UC Riverside

UC Riverside Electronic Theses and Dissertations

Title

FPGA-Based Acceleration: From Cloud to Edge

Permalink

https://escholarship.org/uc/item/0wq5r3hj

Author

Kalantar Chahouki, Amin

Publication Date

2023

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

FPGA-Based Acceleration: From Cloud to Edge

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Amin Kalantar Chahouki

March 2023

Dissertation Committee:

Professor Philip Brisk, Chairperson
Professor Nael Abu-Ghazaleh
Professor Walid Najjar
Professor Daniel Wong

Copyright by
Amin Kalantar Chahouki
2023

The Dissertation of Amin Kalantar Chahouki is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to thank my advisor Dr. Philip Brisk for supporting me through the PhD program, I would not be here without his mentorship and help through these years.

I would like to express my deepest appreciation to my committee Dr. Walid Najjar, Dr. Nael Abu-Ghazaleh, and Dr. Daniel Wong. The completion of my dissertation would not have been possible without their support and guidance.

I would like to thank my labmates and friends, Farzin Houshmand, Dr. Kenneth O'neal, Dr. Jason Ott, Dr. Bashar Ramanous, Dr. Jose Rodriguez Borbon, and Amir Hoseinian, for being there with me through my education. Your friendship and attentivenes is what helped me get through the toughest days.

I would like to thank Dr. Zachary Zimmerman, for being both a coauthor and a friend through most of my time in the PhD program.

I would like to thank my parents for paving the way for me to be where I am today. The sacrifices that they have made for me is beyond words.

To Cecilia: You have changed my life for the better and I cannot thank you enough. Your support, love, laughs, and joy is what keeps me going.

To my parents,

for always loving and supporting me.

ABSTRACT OF THE DISSERTATION

FPGA-Based Acceleration: From Cloud to Edge

by

Amin Kalantar Chahouki

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2023
Professor Philip Brisk, Chairperson

Increasingly, FPGAs have shown promise of high processing power in various fields such as machine learning, computer vision, high frequency trading, and others thanks to their basic nature of configurable system. However, this question remains unanswered: "When will FPGAs become popular and mainstream?" Although hardware description languages are notoriously challenging to design with, current software toolchains play a major role in hindering the adoption of FPGAs. This dissertation investigates performance gains by accelerating time series prediction on different FPGA platforms–from smaller edge computing devices to high end data center cards– as well deploying FPGAs as SmartNICs to data centers, and analyzes the challenges and limitations of current FPGA design flow (through both HDL and High Level Synthesis development).

First, we present FA-LAMP, an FPGA-accelerated implementation of the Learned Approximate Matrix Profile algorithm, which predicts the correlation between streaming data sampled in real-time and a representative time series dataset used for training. We expose several technical limitations of Xilinx DPU for convolutional neural network acceleration

on FPGAs, while providing a mechanism to overcome them. Furthermore, we show how Learned Approximate Matrix Profile algorithm can be deployed on data center FPGA cards. We implement two different versions of FA-LAMP for high throughput and low latency applications- and show how to integrate DPU on the Alveo card with an Ethernet module that allows for processing real-time data streams delivered over a network. We discuss different strategies to connect the Ethernet IP to the DPU and present methods to further increase network throughput.

Finally, we show how FPGAs can be used as SmartNICs to ensure consistency in data centers. We implement different consensus protocols on the FPGA that leverage Remote Direct Memory Access (RDMA) to replicate user requests in memory and to fail-over the system with negligible latency. We evaluate the throughput and response time of our design for three scenarios: No failure, leader failure, and replica failure. We also leverage SMT solvers to come up with an optimal memory layout to allocate replication logs.

Contents

List of Figures **x**

List of Tables **xii**

1 Introduction **1**

2 Related Work **6**
- 2.1 Accelerating Neural Networks on FPGAs . . . 6
  - 2.1.1 Optimizing Accelerator Design . . . 6
  - 2.1.2 Automated Frameworks for DNN Compilation . . . 7
  - 2.1.3 Xilinx DPU . . . 8
- 2.2 FPGAs as SmartNIC . . . 9
- 2.3 RDMA-based Consensus Protocols . . . 10

3 FA-LAMP: FPGA-Accelerated Learned Approximate Matrix Profile for Time Series Similarity Prediction **12**
- 3.1 Introduction . . . 12
- 3.2 Related Work . . . 14
- 3.3 FA-LAMP System Overview . . . 15
  - 3.3.1 Background: Time Series and the Matrix Profile . . . 15
  - 3.3.2 Background: LAMP . . . 16
  - 3.3.3 Xilinx DPU: Objective and Technical Challenges . . . 18
  - 3.3.4 DPU for Edge Processing . . . 20
  - 3.3.5 DPU for Cloud Acceleration . . . 21
  - 3.3.6 HLS Kernel . . . 23
- 3.4 Experimental Setup . . . 31
  - 3.4.1 Model Training . . . 31
  - 3.4.2 Model Inference . . . 33
  - 3.4.3 Measurements . . . 38
  - 3.4.4 Benchmarks . . . 39
  - 3.4.5 Source code and Data Availability . . . 40
- 3.5 Results . . . 41

3.5.1 Edge: Throughput and Resource Utilization . . . 41
3.5.2 Edge: Comparison to a Raspberry Pi 3 and Edge TPU . . . 43
3.5.3 Cloud Prototype: Throughput and Energy . . . 45
3.5.4 Inference Accuracy . . . 47
3.5.5 Comparison to Recent CNN-to-FPGA Compilation Frameworks . . . 48
3.5.6 Case Study: Interpreting the FA-LAMP Output . . . 49
3.6 DPU Integration with Ethernet . . . 51
3.7 Conclusion . . . 55

4 Ensuring Consistency in Data Centers using RDMA-Enabled Consensus Protocols on FPGAs 57
4.1 Disclaimer . . . 57
4.2 Introduction . . . 58
4.3 Background . . . 60
4.3.1 RDMA and RoCE . . . 60
4.3.2 Alveo Accelerator Cards . . . 61
4.3.3 Coordination and Hamband . . . 62
4.4 Design Overview . . . 64
4.4.1 RDMA Replicated Data Types . . . 64
4.4.2 FPGA Architecture . . . 66
4.4.3 Log Allocation Optimizations . . . 68
4.5 Design Methodology . . . 69
4.5.1 RoCE Stack . . . 69
4.5.2 Application Engine . . . 73
4.5.3 Replication Engine . . . 76
4.5.4 Control Flow . . . 83
4.5.5 Memory Allocation Optimizations . . . 84
4.6 Experimental Setup and Results . . . 87
4.6.1 Hypothesis . . . 87
4.6.2 Deployment Platforms and Toolchains . . . 87
4.6.3 Measurements . . . 89
4.6.4 Benchmarks . . . 89
4.6.5 Results . . . 90

5 Conclusion 102
5.1 FA-LAMP Concluding Remarks . . . 102
5.2 FPGA-Based Consensus Concluding Remarks . . . 103
5.3 Future Work for FA-LAMP . . . 103
5.4 Future Work for FPGA-based Consensus . . . 104

Bibliography 105

List of Figures

3.1 Matrix Profile (MP) computation for subsequences of length m: $c_{i,j}$ denotes the Pearson Correlation between the ith 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 ith entry in the MP. 16
3.2 Illustration of the parameters used for LAMP inference on a streaming time series. 17
3.3 The CNN used for LAMP inference. Batch normalization layers are omitted to simplify the presentation. 18
3.4 Zynq DPU architecture. 19
3.5 High-throughput DPUCAHX8H architecture, comprising three DPU instances with multiple batch engines for parallel data processing. 21
3.6 Low-latency DPUCAHX8L architecture, comprising two DPU instances with one convolution engine, scheduler, and code FIFO units. 22
3.7 Column-wise vector-matrix multiplication tiling scheme. 25
3.8 Fully connected layer hardware architecture. 26
3.9 (a) Approximation functions for sigmoid and (b) their error. Both charts were computed using an 8-bit fixed-point data type. 29
3.10 Improvements in custom kernel latency and resource utilization due to HLS optimizations. 30
3.11 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. 31
3.12 Alveo architecture programmed with the high throughput DPU. 32
3.13 Overview of deploying a LAMP model on a DPU. 33
3.14 A snippet of chicken accelerometer data with corresponding labels (Preening: label height = 3, dustbathing: label height = 4, and pecking: label height = 6). 41
3.15 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). 51
3.16 The SLR0 in the Alveo card configured with the Ethernet subsystem and the custom kernel IPs. 51
3.17 Ethernet module throughput on the Alveo card as a function of payload size. 52

4.1 Proposed FPGA architecture. 67
4.2 RDMA Queue Pair states and the possible transitions. 69
4.3 Application Engine comprises memory manager, request scheduler, and app processor units. 73
4.4 Control flow for executing query command. 77
4.5 Control flow for executing reducible methods. 77
4.6 Control flow for executing irreducible conflict-free methods. 79
4.7 Control flow for executing conflicting methods. 80
4.8 RDMA, memory, and consensus commands flow in the system. 82
4.9 Comparison of throughput for reducible methods. 92
4.10 Comparison of throughput for irreducible conflict-free methods. 93
4.11 Comparison of response time for reducible and irreducible conflict-free methods. 94
4.12 Comparison of throughput and response time for Movie benchmark. 94
4.13 Project management throughput and response time. 95
4.14 Effect of failure on the Counter and ORSet use-cases. 96
4.15 The effect of failure on the courseware throughput. 97
4.16 The effect of failure on the courseware response time per method. 98
4.17 Effect of log allocation optimization. 100
4.18 Comparison of energy consumption for Mu, Hamband, and our proposed work. (note the logarithmic scale on the vertical axis). 101

List of Tables

1.1 Comparison between Ultra96-V2 and Alveo U280 FPGA specifications. 2

3.1 Edge Prototype: Throughput (GOPS) and resource utilization comparison between different DPU architectures; (DPU + IP) uses a B2304 DPU. 42
3.2 Edge Prototype: Inference rate and energy consumption of LAMP neural network inference on an Edge TPU, Raspberry Pi 3 and Ultra96-V2 board. . 44
3.3 Cloud prototype: throughput, latency, inference rate and energy consumption: LL=Alveo low-latency, HT=Alveo high-throughput. 45
3.4 FA-LAMP neural network inference accuracy; qa=quantization-aware training, edge=Ultra96, cloud=Alveo. 47
3.5 Performance comparison with other FPGA-based edge and cloud DNN deployment frameworks. 48

4.1 Available memories on the Alveo U280. 86
4.2 Hardware details of ETHZ-HACC. 90
4.3 Alveo accelerator cards specifications. 91

Chapter 1

Introduction

Due to their promising advantages such as fast time-to market, reduced Non-Recurring Engineering (NRE) costs, as well as flexibility to fast implementation and modification of digital designs, Field-Programmable Gate Arrays (FPGAs) are gaining popular usage in a diverse range of use-cases from embedded systems to cloud applications. Although FPGAs are widespread, they cannot be randomly deployed as part of the system. A number of choices must be made in order to make the most efficient use of available hardware resources on the FPGA. These choices include the architecture parameters such as the number of pipeline stages; constraints like hardware area and the memory bandwidth; and most importantly the method of developing.

Two different methods for developing accelerator designs are High Level Synthesis (HLS) and Register Transfer Level (RTL). Both of these methods have advantages and disadvantages: RTL exposes a lower level of abstraction to the programmers which can increase the performance and efficiency of the system at the cost of lower productivity. On

Table 1.1: 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

the other hand, HLS offers a faster development cycle while limiting the developers to only high level pragma optimizations.

The purpose of this dissertation is to investigate the challenges of deploying FPGAs in two areas: Convolutional Neural Network (CNN) inference and FPGAs as SmartNIC with Remote Direct Memory Access (RDMA) capabilities. The FPGA platforms that we use in this dissertation are Xilinx Ultra96-V2 and Alveo U280. Table 1.1 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 rest of this dissertation is organized as follows. Chapter two discusses related work for FPGA-based neural network acceleration, FPGAs as SmartNICs, and RDMA-based object replication in distributed systems. We classify FPGA-based deep neural network (DNN) studies within three areas (a) optimization techniques for an FPGA-based DNN accelerator design (b) automated frameworks for deploying trained DNN models on the FPGAs; and (c) overlay architectures for DNN acceleration.

Chapter three presents 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 chapter 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.

In chapter four we use the Alveo U280 and U250 FPGA cards as SmartNICs to replicate user requests in a data center. To ensure consistency, we base our work on an RDMA-based hybrid consistency model Hamband [43]. Hamband divides the methods of an application into three categories based on whether they require strong or weak consistency and whether they are summarizable and declares coordination requirements for each category. We extend an already existing RDMA stack for FPGAs (StRoM) [88] and implement Hamband's coordination protocols on the FPGA to keep the replication states as close as possible to the network stack. We evaluate our design for different scenarios (no failure, leader failure, replica failure) for various use-cases including Convergent and Commutative Replicated Data Types (CRDTs) and three relational schemata: project management, courseware, and movie. We compare the throughput and response time of our design with similar CPU-based replication systems, namely, Mu [3] and Hamband [43]. Our design improved the throughput by 1.48× and 6.62× compared to Hamband and Mu for CRDTs and reduced the response time by 2.14× and 1.49×, respectively. For a use-case containing method calls in all three

categories of semantics, our design improved the throughput by 1.22× and 1.45× compared to Hamband and Mu.

