FPGA-Based Acceleration of Time Series Similarity Prediction: From Cloud to Edge

AMIN KALANTAR, University of California, Riverside, USA
ZACHARY ZIMMERMAN*, Google Inc., USA
PHILIP BRISK, University of California, Riverside, USA

With the proliferation of low-cost sensors and the Internet of Things, the rate of producing data far exceeds the compute and storage capabilities of today's infrastructure. Much of this data takes the form of time series, and in response, there has been increasing interest in the creation of time series archives in the last decade, along with the development and deployment of novel analysis methods to process the data. The general strategy has been to apply a plurality of similarity search mechanisms to various subsets and subsequences of time series data in order to identify repeated patterns and anomalies; however, the computational demands of these approaches renders them incompatible with today's power-constrained embedded CPUs.

To address this challenge, we present FA-LAMP, an FPGA-accelerated implementation of the Learned Approximate Matrix Profile (LAMP) algorithm, which predicts the correlation between streaming data sampled in real-time and a representative time series dataset used for training. FA-LAMP lends itself as a real-time solution for time series analysis problems such as classification. We present the implementation of FA-LAMP on both edge- and cloud-based prototypes. On the edge devices, FA-LAMP integrates accelerated computation as close as possible to IoT sensors, thereby eliminating the need to transmit and store data in the cloud for posterior analysis. On the cloud-based accelerators, FA-LAMP can execute multiple LAMP models on the same board, allowing simultaneous processing of incoming data from multiple data sources across a network.

LAMP employs a Convolutional Neural Network (CNN) for prediction. This work investigates the challenges and limitations of deploying CNNs on FPGAs using the Xilinx Deep Learning Processor Unit (DPU) and the Vitis AI development environment. We expose several technical limitations of the DPU, while providing a mechanism to overcome them by attaching custom IP block accelerators to the architecture. We evaluate FA-LAMP using a low-cost Xilinx Ultra96-V2 FPGA as well as a cloud-based Xilinx Alveo U280 accelerator card and measure their performance against a prototypical LAMP deployment running on a Raspberry Pi 3, an Edge TPU, a GPU, a desktop CPU, and a server-class CPU. In the edge scenario, the Ultra96-V2 FPGA improved performance and energy consumption compared to the Raspberry Pi; in the cloud scenario, the server CPU and GPU outperformed the Alveo U280 accelerator card, while the desktop CPU achieved comparable performance; however, the Alveo card offered an order of magnitude lower energy consumption compared to the other four platforms. Our implementation is publicly available at https://github.com/aminiok1/lamp-alveo.

1 INTRODUCTION

The proliferation of IoT sensors and the volume of data that they generate creates unique challenges in edge computing [27]. One motivating application, among many, is real-time seismic event prediction, which can inform hazard response strategies and enhance early warning systems [3, 25, 35]. In this case, the relevant question is whether or not the most recent seismic measurements strongly correlate to the relatively short window of time leading up to a previously observed seismic event. Such a system could benefit from increasing the throughput of the near-sensor raw data processing, and acceleration using an FPGA represents one potential avenue to do so.

This article describes an FPGA-based accelerator for a streaming time series prediction scheme called the *Learned Approximate Matrix Profile (LAMP)* [54]. Given the most recent window of data points, LAMP uses a *Convolutional Neural Network (CNN)* to predict whether or not a similarly correlated pattern occurred in the time series used to train the model. Exact methods to compute these correlations are impractical due to the requirement that the streaming time series be archived, and the fact that computing the correlations entails execution of an $O(n^2)$ algorithm on a time series of ever-increasing length [53]. It is certainly more practical to perform inference on a moderately sized CNN; nonetheless, the overhead of CNN inference remains a computational bottleneck that

*Work done while the author was at University of California, Riverside.

limits the achievable sampling rate. Embedded CPU-based solutions are state-of-the-art, but higher performance and lower energy consumption could be achieved through FPGA acceleration.

We call our approach *FPGA-Accelerated LAMP*, or *FA-LAMP*, for short. We implemented our design on both edge- and cloud-based accelerators. We compiled the LAMP model to run on a Xilinx *Deep Learning Processing Unit (DPU)* using the Vitis AI development environment and executed it on a Xilinx Zynq UltraScale+ MPSoC edge device as well as Xilinx Alveo U280 cloud-based accelerator card. Several layers of the CNN were not compatible with the DPU; to complete the system, we implemented these layers as custom hardwrae IP blocks. One challenge involved the output layer, which computes a sigmoid activation function; we considered two approximations and evaluated them in terms of accuracy, performance (latency and throughput), resource utilization, and energy consumption on three time series datasets from the domains of seismology, entomology, and poultry farming. Our highest-performing FA-LAMP system configuration on the Zynq device achieved throughput of 453.5 GOPS with an inference rate 10.7× faster and an 15.8× improvement in energy consumption compared to running LAMP on a Raspberry Pi. Our highest-performing design on the Alveo U280 accelerator card achieved throughput of 5.53 TOPS and demonstrates a 12.3% higher inference rate in comparison to a high-end CPU, while consuming one order of magnitude less energy. Using a dataset obtained from the entomology domain, we show how FA-LAMP can be combined with a post-processing classifier to better understand insect feeding behavior. We also demonstrate how DPU on the Alveo U280 accelerator can be connected to an Ethernet module to process the incoming network data while bypassing the host CPU; this capability allows FA-LAMP to process streaming data coming from external sources across the network.

This article is an extension of our prior work [17], which presented results for the edge-based FA-LAMP implementation on the Xilinx Zynq UltraScale+ MPSoC. This article makes the following contributions as extensions: (1) We extend the evaluation of edge-class devices to include a comparison with a Google Edge TPU. (2) We deploy and evaluate FA-LAMP on an Xilinx Alveo U280 accelerator card, representative of a cloud-based deployment, and we compare FA-LAMP's performance and energy consumption to desktop and server-class CPUs as well as a GPU. (3) We integrate the DPU with a 100 G Ethernet module on the Alveo card and describe steps taken to optimize throughput accounting for both computational and network performance factors. As a methodological difference, we employ quantization-aware training for our edge scenario, whereas our prior work [17] trained the FA-LAMP CNN in a quantization-oblivious manner.

2 RELATED WORK

We classify previous FPGA-based Deep Neural Network (DNN) studies along three axes: (a) techniques to optimize accelerator design from the perspective of computing engine or memory system; (b) user-accessible frameworks that deploy DNNs on FPGAs; and (c) overlays for DNN acceleration. With respect these axes, our work: (1) leverages the Xilinx DPU in conjunction with a custom HLS kernel to enable efficient whole-network acceleration on-chip; (2) evaluates the impact of different DPU configurations on throughput, latency and resource utilization on both edge- and cloud-scale FPGAs; (3) analyzes inherent tradeoffs between different approximate implementations of the sigmoid activation function; and (4) co-optimizes computational and network performance by integrating the Xilinx DPU with a 100 Gigabit Ethernet module.

2.1 Optimizing Accelerator Design

Zhang et al. proposed a novel CNN accelerator architecture that performs loop tiling and transformation to explore the design space and balance computation and memory bandwidth [50]. Another recent accelerator architecture [44] implements a large-scale matrix multiplication algorithm that statically allocates constant weights to physical multipliers, allowing the design to operate at a near-peak FPGA clock rate. A similar, yet effective, strategy for FPGA-based edge acceleration is to pack parameter memories into groups that optimize

BRAM usage, enabling the accelerator to be synthesized onto a smaller FPGA while maintaining throughput compared to a larger device [29].

Colangelo et al. extended Intel's FPGA Deep Learning Acceleration (DLA) Suite [4] to accelerate networks with 8-bit and sub 8-bit activations and weights [8]. Similar techniques achieve high throughput in FPGA-based CNN inference by either quantizing the model's weights or training the model with lower bit precision [31, 33, 47].

We take inspiration from these studies in implementing our handcrafted kernels. We employ loop tiling [50], data reuse [7, 14, 15], and quantization [8] to improve their efficiency.

2.2 Automated Frameworks for DNN Compilation

A number of domain-specific DNN compilers translate a high-level description of a model into synthesizable RTL coupled with an execution schedule. They facilitate DNN deployment on FPGAs but limit opportunities for further optimization, as the generated HLS/RTL code is hard to interpret.

HeteroCL [20] is a Python-based domain-specific language (DSL) extended from TVM [6] that maps high-level specifications of designs to hardware implementations, targeting systolic arrays and stencil architectures. It has been reported that deeply pipelined kernels designed in this framework result in routing congestion in large FPGAs [18]. DNNWEAVER [36] generates target-specific Verilog code for FPGA-based DNN accelerators using hand-optimized design templates; however, the framework can only handle conventional CNNs and does not support quantization. Other automatic DNN generation frameworks include: HLS4ML [10], which targets low-power applications; fpgaConvNet [40] which achieved the best throughput per DSP unit in a recent survey [41]; VTA [6], which uses a TVM-based compiler stack; and FINN [39] which is developed and maintained by Xilinx.

2.3 Xilinx DPU

Recently, Xilinx introduced the Deep Learning Processor Unit (DPU), a programmable engine optimized for CNNs [45]. The DPU supports a variety of deep learning models, including, but not limited to ResNet [51], VGG [37], YOLO [34]. Programmable parameters allow the FPGA designer to control the degree of parallelism and resource utilization of the DPU IP, as we have done in this study. Operations not supported by the DPU can be offloaded to a CPU or to custom IP kernels.

Project Brainwave [11] translates a pre-trained DNN model specified in a graph-baed intermediate representation and partitions it for execution on multiple FPGAs in a datacenter. The tool compiles the FPGA sub-graph to Neural Processing Unit (NPU) instruction set architecture (ISA) binary. The NPU ISA supports matrix-vector and vector-vector operations. Intel DLA [1] applies the Winograd transformation [21] to optimize the performance and bandwidth of convolutional and fully connected layers. Lastly, Light-OPU [49] uses a single uniform computation engine to accelerate light-weight convolutional neural networks.

One key challenge that we faced was that the Xilinx DPU could not execute the three final stages of our FA-LAMP CNN. This required us to design custom kernels to accelerate those functions. It remains an open question as to whether the cost of extending the DPU architecture and ISA to support these functions would be justifiable.

3 FA-LAMP SYSTEM OVERVIEW

3.1 Background: Time Series and the Matrix Profile

A *Time series* $T = \langle t_1, t_2, \ldots, t_n \rangle$ is an ordered sequence of n scalar data points. A *subsequence* of length m starting at position i is denoted $T_{i,m}$ (or just T_i if m is known from context, an assumption that we make here).

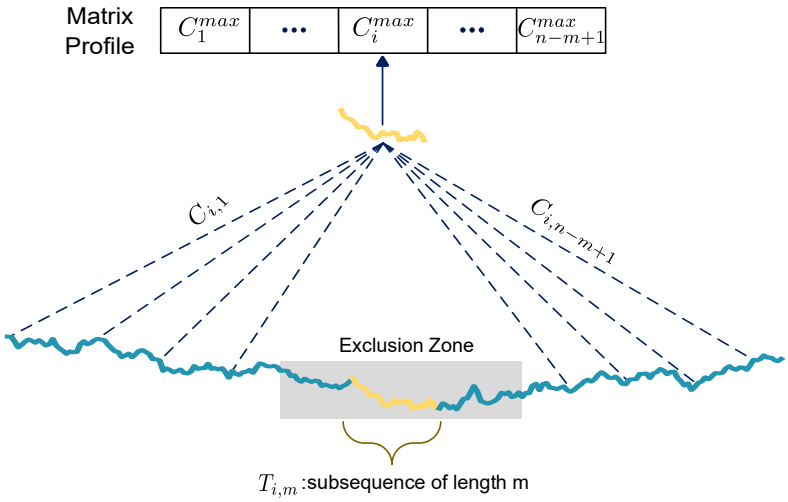

Fig. 1. Matrix Profile (MP) computation for subsequences of length m: $c_{i,j}$ denotes the Pearson Correlation between the i^{th} and j^{th} subsequences, $T_{i,m}$ and $T_{j,m}$ for all j, excluding an exclusion zone surrounding $T_{i,m}$. The maximum Pearson Correlation value c_i^{max} is stored as the i^{th} entry in the MP.

