RAM space to store the tables. Also, few floating point operations can be done per packet, limiting parallelism. FPISA may enable new design options for in-switch resource allocation.

Extending switches' processing capability. Proposed enhancements to the RMT architecture [9] include transactions [102], disaggregated memory [14], and better stateful data plane support [28]. While many focus on improving stateful computations, none address floating point operations.

7 Conclusion

In this work, we propose FPISA, a floating point representation designed to work efficiently in programmable switches. We first implement FPISA on a commodity Intel Tofino switch, but its design limits throughput and accuracy. We then propose hardware changes based on the Banzai programmable switch architecture to avoid these limitations. We demonstrate their feasibility through synthesis using a 15-nm standard-cell library, and find minimal impact on area, power, and timing. Finally, we investigate the benefit of FPISA by implementing accelerators for distributed training application, evaluating its performance on a switch implementing our changes using emulation. We find that FPISA allows distributed training to use 25-75% fewer CPU cores and provide up to 85.9% better throughput in a CPU-constrained environment than the state-of-the-art framework.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, Nov. 2016.

[2] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, J. Brunheroto, C. Caşcaval, J. Castaños, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. Cipolla, P. Crumley, K. Desai, A. Deutsch, T. Domany, M. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. Haring, D. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. Jackson, T. Jamal-Eddine, G. Kopcsay, E. Krevat, M. Kurhekar, A. Lanzetta, D. Lieber, L. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. Moreira, B. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. Tremaine, M. Tsao, A. Umamaheshwaran, P. Verma, P. Vranas, T. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. Seager, J. Vetter, and K. Yates. An overview of the BlueGene/L supercomputer. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02)*, Baltimorem, MD, Nov. 2002.

[3] N. Alachiotis and A. Stamatakis. Efficient floating-point logarithm unit for FPGAs. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW'10)*, Atlanta, GA, May 2010.

[4] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM'14)*, Chicago, IL, Aug. 2014.

[5] Arista. 7130 FPGA-enabled Network Switches. https://www.arista.com/en/products/7130-fpga-enabled-network-switches-quick-look, accessed in 2021.

[6] M. Barnett, L. Shuler, R. van De Geijn, S. Gupta, D. G. Payne, and J. Watts. Interprocessor collective communication library (InterCom). In *Proceedings of the 1994 IEEE Scalable High Performance Computing Conference (SHPCC'94)*, Knoxville, TN, May 1994.

[7] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM'20)*, Virtual Event, Aug. 2020.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3), 2014.

[9] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the 2013 ACM SIGCOMM Conference (SIGCOMM'13)*, Hong Kong, China, Aug. 2013.

[10] Broadcom. NPL: Open, High-Level language for developing feature-rich solutions for programmable networking platforms. https://nplang.org/, accessed in 2021.

[11] Broadcom. Trident4 BCM56880 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, accessed in 2021.

[12] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, and C. Guo. Elastic parameter server load distribution in deep learning clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC'20)*, Virtual Event, Oct. 2020.

[13] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, Oct. 2014.

[14] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. dRMT: Disaggregated programmable switching. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM'17)*, Los Angeles, CA, Aug. 2017.

[15] M. Courbariaux, Y. Bengio, and J.-P. David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[16] P. Cui, H. Pan, Z. Li, J. Wu, S. Zhang, X. Yang, H. Guan, and G. Xie. NetFC: Enabling accurate floating-point arithmetic on programmable switches. In *Proceedings of the 29th IEEE International Conference on Network Protocols*, Virtual Event, Nov. 2021.

[17] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4), Aug. 2020.

[18] B. Darvish Rouhani, D. Lo, R. Zhao, M. Liu, J. Fowers, K. Ovtcharov, A. Vinogradsky, S. Massengill, L. Yang, R. Bittner, A. Forin, H. Zhu, T. Na, P. Patel, S. Che, L. C. Koppaka, X. Song, S. Som, K. Das, S. Tiwary, S. Reinhardt, S. Lanka, E. Chung, and D. Burger. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. In *Advances in neural information processing systems 33 (NeurIPS'20)*, Virtual Event, Dec. 2020.

[19] C. De Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. R. Aberger, K. Olukotun, and C. Ré. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383*, 2018.

