Quark: Implementing Convolutional Neural Networks Entirely on Programmable Data Plane

Mai Zhang[1], Lin Cui[1*], Xiaoquan Zhang[1], Fung Po Tso[2], Zhang Zhen[1], Yuhui Deng[1] and Zhetao Li[1]
[1]*Department of Computer Science, Jinan University, Guangzhou, China.*
[2]*Department of Computer Science, Loughborough University, UK.*

***Abstract*—The rapid development of programmable network devices and the widespread use of machine learning (ML) in networking have facilitated efficient research into intelligent data plane (IDP). Offloading ML to programmable data plane (PDP) enables quick analysis and responses to network traffic dynamics, and efficient management of network links. However, PDP hardware pipeline has significant resource limitations. For instance, Intel Tofino ASIC has only 10Mb SRAM in each stage, and lacks support for multiplication, division and floating-point operations. These constraints significantly hinder the development of IDP. This paper presents *Quark*, a framework that fully offloads convolutional neural network (CNN) inference onto PDP. *Quark* employs model pruning to simplify the CNN model, and uses quantization to support floating-point operations. Additionally, *Quark* divides the CNN into smaller units to improve resource utilization on the PDP. We have implemented a testbed prototype of *Quark* on both P4 hardware switch (Intel Tofino ASIC) and software switch (i.e., BMv2). Extensive evaluation results demonstrate that *Quark* achieves 97.3% accuracy in anomaly detection task while using only 22.7% of the SRAM resources on the Intel Tofino ASIC switch, completing inference tasks at line rate with an average latency of 42.66μs.**

I. Introduction

The development of programmable data plane (PDP) [1] has enabled deep programmability for computer networks, allowing for flexible packet processing and forwarding at line rate. Concurrently, the rapid advancement of machine learning (ML) in networking and the gradual adoption of PDP devices have driven the evolution of the intelligent data plane (IDP) [2]. By deploying ML models on PDP, IDP leverages both the line rate processing capabilities of PDP along with the intelligent analytical abilities of ML.

Existing research has demonstrated the feasibility of IDP, primarily in areas like flow classification [3]–[5], network defense [6], congestion control [7], and network telemetry [8]. Some efforts have been made to offload various types of simple ML models onto PDP, such as decision trees [9], [10], binary neural networks [3], [5], [11]. Other efforts have focused on enhancing PDP with additional hardware (e.g., FPGA) in PDP's pipeline [4], [12], [13] to support ML functions. Convolutional neural networks (CNNs) [14], a cornerstone of deep learning, offer significant advantages in tasks like time series prediction and signal identification due to their ability to detect sequential correlations. However, offloading CNNs to PDP presents unique challenges.

Corresponding author: Lin Cui. Email: tcuilin@jnu.edu.cn

Current attempts to implement CNNs on PDP have faced limitations. Some approaches only provide software switch implementations [15], [16], while others require additional devices such as CPUs [17]. These solutions fall short of fully utilizing PDP hardware, such as Tofino ASIC switches, thus introducing additional latency and compromising line-rate performance.

Fully implementing CNNs on PDP hardware switches is necessary but challenging. For PDP hardware, such as switches with PISA (Protocol-independent switch architecture) [18], which is a well-known hardware pipeline architecture in Intel Tofino ASIC, implementing CNNs on the hardware pipeline has the following challenges: *(i)* The lack of support for floating-point operations in PISA, which are essential for CNNs, limits the effective offloading of these models. While high-precision floating-point computation has been achieved on PDP [19], it exhausts PDP resources, making it unsuitable for CNN offloading. *(ii)* The absence of multiplication and division support in the PISA architecture hinders the implementation of convolution and activation operations. Although multiplication can be implemented using bit shifts and addition, this approach consumes excessive stage resources (e.g., an 8-bit multiplication requires 4 stages) *(iii)* PISA's pipeline-based operation can be decomposed into only a limited number of stages (e.g., Intel Tofino 1 offers 12 stages), each with restricted computational and storage capacities. Additionally, the pipeline does not support looping, further complicating CNNs deployment on PDP.

To address these challenges, we propose *Quark* (Quantized convolutional neural network), a framework for offloading CNN inference onto PDP while ensuring high inference accuracy, reducing computational demands, and minimizing forwarding latency. *Quark* employs quantization techniques to convert network models into fixed-point numbers, overcoming the lack of floating-point support in PDP. *Quark* also uses model pruning techniques to eliminate unnecessary neurons or channels in the CNN, mitigating PDP resource constraints and reducing the number of recirculations, thereby reducing inference latency. Moreover, *Quark* achieves multiplication within one stage using match-actions in SRAM. Besides, *Quark* proposes a modular design by abstracting CNN model into multiple units of varying granularity, and maximizes the use of pipeline resources by combining different units to implement various CNN models. Specifically, our contributions are:

- We propose the *Quark* framework, which completely

offloads CNN inference onto PDP, maximizes resource utilization and achieves line-rate CNN inference on hardware switches.
- We utilize pruning and quantization on control plane to compress CNN models to overcome the limitations of hardware resources and floating-point computations. We also propose a modular design to split and reorganize CNN models to maximize the utilization of PDP resources.
- We have implemented a testbed prototype of *Quark* based on the Intel Tofino ASIC switches. Experimental results demonstrate that *Quark* effectively detects anomalies with an accuracy of 97.3%, and performs inference tasks at line rate with 42.66μs latency.

II. Background and Related Works

A. Intelligent Data Plane

With the increasing integration of ML in networking, IDP aims to offload various ML models onto PDP such as PISA switches. Currently, neural network-based IDP focuses primarily on Binary Neural Networks (BNNs) [3], [5], [11], Recurrent Neural Networks (RNNs) [20], and Multi-Layer Perceptrons (MLPs) [15], [21]. These models have significantly contributed to tasks such as traffic classification [3]–[5], [12], network defense [6], [22], congestion control [4], [7], and load balancing [7].

Beyond these models, CNNs excel at extracting time-series features from data flows, proving particularly effective for network functions, especially for network traffic intrusion detection [23] and traffic classification [24]. However, the complex computations involved in CNN have limited their deployment on PISA hardware switches.

To maintain parallel processing capabilities, PISA's available operations and memory are highly restricted [25]. Various solutions have been employed to encode complex ML computations into basic PISA-supported operations. Typically, there are three main approaches: *(i)* Converting floating-point parameters to fixed-point numbers by using quantization techniques to transform operations into PISA-supported integer computations. *(ii)* Expanding the data plane by adding extra hardware modules to handle PISA's resource constraints. *(iii)* Implementing ML models on FPGA [26] or Network Processor [27], as these platforms support operations unavailable in PISA, such as multiplication, division, and looping.

B. Related Work

Table I summarizes existing works on implementing neural networks on PDP. N2Net [11] and N3IC [3] have deployed BNNs on PDP to address the lack of floating-point computation support. BNNs convert the trained parameters of neural networks to 0 or 1, allowing logic operations to approximate multiplication and use integer-based approximations of floating-point [28]. N3IC applies this approach to traffic analysis and deploys it on SmartNICs. However, the binarization of network parameters affects inference accuracy.

