Leveraging In-Network Computing and Programmable Switches for Streaming Analysis of Scientific Data

Ganesh C. Sankaran
University of California
Davis, Davis, CA, USA
gsankaran@ucdavis.edu

Joaquin Chung
Argonne National Laboratory
Lemont, IL, USA
chungmiranda@anl.gov

Raj Kettimuthu
Argonne National Laboratory
Lemont, IL, USA
kettimut@anl.gov

***Abstract*—With the emergence of programmable network devices that match the performance of fixed function devices, several recent projects have explored in-network computing, where the processing that is traditionally done outside the network is offloaded to the network devices. In-network computing has typically been applied to network functions (e.g., load balancing, NAT, and DNS), caching, data reduction/aggregation, and coordination/consensus functions. In some cases it has been used to accelerate stream-processing tasks that involve small payloads and simple operations. In this work we focus on leveraging in-network computing for stream processing of scientific datasets with large payloads that require complex operations such as floating-point computations and logarithmic functions. We demonstrate in-network computing for a real-world scientific application performing streaming normalization of a 2-D image from a light source experiment. We discuss the challenges we encountered and potential approaches to address them.**

***Index Terms*—in-network computing, programmable switches, scientific streaming analysis**

I. Introduction

The emergence of programmable switches [1] has inspired new ideas in in-network computing [2]. For instance, early proposals have explored implementing network functions such as load balancers fully on the data plane [3]. In-network cache applications [4], [5] that leverage programmable switch hardware to implement key-value stores are among early demonstrations as well. Other researchers have proposed architectures that place accelerators next to networking devices to offload computation on streaming analysis pipelines [6]. However, the most recent trends exclude these types of architectures from in-network computing [7] definitions.

Modern programmable switches have different processing capabilities depending on their hardware and software architectures. In general, their main benefits are high throughput (10 billion packets/s) and low latency (sub-microseconds). However, they support only basic arithmetic/Boolean operations, and they do not allow loops. Although these capabilities may be sufficient to implement network functions or key-value stores in the network, scientific applications require more complex operations. Range normalization and log scaling are key ingredients of many image normalization and scientific computing use cases.

978-1-6654-0522-5/21/$31.00 ©2021 IEEE

In this work we evaluate the offloading of computational functions into programmable switches in the context of projection normalization for tomographic imaging experiments. In contrast to the accelerator offloading approach, we take an algorithm transformation approach, similar to how high-performance computing (HPC) algorithms are optimized to the hardware where they are running. We present the several algorithmic transformations needed to perform computations on programmable switches that are not natively supported. We discuss the tradeoffs of this process and elaborate on new directions that can be explored from this approach to in-network computing. This paper paves the way to achieving these calculations on network switches. The main contributions of this paper are the following:

1) An implementation of an algorithm involving floating-point calculations on a programmable pipeline (BMv2)
2) A pseudo-floating-point (PF) representation for programmable switches
3) An implementation of logarithmic computation on programmable switches using the optimal number of table entries required

The rest of the paper is organized as follows. We provide background and motivation in §II. We describe our proposed solution and implementation in §III and §IV, respectively. We discuss challenges and tradeoffs in §V, and summarize our conclusions in §VI.

II. Background and Motivation

A. Stream Processing in Light Source Facilities

Light sources are crucial tools for addressing grand challenge problems in the areas of life sciences, energy, climate change, and information technology [8]. For instance, the X-rays produced at a light source enable scientists to study internal morphology of materials and samples with very high spatial (atomic and molecular scale) and temporal (<100 ps) resolutions. These experiments can generate massive amounts of burst data. For example, tomographic imaging stations

can collect 1,500 projections (images each with 2,048 x 2,048 pixels) in 9 seconds, generating data at a rate of >8 Gbps. These projections are then processed at remote high-performance computing (HPC) facilities. Tomographic imaging stations (generating the data) are connected to their corresponding HPC compute nodes (analyzing the data) via wide-area networks. Data processing can be performed after all data is generated (postprocessing) or in real time (streaming analysis). Real-time streaming and analysis of tomographic imaging data enable scientists (or the control software) to (1) make timely decisions that can significantly accelerate the execution of experiments and (2) do smart experimentation, such as changing the parameters interactively to enhance the overall efficiency of end-to-end scientific workflow.