Finally, chapter five concludes this dissertation.

Chapter 2

Related Work

2.1 Accelerating Neural Networks on FPGAs

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.

2.1.1 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 [116]. Another recent accelerator architecture [103] 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 [75].

Colangelo et al. extended Intel's FPGA Deep Learning Acceleration (DLA) Suite [7] to accelerate networks with 8-bit and sub 8-bit activations and weights [18]. 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 [78, 81, 110].

We take inspiration from these studies in implementing our handcrafted kernels. We employ loop tiling [116], data reuse [38, 37, 16], and quantization [18] to improve their efficiency.

2.1.2 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 [56] is a Python-based domain-specific language (DSL) extended from TVM [14] 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 [51]. DNNWEAVER [87] 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 [26], which targets low-power applications; fpgaConvNet [96] which achieved the best throughput

per DSP unit in a recent survey [97]; VTA [14], which uses a TVM-based compiler stack; and FINN [95] which is developed and maintained by Xilinx.

Many similar frameworks that map DNNs onto FPGAs have been published; however, they tend to suffer from other shortcomings such as performance deficiencies [118, 115, 114] or untenable resource utilization [99, 96, 107]. The framework proposed in LUTNet [98] implements the desired network model using LUTs as inference operators. Cloud-DNN [15] maps a trained CNN model specified in Caffe [48] to a cloud-based FPGA. Similar frameworks have been proposed that map neural networks onto FPGAs using either HLS: FP-DNN [34], Caffeine [115], and AutoDNNchip [108], RTL: DeepBurning [99], CaFPGA [107], DNNBuilder [118], TuRF [120], and the frameworks in [114, 66], or RTL-HLS hybrid: fpgaConvNet [96]. [34, 115, 108, 15], RTL: [99, 107, 118, 120, 114, 66], or RTL-HLS hybrid: [96, 98]. References [99, 96, 107] produce desirable throughput and latency results; however, their resource utilization in these works is untenable. References [87, 99, 115, 96, 34] achieve appealing performance at the cost of low flexibility and can only handle conventional CNNs. On the other hand [118, 115, 114] support versatile models in exchange for performance deficiencies. Large number of DRAM accesses in [66] leads to unwanted amount of energy consumption. Kernels designed in [56] suffers from routing congestion in large FPGAs.

2.1.3 Xilinx DPU

Recently, Xilinx introduced the Deep Learning Processor Unit (DPU), a programmable engine optimized for CNNs [104]. The DPU supports a variety of deep learning models, including, but not limited to ResNet [117], VGG [89], YOLO [82]. 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 [29] 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 [58] to optimize the performance and bandwidth of convolutional and fully connected layers. Lastly, Light-OPU [113] 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.

2.2 FPGAs as SmartNIC

Deploying FPGAs as SmartNICs has gained increasing attention in recent years due to the growing demand for high-performance and low-latency networking in data centers. FPGAs can be customized to perform specific tasks, making them ideal for accelerating network functions such as packet processing, filtering, and routing. In the context of SmartNICs, FPGAs can be used to offload network processing from the CPU, reducing the load on the server and improving network performance.

KV-Direct [60] leverages FPGAs as programmable NIC to extend RDMA primitives to enable remote direct key-value access to the main host memory. StRoM [88] provides an open source RDMA stack for the FPGAs that can be integrated with user kernels such as cardinality estimation and data shuffling to allow in-network processing of RDMA streams. The work proposed in [47] designs an atomic broadcast (consensus) protocol based on the TCP for the FPGAs. The goal of the paper is to push down the consensus protocol into the network in an efficient manner remove it process from the critical path of the CPU performanc. The nanoPU provides a fast path between the network stack and the application that bypasses the cache and memory hierarchy [46]. The work targets specific datacenter applications: those that utilize many small Remote Procedure Calls (RPCs) with very short (µs-scale) processing time.

FPGAs have been utilized as SmartNICs in data centers as well. Microsoft Azure [28] uses FPGAs to implement Azure Accelerated Networking (AccelNet) which enables VM-VM TCP connections with a significant high throughput of 32 Gbps. NVIDIA provides an FPGA-based smart NIC to improve the efficiency and scalability of the network in AI, HPC, and security applications [71].

2.3 RDMA-based Consensus Protocols

RDMA enables high-speed data transfer with low latency and reduced CPU overhead making it an attractive choice for microsecond-scale [11] replicated services whose availability and low-latency are critical in applications such as finance and control.

Mu [3] uses one-sided RDMA operations (READ and WRITE) to implement a consensus protocol for strong consistency based on paxos [57]. Mu is capable of replicating requests in less than 1.3 microseconds and takes less than a millisecond to fail-over the system. Hamband [43] avoids synchronizing the operations that do not require strong consistency. It divides the methods of an object into three categories based on whether the operations require strong or relaxed consistency and whether they are summarizable. All three categories of methods are replicated using one-sided RDMA operations. Hermes [53] presents a broadcast-based reliable replication with last-write-wins policy and using timestamps for each write. Kite [32] provides the first highly available key-value stores (KVSs) by using the release consistency (RC) model. ECROs [22] based on the replicated data types (RDTs) model, statically analyzes and reorders conflicting operations to have minimal coordination.

Chapter 3

FA-LAMP: FPGA-Accelerated Learned Approximate Matrix Profile for Time Series Similarity Prediction

3.1 Introduction

The proliferation of IoT sensors and the volume of data that they generate creates unique challenges in edge computing [72]. One motivating application, among many, is real-time seismic event prediction, which can inform hazard response strategies and enhance early warning systems [68, 4, 84]. 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 dissertation describes an FPGA-based accelerator for a streaming time series prediction scheme called the *Learned Approximate Matrix Profile (LAMP)* [127]. 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 [126]. It is certainly more practical to perform inference on a moderately sized CNN; nonetheless, the overhead of CNN inference remains a computational bottleneck that 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. We also present the results for the edge-based FA-LAMP implementation on the Xilinx Zynq UltraScale+ MPSoC.

3.2 Related Work

Most FPGA-based deep neural network studies focus on accelerator design, compilation frameworks, and/or domain-specific overlays. Our work borrows ideas from all three areas. Our HLS-generated IP blocks employ optimizations such as loop tiling [116], data reuse [38, 37, 16], weight-stationary multiplication [103], memory packing [75], fixed-point numeric formats [18], and model weight quantization [111, 78, 70, 6, 81, 110].

Commercial frameworks, such as Xilinx's Vitis AI framework, which we used, or Intel's Deep Learning Acceleration (DLA) suite [7], take inspiration from general-purpose

languages and frameworks such as HeteroCL [56, 51], as well as frameworks specific to neural networks [87, 118, 115, 114, 99, 107, 108, 120, 66, 96, 15, 14, 98, 34].

We used the Xilinx programmable DPU overlay [104], which was optimized for well-known convolutional neural networks [117, 89, 82]; similar overlays include Microsoft's Project Brainwave [29], Intel DLA [1], and Light-OPU [113]. Our work includes mixed use of a programmable overlay and custom IP blocks designed using HLS, and a fairly detailed analysis of how to implement the sigmoid activation function.

3.3 FA-LAMP System Overview

3.3.1 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). 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)* [126] (Figure 3.1) 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

[1] Historically, Euclidean distance between z-normalized subsequences is used as the distance function for time series data mining tasks [112]; the use of Pearson correlation, which limits the range of correlation values to [-1, +1], is more recent [126, 127] and is arguably more intuitive as the maximum Euclidean distance value is unbounded.

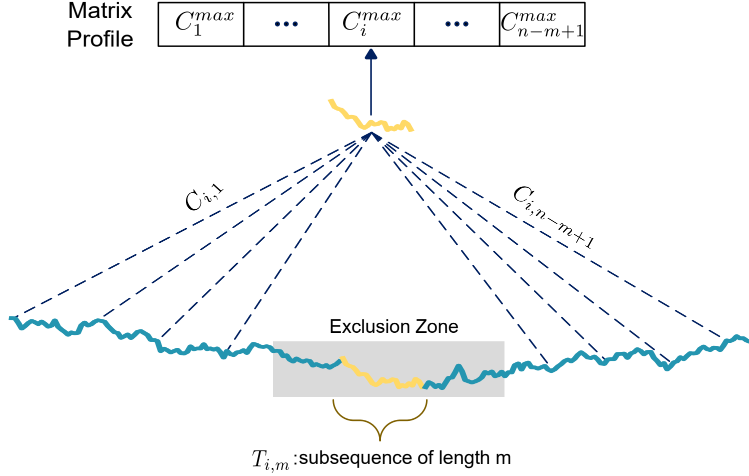

Figure 3.1: Matrix Profile (MP) computation for subsequences of length m: $c_{i,j}$ denotes the Pearson Correlation between the ith and j^{th} subsequences, $T_{i,m}$ and $T_{j,m}$ for all j, excluding an exclusion zone surrounding $T_{i,m}$. The maximum Pearson Correlation value c_i^{max} is stored as the ith entry in the MP.

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 [124].

3.3.2 Background: LAMP

The MP is itself a time series; while the MP can be computed efficiently with GPUs [126], doing so is not amenable to streaming data. While the time complexity to compute the MP is $O(n^2 logn)$ [112], 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) [127] predicts the maximum

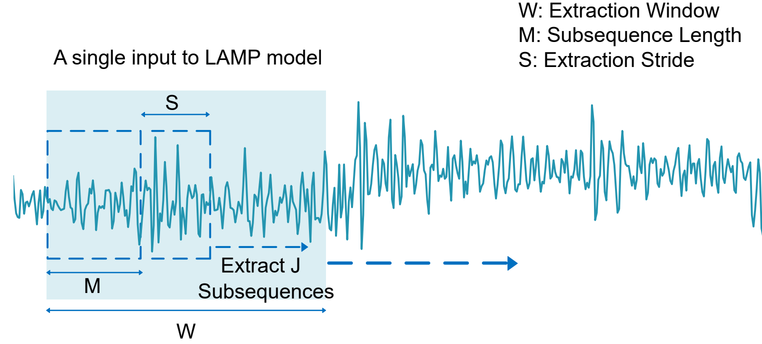

Figure 3.2: Illustration of the parameters used for LAMP inference on a streaming time series.

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 3.2 illustrates the LAMP inference process. Each input consists of $\mathbf{J}$ z-normalized (zero mean and unit variance) subsequences of length $\mathbf{M}$, extracted with stride $\mathbf{S}$. This scheme defines an extraction window in the 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 3.3. 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 [117] for time series classification [100, 127]. Model inputs and outputs are modified to support concurrent predictions. The first layer in the LAMP CNN is batch normalization (omitted from Figure 3.3 for simplicity); each convolutional layer in the model is followed by a batch normalization layer (also omitted from

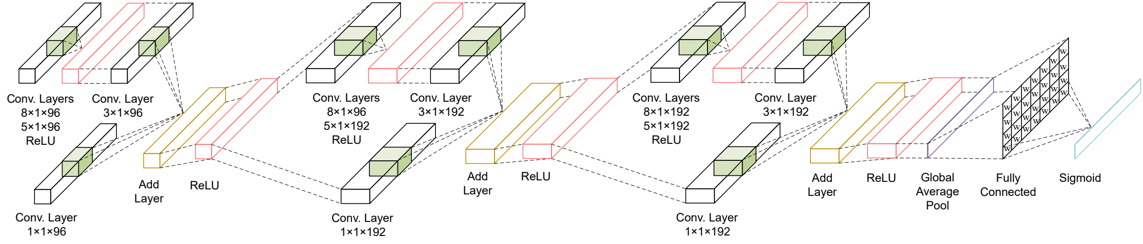

Figure 3.3: The CNN used for LAMP inference. Batch normalization layers are omitted to simplify the presentation.

Figure 3.3), 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 3.3 reports the kernel dimensions and number of filters used below each convolution layer.

3.3.3 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 [104]. 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.

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 3.3; these layers must be implemented in software running on one of the Arm CPU cores

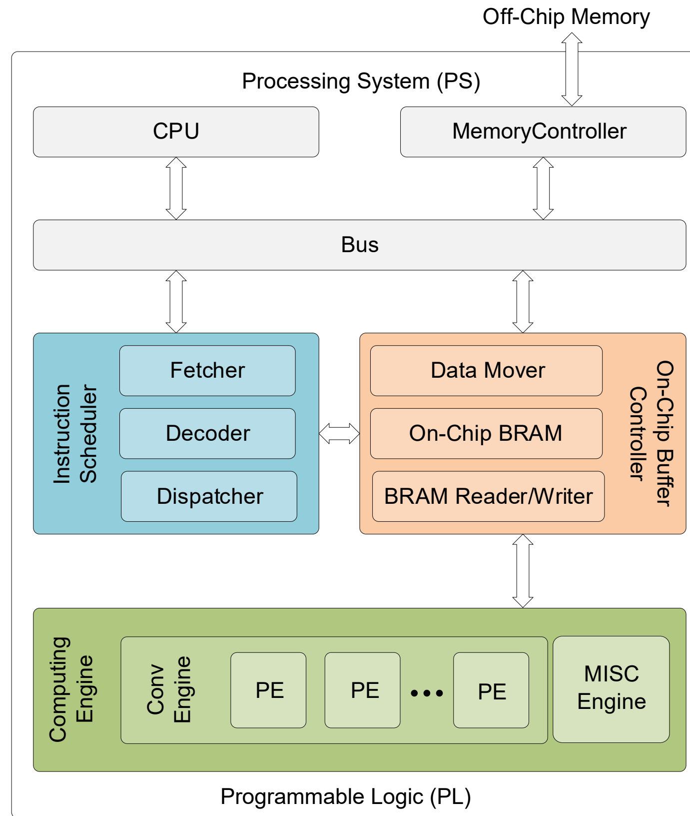

Figure 3.4: Zynq DPU architecture.