The *Pearson correlation*[1] between subsequences T_i and T_j, which measures their similarity, is denoted $c_{i,j}$ ($c_{i,j}$ values closer to 1 indicate strong correlation; values closer to 0 indicate no relationship; values closer to -1 indicate negative correlation). Once we obtain all of the $c_{i,j}$ values, we can extract the nearest neighbor of T_i in T. Subsequence T_j is defined to be the *nearest neighbor* of subsequence T_i if $c_{i,j} \geq c_{i,k}, \forall k \neq j$. The *Matrix Profile (MP)* [53] (Figure 1) is a vector that contains the correlations of the nearest neighbors of each subsequence in T: $P(T) = \langle c_i^{max} | \ 1 \leq i \leq n-m+1 \rangle$, where c_i^{max} is the maximum correlation between T_i and any other subsequence $T_j \in T$, excluding subsequences in an exclusion zone surrounding T_i. Once we compute the MP (correlation to the nearest neighbor of every subsequence), determining *time series motifs* (repeated patterns) and *time series discords* (anomalies) becomes trivial [52].

3.2 Background: LAMP

The MP is itself a time series; while the MP can be computed efficiently with GPUs [53], doing so is not amenable to streaming data. While the time complexity to compute the MP is $O(n^2 logn)$ [48], in the streaming context, the time complexity of updating the MP for each newly sampled data point is $O(nlogn)$ as $n \rightarrow \infty$. In other words, not only is it necessary to store the entire time series as it grows over time, but each new data point requires a super-linear pass over all of the data points that have been stored. To sidestep this issue, the Learned Approximate Matrix Profile (LAMP) [54] predicts the maximum correlation between the mostly recently-sampled length-m window of streaming data points to a representative time series used to train the model. This enables real-time analytics, such as anomaly detection and classification, using predicted MP values. The objective of this article is to accelerate LAMP inference using an FPGA.

Figure 2 illustrates the LAMP inference process. Each input consists of **J** z-normalized (zero mean and unit variance) subsequences of length **M**, extracted with stride **S**. This scheme defines an extraction window in the

[1]Historically, Euclidean distance between z-normalized subsequences is used as the distance function for time series data mining tasks [48]; the use of Pearson correlation, which limits the range of correlation values to [-1, +1], is more recent [53, 54] and is arguably more intuitive as the maximum Euclidean distance value is unbounded.

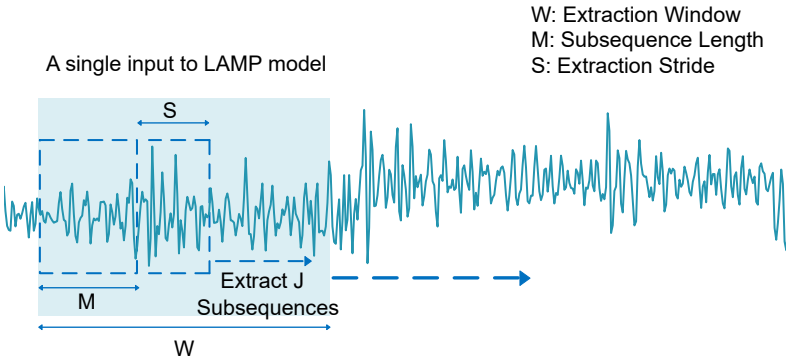

Fig. 2. Illustration of the parameters used for LAMP inference on a streaming time series.

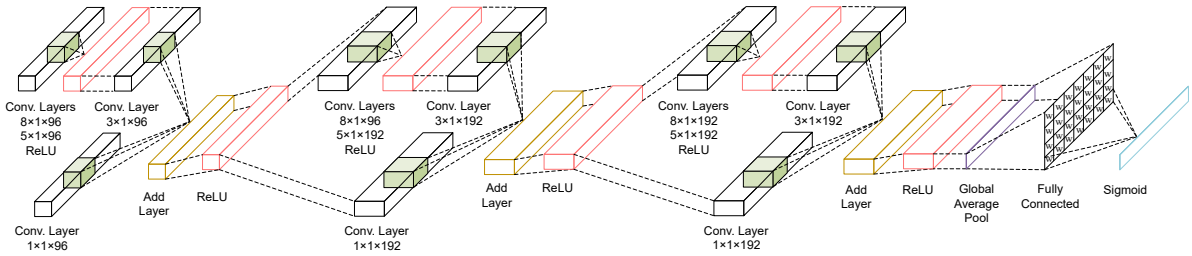

Fig. 3. The CNN used for LAMP inference. Batch normalization layers are omitted to simplify the presentation.

data, $\mathbf{W}$, where $||\mathbf{W}|| = \mathbf{J} \cdot \mathbf{S} + \mathbf{M} - \mathbf{1}$. We slide $\mathbf{W}$ across the time series and extract a new input for the model for each position of $\mathbf{W}$. This procedure generates vectors of length $\mathbf{M}$ with $\mathbf{J}$ channels as inputs to LAMP's neural network (a CNN), shown in Figure 3. For each input, the model predicts $\mathbf{J} \cdot \mathbf{S}$ LAMP values, one for each subsequence in $\mathbf{W}$.

LAMP's CNN is a simplified version of ResNet [51] for time series classification [42, 54]. Model inputs and outputs are modified to support concurrent predictions. The first layer in the LAMP CNN is batch normalization (omitted from Figure 3 for simplicity); each convolutional layer in the model is followed by a batch normalization layer (also omitted from Figure 3), which are aggregated by Addition layers followed by ReLU activation functions. The final three layers are Global Average Pool (GAP), a fully-connected layer, and a sigmoid activation function. Figure 3 reports the kernel dimensions and number of filters used below each convolution layer.

3.3 Xilinx DPU: Objective and Technical Challenges

The Xilinx DPU is a programmable architecture that accelerates many common CNN operations, such as convolution, deconvolution, max pooling, and fully connected layers [45]. The objective of this work to accelerate LAMP neural network inference on the Xilinx Ultra96-V2 and Alveo U280 FPGA boards, leveraging the DPU to achieve a balance between performance and programmability. The on-board Xilinx Zynq UltraScale+ FPGA features two Arm CPUs, and has sufficient capacity to realize at most one DPU, with additional logic remaining to implement custom IP block accelerators; the larger capacity UltraScale+ FPGA in the Alveo U280 card can fit multiple DPU instances.

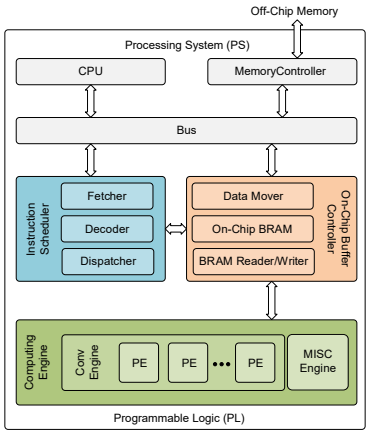

Fig. 4. Zynq DPU architecture.

We ran into several technical challenges. First, the DPU does not support the Global Average Pooling (GAP) and sigmoid layers, shown on the right-hand-side of Figure 3; these layers must be implemented in software running on one of the Arm CPU cores (UltraScale) or as custom hardware IP block accelerators (Ultrascale or Alveo). Second, implementing the fully connected layer, which sits between the GAP and sigmoid layers, would entail significant data transfer overhead between the DPU and the Arm CPU / IP block. Third, the DPU for Ultra96-V2 board uses different configurations to perform the convolutional layer (including accumulation and ReLUs); with space for just one DPU, dynamic reconfiguration during inference would be needed to support the fully connected layer; the alternative, which we adopted, is to implement the fully connected layer externally on the CPU or as an IP block. This approach worked well for both platforms.

3.4 DPU for Edge Processing

Figure 4 depicts the DPU architecture for Zynq devices. The DPU features user-configurable parameters to optimize resource utilization and to select which features are needed for a given deployment scenario. For example, our implementation does not use softmax, channel augmentation, or depthwise convolution. Seven DPU variants exist, which differ in the amount of parallelism provided by the convolution units, with IDs ranging from B512 (smallest, 512 operations per clock cycle) to B4096 (largest, 4096 operations per clock cycle); the largest variant that fits onto the Ultra96-V2 board is the B2304. The DPU compiler translates a neural network model into a sequence of DPU instructions. After start-up, the DPU fetches these instructions from off-chip memory to control the compute engine's operations. The compute engine employs deep pipelining and comprises one or more processing elements (PEs), each consisting of multipliers, adders, and accumulators. DSP blocks can be clocked at twice the frequency of general logic.

The DPU buffers input, output, and intermediate values in BRAM to reduce external memory bandwidth. It directly connects to the Processing System (PS) through the Advanced eXtensible Interface 4 (AXI4) to transfer data. The host program uses the Xilinx Deep Neural Network Development Kit (DNNDK) to control the DPU, service interrupts, and coordinate data transfers. In our design, data transfers were necessary as the final three layers of the CNN (GAP, fully connected, and sigmoid) were performed outside the DPU.

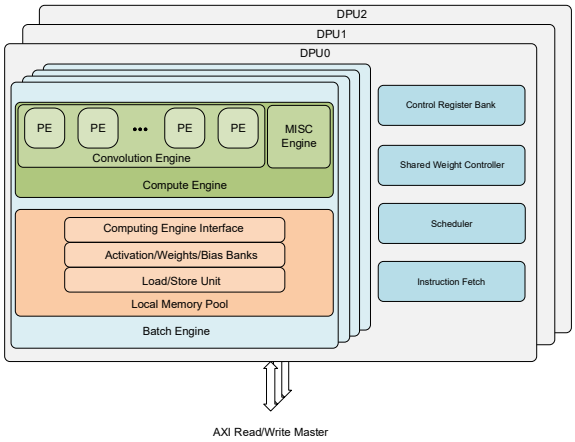

Fig. 5. High-throughput DPUCAHX8H architecture, comprising three DPU instances with multiple batch engines for parallel data processing.

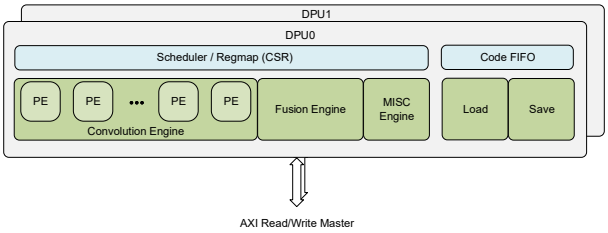

Fig. 6. Low-latency DPUCAHX8L architecture, comprising two DPU instances with one convolution engine, scheduler, and code FIFO units.

3.5 DPU for Cloud Acceleration

Two different DPU architectures are currently available that support the High Bandwidth Memory (HBM)[2] on the Alveo FPGA card, one is high-throughput (Figure 5) and the other is low-latency (Figure 6). The Alveo DPUs are named DPUCAHX8 as they are targeted towards CNN applications (C) for the Alveo platform with HBM (AH) using 8-bit quantization (X8). The two variants are named DPUCAHX8H (high-throughput) and DPUCAHX8L (low-latency) respectively. Both architectures are provided as device binary files and cannot be further configured. The high-throughput architecture is configured with three DPUCAHX8H DPUs; the low latency architecture is configured with two DPUCAHX8L DPUs. The DPU compiler for Alveo allows the user to partition the inference model (a graph) between the FPGA and the host. We use the default partitioning option which divides the model between the layers that are supported by the DPU and those that are not.

Figure 5 depicts high-throughput DPUCAHX8H DPU microarchitecture. The DPUCAHX8H consists of shared weights control logic, an instruction scheduler to fetch, decode and dispatch jobs, a control register bank that provides a control interface between the DPU and host CPU, and can be configured with four or five batch engines

[2]While FA-LAMP is optimized for streaming time series generated by external sensors, we evaluate FA-LAMP by loading the time series into the HBM and streaming it directly into the FPGA.

that allow the DPU to process multiple input data streams simultaneously. The DPU requires all of the batch engines in a kernel to execute the same neural network; the weight buffer, the instruction scheduler, and the control register bank can serve all of the batch engines. The batch engine contains a compute engine which comprises two sub-engines: a convolution engine and a MISC engine, along with a local memory pool that stores trained model parameters (weights). The convolution engine executes regular convolution/deconvolution operations, and the MISC engine handles other operations such as ReLU, pooling, etc. Each batch engine communicates with the device memory through AXI read/write master interfaces.