B. In-Network Computing and Programmable Switches

In-network computing is the process of offloading operations from end hosts into networking devices (e.g., switches, routers, or smart NICs) [2]. It focuses on computing *within* the network, using devices that are already being used to forward traffic [7]. Recent developments in programmable switches have increased the interest in in-network computing. Programmable switches seek to allow network operators and programmers to define exactly how packets are processed in a reconfigurable switch chip [1], [9] or a virtual programmable switch [10], [11] through high-level programming languages such as P4 [12]. Since networking devices are limited in memory (use of expensive TCAMs), set of actions (only arithmetic/Boolean operations), and operations per packet (no loops), only applications that follow a partition/aggregate pattern (e.g., big data analytic and machine learning, graph processing, and streaming analysis) can be offloaded to networking devices [2].

Figure 1 illustrates a generic in-network computing use case. In general, a programmable switch may perform a function on the data stream that would have otherwise been executed on an end node. It computes a result (e.g., future parameters of an experiment) that could be consumed by an application (e.g., a data acquisition node collecting tomographic images).

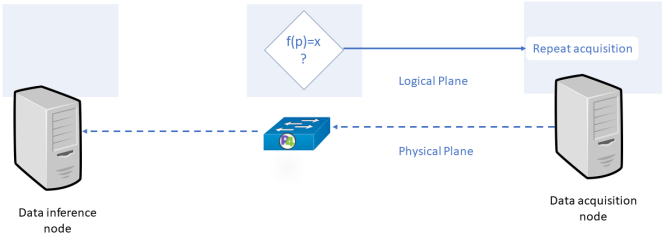


Fig. 1: In-network computing on programmable switches

C. Related Work

In 2015, Dang et al. [13] explored two approaches to deploy the Paxos consensus protocol in network devices: (1) implementation of the full logic of Paxos on SDN switches using OpenFlow extension [14] and (2) an optimistic protocol that does not require changes to the OpenFlow API. In 2016 Katta et al. [3] proposed HULA, a scalable load balancer implemented in programmable switches using P4. In 2017, two independent works presented in-network caching solutions that leverage programmable switches: NetCache [4] and IncBricks [5]. While NetCache is fully implemented in Barefoot's Tofino chip [9], IncBricks is a hardware/software co-designed system that relies on network accelerators.

Sapio et al. [2] proposed DAIET, a system for data aggregation in-network. It was implemented by using P4 and a programmable ASIC and evaluated with a MapReduce-based application. Jepsen et al. [15] studied what abstraction are need to support stateful processing in programmable switches by implementing the LinearRoad benchmark for stream-processing systems in P4.

D. Motivation

As shown in Section II-C, most of the current demonstrations of in-network computing try to find a best match between application and programmable switch functionalities. However, scientific applications are heavy users of floating-point and logarithmic operations (two operations that current programmable switches lack). Thus, we pose the question: *Can we use algorithmic transformations and approximations to offload scientific computation to the network?* Without loss of generality, we use the projection normalization process of tomographic imaging as an example to demonstrate that this strategy is possible.

III. In-Network Normalization

In this section we describe the projection normalization process of tomographic imaging. Next we highlight the challenges of implementing projection normalization on programmable switches. We then introduce our proposed solution.

A. Projection Normalization in Tomographic Imaging

Normalization of incoming projections (i.e., X-ray images) requires simple arithmetic operations. These projections are typically 2,048 x 2,048 pixels in size, with each pixel represented as a 16 bit unsigned integer (i.e., 4M x 16 bits = 8 MB). They are transferred over the network in an uncompressed form as a sequence of pixel values. For normalization, scientists need to collect and store two types of projections before experimental data acquisition starts: dark (d) and white (w). Both d and w projections are collected without a sample between the light source and the detector. While d projection is collected when the light source is turned off, w projection is collected when the light source is turned on. Typically, scientists collect 10–20 images and average them to produce d and w projections. Then the sample is placed on the observation table to capture the sample image. Scientists use d and w projections to normalize images collected during experimentation (i.e., cancel the errors due to equipment setup and illumination).

For each incoming pixel p_i, the normalization procedure produces a normalized pixel n_i by performing the following operations on the image:

1) The difference between a sample image pixel and its corresponding dark pixel average is computed, $\delta_i = p_i - d_i$.
2) This difference is divided by the range computed from the difference between the corresponding white and dark pixels, $r_i = \frac{\delta_i}{(w_i - d_i)}$.
3) After range-based normalization, r_i is transformed to logarithmic scale to complete the normalization process $n_i = \log r_i$.

B. Opportunity

We envision that any switch in the network could execute the required operations for projection normalization, as we split operations along the path between source and destination. The advantage of pipelining data transfer and computation is that it may save time compared with doing it after moving all the data to the HPC compute nodes. Furthermore, by executing these computations in the network, we can free cycles from supercomputers that could be used for more complex tasks.