[20] D. De Sensi, S. Di Girolamo, S. Ashkboos, S. Li, and T. Hoefler. Flare: Flexible in-network allreduce. *arXiv preprint arXiv:2106.15565*, 2021.

[21] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Advances in neural information processing systems 25 (NIPS'12)*, Lake Tahoe, NV, Dec. 2012.

[22] W. Deng, J. Pan, T. Zhou, D. Kong, A. Flores, and G. Lin. DeepLight: Deep lightweight feature interactions for accelerating CTR predictions in ad serving. *arXiv preprint arXiv:2002.06987*, 2020.

[23] A. Devarakonda, M. Naumov, and M. Garland. Adabatch: Adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029*, 2017.

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[25] M. Drumond, T. LIN, M. Jaggi, and B. Falsafi. Training dnns with hybrid block floating point. In *Advances in Neural Information Processing Systems 31 (NeurIPS'18)*, Montreal, Canada, Dec. 2018.

[26] N. Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. PhD thesis, Stanford University, Dept. of Electrical Engineering, 2007.

[27] N. Gebara, P. Costa, and M. Ghobadi. In-network aggregation for shared machine learning clusters. In *Proceedings of the 4th MLSys confrence (MLSys'21)*, Virtual Event, Apr. 2021.

[28] N. Gebara, A. Lerner, M. Yang, M. Yu, P. Costa, and M. Ghobadi. Challenging the stateless quo of programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*, Virtual Event, Nov. 2020.

[29] J. Geng, J. Yan, and Y. Zhang. P4QCN: Congestion control using P4-capable device in data center networks. *Electronics*, 8(3), 2019.

[30] Google Cloud. Using bfloat16 with TensorFlow models. https://cloud.google.com/tpu/docs/bfloat16, accessed in 2021.

[31] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction. In *Proceedings of the 1st Workshop on Optimization of Communication in HPC (COM-HPC'16)*, Salt Lake City, Utah, Nov. 2016.

[32] R. L. Graham, L. Levi, D. Burredy, G. Bloch, G. Shainer, D. Cho, G. Elias, D. Klein, J. Ladd, O. Maor, A. Marelli, V. Petrov, E. Romlet, Y. Qin, and I. Zemah. Scalable hierarchical aggregation and reduction protocol (SHARP) streaming-aggregation hardware design and evaluation. In *Proceedings of the 35th International Conference on High Performance Computing (ISC'20)*, Frankfurt/Main, Germany, June 2020.

[33] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, Boston, MA, Feb. 2019.

[34] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM'18)*, Budapest, Hungary, Aug. 2018.

[35] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[36] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth. A survey on data plane programming with P4: Fundamentals, advances, and applied research, 2021.

[37] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE conference on computer vision and pattern recognition (CVPR'16)*, Las Vegas, NV, June 2016.

[38] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *Advances in neural information processing systems 26 (NIPS'13)*, Lake Tahoe, NV, Dec. 2013.

[39] S. Horvath, C.-Y. Ho, L. Horvath, A. N. Sahu, M. Canini, and P. Richtarik. Natural compression for distributed deep learning. *arXiv preprint arXiv:1905.10988*, 2019.

[40] C. Hwang, T. Kim, S. Kim, J. Shin, and K. Park. Elastic resource sharing for distributed deep learning. In *Proceedings*

of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21), Virtual Event, Apr. 2021.

[41] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer. FireCaffe: Near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, Las Vegas, NV, June 2016.

[42] Intel Corporation. Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, accessed in 2021.

[43] Intel Corporation. Intel Tofino2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html, accessed in 2021.

[44] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA'18)*, Los Angeles, CA, June 2018.

[45] T. Jepsen, L. P. de Sousa, M. Moshref, F. Pedone, and R. Soulé. Infinite resources for optimistic concurrency control. In *Proceedings of the ACM SIGCOMM 2018 Workshop on In-Network Computing (NetCompute'18)*, Budapest, Hungary, Aug. 2018.

[46] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, S. Shi, and X. Chu. Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[47] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Virtual Event, Nov. 2020.

[48] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, Apr. 2018.

[49] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, Shanghai, China, Oct. 2017.

[50] J. Johnson. Rethinking floating point for deep learning. *arXiv preprint arXiv:1811.01721*, 2018.