Figure 6 depicts the low-latency DPUCAHX8L microarchitecture. This microarchitecture comprises convolution and MISC engines and control bank registers, but omits the batch engine and local memory pool. The low-latency architecture is compatible with compiler optimizations such as kernel fusion, which can achieve higher throughput via pipeline-level parallelism.

3.6 HLS Kernel

This subsection summarizes the steps taken to design an IP accelerator that performs the GAP, fully connected, and sigmoid layers using High-Level Synthesis (HLS).

(1) Global Average Pool (GAP): The output of the final convolutional layer in Figure 3 is an array of feature maps $D \in \mathbb{R}^{M \times N}$ corresponding to each of the N channels. The GAP generates an N-dimensional vector $q \in \mathbb{R}^N$ consisting of the average value of each feature map. In other words,

$$q_j \leftarrow \frac{1}{M} \sum_{i=1}^{M} D_{i,j}, \quad 1 \leq j \leq N. \tag{1}$$

The vector q is then passed to the fully connected layer.

(2) Fully Connected Layer: The input to the fully connected layer is a feature vector $q \in \mathbb{R}^N$. The fully connected layer left-multiplies a weight matrix $W \in \mathbb{R}^{N \times M}$ by q and adds a bias vector $b \in \mathbb{R}^M$, to the result, yielding a new feature vector $z \in \mathbb{R}^M$.

$$z \leftarrow qW + b. \tag{2}$$

Initially, we set $z \leftarrow b$ in BRAM. We then process each feature $q_i, 1 \leq i \leq N$ and multiply it by the element in the i^{th} row of the weight matrix, $W_{i,j=1\ldots M}$, adding each scalar product term to z_j, i.e., $z_j \leftarrow q_i W_{i,j}$, once again, storing the accumulated sum in BRAM (We store the weights, biases, and accumulated sum in UltraRAM in our Alveo implementation). This scheme allows the execution of the fully connected layer to start as soon as the first element q_1 produced by the GAP layer arrives; likewise, each feature q_i can be discarded as soon as all of its intermediate products are computed.

We use row-wise vector-matrix multiplication and tiling [32] to optimize performance. We tile the weight matrix W into small $n_c \times n_r$ blocks as shown in Figure 7; each vector element is multiplied by n_r matrix elements, allowing the accelerator to perform $n_c \times n_r$ scalar multiplication operations per cycle. Parameter n_c must be chosen to make sure that the latency of GAP layer is greater than the number of cycles required to process n_c vector elements; n_r is chosen to be as large as possible to increase system parallelism, subject to resource constraints. We set $n_c = 8$ and $n_r = 4$ for the Ultra96-V2 implementation and set $n_c = 16$ and $n_r = 16$ for the Alveo card in our experiments.

Figure 8 depicts the hardware architecture for the fully connected layer. The design starts by reading n_c elements from the previous layer (GAP) and inserting them into n_r FIFOs. During each iteration, a tile of size $n_c \times n_r$ of the weights is read from the BRAM and is multiplied by the corresponding vector, which is provided by the GAP layer. The vector is reused until the final column of the weight matrix is processed; then the next n_c elements are read from the GAP layer and the process repeats. The *Multiply-Accumulate (MAC)* module executes

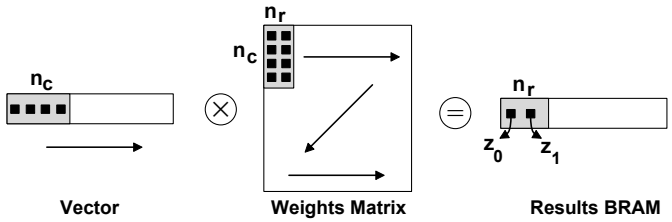

Fig. 7. Column-wise vector-matrix multiplication tiling scheme.

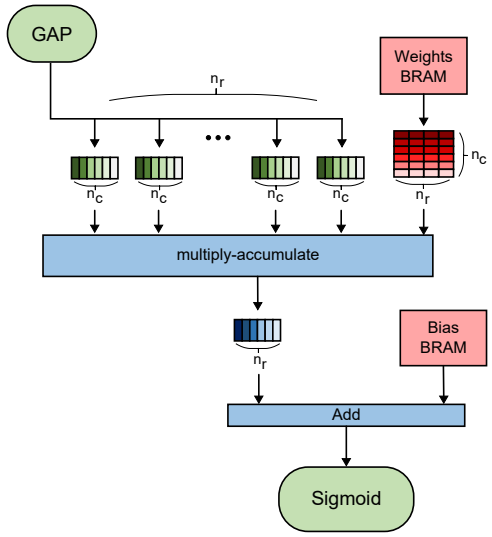

Fig. 8. Fully connected layer hardware architecture.

$n_c \times n_r$ parallel multiplications per clock cycle[3], storing the accumulated sums in a BRAM. The MAC module outputs a vector of length n_r which is added to the bias values stored in a separate BRAM; the resulting sum is then transmitted to the Sigmoid layer.

(3) Sigmoid Activation: The LAMP CNN applies the sigmoid activation function to each scalar element of the feature vector z produced by the fully connected layer. To simplify notation, we present the sigmoid function of a scalar input x which can represent any of the scalars $z_i \in z$:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

Computing the sigmoid function directly on an FPGA is impractical due to the cost of division and exponentiation. Informed by extensive studies regarding sigmoid approximations [13], we chose two variants to evaluate: ultra_fast_sigmoid, a piece-wise approximation used in the Theano library [5]; and sigm_fastexp_512, which expands the exponential function for an infinite limit [38].

There are inherent tradeoffs among these approximations in terms of accuracy, throughput/latency, area, and energy consumption; additionally, their implementation differs radically, depending on the chosen precision and

[3]A single-cycle multiplier is acceptable for our design because we use an 8-bit fixed-point data format; increasing the precision or switching to a floating-point data format may necessitate multi-cycle or pipelined multipliers.

whether they are implemented using fixed- or floating-point arithmetic[4]. A thorough survey of the tradeoffs involved is beyond the scope of this article. The final design, which we evaluate in the following section, uses 8-bit fixed-point arithmetic.

The ultra_fast_sigmoid approximation is defined as follows:

$$f(x) = \begin{cases} 0.5(\frac{\frac{1.5x}{2}}{1+\frac{x}{2}} + 1) & 0 \leq \frac{x}{2} < 1.7 \\ 0.5(1 + 0.935 + 0.045(\frac{x}{2} - 1.7)) & 1.7 \leq \frac{x}{2} < 3 \\ 0.5(1 + 0.995) & \frac{x}{2} \geq 3 \\ 0.5(-\frac{\frac{-1.5x}{2}}{1-\frac{x}{2}} + 1) & -1.7 \leq \frac{x}{2} \leq 0 \\ 0.5(1 - (0.935 + 0.045(-\frac{x}{2} - 1.7))) & -3 < \frac{x}{2} \leq -1.7 \\ 0.5(1 - 0.995) & \frac{x}{2} \leq -3 \end{cases} \tag{4}$$

Due to the relative simplicity of the operations compared to directly computing the sigmoid function, ultra_fast_sigmoid can be implemented as a low-latency kernel.

The sigm_fastexp_512 approximation expands the exponential function in terms of an infinite limit ($n \rightarrow \infty$), using a value of $n = 512$ to render the approximation computable [38]:

$$\exp(x) = \prod_{k=1}^{lg(n)} (1 + \frac{x}{k})^k, \quad n = 512 \tag{5}$$

$$\text{sigm}(x) = \frac{1}{1 + \exp(-x)} \tag{6}$$

We implemented our sigmoid layer in HLS using a loop that takes x as an input from the fully connected layer and approximates the sigmoid using either Eq. (4) or Eq. (6). In both scenarios, we pipelined the loop with an Initiation Interval (II) of 1; the latency of the loop for sigm_fastexp_512 is higher due to the complexity of the operations.

Figure 9 shows the sigm_fastexp_512 and ultra_fast_sigmoid approximations, along with their associated errors, defined as the squared difference between them and an exactly-computed sigmoid function. Neither is uniformly more accurate than the other for all reported values of x, but ultra_fast_sigmoid has noticeably higher error closer to zero. This error is tolerable for classification problems [9], where results are normally determined through comparison, not exact values. The error has a greater impact for regression systems that subsequently process the neural network's calculated output.

(4) HLS Optimizations: We optimized our design using directives provided by Vivado HLS and through manual redesign of the fully connected layer. As shown in Figure 10, we achieved a 20× speedup over our baseline implementation, while increasing resource usage by 1.5×:

- Baseline : our starting point design using a 32-bit floating-point data format.
- Unroll : unrolls the inner loops of the GAP and fully connected layers.
- Pipeline : pipelines the outer computation loops and I/O interface loops to infer burst reads/writes; the three layers execute as a pipeline to maximally overlap computation.
- Fixed-Point: is the design implemented in an 8-bit fixed-point (ap_fixed<8, 3>) data format which reduces the resource utilization by 3× [12].

[4]Alternative implementations, such as logarithmic number systems or Posits, are also possible, but are neither discussed nor evaluated here.

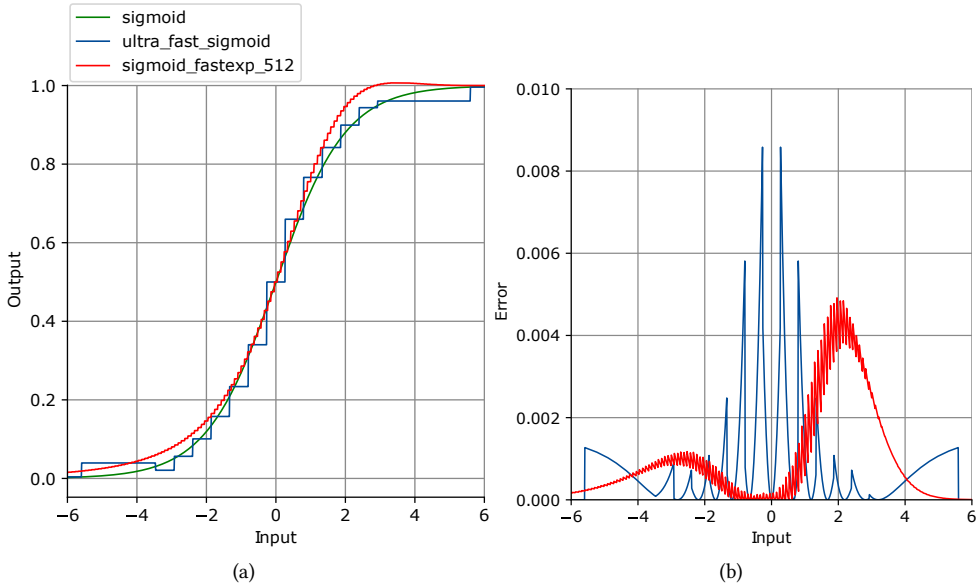

Fig. 9. (a) Approximation functions for sigmoid and (b) their error. Both charts were computed using an 8-bit fixed-point data type.

- Loop-Tiling-n_r tiling the fully connected layer (see Figure 7), while retaining the 8-bit data format.

The average resource axis in Figure 10 is the average percentage of BRAMs, LUTs, DSP blocks, and registers used for each design. Most of the speedup arises from pipelining and unrolling loops, which increases the number of DSP blocks and registers used in a design.

Figure 11 shows the overall design on the Ultra96-V2 board. The HLS kernel implements the GAP, fully connected, and sigmoid layers while the rest of the neural network runs on the DPU. The DPU and HLS kernel connect to the processing system via AXI4 ports to allow access to the DDR memory space. The Zynq UltraScale+ processing system in our platform has four High-Performance (HP) ports and two High-Performance Cache coherent (HPC) ports. The DPU I/O interfaces and HLS kernel connect to the HP ports, which provide lower latency than the HPC ports; the DPU instruction fetch port connects to an HPC port.