When we observe the normalization procedure closely, we see that the input pixel values are in the integer domain $\mathbb{Z}^+$ and the output is in the real domain $\mathbb{R}$. The second step maps data from $\mathbb{Z}^+ \rightarrow [0, 1] \in \mathbb{R}$. Further, the logarithm maps from the real domain onto itself, $\mathbb{R} \rightarrow \mathbb{R}$.

By performing these operations in the network, we could leverage the programmable pipeline and in-network computing capabilities of the network switches. A programmable pipeline is composed of multiple stages, each with a lookup table and ALU resources. These resources can be programmed to realize an in-network function. For instance, lookup tables support a wide variety of key matching capabilities (e.g., exact, longest-prefix, and ternary match), and ALU resources support various arithmetic/logical operations on integers and arbitrary-length bits [12].

C. Challenges

Projection normalization is simple when performed on a CPU- or a GPU-based system that supports floating-point operations. However, three key challenges arise in performing this operation on network switches. Specifically, existing programmable network switches do not support division (only integer divisions are supported), floating-point arithmetic, and logarithmic function.

D. Proposed Solution

Despite these deficiencies, network switches store and forward the pixel values in the packet payload. We note that the entire packet is buffered and then retrieved before it gets forwarded. This process provides a good opportunity for any data transformations on the packet payload. When performing stream and image processing, complete payload transformations are of specific interest, as shown in Fig. 5.

At a high level, we propose to circumvent the lack of division operation by modifying our algorithm to perform logarithmic subtraction instead of division. We propose to use pseudo-floating-point representation to overcome the absence of floating-point support. Furthermore, to add logarithmic operation support, we will use lookup tables to compute linear piecewise approximations.

Fixed point	One variable	u	$u \times 2^0$
Floating point	Two variables	v u	$u \times 2^v$
Pseudo Floating point	**Two variables; v implicit**	$\boldsymbol{u}$	$\boldsymbol{u \times 2^v}$

Fig. 2: Pseudo-floating-point representation

IV. Implementation Details

A. Modified Algorithm

We adapted the sequence of operations for in-network normalization algorithm as follows:

1) Compute $\delta_i = p_i - d_i$: A lookup table provides the d_i value based on a flow tuple. Then, ALU resources compute the difference in $\mathbb{Z}$ (integer) domain.
2) Perform logarithmic transformation on $\log \delta_i$: A lookup table is used to compute the logarithmic equivalent based on a linear piecewise approximation equation $\log x = \log x' + \frac{dy}{dx}\Delta x$. Here, x' is the nearest quantization interval, and the deviation from this interval $\Delta x = x - x'$. For this computation, we store $\log x'$ and the slope $\frac{dy}{dx}$ as the lookup output. The computation output accuracy depends on the log transformation. This is presented in the next section.
3) Replace division with subtraction in the logarithmic domain: Subtract $\log \delta_i$ from the precomputed $\log(w_i - d_i)$. A lookup table provides the precomputed $\log(w_i - d_i)$ value for the corresponding pixel. Then, ALU resources compute the difference.
4) Transform the output into floating-point representation p_i'.

For this computation, besides the payload stream, two other streams carrying average d_i values and $\log(w_i - d_i)$ provide the context for the operation. We assume that these values have been precomputed and stored on the switches as table entries.

B. Pseudo-Floating Point

Programmable switches do not support floating operations natively because of the complexity. For instance, the exponents of the floating-point values must be matched before the subtraction operation in step 3. To eliminate this complexity at every step, we propose a pseudo-floating-point representation.

In a PF representation, all values share a common implicit exponent (see Fig. 2). This implicit exponent, eliminates the need for having to match the exponents at every step of intermediate computations. Thus, we can perform steps 2 and 3 on a network switch without floating-point support. At the end in step 4, this intermediate representation is converted into a well-known floating-point representation.

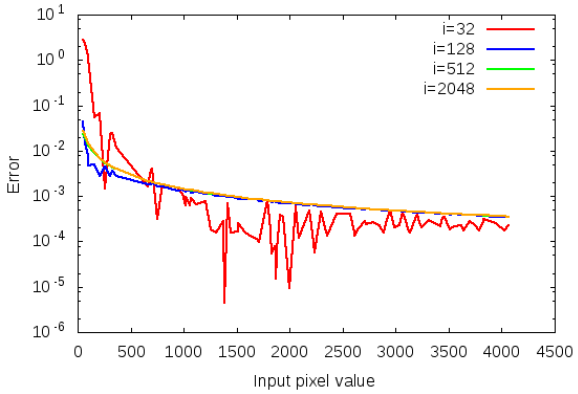


Fig. 3: Error for different numbers of entries

C. Logarithm Approximation

The accuracy of the computation depends on the accuracy of the logarithmic interpolation. The logarithmic function monotonically increases with its input. There is also a one-to-one mapping between its input and the output (i.e., given any real value, it has only one logarithmic value). These properties help increase the accuracy of interpolation.