[51] M. Jose, K. Lazri, J. François, and O. Festor. InREC: In-network real number computation. In *Proceedings of the 2021 IFIP/IEEE International Symposium on Integrated Network Management*, Virtual Event, May 2021.

[52] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[53] D. D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey. A study of bfloat16 for deep learning training. *arXiv preprint arXiv:1905.12322*, 2019.

[54] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *ACM SIGCOMM Computer Communication Review*, 32(4), 2002.

[55] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable load balancing using programmable data planes. In *Proceedings of the 2016 Symposium on SDN Research (SOSR'16)*, Santa Clara, CA, Mar. 2016.

[56] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. TEA: Enabling state-intensive network functions on programmable switches. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM'20)*, Virtual Event, Aug. 2020.

[57] B. Klenk, N. Jiang, G. Thorson, and L. Dennison. An in-network architecture for accelerating shared-memory multiprocessor collectives. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA'20)*, Virtual Event, May 2020.

[58] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1), 2014.

[59] A. Krizhevsky. The CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html, accessed in 2021.

[60] R. Kundel, J. Blendin, T. Viernickel, B. Koldehofe, and R. Steinmetz. P4-CoDel: Active queue management in programmable data planes. In *Proceedings of 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Verona, Italy, Nov. 2018.

[61] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift. ATP: In-network aggregation for multi-tenant learning. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, Virtual Event, Apr. 2021.

[62] A. S. Leon, K. W. Tam, J. L. Shin, D. Weisner, and F. Schumacher. A power-efficient high-throughput 32-thread SPARC processor. *IEEE Journal of Solid-State Circuits*, 42(1), 2007.

[63] A. Lerner, R. Hussein, and P. Cudre-Mauroux. The case for network accelerated query processing. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR'19)*, Asilomar, CA, Jan. 2019.

[64] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, Shanghai, China, Oct. 2017.

[65] J. Li, E. Michael, A. Szekeres, N. K. Sharma, and D. R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, Nov. 2016.

[66] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Virtual Event, Nov. 2020.

[67] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, Oct. 2014.

[68] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*, New York, NY, Dec. 2009.

[69] Y. Li and W. Chu. Implementation of single precision floating point square root on FPGAs. In *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*, Apr. 1997.

[70] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang. Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*, Phoenix, AZ, June 2019.

[71] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. G. Schwing, H. Esmaeilzadeh, and N. S. Kim. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In *Proceedings of the 51st International Symposium on Microarchitecture (MICRO'18)*, Fukuoka, Japan, Oct. 2018.

[72] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica. DistCache: Provable load balancing for large-scale storage systems with distributed caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*, Boston, MA, Feb. 2019.

[73] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen. Open cell library in 15nm FreePDK technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design (ISPD'15)*, Monterey, CA, Mar. 2015.

[74] S. Mathew, M. Anders, B. Bloechel, T. Nguyen, R. K. Krishnamurthy, and S. Borkar. A 4-GHz 300-mW 64-bit integer execution ALU with dual supply voltages in 90-nm CMOS. *IEEE Journal of Solid-State Circuits*, 40(1), 2005.

[75] Mellanox. Mellanox scalable hierarchical aggregation and reduction protocol (SHARP). http://www.mellanox.com/page/products_dyn?product_family=261&mtag=sharp, accessed in 2021.

[76] Mellanox. QM8700 Mellanox Quantum HDR Edge Switch. https://www.mellanox.com/files/related-docs/prod_ib_switch_systems/PB_QM8700.pdf, accessed in 2021.

[77] M. Menth, H. Mostafaei, D. Merling, and M. Häberle. Implementation and evaluation of activity-based congestion management using P4 (P4-ABC). *Future Internet*, 11, 2019.

[78] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM'17)*, Los Angeles, CA, Aug. 2017.

[79] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[80] MLCommons. Mlperf benchmark. https://mlcommons.org/en/training-normal-10/, accessed in 2021.

[81] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. SparkNet: Training deep networks in Spark. *arXiv preprint arXiv:1511.06051*, 2015.

[82] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization*, 19(4), 2009.

[83] A. S. Nemirovsky and D. B. Yudin. *Problem complexity and method efficiency in optimization*. Society for Industrial and Applied Mathematics, 1983.

