NetFC: Enabling Accurate Floating-point Arithmetic on Programmable Switches

Penglai Cui*†‡, Heng Pan*†§, Zhenyu Li†§, Jiaoren Wu¶, Shengzhuo Zhang¶, Xingwu Yang¶
Hongtao Guan† and Gaogang Xie‖
†ICT, CAS, China, ‡University of Chinese Academy of Sciences, China
§Purple Mountain Laboratories, Nanjing, China, ¶Kwai Inc., ‖CNIC, CAS, China
*Co-first authors

***Abstract*—Programmable switches are recently used for accelerating data-intensive distributed applications. Some computational tasks, traditionally performed on servers in data centers, are offloaded to the network on programmable switches. These tasks may require the support of on-the-fly floating-point operations. Unfortunately, the computational capacity of programmable switches is limited to simple integer arithmetic operations. To address this issue, prior approaches either adopt a float-to-integer method or rely on local CPUs of switches, incurring accuracy loss and delayed processing.**

To this end, we propose NetFC, a table-lookup method to achieve on-the-fly in-network floating-point arithmetic operations nearly without accuracy loss. NetFC adopts a divide-and-conquer mechanism that converts the original huge table into several much smaller tables that are operated by the built-in integer operations. NetFC further leverages a scaling-factor mechanism for improving computational accuracy, and a prefix-based lossless table compression method to reduce memory consumption. We use both synthetic and real-life datasets to evaluate NetFC. The experimental results show that the average accuracy of NetFC is above 99.94% with only 448KB memory consumption. Furthermore, we integrate NetFC into Sonata [12] for detecting Slowloris attack, yielding significant decrease of detection delay.

***Index Terms*—In-network computation, floating-point calculation, programmable switch**

I. Introduction

In modern data centers, many applications are data-intensive, such as big data analysis [10], distributed deep learning [27], graph processing [25] and real-time stream processing [5], [15]. Due to frequent data exchanging, these applications may suffer from performance degradation due to extensive network communication overhead. For example, in some of the FaceBook MapReduce jobs, network communication can occupy up to 70% of the execution time [6]; in distributed reinforcement learning, network overhead can be over 80% of the total cost in each iteration [24]. Thus, cutting down network communication is the key factor to accelerate data-intensive applications.

In-network computation has been recently explored for this purpose. The intuition behind this is that the network has been equipped with many new devices (e.g. programmable switches [4] and smart network interface [23]); that said the network has become capable of providing computational capacity. Consequently, some computational tasks, traditionally performed in the host side, can be offloaded to the network devices. In doing so, network traffic can be intercepted and processed by the network devices on the fly based on the pre-offloaded computational logics before it reaches the hosts. Indeed, researchers in the community have already utilized in-network computation to accelerate different distributed applications and achieve remarkable performance improvement. For example, NetCache [19] is co-designed with programmable switches to support over 2B queries per second; ATP [22] performs in-network gradient aggregation, which can improve the training throughput of existing distributed deep learning (DDL) systems up to 66%; Sonata [12] utilizes programmable switches to achieve fast in-band network telemetry.

Unfortunately, the computational capacity of the network is very limited, and even the state-of-the-art programmable switches (e.g. Barefoot Tofino [4]) only support simple integer arithmetic operations (e.g. addition and subtraction). This becomes a barrier to in-network acceleration of applications, because many of them often require processing sophisticated floating-point data and arithmetic operations (e.g multiplication and division). For example, ATP [22] needs to perform floating-point calculation on programmable switches for in-network gradient aggregation during DDL training; Sonata [12] measures the states of the network, where some measurement tasks (e.g. Slowloris attack detection [1]) require in-network multiplication or division.

To overcome the barrier, prior studies mainly adopt two different ways to support floating-point operations indirectly. One is to convert floating-point numbers into integers based on an sophisticated negotiation mechanism on the server side (e.g. SwitchML [29] and ATP [22]). But it does not support floating-point multiplication and division. The other solution is to offload the computational tasks to the local CPUs of switches (e.g. Sonata [12]). Nevertheless, this solution introduces significant delay (see section VI). In summary, to the best of our knowledge, there is a lack of a solution to achieve on-the-fly in-network floating-point arithmetic operations nearly without accuracy loss on programmable switches.

To address this gap, we design and implement NetFC that adopts a table-lookup method to support floating-point arithmetic operations on programmable switches. Intuitively, a simple and direct way is to use a table to enumerate all possible calculation cases ahead. In doing so, for one

978-1-6654-4131-5/21/$31.00 ©2021 IEEE

arithmetic operation, we can use its two operands as a key to look up the table where the corresponding value is the result. However, the generated table is too big to be installed on programmable switches, since it needs to traverse all operands and enumerate their various combinations. For example, for two 16-bit floating-point operands, it will cost about 8 GB memory. To this end, NetFC adopts a divide-and-conquer method. Specifically, it utilizes logarithm projection and transformation to convert the original large table into several much smaller tables that are operated with the built-in integer operations (i.e. addition and subtraction). NetFC further adopts a scaling-factor mechanism to improve computational accuracy, and relies on a prefix-based loss-less compression method to reduce on-chip memory usage. Experimental results demonstrate that the average accuracy of NetFC is above 99.94% with only 448KB memory consumption. In addition, we integrate NetFC into Sonata [12] for detecting and defencing Slowloris attack in real time; with NetFC, the detection latency is reduced from 43.16*ms* to 0.046*ms*.

To sum up, the contributions of this paper are three-fold:

- We design a table-lookup approach, NetFC, to achieve in-network floating-point arithmetic operations nearly without accuracy loss. It adopts a divide-and-conquer method to address the on-chip memory consumption problem.
- We propose a scaling-factor method to improve the computational accuracy of NetFC, and design a prefix-based loss-less compression mechanism to further reduce memory consumption.
- We implement NetFC based on Barefoot Tofino switches. Extensive experiments show that NetFC enables floating-point arithmetic operations on programmable switches with low on-chip memory usage. In addition, we integrate NetFC into Sonata to detect and defense SlowLoris attack in real time.

II. Background and Motivation

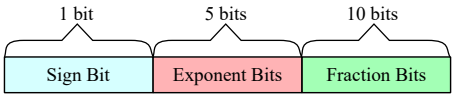
In this section, we first briefly describe in-network computation, and then introduce details of floating-point standards. At last, we give two typical examples to illustrate the limitation of prior arts that motivates our work.

A. Emerging Trends of In-network computation

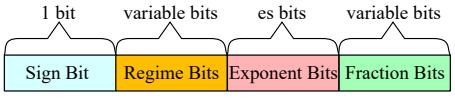
In-network computation that is built on top of programmable switches exploits the computational capacity of switches to offload part of computational tasks from the server side to the network. Barefoot Networks' Tofino switches [4] are one of the popular programmable switches that have been widely used in academy and industry. The Tofino switch chip has a flexible parser and a customized match-action forwarding engine. With the provided programming language and interfaces, network programmers are able to dynamically configure the switches in order to program the network. Tofino switches have two multi-stage pipelines: ingress pipelines and egress pipelines. Each pipeline stage has a fixed amount of time to process packets in memory (TCAM and SRAM). Tofino switches support boolean and simple arithmetic operations (i.e. integer addition and subtraction) using a set of ALUs, but they do not support complex operations (e.g. multiplication and division) and floating-point numbers.