We use a linear piecewise approximation for computing $\log x = \log x' + \frac{dy}{dx}\Delta x$. This makes use of the function value at the quantized interval x' and the slope of the function observed for interpolation. The entire range of x is divided into ranges, and the starting value is used as x'. For instance, in the range $(2, 4)$, $\log x' = \log_2 2 = 1$ and $\frac{dy}{dx} = \frac{1}{2} = 0.5$. With this approach, the maximum error $\epsilon = 0.0849$ when $x = 3$. Figure 3 shows that the error decreases as x increases. Here we present a numerical example considering a single pixel (p_i):

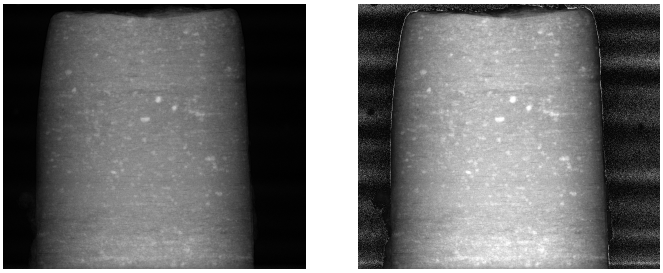
1) $\delta_i = p_i - d_i = 2557 - 99 = 2458$
2) $\log \delta_i = \log x' + \frac{dy}{dx}\Delta x = 11794307 + 617 * (2458 - 2432) = 11810349$ in PF representation with 4 as its implicit exponent.
3) $\log(w_i - d_i) - \log \delta_i = 11815309 - 11810349 = 4960$. This corresponds to 0.00473 in PF 16-bit representation.

. The error with approximation for this pixel is 0.0089 ($< 1\%$).

Next we find the optimal number of entries required to realize the log function by computing the error. We choose both the range of p_i and the number of table entries to be a power of 2. The range is divided on a linear scale. Thus, the last few bits of all table keys is zero and are easy to represent with an "exact" match. From the error graph we can see that 32 entries result in high errors compared with 128 entries. However, adding more entries than 128 does not improve errors by a significant margin. The error in the first interval between 1 and 32 is high. To reduce the error, we use customized entries.

D. P4 Implementation

We implement the algorithm for in-network projection normalization in P4 BMv2 [10] using four tables. Each pixel is parsed as a 16-bit value p_i. The first table maps the flow to dark pixel values d_i, which is used to compute $\delta_i = p_i - d_i$. The second table then is used to compute logarithmic transformation. The key is computed by right shifting one, two, and five bits for different ranges $\{2\}, \{4, 8, 12, \ldots, 28\}, \{32, 64, 96, \ldots, 4096\}$, respectively. Then, the key is exactly mapped to its value and slope corresponding to linear piecewise approximation. Next, the third table maps the flow to $\log(w_i - d_i)$ value. This is used to compute the logarithmic difference between $\log(w_i - d_i)$ and $\log \delta_i$. The fourth table then converts the pixel value in pseudo-floating-point representation to float-32 representation. Our code can be accessed at [16].



(a) Original (b) In-network

Fig. 4: Comparison of pixel normalization with the proposed in-network algorithm

E. Final Normalized Image

The images in Fig. 4 present the pixel normalization with the original procedure and with our proposed in-network algorithm. We can see that the central portion of the image is similar. This shows that the values computed by using the proposed algorithm are a good approximation to the pixel normalization procedure. Considerable amount of noise is observed on the sides of the image when a pixel value p_i is out of range (d_i, w_i). When normalizing on a CPU, any value more than the maximum gets converted into a negative value. The modified algorithm approximation converts this value into a large positive value, resulting in a white dot. We could add a zero slope table entry for out-of-range values to reduce this noise.

V. Discussion

This section outlines the open questions and possible approaches to address these challenges.