Figure 12 shows the Alveo U280 FPGA configured to run the high throughput DPUCAHX8H architecture. The host CPU, which pre-processes the input time series, communicates with the Alveo card via the PCIe bus. The FPGA is partitioned into static and dynamic regions. The static region is a fixed logic partition that contains the board interface logic and cannot be programmed by the user. The dynamic region contains memories, memory interfaces and user kernels compiled using the Xilinx Vitis compiler. The resources in the dynamic region are further divided into three Super Logic Regions (SLR0-2). The DPU architecture consists of three DPUCAHX8H instances, each of which is mapped to a separate logic region. The DPUs in SLR1 and SLR2 are configured with five batch engines for maximum parallelism; the DPU in SLR0 contains four batch engines, in order to leave space for our custom kernel, which implements the GAP, fully connected, and sigmoid layers, and the AXI switch network and HBM controller to connect the device memory. The switch network connects to all three DPU instances, providing 7, 7, and 6 HBM AXI ports respectively, and provides two additional ports to the custom kernel in SLR0.

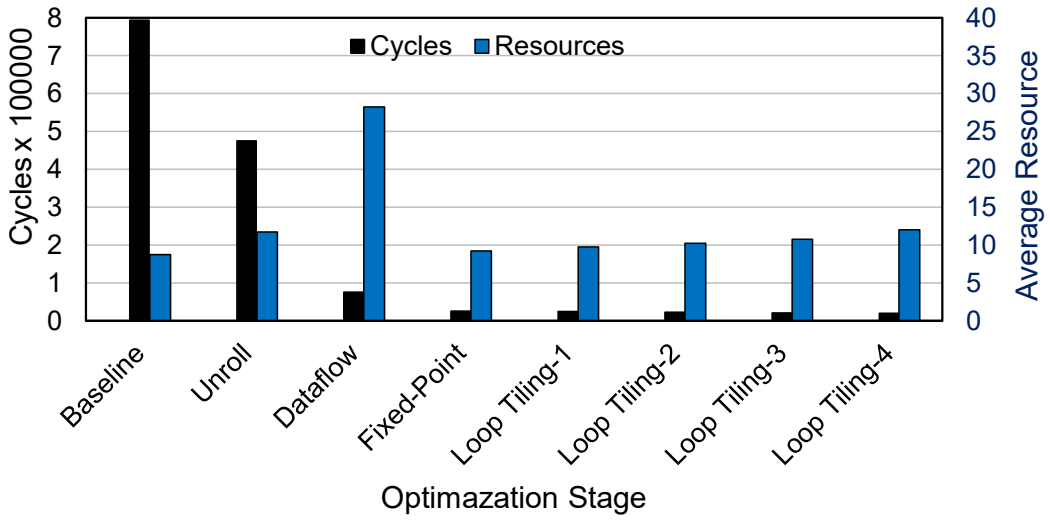

Fig. 10. Improvements in custom kernel latency and resource utilization due to HLS optimizations.

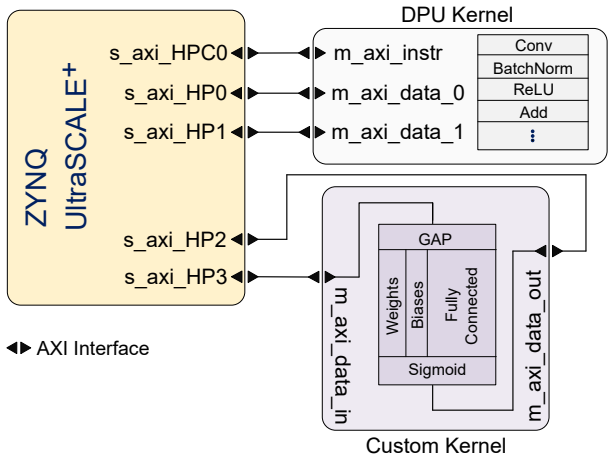

Fig. 11. The FA-LAMP edge implementation comprises a Zynq UltraScale+ processing system, DPU IP, and custom HLS kernel; the HLS kernel implements the GAP, fully connected, and sigmoid layers.

4 EXPERIMENTAL SETUP

Figure 13 depicts the LAMP model training process and DPU deployment workflow; a detailed explanation follows.

4.1 Model Training

FA-LAMP deployment on an FPGA begins by training the model. We set the number of subsequences **J** to 32 [54], the length of window **M** to 100, and the stride **S** to 8. We used the Adam [19] optimizer to train the model using stochastic gradient descent with a learning rate of 1e-3 and a batch size of 128. The training objective

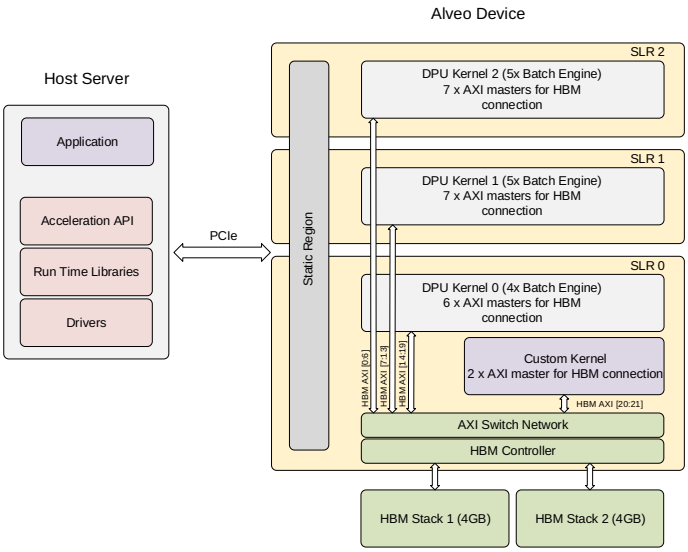

Fig. 12. Alveo architecture programmed with the high throughput DPU.

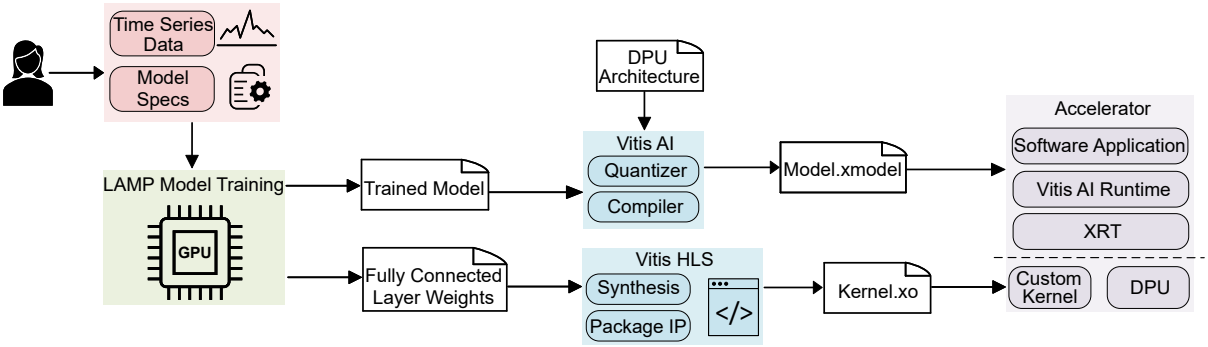

Fig. 13. Overview of deploying a LAMP model on a DPU.

is to minimize the mean squared error loss between the predicted and exact MP values for the training data set. We removed the first batch normalization layer from the LAMP CNN [54]: the Vitis compiler merges each convolutional layer followed by a batch normalization layer followed by a ReLU layer; a CNN with a batch normalization layer preceding the first convolutional layer caused an error, because the Vitis compiler interpreted the CNN as consisting of a sequence of batch normalization layers followed by convolutional layers. Removing the initial convolutional layer was the most straightforward way to rectify the problem.

We rearranged the layers in the original LAMP CNN design [54] so that each convolutional layer is followed by a batch normalization layer followed by a ReLU layer; this enables batch normalization to merge with the convolution layer in the DPU.

We trained a LAMP model for each dataset offline using the TensorFlow quantization-aware training API on an Nvidia Tesla P100 GPU. This API improves the accuracy of the model prior to quantization to INT8, which

Table 1. Comparison between Ultra96-V2 and Alveo U280 FPGA specifications.

	Alveo U280	Ultra96V2
INT8 Peak Throughput	24.5 TOPS	691 GOPS
HBM2 Capacity	8 GB	N/A
HBM2 Bandwidth	460 GB/s	N/A
DDR Capacity	32 GB	2 GB
DDR Bandwidth	38 GB/s	25 GB/s
Look-Up Tables	1,304k	70,560
DSP Slices	9,024	360
Block RAMs	2,016	432
UltraRAMs	960	N/A
Price	$7,500	$250

is performed post-hoc by downstream tools (Vitis AI Quantizer in our case). The model is then calibrated and partitioned in two using Vitis AI: (i) the layers to be executed on a custom kernel (GAP, fully connected, and sigmoid), and (ii) the rest of the model, which runs on the DPU. The custom kernel code includes a header that contains the weights and activations of the fully connected layer for high-level synthesis; the GAP and sigmoid layers do not feature any trained parameters. The second sub-graph of the model is stored in the h5 format file.

4.2 Model Inference

4.2.1 DPU Deployment. We use Vitis AI 1.3 to quantize and compile the trained LAMP model. AI Quantizer converts all of the model weights and activations into a fixed-point INT8 format. The Xilinx Intermediate Representation (XIR)-based Compiler then maps the model to the DPU instruction set and data flow. We specified the custom kernel (fully connected, GAP, and sigmoid layers) in Vitis HLS using C++ and the ap_fixed<8, 3> data type. We synthesized the custom kernel using Vivado HLS 2019.2 and integrated the resulting IP block with the DPU using Vitis 2019.2.

We evaluated the LAMP CNNs on a Xilinx Ultra96-V2 development board and Alveo U280 card. Table 1 compares the resources provided by the two platforms. The Alveo card is 30× more expensive than the Ultra96-V2 board, while providing considerably more logic, memory, and DSP resources and higher off-chip memory capacity and bandwidth.

The Ultra96-V2 integrates two Arm CPUs (an 1.5 GHz Arm Cortex A-53 and a 600 MHz Cortex-R5) with a Xilinx Zynq UltraScale+ MPSoC featuring 70,560 LUTs, 360 DSP slices and 7.5 MB of BRAM. We used a 16 GB SD card to store an embedded Linux image created with PetaLinux 2019.2 along with the input time series datasets for the design that we will use for inference. We wrote a host program in C++ that uses the DNNDK API (VART for Alveo) to communicate with the DPU IP core.

We inserted the Alveo FPGA card into a Dell PowerEdge R730 Rack Server which contains a 6-core 2.60 GHz Intel Xeon E5-2640 processor. The host connects to the FPGA through a PCI Express 4.0 interface. The server features 32 GB of DDR and 8 GB of HBM with 460 GB/s of bandwidth.

In the standard DPU flow, unsupported layers can be offloaded to a host CPU as an alternative to utilizing custom IP blocks. The Zynq FPGA on the Ultra96-V2 development board features two integrated Arm Cores: a Cortex-A53 and a Cortex-R5. As a baseline for comparison for the edge deployment scenario, we implemented the custom kernel layers on the Cortex-A53, which supports a higher clock frequency than the Cortex-R5. The

source code running on the Cortex-A53 employs the same 8-bit fixed-point data type as we used on the FPGA. We use the C++ built-in exp() function (from the <cmath> library) to compute the sigmoid and a for-loop to compute the global average pool layer. For the cloud deployment, we evaluated the software performance of the custom kernel on the Intel Xeon E5-2640 CPU, noting that the latency over the PCIe communication channel is significant.

4.2.2 LAMP Deployment on CPU and GPU. In order to quantify FA-LAMP's performance in the cloud scenario, we implemented LAMP inference on a server CPU, a desktop CPU, and a GPU. The GPU platform comprises two NVIDIA GeForce RTX 2080 cards inserted into a Rack Server containing 16 Intel Core i9-9900 processors operating at 3.1 GHz; the Server CPU includes the 6-core Intel Xeon E5-2640 server described earlier; and the Desktop CPU is an Intel Core i7-8750 CPU with six cores running at 2.2 GHz. All the platforms mentioned above execute CNN inference in Python 3.7 using Keras' Predict Generator class with multiprocessing enabled.