TABLE I
State of the art.

Work	Model	Target Platform	Notes
N2Net [11]	BNN	Unknown Switches	Precision degradation (Binary)
N3IC [3]	BNN	NetFPGA	Precision degradation (Binary)
Taurus [4]	DNN	Intel Tofino ASIC with FPGA	Need additional Hardware Block
INQ-MLT [15]	CNN & MLP	BMv2 (software)	Excessive model complexity
BaNaNa Split [17]	NN	SmartNICs/Unknown Switches with CPU	High latency between PDP and CPU
NetNN [16]	DNN	BMv2 (software)	High cost and latency by multiple switches
Quark (ours)	CNN	Intel Tofino ASIC (hardware) & BMv2 (software)	Entirely on hardware PDP, low latency and high accuracy

Besides binarization, other works have verified the feasibility of IDP on software switches (e.g., BMv2 [29]). NetNN [16] addresses the processing and memory constraints of the PISA architecture by partitioning deep neural networks (DNNs) across multiple switches on a layer-by-layer basis. Additionally, NetNN stores the dot product results in ternary content-addressable memory (TCAM) tables to reduce stage resource consumption. INQ-MLT [15] employs quantization techniques to convert floating-point operations into integer computations supported by PDP, addressing the lack of floating-point support in PISA. The system also uses Quantization Aware Training (QAT) [30] to mitigate accuracy loss caused by quantization.

In addition, some works use additional hardware devices to assist PDP in achieving ML functionality. Taurus [4] extends PISA by incorporating a custom hardware MapReduce block between two stages in the pipeline. The blocks, emulated using FPGA, work alongside parsers and match-action tables (MATs) to handle packet forwarding. BaNaNa Split [17] also uses quantization, splitting the neural network model, with one part processed by the server CPU and the other part quantized and integrated into programmable network devices.

C. Discussion

Although binarization effectively addresses PDP floating-point constraints, it results in notable accuracy loss. Using quantization instead of binarization mitigates this drawback but introduces higher computational demands, requiring further optimization. Schemes deployed on software switches have also proposed methods to address PDP resource limitations, such as using TCAM to store dot product results. However, software switches have much lower processing efficiency compared to hardware switches, and the stricter resource limitations on hardware switches hinder the further development of these schemes. Using extra hardware (e.g., FPGA) to assist the hardware switch introduces inevitable delay between the hardware and PDP. This approach has lower scalability and flexibility and incurs additional costs.

Our objective is to design a solution that simplifies CNN models and fully offloads them onto the data planes of programmable hardware devices (e.g., Intel Tofino ASIC switch) without modifying existing switch hardware. This approach

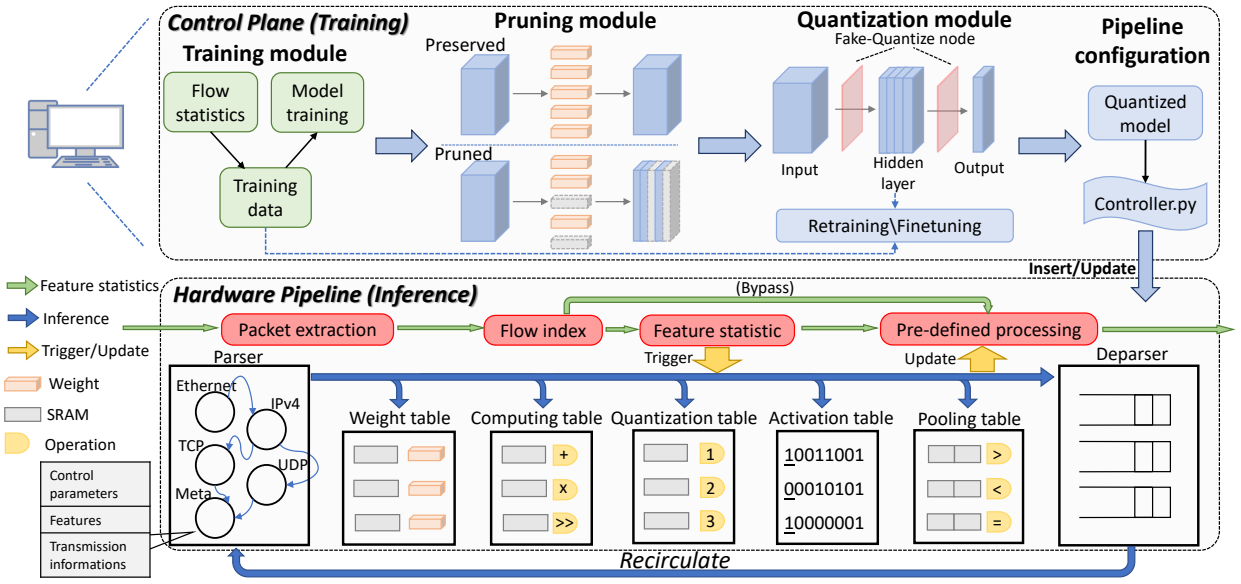


Fig. 1. An overview of the *Quark* architecture.

aims to address the resource and stage limitations of the PISA architecture, achieving efficient inference performance while maintaining high accuracy and low latency.

III. Overview of *Quark*

As shown in Fig. 1, the architecture of *Quark* contains two main components: the control plane and the hardware pipeline (i.e., data plane).

A. Control Plane

The primary task of the control plane is to train a CNN model and insert or update its parameters into the hardware pipeline. The workflow is as follows: *(i)* Model Training: The model training module analyzes flow statistics to obtain training data and trains a floating-point model. *(ii)* Pruning: The pruning module performs channel pruning to remove less important parameter channels from the pre-pruned model. *(iii)* Quantization: The quantization module employs QAT by adding fake-quantize nodes to the pruned model to simulate precision loss during quantized inference. The model is then retrained or fine-tuned for maintaining accuracy (see Section IV). *(iv)* Pipeline configuration: the pipeline configuration extracts the model parameters and uses the controller's API (e.g., P4Runtime) to insert or update them into the pipeline.

This architecture design enables efficient deployment of CNN models on PDP, ensuring high accuracy while reducing computational and storage resource requirements.

B. Hardware Pipeline

The hardware pipeline has two distinct workflows: network flow feature statistics (green arrow) and neural network inference (blue arrow), as shown in Fig. 1.

1) Feature Statistics: This workflow extracts network flow feature statistics (see Section V-B). *(i)* When the switch receives a normal packet, it extracts the header information and retrieves the corresponding data via a flow index (e.g., whether the flow has already been predicted). *(ii)* If prediction information is already indexed, feature statistics are skipped, and predefined processing is executed. *(iii)* If the prediction information is not indexed, feature statistics is performed, and the packet is forwarded. *(iv)* Once certain conditions are met during feature statistics (e.g., counting the first n packets), the neural network inference workflow is triggered.