In-network computation is appealing for several reasons: i) many packets can be consumed and processed during data transmission, which significantly reduces the overhead of the network (e.g. the network queuing latency and I/O overhead); ii) the computational workloads that are offloaded to the network can alleviate the burden of the server CPUs. For example, ATP [22], SwitchML [29] and iSwitch [24] accelerate distributed deep learning via in-network gradient aggregation on programmable switches; NetCache [19] caches data in the network; NetSHa [33] accelerates LSH-based distributed search. We also see some traditional network algorithms [17] (e.g. string matching) and network telemetry tasks [31], [34] (e.g. sketch) are deployed to the network.

B. Floating-point Arithmetic



(a) 16-bit float.



(b) 16-bit posit.

Fig. 1. Data format for 16-bit length floating points.

Two popular technical standards of floating-point computation have been widely adopted: the traditional IEEE 754 standard [2] and the posit standard [13]. Here, we briefly introduce these two standards.

IEEE 754 floating point. IEEE 754 float is the most widely used data type. An IEEE 754 16-bit float representation consists of three parts as shown in Figure 1(a): a sign bit, 5 exponent bits and 10 fraction bits. Let e be the unsigned integer represented by the exponent field. If the fraction bits are $f_1 f_2 ... f_s$, then $f = 1.f_1 f_2 ... f_s$. The value p of a 16-bit floating-point number that does not fall into any exception cases is given by:

$$p = sign * 2^{e-15} * f \tag{1}$$

Modern FPU (floating-point unit) implements floating-point addition, subtraction, multiplication and division as follows. For addition and subtraction, the FPU expresses operands with the same exponent and shift the mantissas accordingly; the shifted mantissas are then added together; for multiplication, the FPU adds the exponents of the two numbers and multiplies their mantissas; Likewise, for division, the FPU subtracts the divisor's exponent from the dividend's exponent, and divides the divisor's mantissa from the dividend's mantissa. The final outputs of the above floating arithmetic are obtained by rounding and normalizing the results.

Posit floating point. Posit is designed as a direct drop-in replacement for IEEE 754 floating-point numbers (floats) [13]. Figure 1(b) shows its data format when using 16-bit length to represent the points. Compared to float, posit data type has an additional regime field. Regime field begins with some number of all 0 or all 1 bits in a row, and terminates when the next bit is opposite, or the end of string is reached. Let es denote the width of exponent bits, which is determined by the size of posit floating-point number. For a 16-bit (32-bit resp.) posit, es = 1 (es = 3 resp.). Let k be the integer represented by regime field, e be the unsigned integer represented by the exponent field. If the fraction bits are $f_1 f_2 ... f_s$, then $f = 1.f_1 f_2 ... f_s$. The value of a posit number p can be represented as:

$$p = sign * 2^{2^{es}k} * 2^e * f \tag{2}$$

Compared to the IEEE 754 float, posit floating point has the following advantages [7], [9], [13].

- **Larger dynamic range.** Dynamic range represents a range from the minimum positive number to the maximum positive number that a number system can express. The dynamic range of 16-bit float is $6 * 10^{-8}$ to $7 * 10^4$, while the dynamic range of 16-bit posit is $4 * 10^{-9}$ to $3 * 10^8$. That said, with the same bitwidth, posit has a larger dynamic range.
- **No representations wasted for NaN or infinity.** For the 16-bit float, when the exponent bits are 11111, it represents NaN or infinity. That said, there are 2,048 representations used to represent Nan or infinity. However, for the 16-bit posit, it represents NaN or infinity only when the representation is 0x8000.
- **Tapered accuracy.** Posit numbers near 1 in magnitude have more accuracy than extremely large or extremely small numbers. This phenomenon is called "golden zone" in [9], in which the accuracy of posit is higher than float. For example, Posit32 has more fraction bits than Float32 numbers whose magnitude ranges from 10^{-6} to 10^6.

Due to the above characteristics, posit is considered to be very advantageous in deep learning by [7], [21], [32]. Indeed, the above two popular standards for floating-point calculation are widely used by different kinds of applications while our NetFC can work very well with both of them. In the following sections, we use float or floating-point (posit resp.) to refer to IEEE 754 floating point (posit floating point resp.).

C. Motivating NetFC

We are motivated by the fact that modern programmable switches only support simple integer arithmetic operations (addition and subtraction). However, floating-point operations in in-network computation are essential since a variety of tasks (e.g. gradient aggregation) offloaded to the network require floating-point operations. As such, they have to rely on other indirect 'layers' to implement floating-point arithmetic operations (either addition and subtraction, or multiplication and division) if needed. Nevertheless, these indirect 'layers' may be not general or increase the delay of the tasks. We take two examples here to illustrate this.

In-network gradient aggregation. In-network gradient aggregation is used to accelerate the training process of distributed machine learning systems [22], [29]. The basic idea is to cache gradients from training workers on programmable switches, and accumulate them when some conditions are met to get aggregated gradients. In this way, the number of gradient packets sent to the parameter servers are decreased, mitigating the communication overhead. Gradients are always floating-point numbers, which require the support of floating-point arithmetic operations when performing aggregation. As programmable switches are unable to support the operations, the current solution in [22] is to introduce a 'shim layer' on end hosts that coverts floating-point numbers to integers by multiplying a scaling factor (*sf*) on workers first; after aggregation by programmable switches (using integer arithmetic operations), the aggregated results will be forwarded to servers where they are restored back to floating-point representation by dividing *sf*. To improve accuracy, some proposals (e.g. SwitchML [29]) adopt a negotiation-based mechanism between workers to decide the value of *sf* during each gradient block transmission. However, it incurs extra overhead due to extensive negotiation, and does not support multiplication and division.

Inband Network Telemetry. Programmable switches are the 'sweet point' to implement network telemetry as they sit in the middle of network paths. We have seen plenty of network telemetry systems built on programmable switches [12], [14], [31]. Many measurement tasks (*e.g.* Slowloris attack detection) require the support of floating-point arithmetic operations (even multiplication and division). Because programmable switches do not support these operations, these systems adopt a slow-path solution, where the intermediate results that require floating-point arithmetic operations are sent to local CPUs for processing. This solution will inevitably introduce huge delay of the tasks. Let us consider the detection of Slowloris attacks [1] in Sonata as a detailed example [12]. The task monitors the traffic of all connections belonging to individual hosts to see whether the average traffic volume of each connection belonging to a host is less than a predefined threshold value. Apparently, this requires floating-point arithmetic operations and has to be implemented in switch local CPUs through the slow path, delaying the detection. That said, without the support of floating-point arithmetic operations in programmable switches, online detection and defense of attacks like Slowloris attacks cannot be implemented in current inband network telemetry.