4.2.3 Raspberry Pi3 and Edge TPU. We also ported the LAMP inference engine to run on a Raspberry Pi 3 board, which provides a 100% software baseline that is representative of edge computing. We wrote a short Python script that converts the pre-trained LAMP model saved in the Keras format to TensorFlow Lite with 8-bit full integer quantization and we configured the optimizer to minimize latency. We performed inference using the trained model on the Raspberry Pi using the TensorFlow Lite Interpreter. The Raspberry Pi 3 features a Quad Core 1.2GHz Broadcom BCM2837 CPU. Ideally, we would have run the full LAMP model in software on either of the two Arm cores on the Ultra96-V2 board; however, it was not possible to do so, as Keras does not support the Ultra96-V2 board at the time of writing.

We also executed LAMP inference on a Coral USB Accelerator [22] which contains a Google Edge TPU coprocessor [23], an ASIC optimized for AI inference. We first quantized the LAMP model to an 8-bit format using TensorFlow quantization-aware training; we then exported the quantized model as a frozen graph, converted it to a TensorFlow Lite model, and used the Edge TPU compiler to convert it to the supported format for the USB accelerator. The Edge TPU allows pipelining to decompose a large model into segments spread across multiple Edge TPUs; this is particularly important for models whose data segments exceeds the Edge TPU cache capacity. Our LAMP model fits within the Edge TPU on-chip memory (8 MB), allowing us to run two models on two Coral USB accelerators concurrently. We inserted the two Coral USB accelerators into two USB 3.0 ports on a desktop PC running Ubuntu 18.04 Linux. We installed the Edge TPU runtime version 13 on Ubuntu and used the increased frequency option, which is known to increase power consumption. We loaded the model and data onto the Edge TPUs using the PyCoral API with Python 3.7.

4.2.4 Comparison to Recent CNN-to-FPGA Compilation Frameworks. In order to quantify DPU's performance, we deployed our LAMP model on several state-of-the-art FPGA edge-based and cloud-based CNN frameworks: HLS4ML [10], fpgaConvNet [40], VTA [6], and FINN [39]. All of these frameworks can target the Ultra96-V2 development board, but only fpgaConvNet and FINN can target the Alveo card.

HLS4ML is a Python package that converts a trained neural network in the ONNX format into an HLS project for synthesis onto an FPGA; layers are implemented by choosing and configuring HLS modules from a template library. We trained and quantized the LAMP model using Tensorflow, and then converted it to ONNX using tf2onnx [24]. HLS4ML performs integer scaling during quantization and can be configured on a per-layer basis. To ensure that the model was synthesizable, we limited the amount of loop unrolling. We also corrected some compilation errors that occurred because HLS4ML did not define the correct AXI Stream interface between modules. We set the precision of all weights as biases to 8-bit fixed-point and used the default resuse factor value. While HLS4ML supports the sigmoid layer using a lookup table implementation, we replaced the last three of the CNN with our own custom layers to ensure a fair comparison.

Similar to HLS4ML, fpgaConvNet converts a trained model in the ONNX format into an HLS project, propagating model quantization settings into its internal representation. samo [26], a design space exploration tool, can optimize the model implementation on the FPGA using simulated annealing; we used samo's rule-based optimizer and selected the latency performance objective.

VTA uses a template deep learning accelerator consisting of load, store, and compute (RISC processor) units. We used TVM to translate a trained LAMP model into a Relay module (TVM's front end compiler) and applied 8-bit quantization (VTA exclusively supports the int-8 format). We then applied constant folding to reduce the number operators and created an object file to load onto the FPGA. The last three layers of the LAMP model are executed in fp32 on the CPU, as VTA's front end compiler is not compatible with custom kernel IP accelerators.

For the FINN framework, we defined our LAMP model in PyTorch and quantized it using Brevitas [28], which exports the model to the FINN-ONNX format. FINN's compiler then converts the model to one or more more FPGA accelerators; the network must be redefined with Brevitas layers, which correspond to standard PyTorch layers, e.g., there is a QuantLinearlayer type. FINN's non-standard use of ONNX restricted our ability to quantize the LAMP model. To target the the Ultra96-V2 board, we quantized weights and activations into a 4-bit representation; to target the Alveo card, we quantized weights and activations into 1-bit and 4-bit representations respectively. FINN was unable to support our custom IP kernels, so we implemented them using the fp32 format on the host CPU.

4.3 Measurements

We report the throughput and the energy consumption of FA-LAMP CNN inference by direct execution of the model on the aforementioned platforms using three time series datasets, which are summarized in the next subsection. The throughput is reported as the total number of multiply-accumulation operations in the model (7.71 GOPs) executed per second. We also report the *inference rate* of each platform, which we define to be the number of Matrix Profile values predicted per second. We measure the Ultra96-V2 and Raspberry Pi power consumption using a commercially available Kuman power meter, which provides power measurements for the entire board.

We estimated the power consumption of the FPGA on the Alveo card by periodically transmitting queries through the *xbutil* tool. *xbutil* measures FPGA power consumption, but does not report the current of the HBM power rails, whichwe omit from our estimation. We estimated the power consumption of the host Intel Xeon CPU using the PyRAPL software toolkit [30], while eliminating all other application programs running under Windows; we could not eliminate any variability arising from the operating system. We report the GPU's power consumption using the NVIDIA System Management Interface (nvidia-smi). We estimate energy consumption by multiplying the power measurement by the time required to perform inference on a batch of size 128. Every batch of data predicts 256 MP values based on the configured LAMP parameters, for a total of 128×256 predictions per inference. Batch sizes larger than 128 led to degraded results on the Raspberry Pi. We report resource utilization results from Vivado's post-implementation reports.

We evaluated the efficiency of all DPU variants that we could fit onto the Ultra96-V2 UltraScale+ Zynq FPGA, which can fit no more than one DPU core. We set the DPU's BRAM and DSP usage to low and disabled the average pool and softmax instructions since the LAMP neural network does not perform these operations. For the Alveo card, we evaluated the efficiency of high-throughput and low-latency DPU kernels. The DPU IP provides two distinct clock inputs: we set the input clock for DSP blocks to 300 MHz and the input clock for general logic to 150 MHz in both evaluated platforms. We set the HBM clock on the Alveo card to 450 MHz.

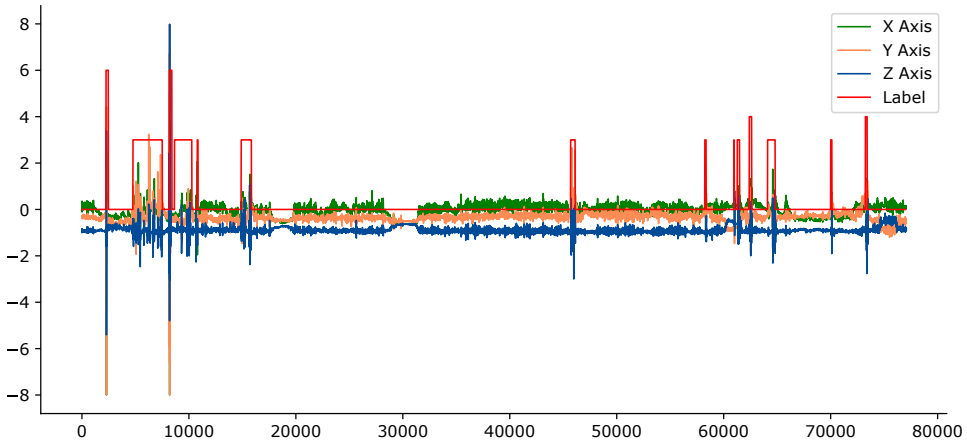

Fig. 14. A snippet of chicken accelerometer data with corresponding labels (Preening: label height = 3, dustbathing: label height = 4, and pecking: label height = 6).

4.4 Benchmarks

We trained neural networks for three time series datasets and measured the error of the model's predictions; this methodology is similar in principle to prior work on LAMP [54].

(1) Seismology Domain: The Earthquake dataset is obtained from a seismic station [53]. Real-time event prediction impacts seismic hazard assessment, response, and early warning systems [3, 25, 35]. We split the time series into 120 million and 30 million data points for training and inference.

(2) Entomology Domain: The Insect EPG dataset is obtained from an Electrical Penetration Graph (EPG) that records insect behavior [53]. This time series is the record of an insect feeding on a plant and observed behaviors were classified by an entomologist as Xylem Ingestion, Phloem Ingestion, or Phloem Salivation. Understanding feeding behavior of insects can help farmers identify vector-bearing pests that may decimate crops. We split the time series into 2.55 million and 5 million data points for training and inference.

(3) Poultry Farming Domain: The Chicken Accelerometer dataset was collected by placing a tracking sensor on the back of a chicken [2]. The sensor outputs acceleration measurements along the x-, y-, and z-axes at a 100 Hz sampling rate. The data was labeled to classify the chicken's behavior into one of three categories: *Pecking*, *Preening*, or *Dustbathing*. This is relevant to disease detection because infected chickens exhibit a marked increase in preening and dustbathing behavior compared to uninfected chickens. Figure 14 depicts a snippet of the dataset corresponding to the x-, y-, and z-axes and behavioral labels. Using only the x-axis measurements, we split the time series into 6 million and 2 million data points for training and inference.

4.5 Source code and Data Availability

We have publicly released all of code, data, code, and LAMP inference models used to produce the results in this article [16].

5 RESULTS

5.1 Edge: Throughput and Resource Utilization

Table 2 summarizes the resource utilization and the measured throughput of FA-LAMP inference using various system configurations on the Ultra96-V2 FPGA board. The DPU + Arm columns report results when the custom

Table 2. Edge Prototype: Throughput (GOPS) and resource utilization comparison between different DPU architectures; (DPU + IP) uses a B2304 DPU.

	DPU + Arm (B512)	DPU + Arm (B800)	DPU + Arm (B1024)	DPU + Arm (B1152)	DPU + Arm (B1600)	DPU + Arm (B2304)	DPU + IP (ultra_fast)	DPU + IP (fastexp_512)
Logic Usage	39K (56%)	42K (59%)	46K (65%)	44K (62%)	49K (70%)	52K (75%)	57K (82%)	60K (86%)
Register Usage	50K (36%)	57K (40%)	65K (46%)	64K (45%)	77K (54%)	87K (62%)	95K (67%)	100K (71%)
DSP Usage	78 (21%)	117 (32%)	154 (42%)	164 (45%)	232 (64%)	289 (79%)	290 (80%)	326 (90%)
On-chip RAM Usage	77 (35%)	95 (44%)	109 (50%)	127 (58%)	131 (60%)	171 (79%)	174 (81%)	174 (81%)
Throughput (GOPS)	70.4	107.0	154.2	167.6	220.2	367.1	**453.5**	**428.3**
Peak Throughput (GOPS)	153	240	307	345	480	691	691	691

kernel (fully connected, GAP, and sigmoid layers) run on the Arm CPU, while the DPU + IP columns report results for the custom kernel implemented as FPGA IP blocks that connect directly to the DPU; the largest and best-performing B2304 DPU is used when reporting results for DPU + IP. Results are reported for the custom kernel implemented using two sigmoid approximations: ultra_fast_sigmoid (ultra_fast) and sigmoid_fastexp_512 (fastexp_512) to approximate the sigmoid function.

The DPU + Arm results in Table 2 show that system throughput increases as DPU size and complexity increases, from B512 to B2304. The highest overall throughput is achieved for the DPU + IP configurations, as the three custom kernel layers that the DPU cannot execute are moved from the Arm CPU to a custom accelerator. Data transfer overhead remains present in both cases between the DPU and Arm CPU / IP block: each read for an input batch of data takes around 0.12 ms and each write takes around 0.1 ms; the port throughput is around 850 MB/s.