[84] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar. Controlled delay active queue management. RFC 8289, 2018. https://tools.ietf.org/html/rfc8289.

[85] NVIDIA. apex: Tools for easy mixed precision and distributed training in Pytorch. https://github.com/NVIDIA/apex, accessed in 2021.

[86] NVIDIA blog. TensorFloat-32 in the A100 GPU accelerates AI training, HPC up to 20x. https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/, accessed in 2021.

[87] S. Oberman. Floating point division and square root algorithms and implementation in the AMD-K7 microprocessor. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, Adelaide, Australia, Apr. 1999.

[88] OpenSwitch. Cavium-XPliant family of programmable ethernet switches. https://www.openswitch.net/cavium/, accessed in 2021.

[89] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems 32 (NIPS'19)*, Vancouver, Canada, Dec. 2019.

[90] P. Patarasuk and X. Yuan. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 69(2), 2009.

[91] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, Huntsville, Canada, Oct. 2019.

[92] Y. Piasetzky, M. Kadosh, M. Pritsak, O. Shabtai, A. Lo, and G. Lu. Switch asic programmability in hybrid mode. In *Proceedings of 2018 IEEE 26th International Conference on Network Protocols (ICNP'18)*, Cambridge, UK, Sept. 2018.

[93] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in datacenter networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, Oakland, CA, May 2015.

[94] D. R. K. Ports and J. Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19)*, Bertinoro, Italy, May 2019.

[95] H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, 1951.

[96] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proceedings of the 2018 IEEE conference on computer vision and pattern recognition (CVPR'18)*, Salt Lake City, UT, June 2018.

[97] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th Workshop on Hot Topics in Networks (HotNets'17)*, Palo Alto, CA, Nov. 2017.

[98] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, Virtual Event, Apr. 2021.

[99] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, Boston, MA, Mar. 2017.

[100] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, Apr. 2018.

[101] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[102] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*, Florianopolis, Brazil, Aug. 2016.

[103] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*, Florianopolis, Brazil, Aug. 2016.

[104] P. Soderquist and M. Leeser. Area and performance tradeoffs in floating-point divide and square-root implementations. *ACM Computing Surveys*, 28(3), 1996.

[105] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys'18)*, Porto, Portugal, Apr. 2018.

[106] A. Svyatkovskiy, J. Kates-Harbeck, and W. Tang. Training distributed deep recurrent neural networks with mixed precision on GPU clusters. In *Proceedings of the Machine Learning on HPC Environments (MLHPC'17)*, Denver, CO, Nov. 2017.

[107] Synopsys. Design Compiler Graphical. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html, accessed in 2021.

[108] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the 2015 IEEE conference on computer vision and pattern recognition (CVPR'15)*, Boston, MA, June 2015.

[109] P.-T. P. Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4), 1990.

[110] M. Tirmazi, R. Ben Basat, J. Gao, and M. Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)* https://arxiv.org/pdf/2004.05076.pdf, Virtual Event, June 2020.

[111] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman. The case for in-network computing on demand. In *Proceedings of the 14th EuroSys Conference (EuroSys'19)*, Dresden, Germany, Mar. 2019.

[112] Y. Tokusashi, H. Matsutani, and N. Zilberman. LaKe: An energy efficient, low latency, accelerated key-value store. *arXiv preprint arXiv:1805.11344*, 2018.

[113] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: Distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*, Prague, Czech Republic, Apr. 2013.

[114] M. Voogel, Y. Frans, and M. Ouellette. Xilinx Versal Premium series. In *HotChips'20*, Virtual Event, Aug. 2020.

[115] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in neural information processing systems 31 (NIPS'18)*, Montreal, Canada, Dec. 2018.

[116] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM Conference (SIGCOMM'21)*, Virtual Event, Aug. 2021.

[117] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin. Net-Lock: Fast, centralized lock management using programmable switches. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM'20)*, Virtual Event, Aug. 2020.

[118] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML (NetAI'20)*, Virtual Event, Aug. 2020.

[119] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu. Flow event telemetry on programmable data plane. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM'20)*, Virtual Event, Aug. 2020.

[120] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. In *Proceedings of the 2019 International Conference on Very Large Data Bases (VLDB'19)*, Los Angeles, CA, Nov. 2019.