2) Neural Network Inference: In this workflow, we design a CNN inference model composed of multiple basic operations such as parser, deparser, MATs, and ALUs. *(i)* The modular design (see Section V-A) splits the CNN inference model into several basic units. *(ii)* If the pipeline cannot fit the entire inference model, packets are cloned and recirculated within the pipeline to achieve complex calculations like convolution operations. This includes the weight table for retrieving weights, the computing table for handling the lack of multiplication operations in PDP, the quantization table for quantizing convolution results, the activation table for performing activation layer operations, and the pooling table for pooling operations. Parsers and deparsers handle packet header parsing and execute recirculate operations. *(iii)* Once inference is completed, the pipeline updates the prediction results in the flow index, which serves as the entry for subsequent packets.

IV. Model Compression for *Quark*

A. Pruning Module

Quark employs model pruning techniques to simplify neural network structures. Given the high programmability and parallel computing capabilities of the PISA, efficient use of computational resources is crucial. Pruning drastically reduces the number of parameters while preserving accuracy [31]. Specifically, *Quark* evaluates the importance of weights to identify and remove channels that minimally contribute to the model's prediction (see Fig. 1). This method achieves significant compression with minimal accuracy loss, prevents overfitting, and enhances the model's generalization ability and stability. Evaluation results demonstrate that the pruned CNN

model of *Quark* exhibits higher computational efficiency and lower resource consumption on PDP while preserving high prediction accuracy.

B. Quantizing CNN Parameters for Quark

Due to the lack of support for floating-point operations within the pipeline, *Quark* employs a quantization module to convert the parameters of convolutional and fully connected layers in CNNs from floating-point precision (e.g., Float32) to lower-precision fixed-point integers (e.g., INT8). Fixed-point calculations can be performed within the PISA pipeline using basic operations, significantly reducing the complexity of neural network inference without substantial accuracy loss.

We use r to denote the floating-point numbers and q to denote the quantized fixed-point integers. The quantization process maps the parameter r from the continuous range $[r_{min}, r_{max}]$ to q within the range $[q_{min}, q_{max}]$. The range bounds q_{min} and q_{max} depend on the bit-width b of the integer q, as well as whether the integer is signed or unsigned.

$$(q_{min}, q_{max}) = \begin{cases} \left(-2^{b-1}, 2^{b-1}-1\right), & signed \\ \left(0, 2^{b}-1\right), & unsigned \end{cases} . \quad (1)$$

Two key definitions for *Quark* are S (scale) and Z (zero-point). The scale S is the factor that converts floating-point numbers to fixed-point numbers while preserving their relative precision and proportion. The definition of S is:

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}} . \quad (2)$$

The zero-point Z represents the integer value corresponding to zero in floating-point numbers after quantization, ensuring consistency of relative relationships between numbers before and after quantization. The definition of Z is:

$$Z = \text{Round}(q_{max} - \frac{R_{max}}{S}) . \quad (3)$$

These definitions allow *Quark* to convert from floating-point numbers to fixed-point integers:

$$r = S(q - Z) , \quad (4)$$

$$q = \text{Clamp}(\text{Round}(\frac{r}{S} + Z), q_{min}, q_{max}) . \quad (5)$$

The function of Clamp is to constrain the quantized fixed-point integer q within the fixed-point range $[q_{min}, q_{max}]$. Thus, *Quark* can use Equation (5) to complete the quantization of parameters such as weights and biases.

C. Quantizing Convolution for Quark

In CNN, both convolutional layers and fully connected layers essentially perform matrix multiplications. Suppose r_1 and r_2 are two $N \times N$ matrices in floating-point representation, and r_3 is the matrix resulting from their multiplication. $r_3^{i,j}$ is the element in the i-th row and j-th column of matrix r_3.

$$r_3^{i,j} = \sum_{k=1}^{N} r_1^{i,k} \cdot r_2^{k,j} . \quad (6)$$

The convolution principle of CNN refers to performing a matrix multiplication between the weight matrix w and the input matrix x, and adding the bias matrix b to obtain the output matrix a. By substituting the above parameters into Equation (6) and simplifying the matrix notation, we obtain:

$$a = \sum_{i=1}^{N} w_i x_i + b . \quad (7)$$

Assuming S_x, Z_x are the scale and zero-point for the input matrix x, and similarly, S_w, Z_w for w and S_a, Z_a for a. Additionally, q_x, q_w, and q_a represent the fixed-point matrices corresponding to matrices x, w, and a, respectively. According to the Equation (4), we can derive:

$$S_a(q_a - Z_a) = \sum_{i}^{N} S_w(q_w - Z_w) S_x(q_x - Z_x) + S_b(q_b - Z_b) , \quad (8)$$

$$q_a = \frac{S_w S_x}{S_a} \sum_{i}^{N} (q_w - Z_w)(q_x - Z_x) + \frac{S_b}{S_a}(q_b - Z_b) + Z_a . \quad (9)$$

Here, the non-integer parts are limited to $S_w S_x$ and S_b. S and Z serve as intermediaries for converting between floating-point numbers and fixed-point integers, which only need to ensure that the conversion between r and q is reversible. Therefore, we can use $S_w S_x$ to replace S_b, and set Z_b directly to zero. Thus, the Equation (9) can be adjusted to:

$$q_a = M(\sum_{i}^{N} (q_w - Z_w)(q_x - Z_x) + q_b) + Z_b , \quad (10)$$

where

$$M = \frac{S_w S_x}{S_a} . \quad (11)$$

Equation (11) can be approximated as a fixed-point number with a bit shift on PISA hardware pipeline. These equations effectively implement quantized convolution operations within CNN on PDP which lack support for floating-point operations, ensuring both computational efficiency and accuracy.

D. Quantization Aware Training

The initial quantization parameters of the CNN was achieved using Equation (5) to calculate directly from floating-point numbers. However, without retraining the model post-quantization, the model parameters' accuracy may be affected.

To address this issue, *Quark* utilizes Quantization Aware Training (QAT), an optimization technique that simulates the quantization process during training, thereby improving the model's performance in deployment [30].

In QAT, fake-quantize nodes are inserted into the network model (see Fig.1). During forward propagation, QAT simulates the Clamp and Round operations of quantization, helping the model adapt to quantization effects and achieve higher accuracy. In backward propagation, the straight-through estimator (STE) [32] approximates the gradient of the quantization operation, as the Round function has zero gradient. This allows the gradient to propagate back to the weights before the fake-quantize nodes. As a result, the weights undergo pseudo-quantization, simulating quantization errors, and the gradients

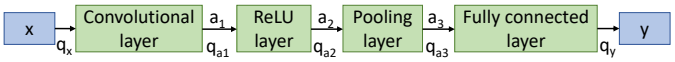


Fig. 2. A example of Quantized CNN model.

of these errors are propagated back to update the original weights, allowing the model to adapt to quantization.