Table 2 also reports the peak (achievable) DPU throughput for each system configuration; this does not include the throughput of the Arm CPU or IP block because the inference procedure, at present, does not lend itself to concurrent execution. The percentage of achievable throughput ranges from 43.6% to 53.1% for the DPU + Arm configurations, and jumps to 65.6% and 62.0% for the two DPU + IP configurations. Even if a hypothetical next-generation DPU could support the three custom kernel operations, the overhead of DPU reconfiguration, which we avoided in the design(s) evaluated here, would also limit the achievable throughput.

DPU resource utilization depends on the degree of parallelism in the chosen configuration; on-chip RAM buffers the weights, bias, and intermediate features. As DPU I/O channel parallelism increases, more on-chip RAM is needed to store more intermediate data and more DSP slices are needed to process that data. When the low DSP usage option is chosen, the DPU uses DSP slices exclusively for multiplication in the convolution layers and offloads accumulation to LUTs. This explains the observed increase in LUT usage as DPU throughput increases.

The custom IP kernels consume additional resources. sigmoid_fastexp_512 performs more multiplication operations and constant division operations than ultra_fast_sigmoid, noting that the latter performs mostly constant multiplications. As a consequence, ultra_fast_sigmoid achieves higher throughput and lower resource utilization compared to sigmoid_fastexp_512; however, as we will see in the next subsection, these benefits come at the cost of lower accuracy.

5.2 Edge: Comparison to a Raspberry Pi 3 and Edge TPU

Next we compare the performance and energy consumption of FA-LAMP neural network inference running on the Ultra96-V2 FPGA board to a Raspberry Pi 3 and the Edge TPU device, being representative of a purely CPU-based edge computing systems.

Table 3 reports the throughput (inference rate), energy consumption (in Joules), and performance per power (GOPs/Watt) of processing a single batch of size 128 on each platform. The runtime of FA-LAMP neural network

Table 3. Edge Prototype: Inference rate and energy consumption of LAMP neural network inference on an Edge TPU, Raspberry Pi 3 and Ultra96-V2 board.

	Edge TPU	Raspberry Pi 3	DPU + Arm	DPU + IP ultra_fast	DPU + IP fastexp_512
Inf. Rate (Hz)	824	2.6K	12.1K	15.0K	14.2K
Energy (J)	161.4	58.8	7.2	6.7	9.1
GOPs/Watt	5.8	10.4	107.9	146.1	135.8

Table 4. Cloud prototype: throughput, latency, inference rate and energy consumption: LL=Alveo low-latency, HT=Alveo high-throughput.

	GPU	Server CPU	Desktop CPU	LL + CPU	LL + IP (ultra_fast)	LL + IP (fastexp_512)	HT + CPU	HT + IP (ultra_fast)	HT + IP (fastexp_512)
Throughput (TOPS)	92.52	69.57	4.91	2.26	3.04	2.53	4.27	5.53	4.83
Latency (ms)	356	227	296	5.68	3.25	3.49	6.29	3.85	4.09
Inference Rate (KHz)	3079	2298	163	75	101	84	142	184	160
Energy (J)	118.32	72.20	38.43	9.15	6.81	8.20	4.86	3.75	4.29
GOPs/Watt	1210	740	68	44	57	46	77	98	89

inference does not depend on the size of the representative dataset used for training; thus, the inference rate and energy consumption is identical across all datasets.

Both the inference rate and energy consumption of all three Ultra96-V2 FPGAs improve by 1 order of magnitude compared to the Edge TPU and ∼6× compared to the Raspberry Pi ; according to our power measurements, the Ultra96-V2 FPGA board consumed ∼3W of power compared to ∼4W for the Raspberry Pi. We consider the nominal power consumption of 4.5W for the Edge TPU devices as reported in the datasheet. As expected, the DPU + IP options achieve a higher inference rate than the reported DPU + Arm configuration. Notably, the DPU + IP option using sigmoid_fastexp_512 consumes more energy than both the DPU + Arm and DPU + IP using ultra_fast_sigmoid; referring back to Table 2, this occurs due to the higher demand for DSP blocks (36 more than ultra_fast_sigmoid) which are clocked twice as fast as the FPGA general logic. All of the evaluated edge platforms exhibit comparable power consumption; however, performance per Watt corresponds, linearly to the inference rate with Ultra96-V2 outperforming the Edge TPU by 1 order of magnitude and the Raspberry Pi by ∼6×. The Edge TPU has the lowest performance among all the edge platforms due to its limited RAM capacity, and its inability to support batch processing; we conclude that it is not a good option for streaming applications.

5.3 Cloud Prototype: Throughput and Energy

Table 4 details the measured performance and energy consumption of FA-LAMP in different scenarios. The columns starting with LL and HT report measurements for the low-latency and high-throughput DPU on the Alveo card. Similar to Table 2, in LL (HT) + CPU columns, the custom kernel (fully connected, GAP, and sigmoid functions) are offloaded to the CPU, while in LL (HT) + IP columns the custom kernel is implemented as FPGA kernel that runs on programmable logic. The FA-LAMP program in all Alveo implementations is multi-threaded to maximize DPU utilization.

Throughput: The server CPU and GPU achieved an order of magnitude higher throughput than the other systems tested, due to their high core count and parallel processing capabilities; the desktop CPU achieves comparable performance to the high-throughput DPU configurations. The high-throughput DPU achieves higher

Table 5. FA-LAMP neural network inference accuracy; qa=quantization-aware training, edge=Ultra96, cloud=Alveo.

Time Series Dataset		FA-LAMP Inference Accuracy						
Name	Train / Test Split	32-bit float	edge: ultra_fast	edge: fastexp	qa_edge: ultra_fast	qa_edge: fastexp	qa_cloud: ultra_fast	qa_cloud: fastexp
Earthquake	120M / 30M	97.4%	91.4%	92.5%	93.8%	94.7%	94.3%	95.1%
Insect EPG	2.5M / 5M	97.2%	90.8%	93.2%	91.9%	94.4%	92.5%	94.8%
Chicken Accel.	6M / 2M	95.8%	86.9%	91.1%	89.5%	93.1%	90.2%	93.7%

throughput than the low-latency DPU. Referring back to Figures 5 and 6, the high-throughput architecture has three DPUs, each with multiple batch engines, while the low-latency architecture has two DPUs with a single compute engine and no local memory pool; the low-latency DPU's fusion engine improves latency, but not throughput.

Latency: We report the latency on each platform as the inference time for a single input. The FPGA-based platforms achieved two orders of magnitude lower-latency compared to the two CPUs and the GPU. The low-latency DPU performs inference approximately 1ms faster than the high throughput DPU, benefiting from compiler optimizations such as layer fusion, as supported by its fusion engine (Figure 6). The hardware IP kernel implemented using the ultra_fast_sigmoid approximation runs around 0.2 ms faster than the sigmoid_fastexp_512 implementation. The FPGA + CPU systems incur the latency associated transferring data between the FPGA and server CPU, and reprogramming the DPU at runtime to execute the fully connected layer on the FPGA.

Inference Rate: The inference rate is the number of predictions per second, which correlates to throughput: the GPU and the Server CPU have the highest inference rate, while the inference rate of the Desktop CPU is comparable to those of the FPGA with high-throughput DPU configurations. The high-throughput DPU connected to the custom kernel with the ultra_fast_sigmoid has the highest overall inference rate among all DPU implementations; this results from the greater arithmetic parallelism provided by the high-throughput DPU compared to the low-latency DPU.

Energy Consumption: The Energy row in Table 4 reports the energy consumption of processing a single batch of size 128 on each platform. The FPGAs are an order of magnitude more energy efficient than the GPU or CPUs. The lowest overall power consumption was achieved using the high throughput DPU and the custom IP kernel with the ultra_fast_sigmoid approximation, which requires far fewer arithmetic operators than sigm_fastexp_512. In terms of performance per Watt, the GPU outperforms all the other platforms while the high throughput DPU with sigm_fastexp_512 improves CPU's performance per Watt by 44%.

5.4 Inference Accuracy

Table 5 summarizes the accuracy of the FA-LAMP neural network models that we evaluated in the preceding section. Columns starting with the label "edge" present the results from our previous implementation [17] and columns labeled with with "qa_edge" and "qa_cloud" detail the results obtained using quantization-aware training. We include results for a 32-bit floating-point CPU-only implementation of the FA-LAMP models as a baseline to quantify the loss in accuracy due to quantization, which is 2.1–2.8 percentage points (pp) for sigmoid_fastexp_512, and 3.1–6.3 pp for ultra_fast_sigmoid. The 6.3 pp accuracy loss for the Chicken Accelerometer dataset for ultra_fast_sigmoid can be attributed to the range of values in the input numbers to the sigmoid kernel. Referring back to Table 3, we note that sigmoid layer's input values line in the range [-0.12 1.85], where ultra_fast_sigmoid has the largest error, when inference is performed on this dataset.

Table 6. Performance comparison with other FPGA-based edge and cloud DNN deployment frameworks.

	HLS4ML [10]	fpgaConvNet [40]	VTA [6]	FINN [39]	DPU	fpgaConvNet [40]	FINN [39]	DPU
FPGA Platform	Ultra96	Ultra96	Ultra96	Ultra96	Ultra96	Alveo U280	Alveo U280	Alveo U280
Precision	fix-8	fix-8	int-8	fix-4	int-8	fix-8	mix	int-8
DSPs	256	220	186	220	326	451	1865	2600
BRAMs	132	164	152	101	174	230	412	628
Throughput	156 GOPS	198 GOPS	101 GOPS	471 GOPS	453 GOPS	2.37 TOPS	6.12 TOPS	5.53 TOPS

Compared to our previous work [17], the results reported in Table 5 achieved 1.6–2.6 pp improvement in sigmoid_fastexp_512 accuracy and 1.7–3.3 pp improvement in ultra_fast_sigmoid, which are due to the use of quantization-aware training in this study. The differences in accuracy reported for the Ultra96-V2 and Alveo implementations is due to different model compilation flows for the two platforms, and potential microarchitectural differences, noting that neither fixed-point nor floating-point addition and multiplication are associative.

5.5 Comparison to Recent CNN-to-FPGA Compilation Frameworks

We deployed our LAMP model on several state-of-the-art FPGA edge-based and cloud-based CNN frameworks and compared their performance; Table 6 reports the resource utilization and throughput of each framework.

For the Ultra96-V2 board, the DPU column represents the results for the DPU integrated with our custom kernsl using ultra_fast_sigmoid; for the Alveo card we picked the high-throughput DPU with ultra_fast_sigmoid as this combination yielded the best performance in our prior experiments. FpgaConvNet achieved a throughput of 164 GOPS, outperforming both HLS4ML and VTA by 1.26× and 1.96× respectively. fpgaConvNet's higher throughput seems to be due to its streaming architecture, which outperforms single computation engine frameworks for large batch sizes. fpgaConvNet also benefits from the design space exploration performed by the samo optimizer. While FINN outperforms fpgaConvNet and the DPU by 2.37× and 1.03×, its low-precision architecture degrades accuracy by more than 30%, which we consider to be unacceptable from the application perspective.

On the Alveo card, the DPU outperforms fpgaConvNet by 2.33×; upon inspection fpgaConvNet was unable to fully utilize the resources provided by the larger FPGA (in comparison to the Ultra96-V2). FINN achieved throughput 1.10× higher than the DPU, while implementing an (almost) binary neural network, whose accuracy was around 55%, which is non-competitive for our purposes.

5.6 Case Study: Interpreting the FA-LAMP Output

The Matrix Profile can be computed using existing methods in an offline context [53], whereas LAMP is used to predict it on streaming data [54]. Regardless of how the Matrix Profile is obtained, subsequent post-processing steps are needed to extract actionable information from it.

As a representative example, we explain how FA-LAMP neural network inference can help a scientist to classify the behavior of an insect in real-time. First, we take the training data (2.5M data points, collected over 7 hours) from an insect feeding on a plant. We then create two classes [43]:

Class **A**: Xylem Ingestion/Stylet Passage

Class **B**: Non-Probing

We take a representative dataset from each class ($\mathbf{R_A}$ and $\mathbf{R_B}$) and train two distinct FA-LAMP models, which we respectively denote as $\mathbf{M_A}$ and $\mathbf{M_B}$. Let $\mathbf{S}$ be a subsequence of streaming data. If $\mathbf{M_A(S) > M_B(S)}$, we predict that behavior **A** is occurring; if $\mathbf{M_A(S) < M_B(S)}$, we predict that behavior **B** is occurring; otherwise, the prediction is inconclusive.