Summary. The support of floating-point arithmetic operations in programmable switches are important and essential to enable practical application of in-network computation. To the best of our knowledge, such a support is overlooked by prior arts, which motivates our work.

III. Design of NetFC

NetFC aims at enabling sophisticated floating-point arithmetic on programmable switches nearly without accuracy loss and additional latency. In this section, we first describe the

basic idea, and then discuss the challenges of NetFC, and finally detail the design.

A. Design Choice

To fix ideas, we assume that 16-bit floating-point numbers follow IEEE 754 standard, but we will relax this assumption in Section III-C. We also assume the use of Barefoot Tofino switches. Intuitively, there are two potential ways to implement floating-point arithmetic on the data plane of programmable switches.

- **FPU method.** A floating-point number is represented as three portions: sign, mantissa and exponent. For floating-point addition (subtraction reps.), it should shift the mantissa of one floating-point operand so that its exponent is identical to that of the other operand. Finally, it adds (subtracts reps.) the two operand mantissas. For floating-point multiplication (division reps.), it needs to perform multiplication (division reps.) between the two operand mantissas and addition (subtraction reps.) between the two operand exponents.
- **Table-lookup method.** Let's use addition to illustrate this method. For any two 16-bit floating-point operands a and b, we perform an addition operation between the two operands and thus get its corresponding result z in advance. After this operation, we obtain a key-value pair whose key is a and b while z constitutes the value. Next we traverse all possible values of a and b to generate multiple key-value pairs and finally constitute an addition table. Subsequently, if we calculate the sum of any other two 16-bit floating-point numbers, we only need to use the two operands as a key to look up the table while the value of the matched table entry is the result. Of course, this method can also be generalized to other arithmetic operations.

However, the question remains: *can either of the above methods be deployed to programmable switches directly?* To explore this, we analyze the capacity limitations of programmable switches as follows:

i. *Limited computation capacity.* Programmable switches (e.g. Barefoot Tofino) only supports some simple integer arithmetic. That said, floating-point numbers and arithmetic operations of multiplication and division have exceeded the switch capacity.

ii. *Scarce on-chip memory.* Switch on-chip memory size is very small (e.g. tens of megabyte in Barefoot Tofino) so that it is impossible to provide huge memory for floating-point arithmetic. Note that, a portion of memory has to be reserved for forwarding rule storage and lookup, further aggravating this problem.

iii. *Limited pipeline stages.* The switch data plane often consists of a pipeline of stages, each of which is a packet processing unit equipped with some computing and storage resources. However, the number of stages is small (e.g. 32 stages in Barefoot Tofino at most), and any two dependent packet processing operations cannot be assigned to the same stage.

Now, let us return back to the FPU and Table-lookup methods. FPU requires on-the-fly variable shifting operations and needs to perform multiplication/division arithmetic. And thus, it is not possible to implement FPU in programmable switches. Thus we turn to the Table-lookup method.

The table-lookup method fits the programmable switches, which abstract the packet processing on data plane as tables that consist of match-action tuples. However, it does not work directly either due to a large amount of memory consumption: a 16-bit floating-point addition arithmetic would consume about 8 GB ($2^{16} \times 2^{16} \times 2B$) memory. Thus, the implementation finally nails down to how to reduce the memory consumption.

B. Divide-and-Conquer Table Lookup

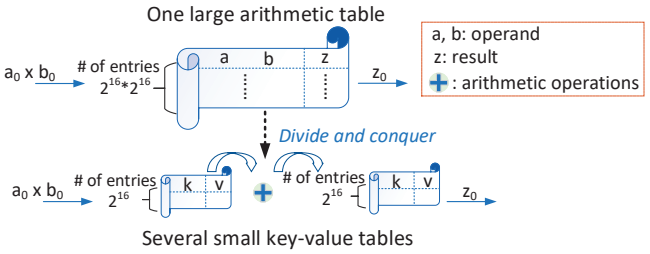


Fig. 2. Example of NetFC.

The original table-lookup method traverses all possible operands and their combinations, which finally constitutes a very large table. For example, considering two 16-bit operands, they will generate $2^{16} \times 2^{16}$ (a.k.a Cartesian product) table entries. Our NetFC adopts a divide-and-conquer approach to address this issue. Specifically, it utilizes logarithm projection and transformation to convert the original large table into several much smaller tables together with some simple integer arithmetic operations. We will provide the details in the subsequent sections. It is noteworthy that the numbers of small tables needed is dependent on floating-point arithmetics. For example, as shown in Figure 2, two small tables are used to replace the original large arithmetic table while the total table entries are reduced from $2^{16} \times 2^{16}$ to 2^{16}+2^{16}. Consequently, NetFC achieves floating-point arithmetic operations via looking up these small tables in sequence and performing some integer arithmetic operations.

One might wonder looking up multiple tables and performing some arithmetic operation would degrade the performance of programmable switches. But in fact, packet-processing pipelines have an all-or-noting characteristic [11]: programs can run at the line rate of the switch pipeline as long as they can run. Next we respectively introduce the details of NetFC for different arithmetic types.

1) Addition and Subtraction: We assume that two floating-point numbers, x and y, are positive; we will relax this assumption later. Note that we mainly discuss addition operation as subtraction also can viewed as one type of addition operations. Let's i (j resp.) denote $\lfloor \log_2(x) \rfloor$ ($\lfloor \log_2(y) \rfloor$ resp.). We note that the round-down operations could degrade the computing

accuracy, and thus propose a novel approach to make up for the accuracy loss (see Section IV-B). Logically, x adds y can be obtained as follow.

$$\begin{aligned} x+y &= 2^{log_2(x+y)} \\ &= 2^{log_2(x)+log_2(1+y/x)} \\ &= 2^{log_2(x)+log_2(1+2^{log_2(y)-log_2(x)})} \\ &= 2^{i+log_2(1+2^{j-i})} \end{aligned} \tag{3}$$

To achieve the above addition, we need to set up three tables (see Figure 3). The first table, *logTable*, is used to record the logarithm values of all possible keys. With this basis, it is straightforward to get the value of i and j via looking up *logTable*. The second table, *miTable*, is used to figure out the value $\sigma(\theta) = \log_2(1+2^\theta)$ for a given θ; we use $j-i$ to look up *miTable*, and then use the result to add i. Thus we can obtain the value of $i + \log_2(1 + 2^{j-i})$. The last table, *expTable*, is to compute (find out) the exponential value for a given key. With this table, we can find out the value of $2^{i+log_2(1+2^{j-i})}$ (a.k.a $x + y$).