(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.

3.3.4 DPU for Edge Processing

Figure 3.4 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.

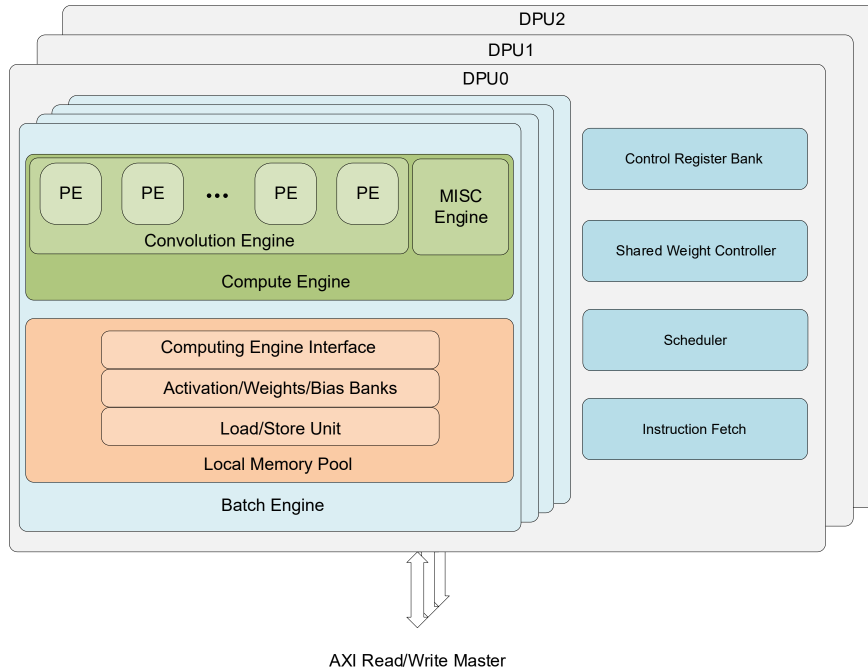

Figure 3.5: High-throughput DPUCAHX8H architecture, comprising three DPU instances with multiple batch engines for parallel data processing.

3.3.5 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 3.5) and the other is low-latency (Figure 3.6). 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

[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.

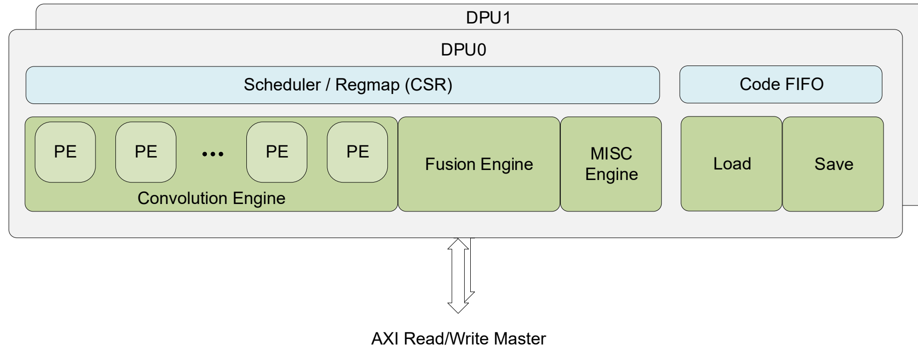

Figure 3.6: Low-latency DPUCAHX8L architecture, comprising two DPU instances with one convolution engine, scheduler, and code FIFO units.

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 3.5 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 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 3.6 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.

3.3.6 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 3.3 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,

$$q_j \leftarrow \frac{1}{M} \sum_{i=1}^{M} D_{i,j}, \quad 1 \leq j \leq N. \tag{3.1}$$

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 $z \in \mathbb{R}^M$.

$$z \leftarrow qW + b. \tag{3.2}$$

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 ith 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 [79] to optimize performance. We tile the weight matrix W into small $n_c \times n_r$ blocks as shown in Figure 3.7; 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 3.8 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 $n_c \times n_r$ parallel multiplications per clock cycle[3], storing the accumulated sums in a

[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.

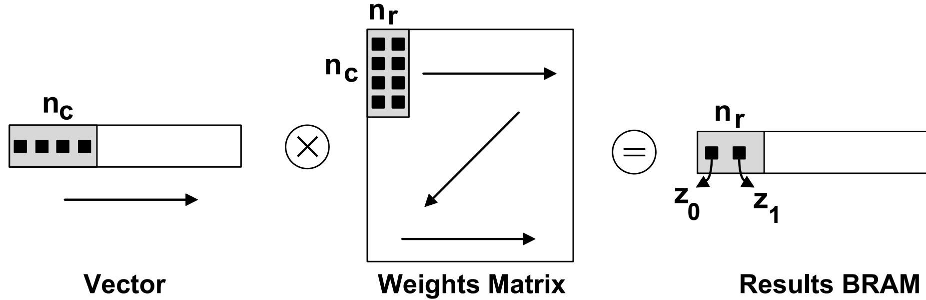

Figure 3.7: Column-wise vector-matrix multiplication tiling scheme.

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 $z_i \in z$:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.3}$$

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 [31], we chose two variants to evaluate: `ultra_fast_sigmoid`, a piece-wise approximation used in the Theano library [12]; and `sigm_fastexp_512`, which expands the exponential function for an infinite limit [92].

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 whether they are implemented using fixed-

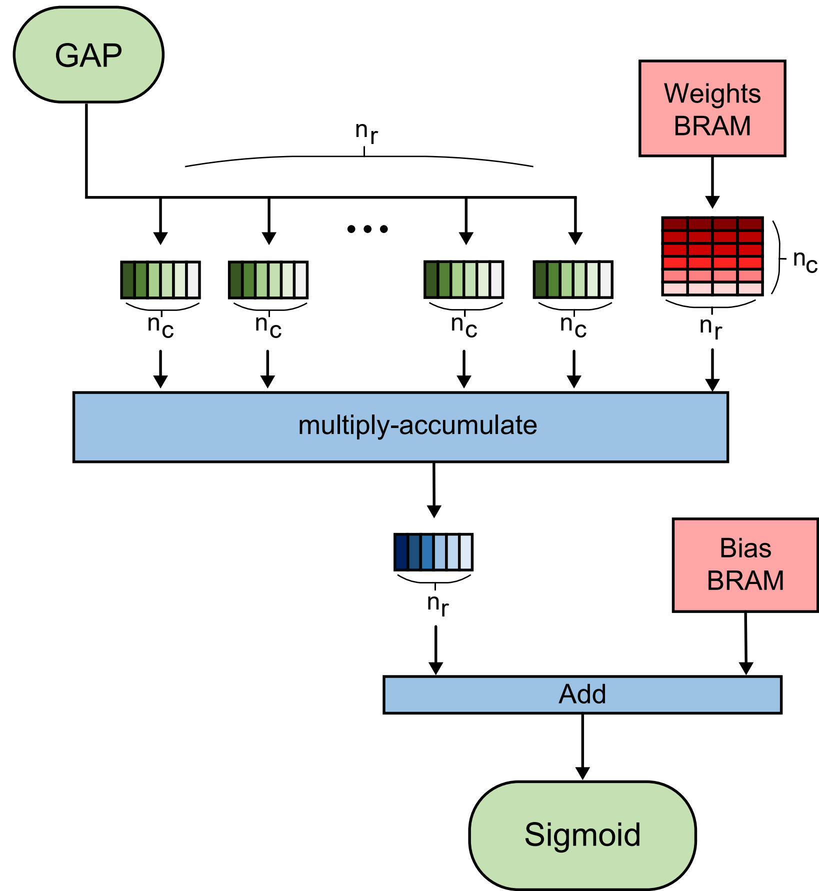

Figure 3.8: Fully connected layer hardware architecture.

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.

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

The ultra_fast_sigmoid approximation is defined as follows:

$$f(x) = \begin{cases} 0.5(\frac{\frac{1.5x}{2}}{1+\frac{x}{2}} + 1) & 0 \leq \frac{x}{2} < 1.7 \\ 0.5(1 + 0.935 + 0.045(\frac{x}{2} - 1.7)) & 1.7 \leq \frac{x}{2} < 3 \\ 0.5(1 + 0.995) & \frac{x}{2} \geq 3 \\ 0.5(-\frac{\frac{-1.5x}{2}}{1-\frac{x}{2}} + 1) & -1.7 \leq \frac{x}{2} \leq 0 \\ 0.5(1 - (0.935 + 0.045(-\frac{x}{2} - 1.7))) & -3 < \frac{x}{2} \leq -1.7 \\ 0.5(1 - 0.995) & \frac{x}{2} \leq -3 \end{cases} \tag{3.4}$$

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 ($n \to \infty$), using a value of $n = 512$ to render the approximation computable [92]:

$$\exp(x) = \prod_{k=1}^{lg(n)} (1 + \frac{x}{k})^k, \;\; n = 512 \tag{3.5}$$

$$\text{sigm}(x) = \frac{1}{1 + \exp(-x)} \tag{3.6}$$

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. (3.4) or Eq. (3.6). 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 3.9 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 [23], 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 3.10, 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× [30].
- Loop-Tiling-n_r tiling the fully connected layer (see Figure 3.7), while retaining the 8-bit data format.

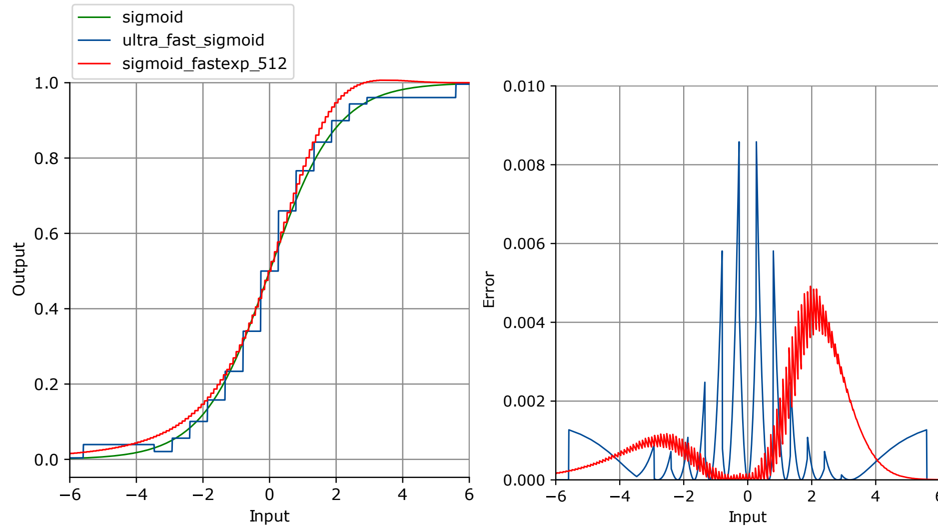

Figure 3.9: (a) Approximation functions for sigmoid and (b) their error. Both charts were computed using an 8-bit fixed-point data type.

The average resource axis in Figure 3.10 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 3.11 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.

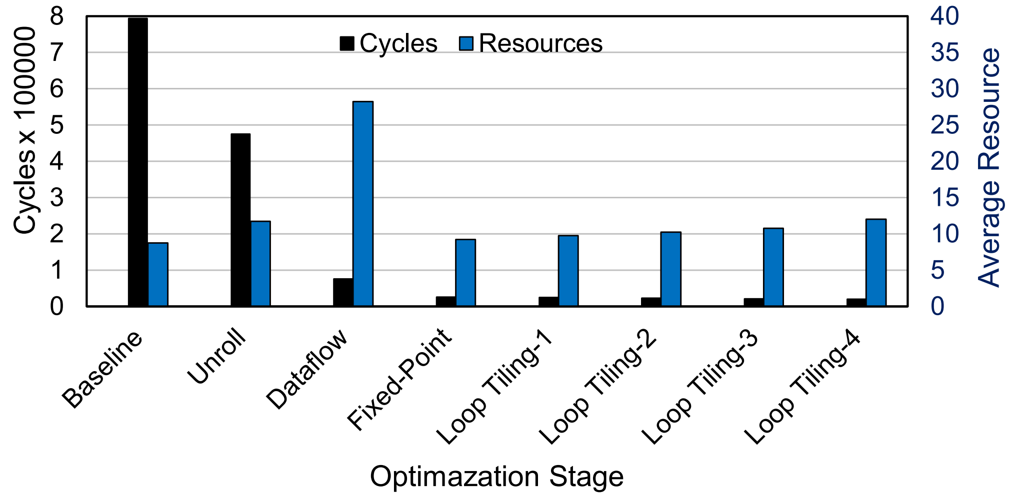

Figure 3.10: Improvements in custom kernel latency and resource utilization due to HLS optimizations.