For evaluation data we consider the inference data (2.5M data points, collected over the next 5 hours from the same insect), whose behavior has also been labeled by an entomologist to provide ground truth. We observed

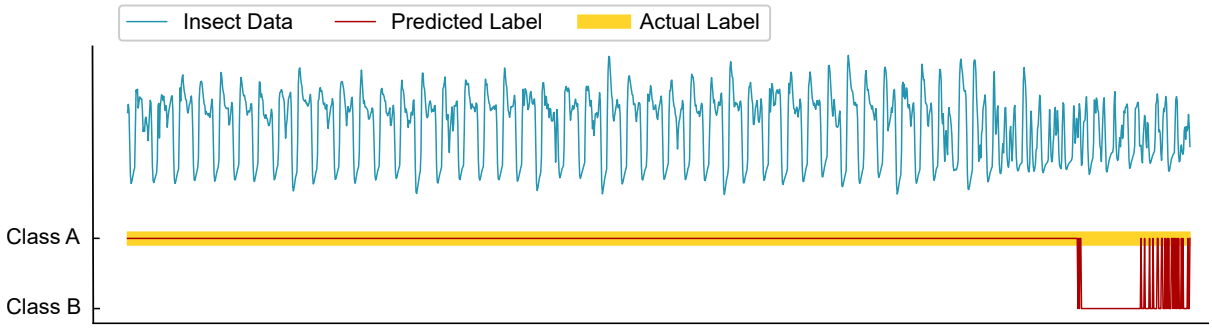

Fig. 15. A snippet of insect EPG time series dataset along with the actual and predicted behavior (Class A: label height=1; Class B: label height=0).

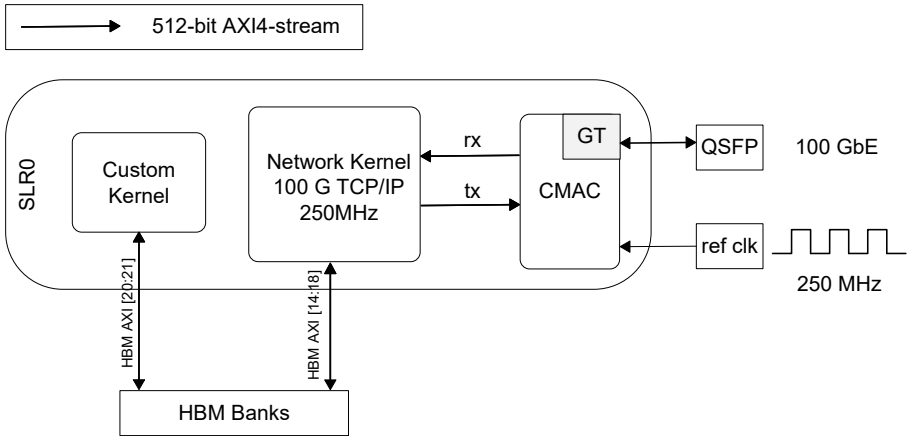

Fig. 16. The SLR0 in the Alveo card configured with the Ethernet subsystem and the custom kernel IPs.

98.2% accuracy in the results of classification using FA-LAMP. Figure 15 shows the time series and the actual and predicted labels reported by the FA-LAMP model for a snippet of test data. To simplify the representation, the time series is rearranged so that the first half represents class **A** and the second half represents class **B**. Figure 15 shows a snippet of the first half.

6 DPU INTEGRATION WITH ETHERNET

In a real world cloud-scale deployment, a plurality of Alveo cards in a server would be connected through a network switch, allowing them to receive data from external sources. For example, multiple edge devices may transmit sensor data to the server in real time over the Internet. To address the needs of such a deployment, this section describes the integration of a high-throughput DPU with a 100 G Ethernet IP allowing an Alveo-based deployment to receive and process data.

We built our design on top of the Xilinx TCP stack IP repository [46], which comprises an UltraScale+ Integrated 100 Gb/s Ethernet (CMAC) and a network layer kernel. The CMAC kernel is connected to the Alveo's GT pins exposed by the Vitis shell and it runs at the frequency of a 100 G Ethernet Subsystem clock, i.e., 322 MHz. It exposes two 512-bit AXI4-Stream interfaces (*S_AXIS* and *M_AXIS*) to the user logic, which run at the same

frequency as the kernel. Internally it has clock domain crossing logic to convert from kernel clock to the 100 G Ethernet Subsystem clock. The network kernel is a collection of HLS IP cores that provide TCP/IP network functionality, consisting of TCP, ICMP, and ARP modules clocked at 250 MHz. The network kernel exposes AXI4-Stream interfaces to enable the user kernel to open and close TCP/IP connections and to send and receive network data.

Figure 16 depicts the Ethernet subsystem and custom kernel IPs implemented in SLR0 in the Alveo card; due to resource constraints, we had to remove the DPU kernel with four batch engines in SLR0 to fit the CMAC and network layer kernels. As mentioned in Section 3.6, the DPUCAHX8H can be configured to have multiple batch engines which execute model inference in parallel. Each batch engine connects to the global HBM memory using a AXI4 memory mapped interface. The DPU also has a *s_axi_control* interface is used to start running a task on a DPU core, wait for the task to finish and clear the DPU's status. Since the network kernel provided by Xilinx has AXI4-Stream interfaces, we cannot directly connect the kernel to the DPU input ports. One solution would be to transmit the network data to the host and then to the DPU using the VART API; however, this would lead to sub-optimal performance.

To address this bottleneck, we added a memory arbiter module to the network kernel that writes the incoming network data to the memory address used by DPU batch engines. This frees up HBM memory channels 14-18 which the memory arbiter uses to divide the incoming network data into equally sized batches and writes the data to memory channels 0-6 for DPU kernel 2 and memory channels 7-13 for DPU kernel 1. The memory arbiter also provides two memory mapped AXI master interfaces that connect to the *s_axi_control* interfaces of the two DPU kernels.

After writing the input data to the corresponding addresses of the five batch engines for each DPU, the memory arbiter starts the execution of that DPU kernel by setting the *reg_ap_control* register to 1 through the *s_axi_control* interface. This allows the Alveo card to process incoming network data without CPU involvement. The memory arbiter waits for DPU's interrupt before it signals the start of a new batch.

We tested the DPU integrated with Ethernet system by directly connecting two Alveo U280 cards through their Quad Small Form-Factor Pluggable (QSFP) ports. We programmed one of the Alveo cards as a producer of data, combining the CMAC and network layer kernel with a custom user TCP kernel. The TCP kernel opens a TCP connection to provide the IP and TCP port of the destination and to transmit the data over the network. To transmit data, a Tx control handshake is required before each payload transfer. The user kernel first transmits the session ID and the payload size and, upon receiving a positive acknowledgment from the TCP module, transmits the data. The second Alveo card is programmed as a consumer, with two DPU kernels, CMAC, and the modified network kernel which includes the aforementioned memory arbiter module.

In order to achieve 100 Gbps, we pipelined the control handshake and payload transfer between the user kernel and the network kernel in the producer FPGA. Since the control handshake is required for each payload transfer and requires 10 to 30 clock cycles, a sequential control handshake-payload transfer may stall. To pipeline the process, we established 32 concurrent connections and pinned them to different threads using the OpenMP API; further increasing the number of concurrent connections yielded no further improvements in our experiments. Next, we transmitted packets whose sizes were a positive integer multiple of 64 bytes. The transmission process buffers portions of the payload in the global memory for retransmission in the event that packet loss and/or memory accesses with unaligned addresses decreases the bandwidth.

Figure 17 shows that the 100 Gigabit QSFP port saturates the available bandwidth at a sufficiently large payload size. We achieved a peak throughput of 86 Gbit/s for payloads larger than 4KiB, which is feasible because the DPU and our custom kernel can achieve an initiation interval of 1, meaning that no stall cycles occur in the design pipeline. At smaller payloads, the control handshake required for each payload transfer impedes throughput. To maximize the Ethernet throughput, optimizations on both the producer and consumer sides are required: in the producer's software code, we leveraged concurrent TCP connections to hide the control handshake latency, and

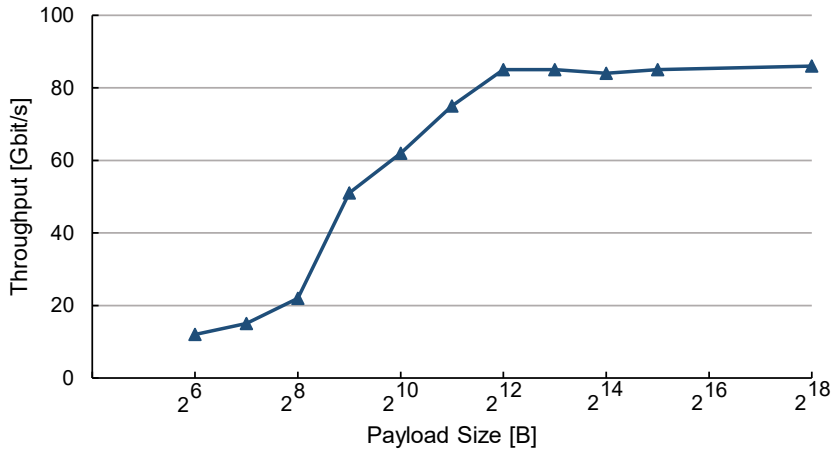

Fig. 17. Ethernet module throughput on the Alveo card as a function of payload size.

in the consumer's hardware deployment, we implemented a memory arbiter module to initiate execution of DPU kernels as soon as network data is received.

7 CONCLUSION AND FUTURE WORK

This article explored FPGA accelerator architectures for time series similarity prediction using CNNs. We integrated a custom IP accelerator block using different Xilinx DPUs to enable whole-model acceleration of the FA-LAMP CNN on two platforms: a Xilinx Ultra96-V2, which is representative of FPGA-accelerated edge computing, and Alveo U280 FPGA, which is representative of a cloud-based system. Compared to a Raspberry Pi 3 and an Edge TPU, our edge design achieved 5.7× and 18.2× higher inference rate and improved the energy efficiency by 8.7× and 24× respectively. We compared the cloud-based accelerator performance with LAMP running on a high-end desktop CPU as well as server CPU processors and a GPU. While the FPGAs could not compete with the server CPU in terms of throughput or inference rate, they reduced latency by two orders of magnitude and energy consumption by one order of magnitude. We also compared the performance of the DPU running FA-LAMP to four state-of-the-art frameworks for CNN compilation onto FPGAs; the result of this experiment showed that the DPU achieves the highest overall performance, with the exception of one framework (FINN) that uses much lower precision and therefore suffers from significant degradation in inference accuracy. Lastly, we integrated the DPU with a Xilinx 100 Gb/s Ethernet module on the Alveo card, demonstrating the ability process streaming data obtained directly from the network without the involvement of a host CPU.

We envision several avenues of future work to improve FA-LAMP. We would like to more thoroughly explore the space of sigmoid approximation functions, including piecewise alternatives to ultra_fast_sigmoid, which might be able to reduce its error, and variants of sigmoid_fastexp_N for values of N other than 512; there is also considerable opportunity to explore the internal architecture and precision of sigmoid_fastexp_N. We also would like to demonstrate that DPU-like overlays can efficiently implement global average pooling and sigmoid approximation functions, which would alleviate the need transfer data out of the overlay. Long-term, we would like to harden the FA-LAMP inference engine so that it can be integrated into a *system-on-chip (SoC)*, creating a near-sensor CNN inference system that can process streaming data.

ACKNOWLEDGMENTS

This work was supported in part by NSF Awards #1740052 and #1763795.