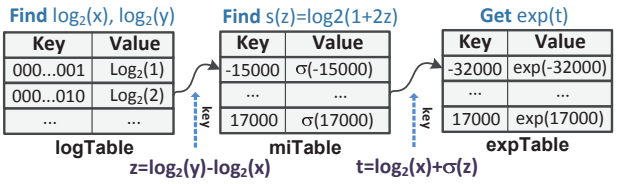


Fig. 3. Floating-point arithmetic on switches.

Next we consider a more general condition that $x \neq 0$ and $y \neq 0$. Thus $x+y$ equals to $\mid x+y \mid$ or $-\mid x+y \mid$. Likewise, we use i (j resp.) to denote $\lfloor \log_2(\mid x \mid) \rfloor$ ($\lfloor \log_2(\mid y \mid) \rfloor$ resp.). $x+y$ can be obtained as follow.

$$\begin{aligned} x+y &= \pm 2^{log_2(|x+y|)} \\ &= \pm 2^{log_2(|x|)+log_2(|x+y|/|x|)} \end{aligned} \tag{4}$$

|x+y| belongs to one of the three situations: |x|+|y|, or |x|-|y|, or |y|-|x|. Finally, we can get Eq. 5.

$$x+y = \pm 2^{i+log(\pm 1 \pm 2^{j-i})} \tag{5}$$

Indeed, due to $\pm$, Eq. 5 has eight possible situations, which are decided by the following three conditions: 1) $x > 0$; 2) $y > 0$; 3) |x|>|y|. The detail is shown in Table I.

Therefore, in the general condition, we still need *logTable*, *miTable* and *expTable*. But the difference is that we require three variants of *miTable* (i.e. $\sigma(\theta) = \log_2(1+2^\theta)$, $\sigma(\theta) = \log_2(1-2^\theta)$ and $\sigma(\theta) = \log_2(-1+2^\theta)$). In summary, NetFC maintains one *logTable* table, three *miTable* tables and one *expTable* table to achieve floating-point addition operation.

Corner cases. There are some corner cases, however. For example, if x (y resp.) equals to 0, NetFC will return y (x resp.) directly. In addition, in the case of $j-i > 15$ ($j-i < -15$ resp.), x (y resp.) is 15 orders larger than y (x resp.) so that the sum of x and y approximately equals to x (y resp.).

TABLE I
DECISION TABLE.

$x>0$	$y>0$	|x|>|y|	formula
T	T	T	$2^{i+log(1+2^{j-i})}$
T	T	F	$2^{i+log(1+2^{j-i})}$
T	F	T	$2^{i+log(1-2^{j-i})}$
T	F	F	$-2^{i+log(-1+2^{j-i})}$
F	T	T	$-2^{i+log(1-2^{j-i})}$
F	T	F	$2^{i+log(-1+2^{j-i})}$
F	F	T	$-2^{i+log(1+2^{j-i})}$
F	F	F	$-2^{i+log(1+2^{j-i})}$

Algorithm 1 In-network floating-point addition.

Require: p, an input data packet.

1: parser floating-point operands x, y from p.
2: **if** x (or y) $\equiv 0$ **then**
3: return y (or x)
4: **end if**
5: get $i = \lfloor log_2(\mid x \mid) \rfloor$, $j = \lfloor log_2(\mid y \mid) \rfloor$ by *logTable*.
6: $n = j - i$.
7: **if** $n >$15 **then**
8: return y
9: **else if** $n <$-15 **then**
10: return x
11: **else**
12: select *miTable* based on table I.
13: get $m = \lfloor log(\pm 1 \pm 2^n) \rfloor$ by looking up *miTable*.
14: $k = i + m$.
15: get $\mid x+y \mid = 2^k$ by looking up *expTable*.
16: set sign bit according to table I.
17: **end if**

Algorithm 1 summarizes how NetFC performs floating-point addition/subtraction operations on programmable switches. First, it parses an incoming packet to obtain two operands x and y (line 1), checks their values (line 2 to 4) and projects them into logarithm space via looking up *logTable* (line 5). After processing corer cases, it decides which *miTable* to use based on Table I. Finally, it further looks up the selected *miTable* and *expTable* to calculate the result (line 13 to line 16). Figure 4 shows the implementation on data plane of programmable switches for addition/subtraction operations.

2) Multiplication and Division: Consider two non-zero floating-point numbers, x and y. We still use i and j to denote $\lfloor log_2(\mid x \mid) \rfloor$ and $\lfloor log_2(\mid y \mid) \rfloor$ respectively. Note that x=-$2^{\log_2(|x|)}$=-2^i (x=2^i resp.) when $x < 0$ ($x > 0$ resp.). Likewise, y=-2^j (y=2^j resp.) when $y < 0$ ($y > 0$ resp.). Thus the multiplication of x and y can be transformed to $x * y$=$\pm 2^{i+j}$ while their division is x/y=$\pm 2^{i-j}$.

In summary NetFC only needs two types of tables, *logTable* and *expTable*, to achieve floating-point multiplication and division operations.

Corner cases. Similarly, some corner cases should be dealt

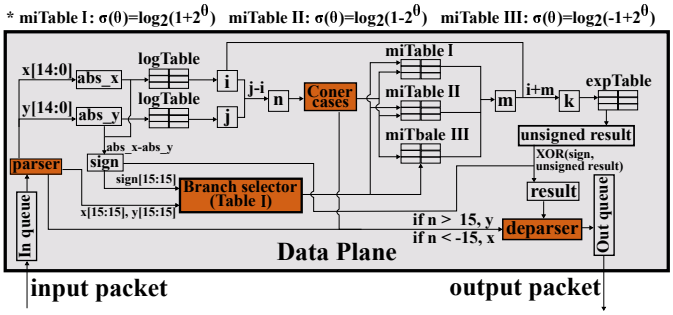


Fig. 4. Implementation of floating-point addition in NetFC.

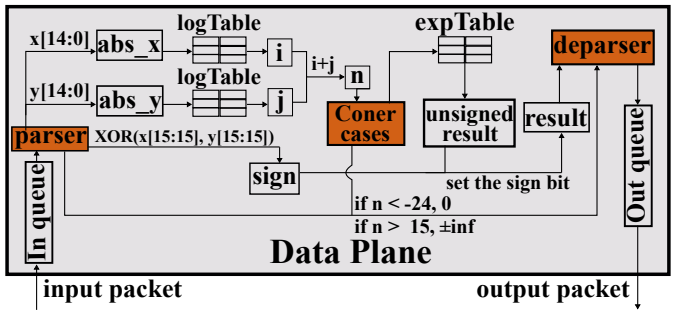


Fig. 5. Implementation of floating-point multiplication in NetFC.

with separately. One case is that one or both operands equal to 0 so that NetFC returns 0 or NaN[1] directly. For another case, the result exceeds the representation range of IEEE 754 floating point and NetFC would return INFINITY[2] or 0 directly.

Algorithm 2 In-network floating-point multiplication.