Figure 3.12 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

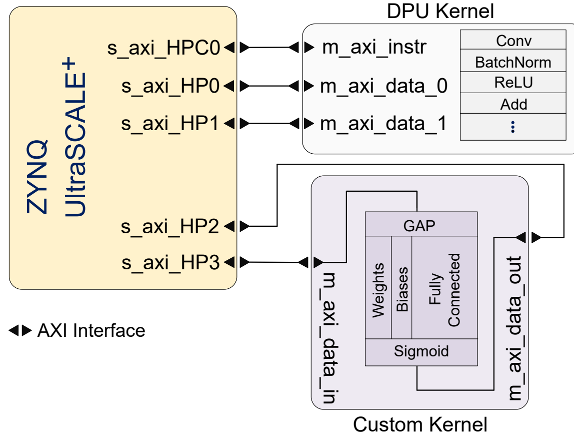

Figure 3.11: The FA-LAMP edge implementation comprises a Zynq UltraScale+ processing system, DPU IP, and custom HLS kernel; the HLS kernel implements the GAP, fully connected, and sigmoid layers.

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.

3.4 Experimental Setup

Figure 3.13 depicts the LAMP model training process and DPU deployment workflow; a detailed explanation follows.

3.4.1 Model Training

FA-LAMP deployment on an FPGA begins by training the model. We set the number of subsequences **J** to 32 [127], the length of window **M** to 100, and the stride **S** to 8. We used the Adam [54] 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 is to minimize the mean squared error loss between the predicted and exact MP values for the training data set. We

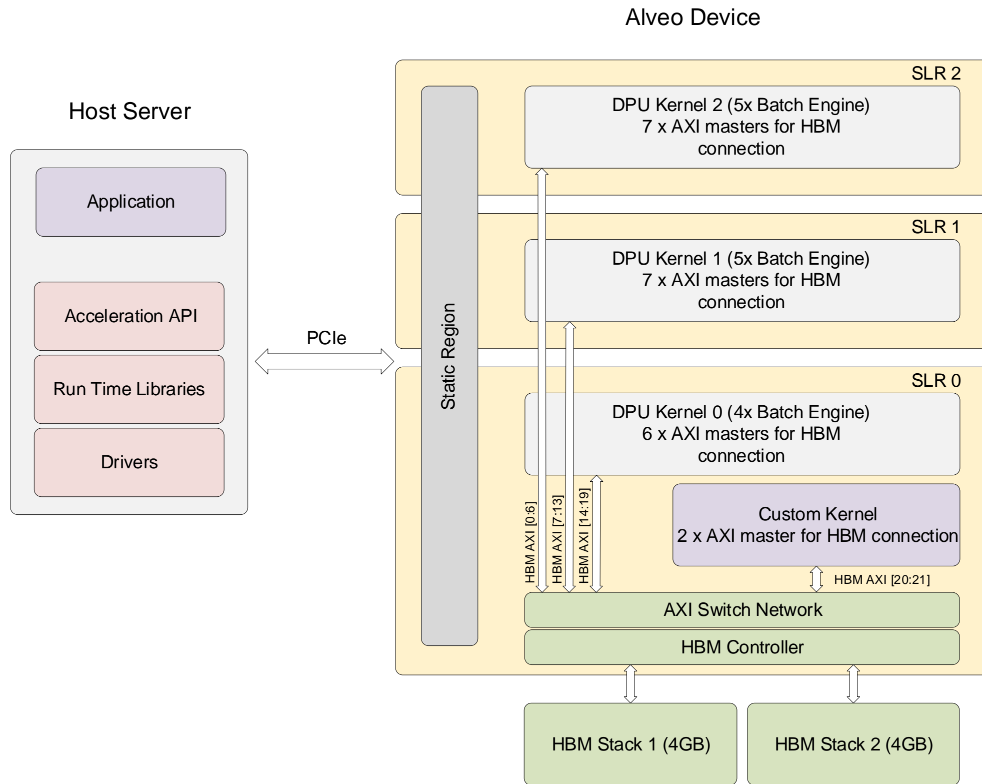

Figure 3.12: Alveo architecture programmed with the high throughput DPU.

removed the first batch normalization layer from the LAMP CNN [127]: 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 [127] 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.

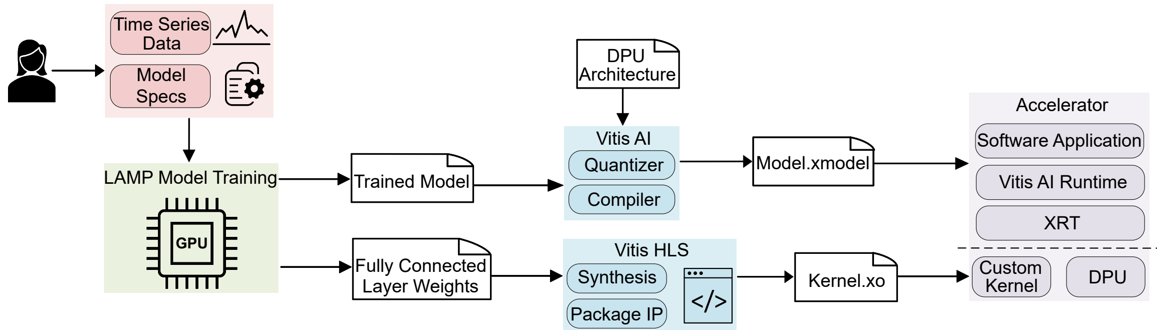

Figure 3.13: Overview of deploying a LAMP model on a DPU.

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 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.

3.4.2 Model Inference

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. 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.

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.

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 [64] which contains a Google Edge TPU coprocessor [65], 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.

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 [26], fpgaConvNet [96], VTA [14], and FINN [95]. 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 [67]. 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 [69], 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 [74], 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.

3.4.3 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 [77], 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.

3.4.4 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 [127].

(1) Seismology Domain: The Earthquake dataset is obtained from a seismic station [126]. Real-time event prediction impacts seismic hazard assessment, response, and early warning systems [68, 4, 84]. 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 [126]. 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 3.14 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.

3.4.5 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 [49].

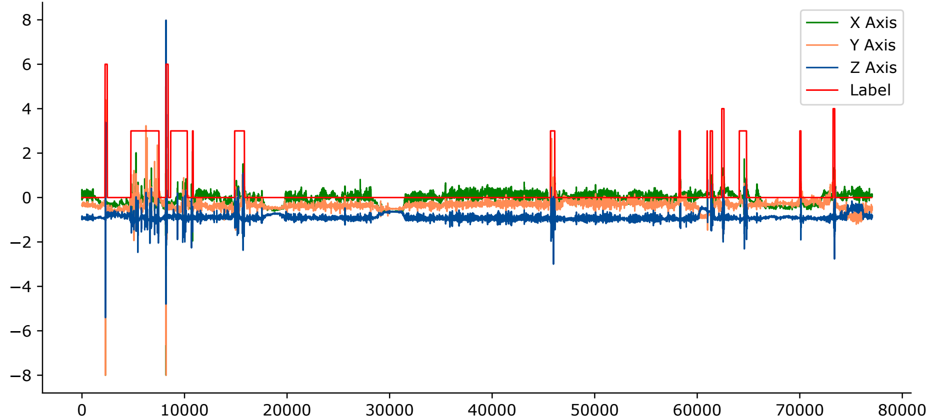

Figure 3.14: A snippet of chicken accelerometer data with corresponding labels (Preening: label height = 3, dustbathing: label height = 4, and pecking: label height = 6).

3.5 Results

3.5.1 Edge: Throughput and Resource Utilization

Table 3.1 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 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 3.1 show that system throughput increases as DPU size and complexity increases, from B512 to B2304. The highest overall throughput

Table 3.1: 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)
Logic Usage	39K (56%)	42K (59%)	46K (65%)	44K (62%)	49K (70%)	52K (75%)
Register Usage	50K (36%)	57K (40%)	65K (46%)	64K (45%)	77K (54%)	87K (62%)
DSP Usage	78 (21%)	117 (32%)	154 (42%)	164 (45%)	232 (64%)	289 (79%)
On-chip RAM Usage	77 (35%)	95 (44%)	109 (50%)	127 (58%)	131 (60%)	171 (79%)
Throughput (GOPS)	70.4	107.0	154.2	167.6	220.2	367.1
Peak Throughput (GOPS)	153	240	307	345	480	691

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 3.1 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.

3.5.2 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 3.2 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 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 $\sim 6\times$ compared to the

Table 3.2: 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

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 3.1, 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.

Table 3.3: 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)
Throughput (TOPS)	92.52	69.57	4.91	2.26	3.04	2.53	4.27	5.53
Latency (ms)	356	227	296	5.68	3.25	3.49	6.29	3.85
Inference Rate (KHz)	3079	2298	163	75	101	84	142	184
Energy (J)	118.32	72.20	38.43	9.15	6.81	8.20	4.86	3.75
GOPs/Watt	1210	740	68	44	57	46	77	98

3.5.3 Cloud Prototype: Throughput and Energy

Table 3.3 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 3.1, 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 throughput than the low-latency DPU. Referring back to Figures 3.5 and 3.6, 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 3.6). 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 3.3 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

Table 3.4: 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
Earthquake	120M / 30M	97.4%	91.4%	92.5%	93.8%	94.7%	94.3%
Insect EPG	2.5M / 5M	97.2%	90.8%	93.2%	91.9%	94.4%	92.5%
Chicken Accel.	6M / 2M	95.8%	86.9%	91.1%	89.5%	93.1%	90.2%

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%.

3.5.4 Inference Accuracy

Table 3.4 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 [50] 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

Table 3.5: Performance comparison with other FPGA-based edge and cloud DNN deployment frameworks.

	HLS4ML [26]	fpgaConvNet [96]	VTA [14]	FINN [95]	DPU	fpgaConvNet [96]	FINN [95]
FPGA Platform	Ultra96	Ultra96	Ultra96	Ultra96	Ultra96	Alveo U280	Alveo U280
Precision	fix-8	fix-8	int-8	fix-4	int-8	fix-8	mix
DSPs	256	220	186	220	326	451	1865
BRAMs	132	164	152	101	174	230	412
Throughput	156 GOPS	198 GOPS	101 GOPS	471 GOPS	453 GOPS	2.37 TOPS	6.12 TOPS

of values in the input numbers to the sigmoid kernel. Referring back to Table 3.3, 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.

Compared to our previous work [50], the results reported in Table 3.4 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.

3.5.5 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 3.5 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.

3.5.6 Case Study: Interpreting the FA-LAMP Output

The Matrix Profile can be computed using existing methods in an offline context [126], whereas LAMP is used to predict it on streaming data [127]. 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 [102]:

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 $\mathbf{M_A(S)} > \mathbf{M_B(S)}$, we predict that behavior **A** is occurring; if $\mathbf{M_A(S)} < \mathbf{M_B(S)}$, 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 98.2% accuracy in the results of classification using FA-LAMP. Figure 3.15 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 3.15 shows a snippet of the first half.

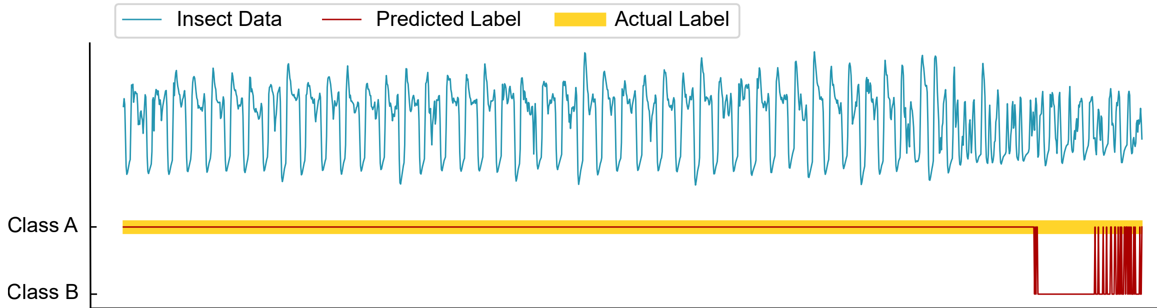

Figure 3.15: A snippet of insect EPG time series dataset along with the actual and predicted behavior (Class A: label height=1; Class B: label height=0).