GPU vs CPU vs Processor-in-Memory Architecture vs Programmable Switch: Figure 5 shows the abstract form of projection normalization. The CPU performs a wide variety of operations on short data types. On the other hand, the GPU performs a small set of operations on wide data widths. The CPU is capable of executing steps both row-wise (Step 1 before Step 2) and column-wise (p_1 before p_2). GPUs are better at executing the steps row-wise. Currently, ToR switches support up to 6 Tbps switching capacity. At 1 GHz clock rate, at least 6,000 bits of payload must be retrieved and processed every clock cycle to meet this switching capacity. Fast memory

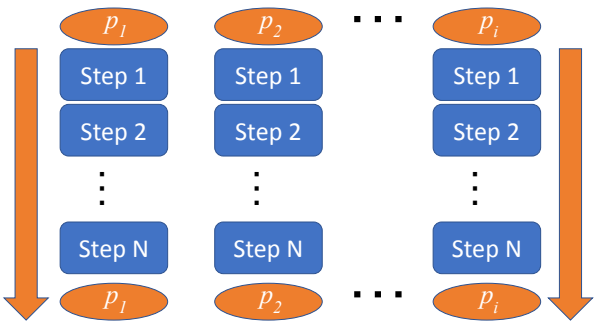


Fig. 5: Pixel-wise projection normalization

modules coupled with wide-width payload processing capability will be required to support this data rate. GPUs with wide width are suitable for this operation. Another option is to explore *processor-in-memory* architectures that can process the payload when it is stored in the packet buffer. This gives a few more cycles for payload processing.

Table Reuse: The current switch pipeline does not allow for multiple fields to use the same table and its entries. For the image-processing example, all pixels use the same logarithm transformation. In a jumbo packet with around a 9000 byte payload of 16-bit pixel values, the current pipeline requires thousands of logarithm tables. This requirement is both a language issue and a run-time issue that is worth looking at.

Distribution: Since the normalization operation is now laid out as a sequence of pipeline stages, this computation can be realized within a single network switch or across multiple switches. This presents many interesting options for processing different portions of the payload on different network switches. A more interesting distribution approach is to look at both the algorithmic steps and different portions of the payload. As shown in Fig. 5, steps and pixels can be mapped in various ways onto network switches.

Routing for Computation: Traditionally, routing algorithms find the shortest path between the source and destination for transferring packets. From an in-network computing perspective, a longer network path capable of meeting the latency and compute requirements is more suitable than choosing the shortest path.

Super Jumbo Frames: When higher-precision output is desired, the payload size is likely to increase. For instance, when float-32 representation is the desired output for a 16-bit input, the payload bloats by a ratio of 2 (i.e., $\frac{32}{16}$). When using jumbo payloads, super jumbo frames must be supported on the network to accommodate the computation results.

VI. Conclusion

In this paper we demonstrated how to leverage in-network computing for performing scientific operations (that are not supported in network devices) using approximations in a streaming fashion. We used a real-world light source science application for our study. We presented an algorithmic approach to perform unsupported operations such as floating-point and log operations on network switches. We showed how to minimize the error introduced by the approximation. We discussed the challenges, open issues, and possible directions to address them.

Acknowledgments

We thank Tekin Bicer for sharing his expertise in stream processing in light source facilities and for providing the image samples for evaluation experiments. This material is based upon work supported by the U.S. National Science Foundation (NSF), under award 2019073 and the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References

[1] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 99–110.

[2] A. Sapio *et al.*, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017, pp. 150—-156.

[3] N. Katta *et al.*, "HULA: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16. New York, NY, USA: Association for Computing Machinery, 2016.

[4] X. Jin *et al.*, "NetCache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 121—-136.

[5] M. Liu *et al.*, "IncBricks: Toward in-network computation with an in-network cache," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 795—809.

[6] S. Bhattacharyya, D. Katramatos, and S. Yoo, "Why wait? let us start computing while the data is still on the wire," *Future Generation Computer Systems*, vol. 89, pp. 563–574, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18302450

[7] N. Zilberman, "In-network computing," https://www.sigarch.org/in-network-computing-draft/.

[8] "The Report of the BES Advisory Subcommittee on Future X-ray Light Sources," https://science.osti.gov/-/media/bes/besac/pdf/Reports/Future_Light_Sources_report_BESAC_approved_72513.pdf, 2013, accessed: 2020-09-08.

[9] Barefoot Networks, "Tofino - World's fastest P4-programmable Ethernet switch ASICs," https://barefootnetworks.com/products/brief-tofino/.

[10] "Working with P4 in Mininet on BMV2," https://cs344-stanford.github.io/deliverables/p4-mininet/.

[11] M. Shahbaz *et al.*, "PISCES: A programmable, protocol-independent software switch," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 525–538. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934886

[12] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, July 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[13] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "NetPaxos: Consensus at network speed," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2774993.2774999

[14] ONF, "OpenFlow switch specification version 1.5.1," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf, Mar 2015.

[15] T. Jepsen *et al.*, "Life in the fast lane: A line-rate linear road," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3185467.3185494

[16] "xGitLab programmable data planes," https://xgitlab.cels.anl.gov/i2s/prog-data-planes/-/tree/master/normal, (Accessed on 01/14/2021).