Require: p, an input data packet.
1: parser floating-point number x, y from p.
2: **if** $x \equiv 0$ or $y \equiv 0$ **then**
3: return 0.
4: **end if**
5: get $i = \lfloor log(\mid x \mid) \rfloor$, $j = \lfloor log(\mid y \mid) \rfloor$ by *logTable*.
6: sign(x*y)=XOR(sign(x), sign(y)).
7: $n = i + j$.
8: **if** $n >$15 **then**
9: return INFINITY
10: **else if** $n <$-24 **then**
11: return 0.
12: **else**
13: get $\mid x * y \mid = 2^n$ by looking up *expTable*.
14: set the sign bit
15: **end if**

Algorithm 2 shows how NetFC performs the floating-point multiplication. NetFC first parses an input packet to obtain two operands x and y and decides whether operands equal to 0 or not (line 2 to 4). Then it looks up *logTable* to find out the value of i and j, whose sum is used to detect the corner cases (line 5 to 11). After processing the corner cases, NetFC further uses the sum (i+j) to look up *expTable* and decides the sign of the result (line 13 to 14). The processing logic of floating-point division is similar to Algorithm 2. Their major differences lie in the corner cases and line 7 in Algorithm 2 (*i.e.,* $n = i-j$). Figure 5 shows the implementation of multiplication and division for floating-point numbers on programmable switches.

[1]NaN (Not a number) is a representation of exceptions in IEEE 754.
[2]All exponent bits are assigned to 1, and all fraction bits are assigned to 0.

C. Working with Posit

The divide-and-conquer table lookup approach applies to both IEEE 754 floats and posit floating points. The major difference lies in the representation of the keys of *logTable* and the values of *expTable*, both of which are dependent on the used float standards of applications. The other difference is about the corner cases, because posit has larger dynamic range than that of IEEE 754 float (see section II-B). For example, for float addition following IEEE 754, the corner case is $|j - i| > 15$ (see section III-B1); while for posit addition, the corner case is $|j - i| > 64$. That said, our approach in principle can apply to different data types.

IV. Implementation and Optimization

In this section, we first present implementation details on programmable switches, and then introduce a few optimizations to improve the computational accuracy and reduce the overhead.

A. Implementation

We implement our NetFC on a Barefoot Tofino switch (3.2Tb/s) using $P4_{16}$ language. The switch has some restrictions on the resource usage. A particular restriction is that it cannot support complicated programming logics due to the limited pipeline stages. However, NetFC requires some *if-else* conditions to decide *miTable*. To address this issue, NetFC uses a separated table with pre-issued entries that covers all cases shown in Table I to choose which *miTable* it should use. This eliminates multi-layer nested *if-else* conditions and reduces the usage of pipeline stages.

B. Optimization: scaling factor

As mentioned before, NetFC uses $\lfloor log_2(x) \rfloor$ to approximate $log_2(x)$, which inevitably incurs accuracy loss since the decimal fraction of $log_2(x)$ has been ignored. To cope with this problem, NetFC utilizes a scaling factor, k, to multiply $log_2(x)$ for amplifying its decimal fraction and avoiding being ignored. NetFC also divides this scaling factor in subsequent steps for guaranteeing the correctness of floating-point arithmetic operation.

To illustrate this, let us consider two operands, x and y. We first get $i = \lfloor log(x) * k \rfloor$ and $j = \lfloor log(y) * k \rfloor$ by looking up *logTable* respectively, and then use $\theta = i - j$ to look up *miTable* for obtaining $\gamma = \lfloor log(\pm 1 \pm 2^{\frac{\theta}{k}}) * k \rfloor$. Finally, it looks up *expTable* for the result $\lfloor \pm 2^{\frac{i+\gamma}{k}} \rfloor$. It is clear that one scaling down operation (diving k) follows every scaling up operation (multiplying k). A larger k brings higher accuracy,

but also consumes more table entries (i.e. memory). Thus this is a tradeoff between accuracy and memory, which will be evaluated in Section V.

C. Optimization: Prefix-Based Lossless Compression

We next discuss the optimization for NetFC memory overhead. Specifically, for a table in NetFC, it is possible that many continuous entries have the same value, consequently their corresponding keys can be merged. Thus we propose a prefix-based compression mechanism.

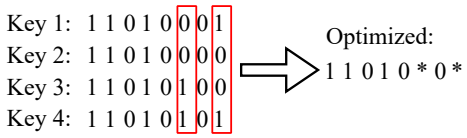


Fig. 6. Example of the prefix-based compression mechanism.

Figure 6 shows an example that NetFC compresses four keys (i.e. table entry) into one key via a wildcard representation, which can be loaded to the switch TCAM memory. It is noteworthy that such compression method is lossless. Our experimental results show that it can save about 25% memory consumption.

V. Evaluation

We raise questions about the computational accuracy and the overhead of NetFC:

- Q1: In terms of floating-point addition/subtraction, how well does NetFC perform comparing to the state-of-the-art approach (i.e. float-to-integer [22]) (Section V-B)?
- Q2: How well does NetFC perform for floating-point multiplication/division (Section V-C)?
- Q3: Does NetFC work well for the posit standard (Section V-D)?
- Q4: How much resources will NetFC consume (Section V-E)?
- Q5: How does the scaling factor affect NetFC (Section V-F)?

Note that NetFC uses 1,024 as the default scaling factor when evaluating the accuracy of floating-point operations.

A. Methodology

Experimental setup. We evaluate NetFC on a testbed with two commodity servers, each of which is equipped with 32 cores of Intel(R) Xeon(R) E5-2682 CPU @ 2.5GHz, 256GB RAM with Ubuntu 16.04 and Linux kernel 4.15.0-132. All servers are directly connected to a Barefoot Tofino switch (3.2 Tbps). We run NetFC on the programmable switch, and deploy a "sender" on one server and a "receiver" on the other server. The sender reads datasets and constructs floating-point operands and operators, which constitute NetFC packets to be forwarded to the switch. And the switch identifies NetFC packets and performs floating-point arithmetic operations while the receiver receives, parses and checks the results from the switch.

Baseline. In the experiments, we use float-to-integer method and InREC [20] as baselines. Float-to-integer method has been widely used in in-network computation (see section II-C). Its accuracy is determined by a scaling factor. For fairness, we configured the factor of the float-to-integer method to achieve the best accuracy in each experiment. In addition, InREC adopts a bit-shifted method to achieve floating-point operations.

Benchmarks. We use two random (synthetic) datasets and one real dataset to evaluate our NetFC. The first random dataset, denoted as Dataset I, consists of ten thousand randomly generated 16-bit floating-point numbers; the other random dataset, denoted as Dataset II contains ten thousand randomly generated 16-bit floating-point decimals. The real dataset, denoted as Dataset III, makes up of gradient updates (50,000 records) from a real distributed deep learning model training [18].