E. Quantizing CNN Training and Inference

We use an example CNN model in Fig. 2 to illustrate the CNN quantization process of *Quark* during forward propagation. Let the input and output be x and y, with the outputs of the hidden layers denoted as a. The quantized fixed-point numbers corresponding to these parameters are q_x, q_y, and q_a.

During training, forward passes with sufficient data are performed to record the value ranges $[r_{min}, r_{max}]$ for x, a_1, a_2, a_3, and y. At the end of training, these recorded ranges are used to pre-calculate certain elements (e.g., S, M, Z_x, and Z_w) using Equations (2), (3) and (11).

During inference, *Quark* first quantize the input x to a fixed-point integer q_x, and then use Equation (10) to compute the result of the first layer q_{a1}. This process continues through the ReLU and Pooling layers. For fully connected layers, the quantization process is similar to that of convolutional layers, allowing us to compute q_y and obtain the final prediction.

This quantization method is scalable to larger networks due to its consistent underlying principles, and therefore establishing a robust quantization approach for CNN inference.

V. *Quark* Design in Data Plane

This section details the design and deployment of a quantized CNN on the data plane of a PISA hardware switch.

A. Unit-based Modularization of CNN

To fit a CNN inference model into a PDP hardware pipeline, we propose a modular design that abstracts and splits a CNN model into multiple units of varying granularity. For example, activation and pooling layers, with relatively low computational complexity, can be combined with convolutional layers into a single unit (see Section V-C for an example). Similarly, fully connected layers and activation layers can be merged into another unit. Different units can be defined by varying scales, allowing each of them to process one or multiple features. This flexibly allows *Quark* to tailor the CNN units to fit within the constraints of the available pipeline stages.

A CNN inference model can be achieved through the combination of one or multiple units. In PISA-based programmable devices, the data planes of different hardware devices offer varying amounts of resource space. We recommend deploying as many units as possible within a single pipeline. When necessary, the recirculation technique can be employed to force data packets to enter the pipeline multiple times for repeated unit processing. This modular design maximizes the utilization of the data plane's resources, completes CNN inference with minimal latency, and provides a flexible and scalable solution for deploying CNNs on the PDP.

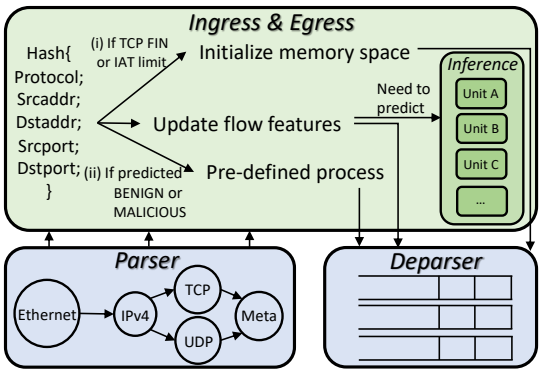


Fig. 3. Packet processing pipeline.

B. Quantized CNN Design in Pipeline

We have implemented *Quark* on both P4 hardware (Intel Tofino ASIC) switches and BMv2. The processing pipeline of Quantized CNN in the PDP is illustrated in Fig. 3.

Parser: The parser extracts features from the packet header (e.g., packet length, TCP flags, IAT). These features play various roles in the ingress stage based on the flow's state.

Ingress & Egress: Packets are hashed into the corresponding flow and processed as follows: *(i)* If the packet header carries flags such as TCP FIN flag or exceeds the IAT limit, the corresponding flow's memory space is initialized; *(ii)* If the flow matches pre-defined operations, it is classified according to the pre-defined results; *(iii)* The flow's features are updated. If the packet triggers a specific condition (e.g., the arrival of the n-th packet), neural network inference is triggered, and the inference result is updated in the pre-defined results.

Parameters: Since biases and weights are fixed after training and quantization, storing them in match-action tables (MATs) is efficient. Due to the pipeline's characteristics, metadata is initialized to zero during recirculation, so *Quark* stores the layer outputs in the header. Additionally, parameters to control the inference process are required (see Table II).

C. Implementation of CNN in the P4 Switch

For simplicity, we use one-dimensional CNN (1D-CNN) for this introduction, with similar concepts extendable to multi-dimensional CNNs. A 1D-CNN unit, shown in Fig. 4, is implemented on the Intel Tofino ASIC switch. Given the simplicity of activation and pooling layer computations, requiring only one stage each, we combined convolutional/fully connected, activation, and pooling layers into a single unit, termed as CAP-Unit (Convolutional, Activation and Pooling Layer Unit). To maximize pipeline resource utilization, we designed

TABLE II
Parameter Description Table.

Parameter	Description
layer_index	The current inferencing layer. All features reading and storage depend on this variable.
channel_index	The channel currently being computed. It manages data storage and reading together with the *layer_index*.
conv_flag	Whether accumulation is completed to determine if activation and pooling operations can be performed.
input_index	Which two sets of features in the input matrix are currently being computed.

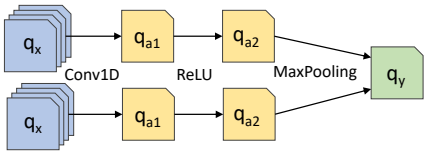


Fig. 4. The structure of a CAP-Unit.

```
apply {
    /* (i) Retrieve the inputs from the header */
    get_input_tbl.apply();
    /* (ii) Retrieve the weights and biases */
    weight_tbl.apply(); bias_tbl.apply();
    /* (iii) Computing two sets of features
         separately and storing the accumulated
         convolution results */
    multi_tbl_1.apply(); multi_tbl_2.apply();
    hdr.hdr_meta.acc_temp_1 += multi_result_1;
    hdr.hdr_meta.acc_temp_2 += multi_result_2;
    /* When all channels have been accumulated,
         proceed with steps (iv)~(vii) */
    if (conv_flag == current_channel_num) {
       /* (iv) Quantize accumulation results */
       hdr.hdr_meta.acc_temp_1 += bias;
       hdr.hdr_meta.acc_temp_2 += bias;
       quanti_tbl_1.apply(); quanti_tbl_2.apply();
       /* (v) ReLU module */
       if (result_1 < 0) { result_1 = 0; }
       if (result_2 < 0) { result_2 = 0; }
       /* Skip (vi) when fully connected layer, do
          result_1 += result_2; */
       /* (vi) Maxpooling module */
       maxPooling_tbl.apply();
       /* (vii) Storage */
       if (layer_index == 1) {
          storage_tbl_1.apply();
          /* Restart next Accumulation*/
          conv_flag=0; channel_index++;
          /* When finish current input features*/
          if(channel_index > current_channel_num){
             input_index++; channel_index = 0;}
          /* When finish current layer*/
          if (input_index > features_num) {
             layer_index++; input_index = 0;}
       }else if(layer_index == 2){
          //... }}
    /* recirculate */
    //...
}
```

Listing 1. $P4_{16}$ code of CAP-Unit.

each CAP-Unit to process two features simultaneously. Given Tofino's 12-stage limit, one CAP-Unit is deployed per pipeline, with the recirculate technique used to reprocess inference packets and complete the full CNN inference.