REFERENCES

[1] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, and Gordon R. Chiu. 2018. DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration. In *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. IEEE Computer Society, 411–418. https://doi.org/10.1109/FPL.2018.00077

[2] Alireza Abdoli, Sara Alaee, Shima Imani, Amy Murillo, Alec Gerry, Leslie Hickle, and Eamonn Keogh. 2020. Fitbit for Chickens? Time Series Data Mining Can Increase the Productivity of Poultry Farms. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*. ACM, 3328–3336.

[3] Richard Allen, Holly Brown, Margaret Hellweg, Oleg Khainovski, Peter Lombard, and Doug Neuhauser. 2009. Real-time earthquake detection and hazard assessment by ElarmS across California. *Geophysical Research Letters - GEOPHYS RES LETT* 36 (2009). https://doi.org/10.1029/2008GL036766

[4] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL™ Deep Learning Accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 55–64.

[5] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math compiler in python. In *Proceedings of the 9th Python in Science Conference*. 3–10.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594.

[7] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2017. Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators. *IEEE Micro* 37, 3 (2017), 12–21. https://doi.org/10.1109/MM.2017.54

[8] Philip Colangelo, Nasibeh Nasiri, Eriko Nurvitadhi, Asit Mishra, Martin Margala, and Kevin Nealis. 2018. Exploration of Low Numeric Precision Deep Learning Inference Using Intel FPGAs. In *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 73–80. https://doi.org/10.1109/FCCM.2018.00020

[9] Sergio Decherchi, Paolo Gastaldo, Alessio Leoncini, and Rodolfo Zunino. 2012. Efficient Digital Implementation of Extreme Learning Machines for Classification. *IEEE Trans. Circuits Syst. II Express Briefs* 59-II, 8 (2012), 496–500. https://doi.org/10.1109/TCSII.2012.2204112

[10] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, Luca P. Carloni, Giuseppe Di Guglielmo, Philip Harris, Jeffrey Krupa, Dylan Rankin, Manuel Blanco Valentin, Josiah Hester, Yingyi Luo, John Mamish, Seda Orgrenci-Memik, Thea Aarrestad, Hamza Javed, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol, Sioni Summers, Javier Duarte, Scott Hauck, Shih-Chieh Hsu, Jennifer Ngadiuba, Mia Liu, Duc Hoang, Edward Kreinar, , and Zhenbin Wu. 2021. hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices. https://doi.org/10.48550/ARXIV.2103.05579

[11] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, Murali Annavaram, Timothy Mark Pinkston, and Babak Falsafi (Eds.). IEEE Computer Society, 1–14. https://doi.org/10.1109/ISCA.2018.00012

[12] Yao Fu. 2017. Deep Learning with INT8 Optimization on Xilinx Devices. Retrieved July 12, 2021 from https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf

[13] Yunqi Gao, Feng Luan, Jiaqi Pan, Xu Li, and Yaodong He. 2020. FPGA-based implementation of stochastic configuration networks for regression prediction. *Sensors* 20 (2020), 4191.

[14] Xushen Han, Dajiang Zhou, Shihao Wang, and Shinji Kimura. 2016. CNN-MERP: An FPGA-based memory-efficient reconfigurable processor for forward and backward propagation of convolutional neural networks. In *34th IEEE International Conference on Computer Design, ICCD 2016, Scottsdale, AZ, USA, October 2-5, 2016*. IEEE Computer Society, 320–327. https://doi.org/10.1109/ICCD.2016.7753296

[15] Martin Hardieck, Martin Kumm, Konrad Möller, and Peter Zipf. 2019. Reconfigurable Convolutional Kernels for Neural Networks on FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 43–52.

[16] Amin Kalantar. 2021. FA-LAMP source code repository. https://github.com/aminiok1/lamp-alveo.

[17] Amin Kalantar, Zachary Zimmerman, and Philip Brisk. 2021. FA-LAMP: FPGA-Accelerated Learned Approximate Matrix Profile for Time Series Similarity Prediction. In *29th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2021, Orlando, FL, USA, May 9-12, 2021*. IEEE, 40–49. https://doi.org/10.1109/FCCM51124.2021.00013

[18] Kamalavasan Kamalakkannan, Gihan R. Mudalige, Istvan Z. Reguly, and Suhaib A. Fahmy. 2021. High-Level FPGA Accelerator Design for Structured-Mesh-Based Explicit Numerical Solvers. *arXiv preprint arXiv:2101.01177* (2021). arXiv:2101.01177

[19] Diederik Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *3rd International Conference on Learning Representations, ICLR* (2014).

[20] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*, Kia Bazargan and Stephen Neuendorffer (Eds.). ACM, 242–251. https://doi.org/10.1145/3289602.3293910

[21] Andrew Lavin and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 4013–4021. https://doi.org/10.1109/CVPR.2016.435

[22] Google LLC. 2020. Coral toolkit. https://coral.ai/.

[23] Google LLC. 2021. Edge TPU. https://cloud.google.com/edge-tpu.

[24] Microsoft. 2022. tensorflow ONNX. https://github.com/onnx/tensorflow-onnx.

[25] Sarah E. Minson, Men-Andrin Meier, Annemarie S. Baltay, Thomas C. Hanks, and Elizabeth S. Cochran. 2018. The limits of earthquake early warning: Timeliness of ground motion estimates. *Science Advances* 4, 3 (2018). https://doi.org/10.1126/sciadv.aaq0504 arXiv:https://advances.sciencemag.org/content/4/3/eaaq0504.full.pdf

[26] Alex Montgomerie. 2022. Streaming Architecture Mapping Optimiser. https://github.com/AlexMontgomerie/samo.

[27] Emmanuel Oyekanlu. 2017. Predictive edge computing for time series of industrial IoT and large scale critical infrastructure based on open-source software analytic of big data. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, Jian-Yun Nie, Zoran Obradovic, Toyotaro Suzumura, Rumi Ghosh, Raghunath Nambiar, Chonggang Wang, Hui Zang, Ricardo Baeza-Yates, Xiaohua Hu, Jeremy Kepner, Alfredo Cuzzocrea, Jian Tang, and Masashi Toyoda (Eds.). IEEE Computer Society, 1663–1669. https://doi.org/10.1109/BigData.2017.8258103

[28] Alessandro Pappalardo. 2021. *Xilinx/brevitas*. https://doi.org/10.5281/zenodo.3333552

[29] Lucian Petrica, Tobias Alonso, Mairin Kroes, Nicholas Fraser, Sorin Cotofana, and Michaela Blott. 2020. Memory-Efficient Dataflow Inference for Deep CNNs on FPGA. *arXiv preprint arXiv:2011.07317* (2020). arXiv:2011.07317

[30] PyRAPL software toolkit 2019. https://github.com/powerapi-ng/pyRAPL.

[31] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. 2016. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 26–35.

[32] Zhiqiang Que, Hiroki Nakahara, Eriko Nurvitadhi, Hongxiang Fan, Chenglong Zeng, Jiuxi Meng, Xinyu Niu, and Wayne Luk. 2020. Optimizing Reconfigurable Recurrent Neural Networks. In *28th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2020, Fayetteville, AR, USA, May 3-6, 2020*. IEEE, 10–18. https://doi.org/10.1109/FCCM48280.2020.00011

[33] SeyedRamin Rasoulinezhad, Hao Zhou, Lingli Wang, and Philip H. W. Leong. 2019. PIR-DSP: An FPGA DSP Block Architecture for Multi-precision Deep Neural Networks. In *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019*. IEEE, 35–44. https://doi.org/10.1109/FCCM.2019.00015

[34] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 779–788. https://doi.org/10.1109/CVPR.2016.91

[35] Claudio Satriano, Anthony Lomax, and Aldo Zollo. 2008. Real-Time Evolutionary Earthquake Location for Seismic Early Warning. *Bulletin of the Seismological Society of America* 98 (2008), 1482–1494. https://doi.org/10.1785/0120060159

[36] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 17:1–17:12. https://doi.org/10.1109/MICRO.2016.7783720

[37] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2015). arXiv:1409.1556

[38] Nicholas Gerard Timmons and Andrew Rice. 2020. Approximating Activation Functions. *arXiv preprint arXiv:2001.06370* (2020).

[39] Yaman Umuroglu. 2022. FINN: Fast, Scalable Quantized Neural Network Inference on FPGAs. https://github.com/Xilinx/finn.

[40] Stylianos I. Venieris and Christos-Savvas Bouganis. 2016. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In *24th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2016, Washington, DC, USA, May 1-3, 2016*. IEEE Computer Society, 40–47. https://doi.org/10.1109/FCCM.2016.22

[41] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. 2018. Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions. *ACM Comput. Surv.* 51, 3, Article 56 (jun 2018), 39 pages. https://doi.org/10.1145/3186332

[42] Zhiguang Wang, Weizhong Yan, and Tim Oates. 2017. Time series classification from scratch with deep neural networks: A strong baseline. In *2017 International Joint Conference on Neural Networks, IJCNN 2017, Anchorage, AK, USA, May 14-19, 2017*. IEEE, 1578–1585.

https://doi.org/10.1109/IJCNN.2017.7966039

[43] Denis S. Willett, Justin George, Nora S. Willett, Lukasz L. Stelinski, and Stephen L. Lapointe. 2016. Machine Learning for Characterization of Insect Vector Feeding. *PLOS Computational Biology* 12, 11 (11 2016), 1–14. https://doi.org/10.1371/journal.pcbi.1005158

[44] Ephrem Wu, Xiaoqian Zhang, David Berman, Inkeun Cho, and John Thendean. 2019. Compute-Efficient Neural-Network Acceleration. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 191–200. https://doi.org/10.1145/3289602.3293925

[45] Xilinx. 2019. DPU for Convolutional Neural Network v3.0, DPU IP Product Guide. Retrieved July 12, 2021 from https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_0/pg338-dpu.pdf

[46] Xilinx. 2022. Xilinx Vitis Network Example. https://github.com/Xilinx/xup_vitis_network_example.

[47] Zhilin Xu, Jincheng Yu, Chao Yu, Hao Shen, Yu Wang, and Huazhong Yang. 2020. CNN-based Feature-point Extraction for Real-time Visual SLAM on Embedded FPGA. In *28th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2020, Fayetteville, AR, USA, May 3-6, 2020*. IEEE, 33–37. https://doi.org/10.1109/FCCM48280.2020.00014

[48] C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh. 2016. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 1317–1322. https://doi.org/10.1109/ICDM.2016.0179

[49] Yunxuan Yu, Tiandong Zhao, Kun Wang, and Lei He. 2020. Light-OPU: An FPGA-Based Overlay Processor for Lightweight Convolutional Neural Networks. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 122–132.

[50] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 161–170. https://doi.org/10.1145/2684746.2689060

[51] Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[52] Yan Zhu, Shaghayegh Gharghabi, Diego Furtado Silva, Hoang Anh Dau, Chin-Chia Michael Yeh, Nader Shakibay Senobari, Abdulaziz Almaslukh, Kaveh Kamgar, Zachary Zimmerman, Gareth J. Funning, Abdullah Mueen, and Eamonn J. Keogh. 2020. The Swiss army knife of time series data mining: ten useful things you can do with the matrix profile and ten lines of code. *Data Min. Knowl. Discov.* 34, 4 (2020), 949–979. https://doi.org/10.1007/s10618-019-00668-6

[53] Zachary Zimmerman, Kaveh Kamgar, Nader Shakibay Senobari, Brian Crites, Gareth J. Funning, Philip Brisk, and Eamonn J. Keogh. 2019. Matrix Profile XIV: Scaling Time Series Motif Discovery with GPUs to Break a Quintillion Pairwise Comparisons a Day and Beyond. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 74–86. https://doi.org/10.1145/3357223.3362721

[54] Zachary Zimmerman, Nader Shakibay Senobari, Gareth J. Funning, Evangelos E. Papalexakis, Samet Oymak, Philip Brisk, and Eamonn J. Keogh. 2019. Matrix Profile XVIII: Time Series Mining in the Face of Fast Moving Streams using a Learned Approximate Matrix Profile. In *2019 IEEE International Conference on Data Mining, ICDM 2019, Beijing, China, November 8-11, 2019*, Jianyong Wang, Kyuseok Shim, and Xindong Wu (Eds.). IEEE, 936–945. https://doi.org/10.1109/ICDM.2019.00104