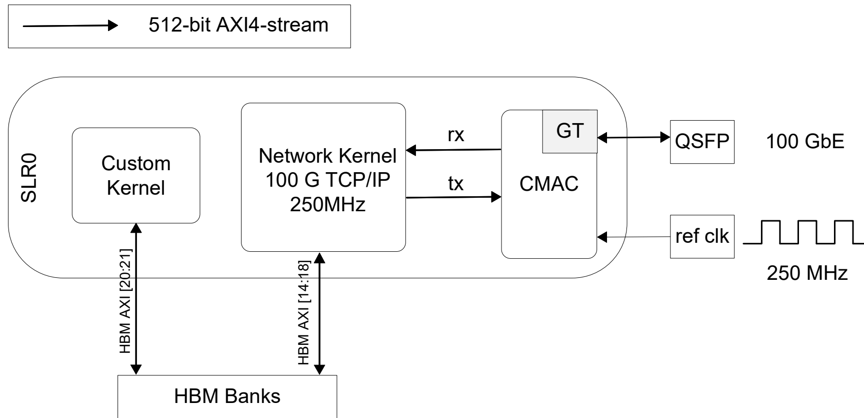

Figure 3.16: The SLR0 in the Alveo card configured with the Ethernet subsystem and the custom kernel IPs.

3.6 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.

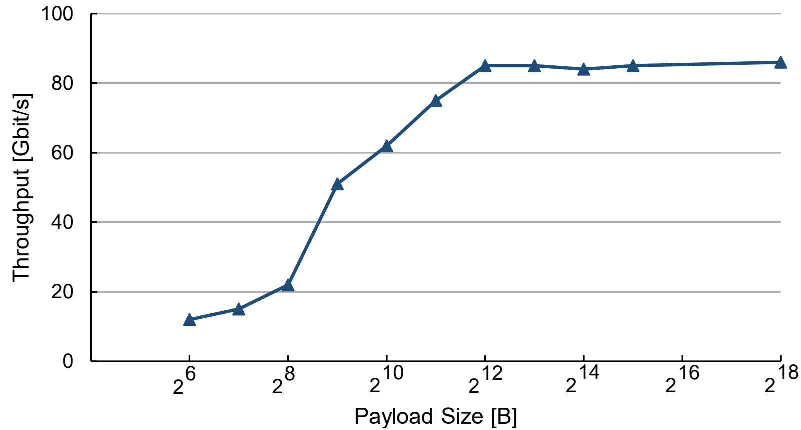

Figure 3.17: Ethernet module throughput on the Alveo card as a function of payload size.

We built our design on top of the Xilinx TCP stack IP repository [105], 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 3.16 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 3.3.6, 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 3.17 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 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.

3.7 Conclusion

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.

Chapter 4

Ensuring Consistency in Data Centers using RDMA-Enabled Consensus Protocols on FPGAs

4.1 Disclaimer

This chapter contains work which has not (as of the writing of this dissertation) been peer-reviewed. While we have made our best effort to present only facts in this chapter, the reader should assume that the claims presented in the following sections could be proven incorrect upon further experimentation.

4.2 Introduction

Data replication is a critical aspect of datacenter design and management as it ensures high availability, scalability, and fault tolerance of applications. Replication refers to the process of creating multiple copies of an application and distributing them across multiple servers or datacenters. This allows for load balancing and the ability to quickly recover from failures, ensuring uninterrupted service to end-users. In addition, replication can improve application performance by allowing for local data access and reducing network latency. Ensuring consistency in data replication is critical for maintaining data integrity and avoiding data loss or corruption. For example, in a distributed database where data is replicated across multiple nodes, consistency is essential to ensure that queries return accurate results, and updates to the data are propagated correctly to all nodes.

While strong consistency ensures that all nodes in a distributed system see the same data at the same time, it comes at the cost of increased latency and reduced availability. Relaxed consistency, on the other hand, allows for faster updates and improved scalability but may result in temporary inconsistencies in the data as it forgoes the total order of operations across replicas. To address this, Convergent and Commutative Replicated Data Types (CRDTs) have been introduced which formally define replicated data types that converge under relaxed consistency. However, not all operations can preserve convergence and integrity under relaxed consistency. To strike an appropriate balance between the two notions of consistency, several projects [41, 33, 10, 8, 9, 61] considered hybrid models where each operation is executed under either relaxed or strong consistency based on its semantics. All these projects utilize the traditional message-passing network model which

incurs intolerable response time for applications whose availability and latency is critical such as control and finance.

To tackle this issue, several works leveraged Remote Direct Memory Access (RDMA) network interfaces to implement consensus and broadcast protocols [3, 43, 76]. RDMA is a network protocol that works by bypassing the operating system and network stack to access memory directly, allowing for efficient data transfer between nodes in a server. The RDMA-based replication designs implemented on CPUs take the processing power away from the main task due to significant cost of coordination. Hence, the choice of a specialized networking hardware to implement consensus while boosting performance and reducing overall overhead seems attractive. Specifically, Field Programmable Gate Arrays (FPGAs) as they offer the opportunity of low energy consumption and do not suffer from some of the traditional limitations that conventional CPUs face in terms of data processing at line-rate.

In this work we propose an FPGA-based architecture that implements different coordination protocols based on operational semantics introduced in Hamband [43]. Given an object, Hamband divides the methods of the object class into three categories: reducible, irreducible conflict-free, and conflicting, and declares distinct coordination requirements for each. We build our design based on an already existing open source RDMA stack for FPGAs. We reinforce the RDMA stack with the missing RDMA semantics to implement the coordination protocols for the three categories of methods.

We implemented our design on Xilinx Alveo accelerator cards using High Level Synthesis (HLS) and Hardware Description Language (HDL) and executed our experiments on a state-of-the-art compute cluster. Our implementation showed significant improvement

in the throughput and response time compared to CPU-based designs. Furthermore, our experiments show that our design is highly scalable and performs with negligible throughput or latency loss in the case of a node failure.

In summary, this paper makes the following contributions:

- It introduces an FPGA architecture that is capable of implementing hybrid replicated data types based on one-sided RDMA operations. The design is highly pipelined and allows convenient integration with user-defined use-cases.
- It presents a method for memory allocation of replication logs that optimizes the throughput and response time using SMT solvers.
- It empirically shows that the design outperforms the throughput, response time, and power consumption of existing implementations by evaluating various scenarios (leader failure, follower failure, no failure) across a comprehensive CRDT use-cases and relational schema.

4.3 Background

4.3.1 RDMA and RoCE

RDMA is a direct memory access from the memory of one node into that of another without involving the processor. RDMA enables the communication using queue pairs: a set of a send queue, receive queue, and completion queue. Send and receive queues are always managed by the same queue pair. Read or write requests are generated by posting a work request to the receive or the send queue respectively. Once a request has been processed, a work completion is placed into the Completion Queue. Applications allocate

local memory for remote access by registering local virtual memory regions with the RDMA driver. Both memory regions and queue pairs have access modes (read-only, write-only, read/write) enabling remote nodes having different access rights for each other. These access permissions are specified when initializing memory regions and queue pairs and can be changed later. RDMA is available over network fabrics such as Infiniband or Ethernet (RoCE). RoCE allows Inifiniband packets to be transmitted as Ethernet frames.

RDMA provides two categories of communication primitives: (1) one-sided primitives i.e. READ, WRITE, and ATOMIC; and (2) two-sided primitives i.e. SEND and RECV. The former functions similarly to the conventional message-passing model where one node sends a message through a send operation, and the receiving node must perform an explicit receive operation to retrieve and handle the message. On the other hand, the latter operates similarly to the shared memory model, where a node can write or read directly to another node's memory without engaging its CPU. One-sided communication tends to have a shorter response time as it avoids involving the network and operating system stack.

4.3.2 Alveo Accelerator Cards

FPGAs are integrated circuits designed to be highly customizable, allowing users to configure the circuitry to perform specific functions or tasks. Unlike traditional Application-Specific Integrated Circuits (ASICs), which are hard-wired for a specific purpose during fabrication, FPGAs offer a flexible and programmable alternative. This makes them ideal for a range of applications, from digital signal processing and computer vision to artificial intelligence and machine learning. With FPGAs, users can rapidly prototype, test, and

deploy their designs, making them an increasingly popular choice for engineers and developers looking to accelerate the development of complex hardware solutions.

Recently, Xilinx introduced the larger capacity UltraScale+ Alveo FPGAs targeting data center and cloud-based applications. Alveo cards consist of a static region and a dynamic region. The static region enables communication with external PCIe host and provides basic clocking and reset for card operation. The accelerated kernels are placed in the dynamic region which consists of resources for implementing user functions (LUTs, DSPs, etc.). Some of the Alveo cards benefit from a High Bandwidth Memory (HBM) e.g. Alveo U280 incorporates two 4 GB HBM stacks. Each stack consists of 16 pseudo channels which can access the memory in parallel.

4.3.3 Coordination and Hamband

Coordination is a fundamental concept in distributed systems that ensures components work together effectively. Distributed systems require coordination mechanisms that allow different components to cooperate and communicate to achieve their objectives [20]. Coordination can involve ensuring that requests are executed in the same total order across replicas to provide strong consistency. This can be achieved through a range of mechanisms such as State Machine Replication (SMR) [85].

Hamband [43] introduces hybrid replicated data types for the RDMA network model. The proposed operational semantics divides methods of a given object into three categories: reducible, irreducible conflict-free, and conflicting. A method is reducible if it is conflict-free, dependence-free and summarizable. The `deposit` method for a bank account is an example, since two deposit calls can be summarized to a single deposit with an amount

equal to the sum of the two amounts without having any conflicts or dependencies with the other methods (e.g. withdraw). A method is irreducible conflict-free if it is conflict-free but either not summarizable or not dependence-free. As an example, in a grow-only set with methods contains and add (to add an element), the method add is irreducible conflict-free. Finally, the conflicting calls have a conflict with another method e.g. the method withdraw in the bank account example.

Hamband also declares distinct coordination requirements for each of the method categories. In particular, the reducible calls can be summarized at the issuing process and propagated to other processes by a single remote RDMA WRITE. Similar to reducible methods, the irreducible conflict-free methods can avoid synchronization. Each process replicates a buffer of calls for all the irreducible conflict-free calls of other processes. When a process p issues an irreducible conflict-free call, it remotely appends it to the buffers that each other process stores for p (via remote RDMA WRITEs). Finally, the other processes periodically traverse their buffer, locally apply the calls and then discard them. The conflicting methods are broken into *synchronization groups* (each connected component of the conflict graph is a group). Each synchronization group has a leader process that replicates the calls using a consensus protocol. Similar to irreducible conflict-free, each process replicates a buffer of calls that traverses periodically; however, each buffer corresponds to a *synchronization group* rather than a process.

In order to preserve the dependency between the calls, Hamband stores a mapping A for each method u that keeps track of the number of locally applied calls on u for a process p. If a call is sent to a remote buffer, it is accompanied by information about its

dependencies. The dependency map D for a call on a method u is determined by projecting the applied map A of the issuing process onto the methods that u depends on. To ensure that dependencies are respected, a process only applies a call while traversing a buffer if its local applied map A is point-wise greater than the dependency map D that comes with the call. When a call is applied, the local applied map A for the issuing process is updated.

4.4 Design Overview

4.4.1 RDMA Replicated Data Types

In this section we present a summary of the operational semantics of RDMA replicated data types introduced in Hamband [43]. The semantics divides methods into three categories, reducible, irreducible conflict-free, and conflicting, and presents dedicated coordination requirements for each.

Two methods u and u' are considered conflicting if there exist arguments v and v' that result in conflicting calls of $u(v)$ and $u'(v')$. If a method has no conflicting method, it is referred to as conflict-free, otherwise, it is considered conflicting. Also, a method is dependent on another if a call to the former relies on a call to the latter. The set of methods that a method depends on is denoted by $Dep(u)$. If a method has no dependencies, it is referred to as dependence-free.

Calls to conflicting methods need to maintain the same order across different processes. The relationship between conflicting methods forms an undirected graph called the conflict graph. The synchronization group $SyncGroup(u)$ of a method u is the connected

component of the method in the conflict graph. Synchronization groups consist of methods that synchronize with each other.

The summary of two calls c and c', written as $Summarize(c, c')$, is a call c'' iff for all states σ, $c \circ c'(\sigma) = c''(\sigma)$. For example, the summary of $deposit(3)$ and $deposit(4)$ is $deposit(7)$. A group of methods is called a summarization group if the calls made to them can be summarized into a single call. This means that a sequence of calls to the methods in the group can be summarized into one single call. A method is considered summarizable if it belongs to a summarization group, which is represented as $SumGroup(u)$. If a method is not a member of any summarization group, it is referred to as not summarizable, and we write it as $SumGroup(u) = \perp$. A method is considered reducible if it is conflict-free, dependence-free, and summarizable. If it does not meet these conditions, it is considered irreducible.

The semantics used in Hamband take advantage of the remote write feature of RDMAs to allow for direct communication between processes when updating state information. Each process maintains a local replica of the state to ensure fault tolerance and low latency for query methods, and only performs remote writes (not remote reads). The methods in the system are divided into three categories: conflicting methods, irreducible conflict-free methods, and reducible methods. Each category requires different coordination methods.

For conflicting methods, a leader process is assigned to each synchronization group and all processes replicate a buffer of calls for each group. The leader orders the calls and remotely appends them to the buffer of each process using a consensus protocol. Other processes periodically traverse their buffers and apply the calls locally.

Irreducible conflict-free methods do not require synchronization and each process autonomously issues and propagates them. Each process replicates a buffer of calls for all the irreducible conflict-free calls of other processes. When a process issues a call, it remotely appends it to the buffers that each other process stores for it. Other processes periodically traverse their buffers, apply the calls, and discard them.