Listing 1 presents the general logic of $P4_{16}$ code of our design. We divide the computation process of the CAP-Unit into seven main steps (see Fig. 5). Initially, *Quark* extracts the intermediate and control parameters in the header using the parser. (*i*) Using the *layer_index* and *input_index*, the input q_x corresponding to the current layer are obtained and sent to the next stage. (*ii*) Based on the *layer_index*, the weights and biases of the current layer are extracted from the MATs.(*iii*) Since multiplication is not supported in the Tofino switch, a multiplication MAT is designed to perform the calculations. Although bit-shift multiplication is possible, it requires excessive pipeline resources (e.g., an 8-bit multiplication requires 4 stages). Instead, *Quark* stores

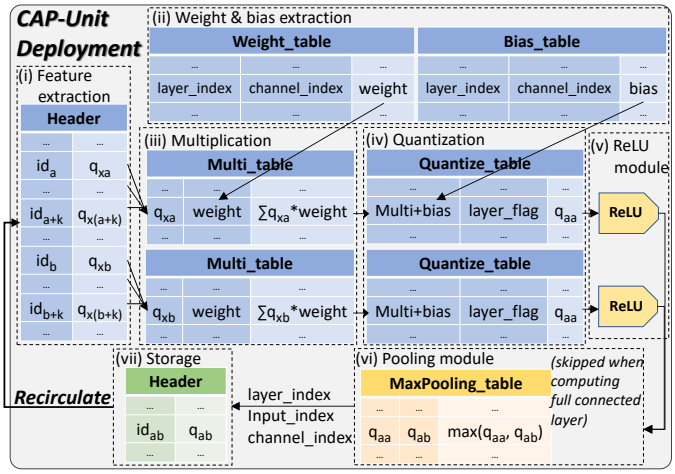


Fig. 5. Deployment of the CAP-Unit in P4 pipeline.

all multiplication results in a MAT to reduce the number of required stages. (*iv*) According to Equation (10), after completing the $\sum (q_w - Z_w)(q_x - Z_x)$ operation, the accumulation result adds the bias, multiplies by M, and adds the zero-point to complete the quantization.Since floating-point operations are not supported in PISA and M remains constant during inference, *Quark* pre-compute the product of M and the previous results, storing them in a MAT to utilize relatively abundant SRAM resources and improve computation efficiency. (*v*) *Quark* uses ReLU for the activation layer, setting the convolution result to zero if it is less than zero, thus introducing non-linearity into our CNN. (*vi*) For the pooling layer, *Quark* uses maxpooling to extract features and reduce dimensionality by selecting the maximum value in each region, retaining the most significant features while reducing the computational load. This step is skipped when computing the fully connected layers. (*vii*) *Quark* stores the results into the header, controlled by *layer_index*, *input_index*, and *channel_index*. The storage method for *Quark* ensures that inputs are fully utilized before being overwritten, and avoids dirty data reads (see Section V-D2 for more details). If all computations for the current layer are complete, *Quark* modify the *layer_index* to proceed to the next layer.

Additionally, the output of a single CAP-Unit corresponds to the output of a single channel in the CNN. Since hidden layers in CNN typically consist of multiple channels, the final output of a hidden layer is the sum of the outputs from each channel. To facilitate this, *Quark* utilize a temporary variable in the header to store the accumulated results from multiple CAP-Units during the recirculation operation (e.g., *hdr_meta.acc_temp_1* and *hdr_meta.acc_temp_2* in Listing 1).

D. Discussion and Analysis

1) Number of Recirculations: With the modular design, if the PDP pipeline can not accommodate all units of a CNN model due to resource constraints, recirculations will be required. We analyze the upper bound of the number of recirculations when the CAP-Unit is used for the modularization. Table III defines the symbols.

Theorem 1. *When deploying a CNN model on PDP pipeline using the CAP-Unit, the maximum recirculations required is* $\lceil (T + L_{conv} + L_{fc}) \cdot C^2 \rceil$, *where* $C = \max_{1 \le n \le L_{conv}, 1 \le m \le L_{fc}} \{C_{in}^{(n)}, C_{out}^{(n)}, T_{in}^{(m)}, T_{out}^{(m)}\}$.

Proof. Let U denote the total number of CAP-Units of a CNN model:

$$U = \sum_{n=1}^{L_{conv}} C_{in}^{(n)} \cdot C_{out}^{(n)} \cdot \lceil T/2^n \rceil + \sum_{m=1}^{L_{fc}} T_{out}^{(m)} \cdot \lceil T_{in}^{(m)}/2 \rceil \,.$$

Assume the PDP pipeline can accommodate at most p CAP-Units where $p \ge 1$, then the number of recirculations required is $R = \lceil U/p \rceil$. Considering C is the maximum of the all layers' input/output channels or features ($C \ge 2$), we have

$$\begin{aligned} R = \lceil U/p \rceil &\le \lceil (\sum_{n=1}^{L_{conv}} C^2 \cdot \lceil T/2^n \rceil + \sum_{m=1}^{L_{fc}} C \cdot \lceil C/2 \rceil)/p \rceil \\ &\le \lceil (T \cdot C^2 + L_{conv} \cdot C^2 + L_{fc} \cdot (C^2/2 + C))/p \rceil \\ &\le \left\lceil (T + L_{conv} + L_{fc}) \cdot C^2/p \right\rceil \,. \end{aligned}$$

In the worst case, the pipeline can only deploy one CAP-Unit, i.e., $p = 1$. Therefore, $R \le \lceil (T + L_{conv} + L_{fc}) \cdot C^2 \rceil$. ■

This allows us to accurately determine the number of recirculations required for the entire CNN, which is useful when deploying *Quark* in resource-constrained PDP hardware.

2) Header Bits Allocation: Since metadata is initialized to zero during recirculation, *Quark* stores the input and hidden layers outputs in the header. However, the header size is constrained, so *Quark* reuses the allocated bit positions in the header. The outputs q_a computed by a CAP-Unit layer lose their utility after serving as inputs q_x for the subsequent CAP-Unit layer. Thus, they can be overlaid by other outputs.

For convolutional layers, we need to allocate bits positions to a layer and the first group in the subsequent layer:

$$Conv\_bits = (C_{out}^{(k)} \cdot \lceil T/2^k \rceil + C_{in}^{(k+1)}) \cdot b \,,$$

where b represents the quantization bit levels of *Quark*, and $k = \text{argmax}_k \, C_{\text{out}}^{(k)} \lceil T/2^k \rceil$.

For fully connected layers, we need to allocate bit positions to the layer input features and output features:

$$Fc\_bits = (T_{in}^{(l)} + T_{out}^{(l)}) \cdot b \,,$$

where $l = \text{argmax}_l (T_{in}^{(l)} + T_{out}^{(l)})$.

Thus, the maximum number of header bits required is:

$$Header\_bits = \max(Conv\_bits, Fc\_bits) \,.$$

TABLE III
SYMBOLS AND DESCRIPTIONS.