Metrics. Let $\otimes$ denote +, -, $\times$ and $\div$. Overall, We utilize the following formula to quantize the $accuracy$:

$$\begin{aligned} expect\_result &= x \otimes y \\ result &= \text{NetFC}(x \otimes y) \\ accuracy &= e^{-\frac{|expect\_result - result|}{|expect\_result|}} \end{aligned} \tag{6}$$

where $expect\_result$ is the exact results that are calculated by the receiver CPUs, and $accuracy$ represents the proportion of error to $expect\_result$. Thus $accuracy$ lies in between 0 and 1, a higher $accuracy$ indicates that $result$ is more close to $expect\_result$. To demonstrate this point, we assume that $accuracy$ is close enough to 1. Consequently, we can see that $|\ expect\_result - result| \approx (1 - accuracy) * |expect\_result|$. This conclusion is easily to be proved via Taylor series [3]. Our experiments show that this approximation holds as long as $accuracy$ is larger than 0.95.

We also use MSE (Mean Square Error) to measure the deviation of $expect\_result$ and $result$:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(expect\_result_i - result_i)^2 \tag{7}$$

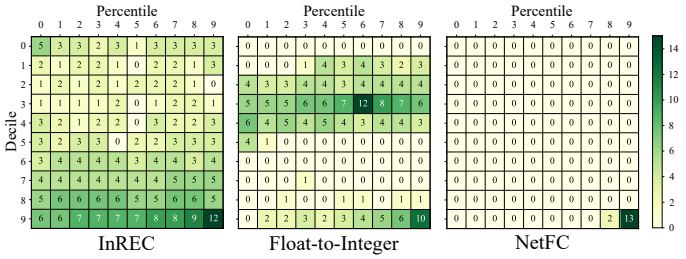
where n represents the total number of pairs in the dataset. MSE measures the average squared difference between $expect\_result$ and $result$, and smaller MSE values means better accuracy.

B. Addition/Subtraction Performance

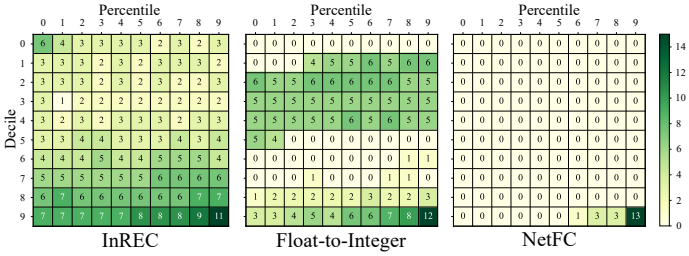
For each of the three datasets, we plot a heatmap to compare the accuracy of NetFC and Float-to-integer method (see Figure 7). The horizontal axis represents the percentile of accuracy, and the vertical axis represents the decile of accuracy. The value in the grid that corresponds to (j, i), where j is the horizontal axis value and i is the vertical axis value, represents the logarithm of the number of data with accuracy between $0.1 * i + 0.01 * j$ and $0.1 * i + 0.01 * (j + 1)$. For example, let us consider the grid that corresponds to (6, 3),

TABLE II
ACCURACY TABLE

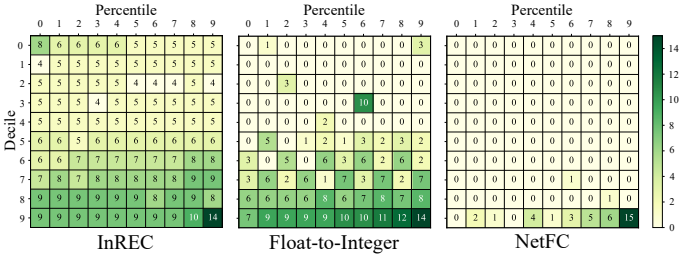
	dataset name	IEEE 754 floating point					posit floating point		
		+,- (NetFC)	+,- (Float-to-integer)	+,- (InREC)	×	÷	+,-	×	÷
average accuracy (%)	I	99.94	48.19	93.82	99.96	99.96	99.86	99.91	99.91
	II	99.95	84.83	88.70	99.94	99.98	99.86	99.87	99.95
	III	99.96	95.28	89.21	99.97	99.98	-	-	-
median accuracy (%)	I	99.94	36.79	99.85	100	100	99.91	99.94	99.94
	II	100	99.93	97.55	100	100	99.94	99.87	99.96
	III	100	99.24	100	100	100	-	-	-
minimum accuracy (%)	I	77.88	13.72	0	60.65	95.56	36.79	93.11	89.48
	II	68.55	4.54	0	71.65	99.85	36.79	93.11	98.20
	III	76.74	0.01	0	36.79	99.88	-	-	-



(a) Dataset I accuracy heatmap.



(b) Dataset II accuracy heatmap.



(c) Dataset III accuracy heatmap.

Fig. 7. Floating-point addition/subtraction heatmap. (Note that we adopt logarithm for the values in grids.)

the value in the grid is 12 (see figure 7(a)), which means that the number of data with accuracy between 0.36 and 0.37 is about 4,096 (2^{12}).

As shown in Figure 7(a), the accuracy of Float-to-integer method mainly distributed near (6,3), while accuracy of NetFC and InREC concentrate around (9,9). Overflow is the main reason for the poor accuracy of Float-to-integer method. Overflow is a phenomenon that an arithmetic operation attempts to create a numeric value that is outside the range that can be represented with a given number of bits. For example, considering two 16-bits floating-point numbers $x = 4.765625$ and $y = -0.005203$, if we use a scaling factor of 10,000, Float-to-integer method converts x to 47656 and y to -52 firstly. Unfortunately, 47656 is out of the range of 16-bits integers, it will be mistaken for -17880 by the switch. This example reveals drawback (significant errors) of the Float-to-integer method: overflow occurs when the factor is large, while loss of decimal parts occurs when the factor is small. Due to this property, Float-to-integer method performs well only on decimal datasets.

Figure 7(b) shows the performance of the three methods on the dataset II. InREC achieve the worst accuracy. The accuracy of Float-to-integer method were concentrated in the range of 0.97 to 1.0, which is much better than the result in Figure 7(a), but still not as good as NetFC (see Figure 7(b) right). For NetFC, we found that only 20 out of 10,000 data have an accuracy of less than 99%.

In Figure 7(c), we compare the performance of the three methods on the dataset sampled from a distributed machine learning system. Again, NetFC achieves better accuracy. It is noteworthy that this dataset takes values in the range of -0.01 to 0.01, which is detrimental to our approach because the non-decimal part of the implementation is completely wasted. Thus we can remove those table entries whose value is larger than 1 since it is impossible that they would be matched. With this basis, NetFC can save more on-chip memory for further improving computational accuracy via increasing scaling factor.

These results demonstrate the advantage of NetFC over Float-to-integer method and InREC in accuracy. Table III compares the three methods in terms of MSE. NetFC performs well on all three datasets; Float-to-integer method is acceptable on dataset III, but performs poorly on dataset I and II; InREC performs better than float-to-Integer method on dataset I and II, but worse on dataset III.