Reducible methods are special because they can be reduced together locally and then remotely written for other processes. Each process replicates a single call for each summarization group of methods and a process, instead of a buffer of calls. This saves space and time because it eliminates buffer traversals by the target processes. Direct RDMA writes are used so that other processes receive updates without receiving and traversing messages. Each summarization call is written by only a single remote process and is only read by the local process.

4.4.2 FPGA Architecture

Figure 4.1 shows the high level FPGA architecture. Three main building blocks of our design are CMAC, Network Handler, and Coordination Engine. The CMAC kernel comprises the UltraScale+ Integrated 100 Gb/s Ethernet which 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

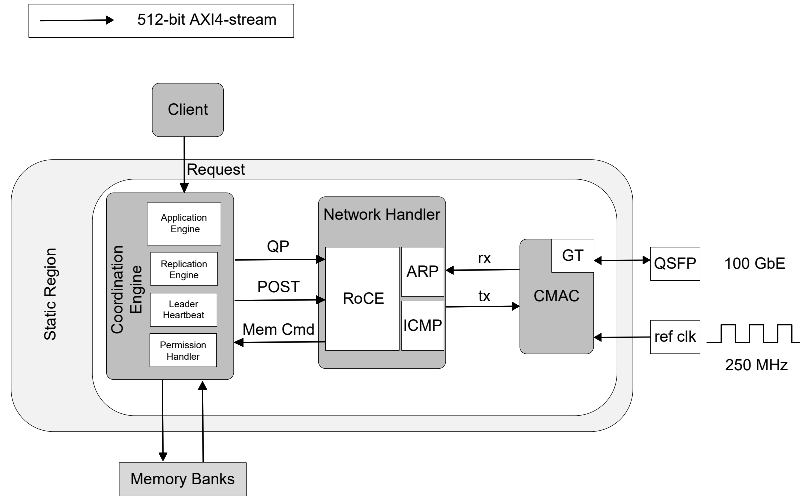

Figure 4.1: Proposed FPGA architecture.

clock. The Network Handler is a collection of HLS IPs that consist of ARP, ICMP, and more importantly RDMA stack (RoCE).

The Coordination Engine comprises several modules. Clients send their requests from host to the Application Engine. The Application Engine serializes the requests and forwards them to the Replication Engine. The Application Engine also executes the incoming method calls. The Replication Engine is responsible for generating RDMA WRITE commands based on the method call type. For the reducible and irreducible conflict-free methods, the Replication engine looks up the log address for each remote node, generates the request based on the address and the payload handed down by the Application Engine, and finally sends the request to the RDMA stack. The Replication Engine also runs a consensus protocol (based on Mu [3]) to serialize calls for the conflicting methods. The Leader Heartbeat periodically scans the leader's counter to detect a failure. In the case of leader failure, the Permission

Handler manages handing the write permission to the synchronization logs to the newly elected leader.

4.4.3 Log Allocation Optimizations

In order to increase the throughput of the system, we assign a thread to each replication log. These threads synchronously traverse the logs and process the method calls. Furthermore, based on the call frequency of methods we assign priorities to these log processing threads. A log with a higher priority will have more elements processed in a unit of time compared to a log with lower priority. However, choosing priorities for the replication logs is not straight forward.

Another design parameter that we have to choose is the type and amount of memory that we allocate for each log. Besides the on-chip BRAM memory, Alveo U280 card offers two off-chip memories: DDR and HBM. DDR and HBM are shared between the FPGA and the CPU. By default, we store the replication states that are frequently used in BRAM. This includes the queue pairs information, minimum proposal number, and the first undecided offset (used by Mu's replication algorithm). Naturally, we want to keep the replication logs as close as possible to the replication engine (on the FPGA); however, the BRAM is limited and we have to reuse the logs (circular logs) or offload some of them to the off-chip memory.

In order to find these parameters (log priority, memory type, and log size) we define a proper set of constraints and use SMT solvers to come up with optimal answer. We take into consideration the latency and size of each memory type, the latency to process a method call as well as the frequency and priority of each method call (defined by the client) to define these constraint. The objective is to maximize the throughput (defined as number of method

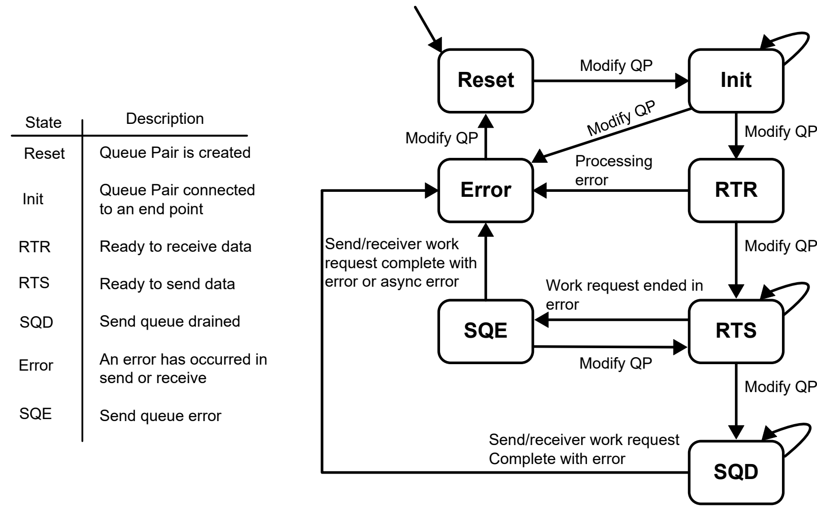

Figure 4.2: RDMA Queue Pair states and the possible transitions.

calls processed in a second) of the design. Given the constraints and the object, the SMT solver comes up with the optimal mentioned parameters.

4.5 Design Methodology

4.5.1 RoCE Stack

We implement the RDMA protocol on the FPGA by using StRoM [88] as our RDMA implementation core. StRoM is an open source, extensible RDMA stack for FPGAs that is available over RoCE and supports different RDMA semantics such as queue pair operations and one-sided READ and WRITE. The architecture of the RDMA stack consists of two data paths: one for incoming and one for outgoing packets. The following protocol headers are processed in both datapaths at each stage: IP, UDP, BTH (Base Transport

Header), RETH (RDMA Extended Transport Header) and AETH (ACK Extended Transport Header).

At each stage the current packet header is processed and the useful metadata is extracted and then the remaining packet is passed to the next stage. Packet processing is pipelined to achieve line-rate bandwidth. The final stage in the outgoing path processes the RETH and AETH headers and based on the RDMA op-code decides to whether issue DMA commands and requests to generate response packets. For the incoming path a request handler generates the DMA command before moving to the RETH/AETH processing stage. For data transfers, the DMA IP core is configured with two 32-byte streaming interfaces. One stream for writing data to memory and the other to retrieve data from memory.

Although StRoM provides an RDMA stack, it lacks three major capabilities that are needed for our replication engine: (1) Queue Pair states; (2) Access flags; and (3) Hybrid memory support. In the following, we describe how we integrate these two features to the already existing stack.

Queue Pair States

Each Queue Pair (QP) maintains several states during its lifetime. Figure 4.2 shows these states and the possible transitions from each one. Every QP start from the Reset states at which it is not associated with any endpoints and cannot send or receive data. In the Init state, the QP is associated with a remote endpoint and they exchange certain information such QP's attributes and its address. In the Ready to Receive (RTR) and Ready to Send (RTS) states a QP can receive or send and recieve data, respectively. The Send Queue Drained (SQD) state ensures that all the work requests are completed before the QP

can process any new incoming requests. A transition from a state to another happens when one calls the *modify QP* API except for the error states (Error and SQE) where transition happens automatically when an error occurs in the send or receive queues. Otherwise, the Queue Pair maintains its state.

To add support for the QP states, we store an array called State Table on the BRAM. Each row stores the state for a corresponding QP. We also add a function modify_state_table that exposes three AXI Stream buffers to interact with the State Table: One for updating a row, and two for querying the state of a QP and returning the response. The update requests to the State Table can originate from the host or within the RDMA stack. The request is checked to make sure the transition is valid and upon success the corresponding row is updated. The response buffer simply returns the state of a QP requested in the query buffer.

In the transmit path of the RDMA stack, right after receiving RDMA READ or WRITE requests, we check the QP state of the issuing request against the State Table. If the operation is not allowed, the request is silently dropped without returning any errors. We assume that the issuing function ensures the requests are valid; otherwise, there is no guarantee whether the request is processed. By doing so, we invalidate the request before it reaches to the packet generation pipeline.

In the receive path the destination QP is extracted after the IP and UPD headers are processed (in the BTH processing stage). Similar to the transmit path, we check the QP's state and drop the request if the QP is not in the valid state to send or receive data.

Access Flags

After the creation of a QP, access flags can be configured for that QP that indicate whether the local or remote node has read or write access to that QP. This is an important attribute for our replication engine, since each node only has write access to its own log on a remote node for non-conflicting operations. Furthermore, it ensures that each time only one leader has write access to the replicas for synchronizing conflicting methods. The access flags are stored in the State Table as well when a QP is created. We reinforce the already existing checks for the QP states with the access flag checks and drop the request if the check fails.

Hybrid memory support

The last modification we make in the RDMA stack is regarding the routing to different memory types. Since our design allows log allocation in DDR, HBM, or BRAM we need a mechanism to differentiate between these memories. In order to do so, we encode the memory type in the virtual address of the RDMA packet. Each entry in the Translation Lookaside Buffer (TLB) stores one 48 bit physical address corresponding to a 2 MB page which is contiguous in the physical address space. The TLB has 16,384 entries which allows the FPGA to address up to 32 GB of memory. This means that we only need 16 bits to index the TLB. The last two bits of the virtual address in our design correspond to the type of memory that the request must be routed to (00 for BRAM, 01 for DDR, and 10 for HBM).

During the AETH processing stage we check the 2 bits of the virtual address corresponding to the memory type. If it's a request for BRAM, we route the request to the AXI Stream FIFOs connected to the BRAM. Otherwise, we push the request to the FIFOs

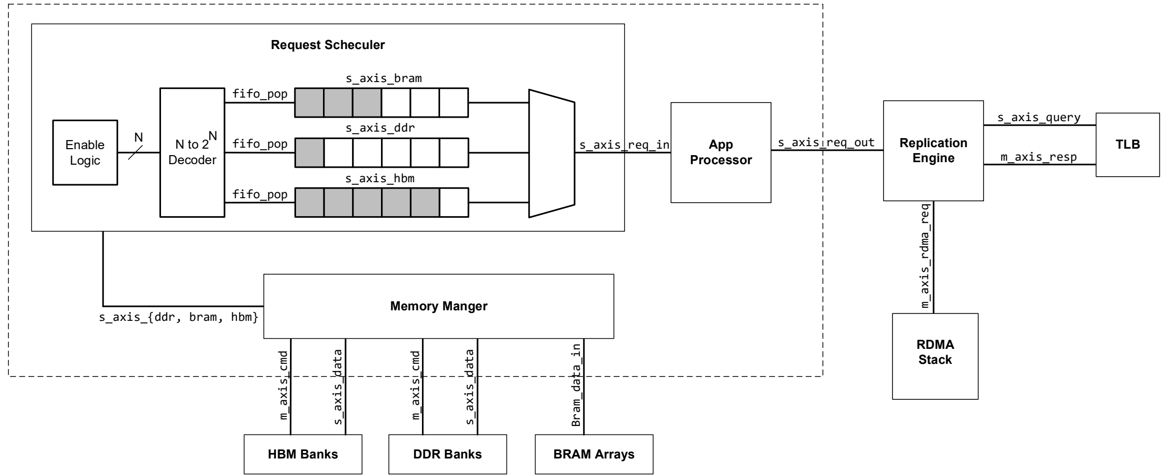

Figure 4.3: Application Engine comprises memory manager, request scheduler, and app processor units.

connected to the AXI DMA Controller. The memory command merger demultiplexes the request based on the 2-bit opcode and decides whether to push the request to DDR or HBM DMA controller.

4.5.2 Application Engine

We implement the client applications on the FPGA. An application is a class of objects that is replicated on the set of processes p. Any process p can accept and issue update calls u or query calls q. Clients can request method calls at every process, and the processes coordinate these calls.

```
void appliction_kernel (
    stream<memCmd>& hbmCmdOut,
    stream<memCmd>& ddrCmdOut,
    stream<ap_uint<512> >& ddrDataIn,
    stream<ap_uint<512> >& hbmDataIn,
    stream<ap_uint<512> >& requestDataOut,
    ap_uint<128> *bramDataIn);
```

Listing 4.1: Function interface of a kernel.

Listing 4.1 shows the kernel interface in C++, the stream type in Vivado HLS [106] maps to FIFOs in hardware. To access the HBM and DDR, the kernel can issue DMA commands consisting of adress and length over the hbmCmdOut and ddrCmdOut, data from the DMA engine is sent over the hbmDataIn and ddrDataIn, respectively. The interface also consist of a 128-bit bus to access the logs stored in BRAMs as well as a stream to send out the user requests.