Symbol	Description
T	Number of input features to the first layer
L_{conv}	Number of convolutional layers
L_{fc}	Number of fully connected layers
$C_{in}^{(n)}$	Number of input channels for the n-th convolutional layer
$C_{out}^{(n)}$	Number of output channels for the n-th convolutional layer
$T_{in}^{(m)}$	Number of input features for the m-th fully connected layer
$T_{out}^{(m)}$	Number of output features for the m-th fully connected layer

TABLE IV
FEATURES AND DESCRIPTION.

Feature Name	Description
length_max	The maximum packet length in the flow
length_min	The minimum packet length in the flow
length_total	The total packet length in the flow
TCP_flag	Cumulative number of occurrences of the FIN, SYN, ACK, PSH, RST, ECE flag
IAT	Inter-arrival time between two adjacent packets

Quark efficiently manages and stores computational data within the limited header space, ensuring optimal utilization of pipeline resources.

VI. EVALUATION

We have implemented a testbed of *Quark* on both P4 hardware switch (with Intel Tofino ASIC) and P4 software switch (i.e., BMv2) using P4_{16} language.

A. Experiment Setup

Testbed: We evaluate the performance of *Quark* by deploying a CNN inference model on a P4 hardware switch (Flnet S9180-32X with Intel Tofino ASIC) connecting two servers (each equipped with Intel Core i7-4771 CPU @ 3.50GHz and 40GbE network interface cards). One server is the sender to replay the traffic of datasets, and the other server is the receiver. Links connecting the P4 switch and servers are 40Gbps fiber optics.

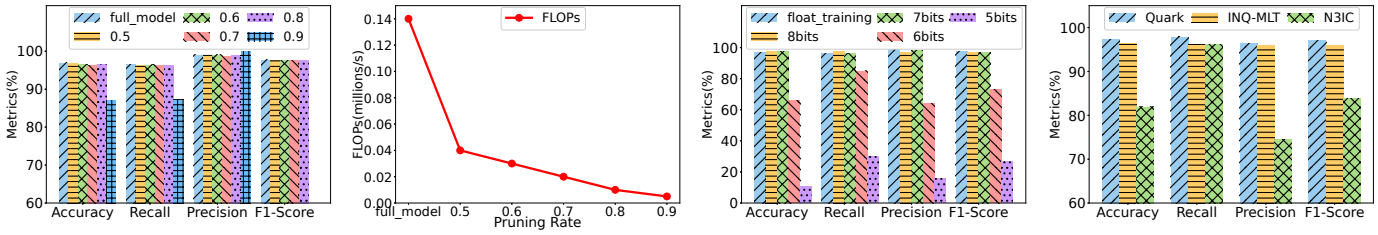
In addition to testbed with hardware switch, we also build a similar network topology in Mininet [33] consisting of one P4 software switch (BMv2) and two hosts.

Datasets and Usecases: We utilize publicly available datasets containing network traffic traces to train and test the CNN model. The features of the first eight packets extracted from datasets are listed in Table IV.

- *Anomaly Detection*: ISCX Botnet Dataset [34] collects heterogeneous botnet and malware traffic alongside non-malicious traffic from real-world scenarios. We extract the features from the TCP and UDP flows of this dataset, classifying the traffic into *Benign* and *Malicious* categories. The training and test sets are divided according to the official guidelines provided by ISCX.
- *Flow Classification*: CICIDS-2017 dataset [35] includes various types of network traffic and attack scenarios. We extract flow features from the pcap files and classified the flows into *Benign*, *DDoS*, *Patator* and *PortScan* categories. Undersampling is used to handle the imbalance of the dataset. The dataset is split into 60% training, 20% validation, and 20% testing subsets.

Model Training: We train the CNN model on a GPU server equipped with an NVIDIA GeForce RTX 2080 Ti. The CNN model consists of three convolutional layers (c_1=16, c_2=16, c_3=16) and two fully connected layers (l_1=16, l_2=15). Each convolutional layer is followed by a ReLU layer and a maxpooling layer, and each fully connected layer is followed by a ReLU layer.

Comparison Schemes: We compare *Quark* with N3IC [3] and INQ-MLT [15]. Since N3IC is designed on NetFPGA



(a) Impact of different pruning rate. (b) FLOPs of different pruning rate. (c) Impact of different quantization bit. (d) Metrics of anomaly detection.

Fig. 6. Model performance of *Quark* on anomaly detection.

using BNN model, we only evaluate its model performance on control plane. INQ-MLT deploys CNN model on software switches and we have implemented it on BMv2 in our evaluation. Both N3IC and INQ-MLT can not be implemented on P4 hardware switches (with Intel Tofino ASIC) due to the restriction of hardware pipeline.

B. Impact of Pruning Rate and Quantization Bit Level

We first conduct two experiments on anomaly detection to evaluate model performance of *Quark*. The first experiment, we fix the model size and vary the pruning rates to observe their impact on model performance. The second experiment, we fix the model size and pruning rate, then vary the quantization bit levels to assess their effect on model performance.

Pruning Rate: As shown in Fig. 6a, pruning rates between 0.5 and 0.8 result in less than a 1% decrease in all metrics compared to the original pre-compressed model, while the number of floating-point operations per second (FLOPs) decreases from 0.14M to 0.01M (shown in Fig. 6b). This trade-off, sacrificing less than 1% of performance for a 92.9% reduction in computation (at a pruning rate of 0.8), is both acceptable and beneficial for deploying large models on PDP hardware. However, when the pruning rate reaches 0.9, the recall increases to 99.99% and other metrics drop significantly. This indicates the model predicts almost all samples as positive due to the severely reduced number of channels after pruning. Therefore, the pruning rate should not be set too high.

Quantization Bit: We set the pruning rate to 0.8 as the baseline for floating-point training. As shown in Fig. 6c, reducing the quantization to 7 bits only causes a slight performance drop (less than 1%) but significantly reduces memory usage, which is crucial for the limited SRAM in PDP hardware pipeline. However, reducing the quantization to 6 bits or lower leads to severe performance degradation. At 5 bits, accuracy drops to 10.6%, which is unacceptable.

In conclusion, selecting an appropriate pruning rate and quantization bit level can effectively conserve memory resources while preserving model performance. In the following experiments, *Quark* uses a pruning rate of 0.8 and 7 bits quantization for model compression.

C. Model Performance Comparison

We compare the model performance of *Quark* with N3IC [3] and INQ-MLT [15] on a server (i.e., control plane). For fairness, we use the same training data for all methods, and

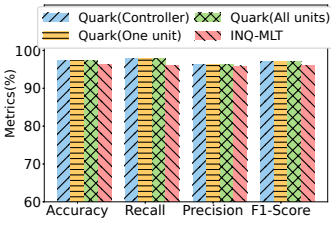


Fig. 7. Anomaly detection metrics on P4 software switch.