C. Multiplication/Division Performance

Figure 8 shows the accuracy distribution of NetFC for multiplication and division using dataset II. Note that some similar results were found on the other two datasets. We see a high accuracy no matter which datasets are used: almost all computations achieve a accuracy over 99% for division, and only 50 (out of 10,000) computations has a accuracy of less than 99%. Table III demonstrates that the deviation from

TABLE III
MSE TABLE

	Dataset I	Dataset II	Dataset III
+,- (NetFC)	91.32	2.60×10^{-8}	1.95×10^{-13}
+,- (Float-to-integer)	198106067.16	0.11	4.17×10^{-11}
+,- (InREC)	1437284.00	2.53×10^{-3}	4.86×10^{-8}
×	28.13	1.52×10^{-9}	2.72×10^{-16}
÷	27.29	0.98	1.12×10^{-4}

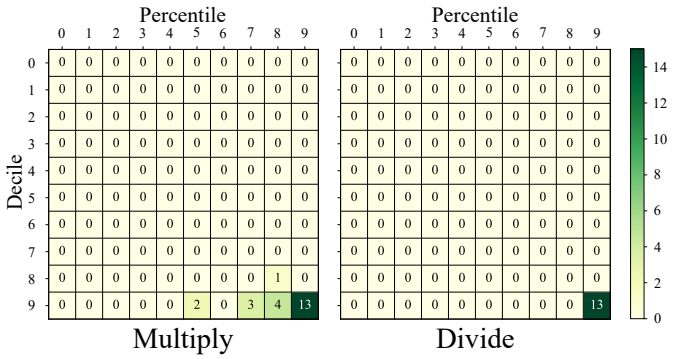


Fig. 8. Dataset II accuracy heatmap for multiplication and division.

$expect\_result$ of NetFC's result is very low. Table II shows the median accuracy of NetFC, which is close to 100% in all cases.

D. Posit Performance

Table II shows the accuracy of NetFC for floating points of posit standard (scaling factor = 512). Again, we see very high accuracy with an average accuracy above 99.86% and a median accuracy above 99.87%, for all four types of operations. These results prove NetFC's good support of floating point arithmetic operations for both IEEE 754 standard and the posit standard.

We see a slightly lower accuracy for posit standard than IEEE 754 standard in Table II. The reason is that posit standard has a larger dynamic range and thus, with the same amount of memory as IEEE 754 standard, it has to use a smaller scaling factor, which is 512 (1,024 for IEEE 754 standard in our experiments). A smaller scaling factor leads to the loss in accuracy.

E. Overhead analysis

NetFC requires several tables on the switch data plane to implement floating-point arithmetic operations. This does consume the on-chip memory of programmable switches. We next discuss its overhead.

NetFC generates 5 tables at most: two *logTable*[3] ($2*2^{15}$ 16-bit entries in total, ≈128KB), three *miTable* (2^{15} 16-bit entries, ≈192KB) and one *expTable* (2^{16} 16-bit entries, ≈128KB). As a result, NetFC consumes about 448KB on-chip memory in total, which is reasonable considering that our lowe-end Barefoot Tofino switches are equipped with 20MB memory. Note that the prefix-based loss-less compression can further reduce the memory usage (see Section IV). By analysing pipeline usage through p4i[4], we find that NetFC only consumes 5 pipeline stages so that it easily runs on the data plane of switches. In addition, we also find that about 30 out of 1152 16-bit PHV (Packet Header Vector) ALU and 17 out of 384 VLIW (Very Long Instruction Word) are used.

F. Sensitiveness to Scaling Factor

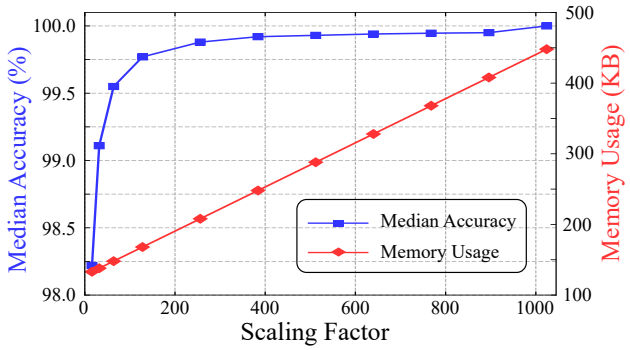


Fig. 9. Median accuracy and memory usage with scaling factor on the dataset II.

Recall the scaling factor is used to compensate for using round-down operation on accuracy: a larger scaling factor increases the accuracy but also the memory consumption. Figure 9 shows the memory usage and median accuracy on the dataset II when increasing scaling factor for floating-point addition. We can see that memory usage is proportional to the value of scaling factor. The accuracy increases significantly with the growth of the scaling factor when the factor below 256; beyond this point the improvement become marginal. That said, NetFC can achieve an significant accuracy improvement using limited extra cost.

Indeed, the scaling factor reflects the tradeoff between computational accuracy and on-chip memory usage. To guide the setting of the scaling factor, we need first to obtain the corresponding dataset. In practice, we can get the dataset in two ways. The first one is to randomly generate data points in the range of floating points in the considered context, while the second is to directly gather the real workloads. Though the real-world dataset is preferred, it is harder for collection. With either the synthetic dataset or the real-world dataset, one can then determine the scaling factor as what we did in Figure 9 by observing its impact empirically on accuracy and memory usage, and finally deploy it on the programmable switches.

VI. USE CASE: ONLINE DETECTION OF SLOWLORIS ATTACKS

To further show the feasibility of our NetFC in real applications, we integrated NeFC into Sonata [12] for detecting Slowloris attacks.

To detect Slowloris attacks, Sonata deploys a measurement task on programmable switches. This task monitors the traffic

[3]In practice, each operand needs to look up an exclusive *logTable*.

[4]A visualization tool offered by Barefoot company.

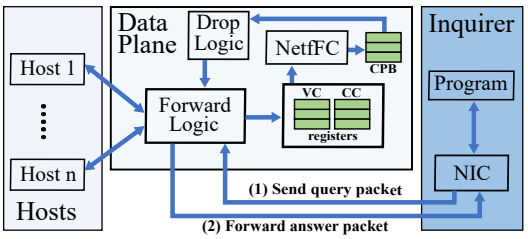


Fig. 10. Slowloris attack detection measurement framework with NetFC.

of all connections belonging to individual hosts to see whether the average traffic volume of each connection belonging to a host is less than a predefined threshold value. Thus Sonata assigns two switch registers for each host, and counts its corresponding connections (denoted by c) and traffic volume (denoted by v). In the conventional fashion, Sonata sends both c and v to the switch local CPU to calculate v/c. This delays the attack detection. We deploy our NetFC as a module for Sonata, and use a CPB (connection per byte) register to record the result of v/c (see Figure 10).

Evaluation Setup. To evaluate how fast NetFC can accelerate Slowloris attack detection, we build a testbed that contains 5 physical machines connecting to a Barefoot Tofino switch. Each machine is equipped with 32 CPU cores and a Mellanox dual-port 100G NIC. Note that we use four machines as hosts and one machine as an inquirer.