Figure 4.3 shows the architecture of the application kernel (the dotted line square) and how it interfaces with the other modules. A memory manager module traverses all the logs in the system, this includes logs for irreducible conflict-free, and conflicting method calls in HBM, BRAM, and DDR. The memory manager accesses logs in the HBM and DDR through Xilinx AXI Direct Memory Access (DMA) IP; for the logs stored in the BRAM it simply stores a pointer to those logs. The memory manager keeps the head of each log and once a new request is pushed to a log, it inserts the request to the corresponding FIFO in the request scheduler (request scheduler keeps a separate FIFO for each log).

The request scheduler assigns quotas to the logs and processes them based on their quota. A log with a higher priority gets more quota or in the other words pops requests for processing more frequently. The enable logic is simply an N-bit counter that increments based on the quota assigned to the logs. For example if we have 32 logs present, the enable logic is a 5-bit counter passed to a decoder that enables pop from a FIFO at each time. If a FIFO has higher priority, a delay circuit in the counter forces it to stop counting based on the quota assigned to that log. In Section 4.5.5 we explain how we can use SMT solvers to find the optimum number of logs and assign priorities to them.

The request scheduler passes the requests in a single FIFO s_axis_req_in to the App Processor module. The App Processor is the actual module that implements the object and applies method calls to it. We only need to swap this module based on the user's use-cases. The App Processor is responsible for serializing and de-serializing the requests. We align the request payload to 512 bits so it matches the network interface bit-width of our FPGA design. Furthermore, the App Processor checks that the method calls are permissible. Finally, it returns a response to query methods and executes update methods.

Clients update requests are first routed to the App Processor as well. The App Processor checks that the request is permissible and if true, serializes the request and sends it to the replication engine through the m_axis_req_out. The replication engine is responsible for replicating the method calls by issuing RDMA WRITE requests. We will discuss this further in the next subsection.

4.5.3 Replication Engine

As mentioned in the previous subsection, the client requests are first arrived at the App Processor. When a client makes a call request, there are four potential outcomes depending on the method's category. First, if the call is a query, it is performed locally at the App Processor and the outcome is sent back to the client. Second, if the call is reducible, it is reduced in the local summary, and the result is sent to the remote summary locations. Third, if the call is irreducible conflict-free it is executed locally and written to the remote buffers F. Before propagation, the App Processor assigns a unique id to the call, pairs it with its dependency arrays and serializes it into a byte stream. Fourth, if the call is conflicting, the Replication Engine uses a consensus protocol to order the calls in L buffers. In the following we show how these methods are handled.

Query Methods

Figure 4.4 shows how the App Engine handles user's query methods in a system with three replicas (units involved are highlighted in red). The Request Scheduler continuously traverses all the logs in the system and sends them to the App Engine to execute them and update its state (σ). Upon receiving a query method, the App Engine applies the summarized calls ($S1$, $S2$, and $S3$ in the figure) to the stored state σ and returns the response.

Reducible Methods

A reducible method can be propagated by simply overwriting the remote summary location of each node. However, in order to generate an RDMA request, the Replication

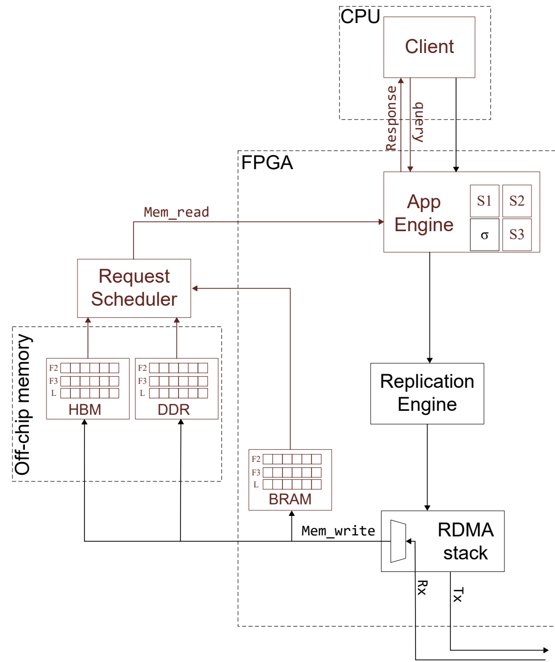

Figure 4.4: Control flow for executing query command.

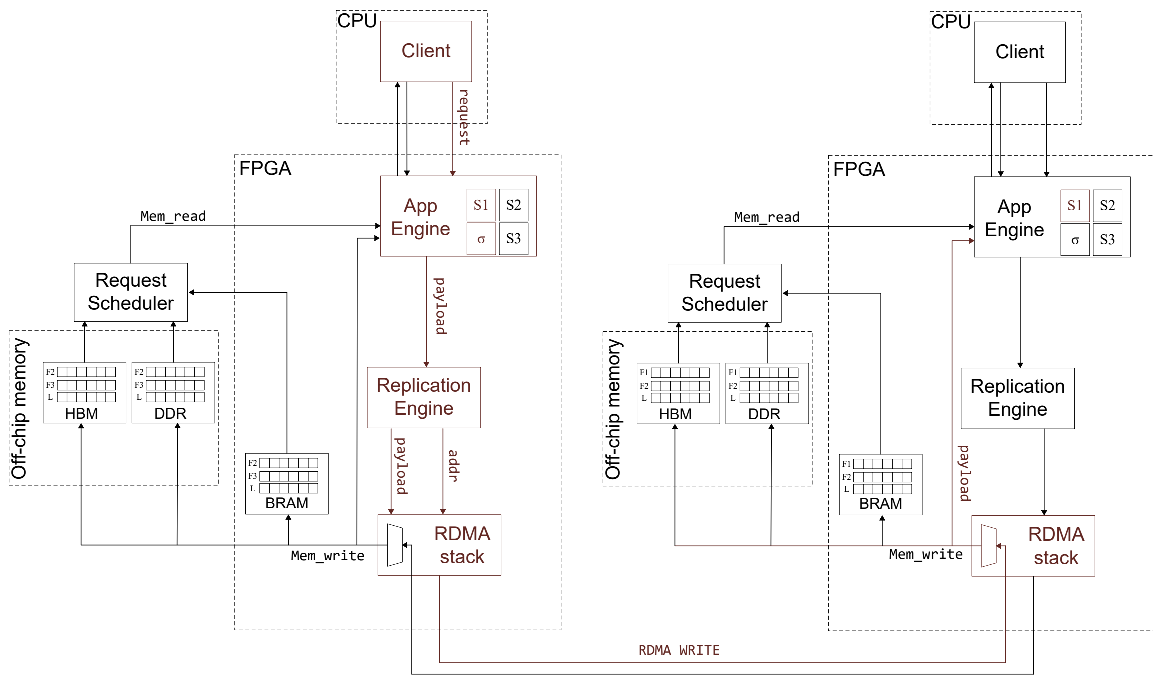

Figure 4.5: Control flow for executing reducible methods.

Engine needs to know the memory type and the memory address of each remote node. We store this information in a TLB that the Replication Engine can query by sending the node number and the method call as shown in Figure 4.3. For each remote node, the TLB stores a table that contains the corresponding memory type and address for a method call. The TLB is initialized once at the system startup based on the output of the SMT solver explained in Section 4.5.5. The memory type for reducible methods is always BRAM as they only occupy a single cell. Since BRAMs on the FPGA do not have an address, for each log stored in the BRAM, we assign an id to that log which is stored in the TLB.

Figure 4.5 shows the datapath for executing a reducible method call for a system with three nodes (only two nodes are shown in the figure for the sake of brevity). The client issues the call to the node shown on the left by sending the request to the APP Engine. The App Engine, after checking that the call is permissible, updates its summary cell ($S1$). After multiple calls are summarized, the App Engine sends the payload (summary) to the Replication Engine which in turn issues RDMA WRITES for the other two nodes by sending the payload and the address (memory address for $S1$ cell in each node). The receiving node (shown on the right in the figure), upon receiving the RDMA WRITE request for the reducible call, routes the payload to the corresponding summary cell ($S1$) and overwrites it.

Irreducible Conflict-Free Methods

Similar to reducible method calls, for irreducible conflict-free methods the Replication Engine queries the TLB to find the memory type and the log address for each remote node. After that it generates an RDMA WRITE command which is sent to the RDMA

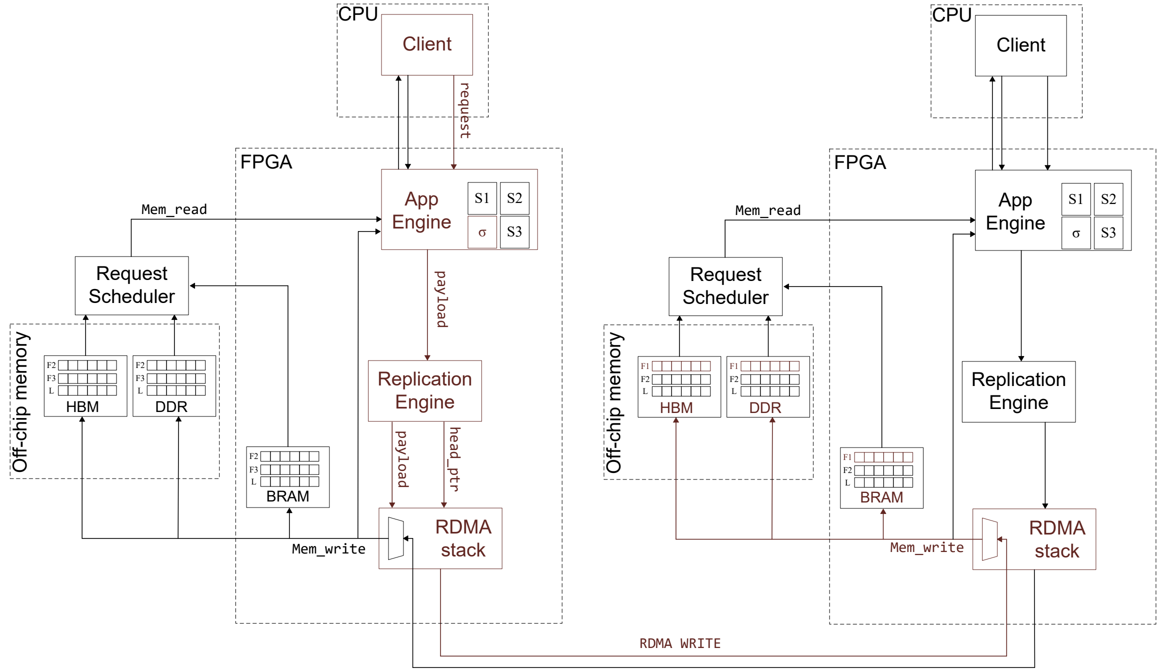

Figure 4.6: Control flow for executing irreducible conflict-free methods.

stack's pipeline to generate a packet and send it to the network. Figure 4.6 shows the call flow for executing irreducible conflict-free methods. The major difference compared to reducible methods is that Replication Engine, instead of passing a static address to the RDMA stack, sends the head pointer to the replication log ($F1$) and after each request increments the pointer. The RDMA stack in the receiving node demultiplexes the address and based on the memory type bits writes the payload to the replication log $F1$ in either BRAM, DDR, or HBM. Note that $F1$ only exists in one of these memories.

We use a similar approach to Hamband [43] to ensure the agreement property of reliable broadcast which states that if a message m is delivered by some correct node, then m is eventually delivered by every correct node. To ensure agreement, the source node keeps a shared memory location and allows other nodes to read it. The source node writes to

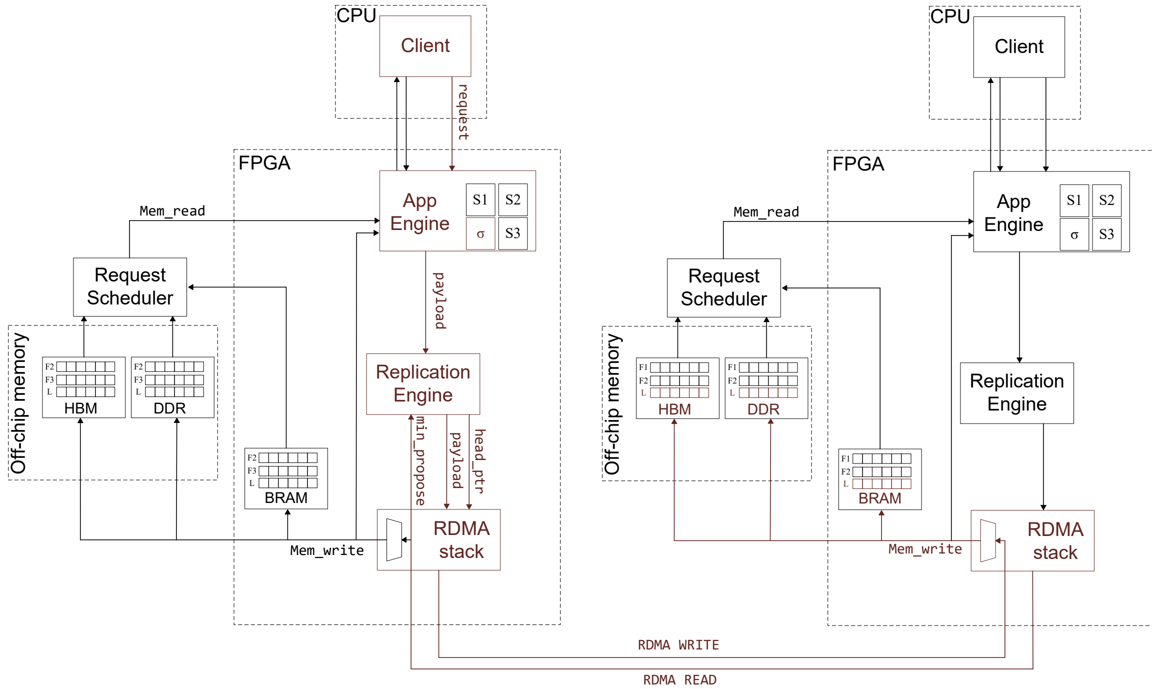

Figure 4.7: Control flow for executing conflicting methods.

the shared location before remotely writing the message for others and clears the location afterward. The shared location acts as a backup. Each node has a heartbeat thread that periodically updates a local counter, which is read by other nodes to check if that node is still alive. If other nodes detect that the source node has failed, they read the shared location remotely, retrieve any pending messages, and check if they have already received it. If not, they deliver the message.