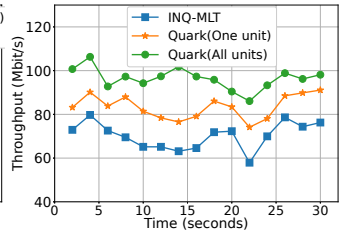


Fig. 8. Throughput on P4 software switch.

select the largest binary MLP model [128, 64, 10] for N3IC and the same 1D-CNN model for *Quark* and INQ-MLT.

The evaluation results for the two tasks are shown in Fig. 6d and Table V. In anomaly detection, *Quark* performs the best among the three schemes, with an F1-Score improvement of 0.130 over N3IC and a slight increase of 0.010 over INQ-MLT. In the more complex flow classification scenario, *Quark* also demonstrates an average F1-Score improvement of 0.130 compared to N3IC. Both *Quark* and INQ-MLT exhibit distinct advantages in four classifications, with no significant difference in overall F1-Score.

The inferior performance of N3IC is primarily attributed to the accuracy loss caused by binarizing the model weights. The performance of INQ-MLT and *Quark* align with our assessments of the pruning rate's impact (see Fig. 6a). *Quark* utilizes a pruning rate of 0.8, which significantly reduces the model size while maintaining superior performance.

D. Performance in the P4 Software Switch

We implement and compare *Quark* with INQ-MLT on BMv2. For *Quark*, only one CAP-Unit is deployed in the pipeline of BMv2. Additionally, we propose deploying as many units as possible within a single pipeline. Given the ample resources of BMv2, *Quark* offers an alternative scheme to deploy all units within a single pipeline. We first verify whether deploying *Quark* to BMv2 had any impact on model performance. As shown in Fig. 7, the performance of *Quark* is consistent on both controller and BMv2 (PDP pipeline).

Comparison with INQ-MLT: As shown in Fig. 7, *Quark* demonstrates approximately a 1% improvement in model performance. However, benefiting from its pruning module, *Quark* only utilizes 20% of the model size (with a pruning rate of 0.8) and 7.1% of the computations (see Fig. 6b) of INQ-MLT, resulting in a significant throughput increase. As illustrated in Fig. 8, *Quark* achieves an average throughput improvement of 18.8% over INQ-MLT.

TABLE V
PERFORMANCE FOR FLOW CLASSIFICATION.

Method	*Quark*			N3IC			INQ-MLT		
Metrics	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Benign	0.931	0.942	0.935	0.783	0.832	0.807	0.918	0.951	0.934
DDoS	0.619	0.625	0.615	0.494	0.463	0.478	0.637	0.612	0.624
Patator	0.563	0.635	0.593	0.307	0.434	0.360	0.541	0.642	0.587
PortScan	0.597	0.886	0.706	0.631	0.746	0.684	0.612	0.901	0.729
Overall F1		0.712			0.582			0.718	

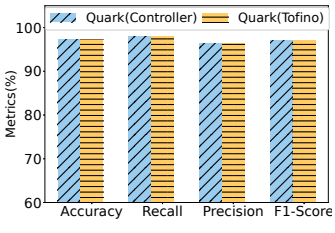


Fig. 9. Anomaly detection metrics on P4 hardware switch (Tofino).

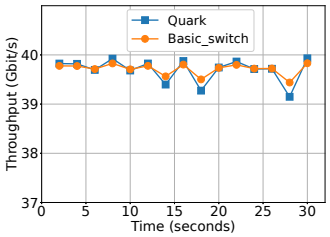
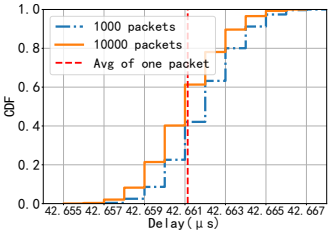
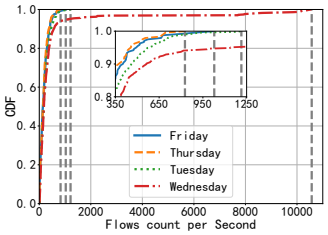


Fig. 10. Throughput on P4 hardware switch (Tofino) with 40Gbps fibers.





(a) Inference latency under different numbers of concurrent flows. (b) The number of concurrent flows.

Fig. 11. Inference latency for CNN on P4 hardware switch.

TABLE VI
HARDWARE RESOURCE CONSUMPTION ON INTEL TOFINO ASIC.

Computational	eMatch xBar	tMatch xBar	Gateway	VLIW	Hash bits
Usage	3.26%	0.00%	13.02%	20.83%	14.20%

Memory	SRAM	TCAM	Map RAM	TableID	Stash
Usage	24.27%	0.00%	0.00%	25.00%	9.38%

PHV	8bits used	16bits used	32bits used	overall bits used
Usage	8.01%	27.50%	4.69%	13.60%

Comparison with All Units per Pipeline: As shown in Fig. 7, both *Quark* solutions exhibit consistent model performance. In terms of throughput (see Fig. 8), the solution deploying all units per pipeline shows an improvement of 15.6%. Compared to the INQ-MLT scheme, there is a significant 37.3% increase in throughput. This indicates that deploying as many units as possible in the pipeline effectively enhances forwarding performance. However, due to resource constraints in Tofino, only one unit can be deployed per pipeline.

E. Performance in the P4 Hardware Switch

We have successfully implement one CAP-Unit in the P4 pipeline of Tofino hardware switch. With 102 recirculations, *Quark* deploys the compressed CNN model, utilizing a pruning rate of 0.8 and 7-bit quantization.

Accuracy: As shown in Fig. 9, *Quark*'s prediction metrics on Intel Tofino ASIC are consistent with those of the controller, achieving a 0.971 F1-Score and a 97.3% accuracy in anomaly detection.

Throughput: To evaluate *Quark*'s throughput, we implement a simple switch forwarding application, called basic_switch, as a baseline for comparison. As shown in Fig. 10, *Quark* achieves 39.696 Gbps throughput, only 0.04% below the baseline's 39.712 Gbps, demonstrating line-rate operation.

Inference Latency: The inference latency refers the delay from the start of inference when a packet enters the switch to the completion of inference and packet forwarding. We evaluate both single and concurrent inference latency, with the CDF results shown in Fig. 11a. For single inference, an average latency of 42.66μs is observed from 1000 individual tests (see red dashed line).

Analysis of the CICIDS dataset reveals that, in most cases, approximately 1000 flows arrive simultaneously within one second, with a maximum of 10571 flows on Wednesday (see Fig. 11b). In the concurrent flow test, the blue dotted line represents 1000 concurrent inference latency, showing an average latency of 42.66μs with fluctuations under 0.01μs. The orange solid line shows similar performance for 10000 concurrent inferences. These evaluations demonstrate that *Quark* maintains the same performance as with a single flow, even under high concurrency conditions.

Resource efficiency: Table VI shows the hardware resource consumption of *Quark* on Tofino switch. In terms of memory resources, MATs of *Quark* entirely rely on exact matching, which does not consume TCAM or Map RAM resources, leaving 75.73% of the SRAM available. Additionally, 75.00% of the table ID resources in Tofino ASIC remain unused. Furthermore, *Quark* consumes only 13.6% of the PHV bits. Overall, the evaluation results demonstrate that *Quark* only consumes small portions of resources on Tofino ASIC, allowing the switch to concurrently deploy many other functionalities.