The experiment can be divided into two steps. First, we use the programmable switch to record the number of connections and traffic volume for individual hosts during their communication. Meanwhile, we utilize NetFC to compute their quotient, and store it into the corresponding PCB register. Second, the inquirer sends a query to the programmable switch, and the switch searches PCB registers based on the IP address carried by the query. The result will be sent back to the inquirer.

Results. By capturing packets from the inquirer's NIC, we measured the time elapsed from the time when the query packet was sent to the time when we got the answer packet. Experiments show that the time to complete a query is decreased from 43.156ms to 0.046ms by NetFC, showing the significant benefits of removing the involvement of switch local CPUs by NetFC. With such a short response time, network operators can quickly detect possible attacks and effectively protect network from the attacks.

VII. Related Work

FPUs. FPU is a relatively conventional topic in the community. Recent researches mainly focus on how to improve the performance of floating-point calculation or increase the computational accuracy [8], [26], [28], [30]. However, due to a specific programming model and limited computing capacity, it is not easy to deploy FPUs into programmable switches.

Fixed-Point Method. [16] implements 16-bit floating-point addition by converting 16-bit floating-point number into 43-bit fixed-point number. However, the state-of-the-art programmable switches only support 32-bit integer operations at most. Therefore, it has to operate two 32-bit integer vectors for each fixed-point number. This leads to more stage consumption. InREC [20] adopts a bit-shifted method to achieve floating-point operations at cost of low accuracy and high memory usage.

In-network computation. There are a lot of works focusing on in-network computation and achieving significant performance improvement. For example, [22], [29] perform in-network gradient aggregation to accelerate distributed deep learning. NetCache [19] implements a high-performance key-value cache in programmable switches. Sonata [12] utilizes programmable switches to achieve fast In-band network telemetry. However, these works do not fundamentally solve the floating-point calculation issue on the data plane of programmable switches.

VIII. Discussion and Future Work

Summary. In-network computation is an emerging trend to reduce the network overhead via offloading some tasks to programmable switches. However, it suffers from the limited computational capability of programmable switches (e.g. floating-point operations). To address this problem, we design NetFC, a table-lookup method, to achieve on-the-fly in-network floating-point operations nearly without accuracy loss. NetFC adopts a prefix-based lossless compression mechanism to reduce its memory consumption. Our experimental results show that the average accuracy of NetFC is above 99.94% with only 448KB memory consumption. Furthermore, we integrate NetFC into Sonata for detecting Slowloris attack, yielding significant decrease of detection delay. We believe that our NetFC can be a building block for in-network computation.

Multiple FP operations. NetFC can support multiple FP operations via deploying different computational types of lookup tables in sequence. For example, we can sequentially deploy the addition and multiplication lookup tables to implement the operation of an addition at first followed by a multiplication. This of course will take more stages. That said, the number of FP operations NetFC can support for each packet depends on the available stages of the data plane. In addition, we can further utilize the recirculation operation provided by Barefoot Tofino switches to change the order of different FP operations.

32-bit FP operations. The current implementation of NetFC does not support 32-bit floating points due to the limitation of on-chip memory. In theory, we can adopt an approximate method based on Taylor series [3] to reduce memory consumption and support 32-bit FP operations. We leave it as our future work.

Acknowledgments

We thank our shepherd Zaoxing Liu and the anonymous reviewers for their insightful feedback. This work is supported in part by National Key R&D Program of China (Grant No. 2019YFB1802800), the National Natural Science Foundation of China (Grant No. U20A20180 and 62002344) and CAS-Austria Joint Project (Grant No. 171111KYSB20200001). Corresponding author: Zhenyu Li.

References

[1] Slowloris http dos. https://web.archive.org/web/20150426090206/http://ha.ckers.org/slowloris, 2009.

[2] Ieee 754 standard. http://mathcenter.oxford.emory.edu/site/cs170/ieee754/, 2021.

[3] Taylor series. https://simple.wikipedia.org/wiki/Taylor_series, 2021.

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

[5] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736, 2013.

[6] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM Computer Communication Review*, 41(4):98–109, 2011.

[7] Marco Cococcioni, Emanuele Ruffaldi, and Sergio Saponara. Exploiting posit arithmetic for deep neural networks in autonomous driving applications. In *2018 International Conference of Electrical and Electronic Technologies for Automotive*, pages 1–6. IEEE, 2018.

[8] Thierry Cretegny, Thierry Dauxois, Stefano Ruffo, and Alessandro Torcini. Localization and equipartition of energy in the β-fpu chain: Chaotic breathers. *Physica D: Nonlinear Phenomena*, 121(1-2):109–126, 1998.

[9] Florent De Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. Posits: the good, the bad and the ugly. In *Proceedings of the Conference for Next Generation Arithmetic 2019*, pages 1–10, 2019.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 44–61, 2020.

[12] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*, pages 357–371, 2018.

[13] John L Gustafson and Isaac T Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 4(2):71–86, 2017.

[14] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 576–590, 2018.

[15] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. *ACM SIGCOMM Computer Communication Review*, 43(4):219–230, 2013.

[16] Masoud Moshref Javadi, Changhoon Kim, Patrick W Bosshart, and Anurag Agrawal. Forwarding element data plane performing floating point computations, April 20 2021. US Patent 10,986,042.

[17] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. Fast string searching on pisa. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 21–28, 2019.

[18] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, et al. Xdl: an industrial deep learning framework for high-dimensional sparse data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, pages 1–9, 2019.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[20] Matthews Jose, Kahina Lazri, Jérôme François, and Olivier Festor. Inrec: In-network real number computation. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 358–366. IEEE, 2021.

[21] Seyed Hamed Fatemi Langroudi, Tej Pandit, and Dhireesha Kudithipudi. Deep learning inference on embedded devices: Fixed-point vs posit. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, pages 19–23. IEEE, 2018.

[22] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. Atp: In-network aggregation for multi-tenant learning. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 741–761, 2021.

[23] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. Uno: Uniflying host and smart nic offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 506–519, 2017.

[24] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating distributed reinforcement learning with in-switch computing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 279–291. IEEE, 2019.

[25] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[26] Stuart Franklin Oberman. *Design Issues in High Performance Floating Point Arithmatic Units*. PhD thesis, Citeseer, 1996.

[27] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[28] Bob Rink. Symmetry and resonance in periodic fpu chains. *Communications in Mathematical Physics*, 218(3):665–685, 2001.

[29] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. 2021.

[30] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[31] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[32] Hao Zhang, Jiongrui He, and Seok-Bum Ko. Efficient posit multiply-accumulate unit generator for deep learning applications. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2019.

[33] Penghao Zhang, Heng Pan, Zhenyu Li, peng He, Zhibin Zhang, Gareth Tyson, and Gaogang Xie. Accelerating lsh-based distributed search with in-network computation. 2021.

[34] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 991–1010, 2021.