VII. CONCLUSION

In this paper, we proposed *Quark* to enable CNN inference to be deployed entirely at line speed on the programmable hardware data plane. *Quark* incorporates model compression to reduce network size while maintaining high accuracy and applies quantization to address floating-point limitations on PISA switches. Additionally, *Quark* modularizes CNNs into unit-based combinations to optimize pipeline resource utilization and enable deployment on PDP Experimental results demonstrate that *Quark* achieves high inference accuracy with microsecond-level latency at line rate. The authors have provided public access to their code and/or data at https://github.com/AntLab-Repo/Quark-CNN-P4.

ACKNOWLEDGEMENT

This work is partially supported by National Natural Science Foundation of China (NSFC) under Grant 62172189; the Innovate UK grants 10098627 and 10106629.

REFERENCES

[1] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–36, 2021.

[2] W.-X. Liu, C. Liang, Y. Cui, J. Cai, and J.-M. Luo, "Programmable data plane intelligence: Advances, opportunities, and challenges," *IEEE Network*, vol. 37, no. 5, pp. 122–128, 2023.

[3] G. Siracusano, S. Galea, D. Sanvito, M. Malekzadeh, G. Antichi, P. Costa, H. Haddadi, and R. Bifulco, "Re-architecting traffic analysis with neural network interface cards," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022, pp. 513–533.

[4] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun, "Taurus: A data plane architecture for per-packet ml," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 1099–1114.

[5] Q. Qin, K. Poularakis, K. K. Leung, and L. Tassiulas, "Line-speed and scalable intrusion detection at the network edge via federated learning," in *IFIP Networking Conference (Networking)*, 2020, pp. 352–360.

[6] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos, and A. Madeira, "Flowlens: Enabling efficient flow classification for ml-based network security applications," in *Network and Distributed System Security Symposium (NDSS)*, 2021.

[7] T. Mai, S. Garg, H. Yao, J. Nie, G. Kaddoum, and Z. Xiong, "In-network intelligence control: Toward a self-driving networking architecture," *IEEE Network*, vol. 35, no. 2, pp. 53–59, 2021.

[8] R. Parizotto, B. L. Coelho, D. C. Nunes, I. Haque, and A. Schaeffer-Filho, "Offloading machine learning to programmable data planes: A systematic survey," *ACM Computing Surveys (CSUR)*, vol. 56, no. 1, pp. 1–34, 2023.

[9] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "pHeavy: Predicting heavy flows in the programmable data plane," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4353–4364, 2021.

[10] B. M. Xavier, R. S. Guimarães, G. Comarela, and M. Martinello, "Programmable switches for in-networking classification," in *IEEE Conference on Computer Communications (IEEE INFOCOM)*, 2021, pp. 1–10.

[11] G. Siracusano and R. Bifulco, "In-network neural networks," *arXiv preprint arXiv: 1801.05731*, 2018.

[12] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? Toward in-network classification," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, 2019, p. 25–33.

[13] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "IIsy: Practical in-network classification," *arXiv preprint arXiv:2205.08243*, 2022.

[14] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 6999–7019, 2022.

[15] K. Zhang, N. Samaan, and A. Karmouch, "A machine learning-based toolbox for p4 programmable data-planes," *IEEE Transactions on Network and Service Management*, 2024.

[16] K. Razavi, S. D. Fard, G. Karlos, V. Nigade, M. Mühlhäuser, and L. Wang, "NetNN: Neural intrusion detection system in programmable networks," in *29th IEEE Symposium on Computers and Communications (ISCC)*, 2024.

[17] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the ai accelerator?" in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, 2018, pp. 20–25.

[18] "Intel Tofino switch ASIC," 2024, https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html, accessed on Jul. 30, 2024.

[19] 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 *IEEE 29th International Conference on Network Protocols (ICNP)*, 2021, pp. 1–11.

[20] J. Yan, H. Xu, Z. Liu, Q. Li, K. Xu, M. Xu, and J. Wu, "Brain-on-Switch: Towards advanced intelligent network data plane via NN-Driven traffic analysis at line-speed," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024, pp. 419–440.

[21] X. Zhang, L. Cui, F. P. Tso, W. Li, and W. Jia, "IN3: A framework for in-network computation of neural networks in the programmable data plane," *IEEE Communications Magazine*, vol. 62, no. 4, pp. 96–102, 2024.

[22] M. C. Luizelli, R. Canofre, A. F. Lorenzon, F. D. Rossi, W. Cordeiro, and O. M. Caicedo, "In-network neural networks: Challenges and opportunities for innovation," *IEEE Network*, vol. 35, no. 6, pp. 68–74, 2021.

[23] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, A. Al-Nemrat, and S. Venkatraman, "Deep learning approach for intelligent intrusion detection system," *IEEE Access*, vol. 7, pp. 41 525–41 550, 2019.

[24] W. Wang, M. Zhu, J. Wang, X. Zeng, and Z. Yang, "End-to-end encrypted traffic classification with one-dimensional convolution neural networks," in *IEEE international conference on intelligence and security informatics (ISI)*, 2017, pp. 43–48.

[25] X. Zhang, L. Cui, F. P. Tso, Z. Li, and W. Jia, "Dapper: Deploying service function chains in the programmable data plane via deep reinforcement learning," *IEEE Transactions on Services Computing*, vol. 16, no. 4, pp. 2532–2544, 2023.

[26] S. Gandhare and B. Karthikeyan, "Survey on FPGA architecture and recent applications," in *International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*, 2019, pp. 1–4.

[27] G. Gadre, S. Badhe, and K. Kulkarni, "Network processor - A simplified approach for transport layer offloading on nic," in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2016, pp. 2542–2548.

[28] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, p. 107281, 2020.

[29] "p4lang/behavioral-model," https://github.com/p4lang/behavioral-model, accessed on Jul. 30, 2024.

[30] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," *arXiv preprint arXiv:2106.08295*, 2021.

[31] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, 2021.

[32] P. Yin, J. Lyu, S. Zhang, S. J. Osher, Y. Qi, and J. Xin, "Understanding straight-through estimator in training activation quantized neural nets," in *International Conference on Learning Representations (ICLR)*, 2019.

[33] "mininet/mininet," https://github.com/mininet/mininet, accessed on Jul. 30, 2024.

[34] E. Biglar Beigi, H. Hadian Jazi, N. Stakhanova, and A. A. Ghorbani, "Towards effective feature selection in machine learning-based botnet detection approaches," in *IEEE Conference on Communications and Network Security*, 2014, pp. 247–255.

[35] I. Sharafaldin, A. H. Lashkari, A. A. Ghorbani *et al.*, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," in *International Conference on Information Systems Security and Privacy (ICISSp)*, vol. 1, 2018, pp. 108–116.