Conflicting Methods

For the conflicting methods, the Replication Engine implements Mu's consensus protocol [3]. Mu is a State Machine Replication (SMR) implementation that replicates user

requests in memory and fail-overs the system to make apps fault-tolerant. In Mu, each node either takes the role of a leader or a follower. Leader is responsible for replicating the requests to the remote replicas. The followers are silent and communicate information passively by publishing it to their memory. Figure 4.7 shows how the conflicting methods are synchronized in our design. We encourage the readers to refer to Mu [3] for a complete description of the replication algorithm but in a nutshell, the leader carries out two phases: Prepare and Accept.

In the prepare phase, the Replication Engine in the leader node reads a *minimum proposal* number from each follower's log and picks a new proposal number higher than any *minimum proposal* number seen so far. After that, the Replication Engine writes its proposed number to all the followers and reads the head pointer's entry of the replication log (L) for each follower. If all the entries are empty, the leader enters the Accept phase and writes the payload to the entries and increments the head pointer; otherwise, it repeats the Prepare phase by picking a new proposal number. All the steps mentioned are carried out by RDMA READ and WRITES issued by the leader.

Leader Failover

As mentioned earlier, only one leader has write permission to the conflicting logs. In order to detect a leader failure, each node has a local heartbeat that is stored in a register and is incremented every 100 clock cycles. Replicas ensure the liveness of the leader by periodically sending RDMA READ commands to get the value of the heartbeat. These heartbeat scans are treated differently compared to the normal RDMA READs in the RDMA stack. We assign a reserved address (0xffffffff) for these packets. Whenever the leader sees a

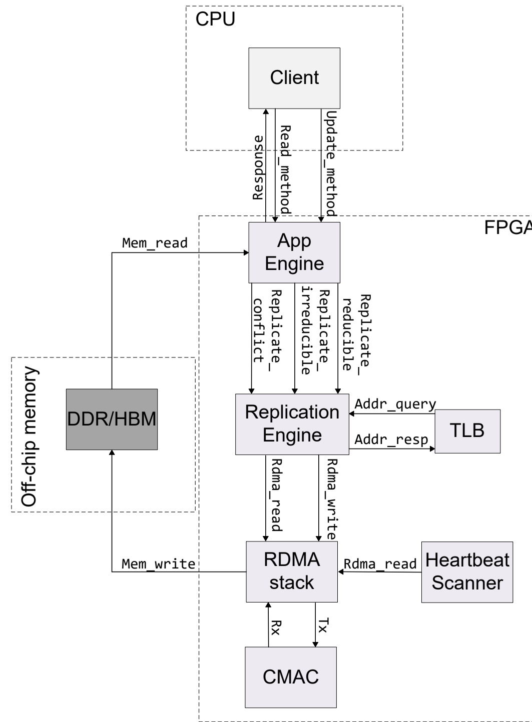

Figure 4.8: RDMA, memory, and consensus commands flow in the system.

packet with this address instead of generating a DMA command, it replies with the value of the heartbeat register. The followers upon receiving the response update their liveness score for the leader, if the score falls below a threshold the leader is considered failed. We use a similar approach to Mu where in case of leader failure where the next replica with the minimum index is considered the new leader. After being selected as the new leader, the replica sends a permission request to all the other nodes. The followers revoke the write permission of the old leader and grant access to the new one by updating the State Table.

4.5.4 Control Flow

Figure 4.8 shows a summary of interactions between the described modules and how the commands flow in our design between the client (host), FPGA, and the off-chip memories. The clients can send a query request from the CPU to the App Engine on the FPGA. Queries are processed locally and a response is returned to the client. If a client requests an update method, the App Engine first checks that the method is permissible and then serializes the payload and sends a request to the Replication Engine. Based on the method category, the App Engine places the payload in one of the FIFOs. App Engine also executes the requests from other replicas placed on the off-chip memories. The Replication Engine based on the type of the method either broadcasts the request or invokes Mu's consensus protocol to synchronize the request. Besides the payload coming from the App Engine, the Replication Engine needs the replica's log address to generate the RDMA WRITE request. It obtains the address by sending a query to TLB. During the system startup, the Replication Engine also sends RDMA READ requests to obtain the head pointer for the replica's logs. The Heartbeat Scanner periodically sends RDMA READ requests to read the remote replica's heartbeats and update their score. The RDMA stack generates the corresponding packets from the requests coming from the Replication Engine or the Heartbeat Scanner, and sends them to the networks by passing the payload to the CMAC kernel. Also, it receives the RDMA network packets from the CMAC and writes the requests to the memory logs.

4.5.5 Memory Allocation Optimizations

As mentioned in the previous subsection, the Request Scheduler assigns priorities to the logs and processes them based on that. Furthermore, as shown in Table 4.1 Alveo card has three different types of memory with different capacity, throughput, and latency. Ideally, we would want to allocate all the logs on the BRAM, since it has negligible latency and the highest throughput; however, the available on-chip BRAM memory is limited and we might be forced to allocate some of the logs on DDR or HBM. In the following we show how we can form constraints and objectives and use solvers to find the optimal memory location, allocated size, and priority for each log.

For the conflicting methods, we have P logs corresponding to P *synchronization groups*. For each *synchronization group*, we want a total order. Based on the frequency of calls, we might want to merge some of these logs. If we put two of them in the same log, there will be a total order for all methods in the log, which means a total order for each *synchronization group*. For the irreducible conflict-free methods, we have a log per process for all those methods. Thus, each process will have *N-1* logs. Based on the frequency of the logs, we may want to split these logs.

For maximum flexibility, we consider each *synchronization group* separately, and each irreducible conflict-free method separately. In the following, we call each such unit a *method*. *Methods* do not merge but split as much as possible. The size of a log is proportional to the frequency of the calls that it stores.

We have three different types of memory, let $m = 1, 2$, and 3. We define S_m to be the size of memory m, and rt_m to be the latency of memory m, f_i to be the frequency of

calls to each method i, u_i to be the priority of method i (a value between 0 and 1), and finally l_i to be the computation load to process a call to method i. We assume that f_i and u_i are provided by the client.

We can calculate $p_{i,m}$ the frequency of processing a method i on a memory m. In other words, the number of calls on the method i that the memory m can process in one second.

$$p_{i,m} = \frac{1}{l_i + rt_m} \tag{4.1}$$

The goal is to find the following variables: (1) $q_{i,m}$: The thread execution quota allocated for the log of method i in memory m. We take a second to be the total time and this is a fraction of a second. (2) $s_{i,m}$: The space allocated to method i in memory m. We form the following constraints: First, each method is hosted on only one memory.

$$\begin{aligned} q_{i,1} \neq 0 &\Rightarrow q_{i,2} = 0 \wedge q_{i,3} = 0 \\ q_{i,2} \neq 0 &\Rightarrow q_{i,1} = 0 \wedge q_{i,3} = 0 \\ q_{i,3} \neq 0 &\Rightarrow q_{i,1} = 0 \wedge q_{i,2} = 0 \end{aligned} \tag{4.2}$$

The same set of constraints also goes for $s_{i,m}$.

For each memory m, we need the sum of the quotas for all methods to be less than one second.

$$\sum_{i=1} q_{i,m} \leq 1 \tag{4.3}$$

For each memory m, we need the size of the logs in that memory to be less than the size of that memory.

$$\sum_{i=1} s_{i,m} \leq S_m \tag{4.4}$$

Table 4.1: Available memories on the Alveo U280.

	Capacity	Bandwidth	Latency
DDR	32 GB	77 GB/s	70 ns
HBM	8 GB	460 GB/s	100 ns
BRAM	43 MB	37 TB/s	1 Cycle

Next, the computation quota must be enough for the frequency of methods for that log. In one second, the number of processed calls must be more than the number of calls that appear.

$$q_{i,m} \times p_{i,m} \geq f_{i,m} \tag{4.5}$$

We need to prevent overflow in the logs during the time it is populated but not processed. Therefore,

$$s_{i,m} \geq (1 - q_{i,m}) \times f_{i,m} \tag{4.6}$$

Finally, we define our objective function to maximize the throughput (the number of processed calls per second), weighted over the priority of methods:

$$\sum_{m=1}^{3} \sum_{i=1} q_{i,m} \times p_{i,m} \times u_i \tag{4.7}$$

Solving these constraints will give us $q_{i,m}$ and $s_{i,m}$ for each *method* which tells us where and how much memory to allocate for the *method* and set the quota for it.

4.6 Experimental Setup and Results

4.6.1 Hypothesis

We aim to answer the following questions in our evaluation:

1. What is the latency and the throughput of our design? How does it change with different benchmarks and use cases? How does it perform compared to other CPU-based solutions?

2. What is the energy consumption of the design? Are there any benefits compared to a similar implementation on CPUs?

3. What is the impact of optimizations introduced in Section 4.5.5 on latency and throughput?

4. How does the design perform under failure?

4.6.2 Deployment Platforms and Toolchains

Our design is implemented on top of StRoM [88] using both RTL and HLS. The network stack and the log scheduler described in Section 4.5.5 are written in Verilog, the rest of the design is implemented using HLS C++. We used Xilinx Vitis 2022.1 toolchain to synthesize and generate the bitstream for our design. We used the Z3 SMT solver [21] to solve the constraints described in Section 4.5.5. The code was written in z3 C++ bindings and compiled using MSVC++. The host on the FPGA is solely responsible for initializations and starting the kernels. These initializations include establishing Queue Pairs and exchanging the Queue Pair ID and memory regions information. Usually, nodes establish a separate

TCP connection to exchange these information; however, we hard coded these in the host code, as it does not affect the performance in any way. The host code was written in C++ as well.

We ran our FPGA-based experiments on ETH Heterogeneous Accelerated Compute Cluster (ETHZ-HACC). ETHZ-HACC is equipped with Xilinx hardware and software technologies for adaptive compute acceleration research. Table 4.2 details the hardware configuration of the cluster. Currently, there are four Alveo U280 and six Alveo U250 accelerator cards available on the cluster. Table 4.3 shows the hardware specifications of these two cards. Alveo U280 supports High Bandwidth Memory (HBM) unlike Alveo U250. Alveo U250 on the other hand offers slightly more resources and a higher peak throughput. Both cards have the same networking infrastructure. In our experiments we increase the number of nodes from 3 to 7. For experiments running on 3 or 4 nodes we only use Alveo U280 and for experiments on 5 nodes or more we use the four Alveo U280 cards plus the remaining Alveo U250 FPGAs.

We compare our work with message-passing CRDTs and RDMA SMRs (i.e. Mu [3] and Hamband [43]). The experiments for these CPU-based replication designs were conducted on a cluster consisting of 7 nodes, with each node featuring 8 AMD Opteron 6376 cores and 50 GB of memory. The nodes were interconnected via a 40 Gbps Infiniband network and ran CentOS 7.4 Linux x86_64 kernel version 3.10. Furthermore, all the programs used in the experiments were compiled using gcc-7.4.0.

4.6.3 Measurements

We report the throughput, response time and the energy consumption of our design by direct execution on the ETHZ-HACC platform the benchmarks summarized in the next subsection. To calculate the throughput, we divided the overall number of update calls by the duration it takes for these calls to be replicated across all nodes. To determine the response time, we compute the average response time of all the calls. We conducted each experiment three times and report the average results.

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, which we omit from our estimation. We estimated the power consumption of the CPU using RAPL [80].

Unless otherwise specified, all experiments are conducted using 4 million operations. Method calls are generated randomly, and update calls are evenly distributed among the updated methods. If there are conflicting method calls, they are automatically redirected to the leader node(s) responsible for handling them. All other types of calls, such as conflict-free and query calls, are evenly distributed among the nodes.

4.6.4 Benchmarks

We experimented with and utilized five different CRDTs [86]: Counter, Last-writer-wins register (LWW), Grow-only set (GSet), Observed-Remove Set (ORSet), and Shopping Cart. In addition, we also employed three relational schemata [41, 73]: project management, courseware, and movie. The project management class encompasses five methods, namely,

Table 4.2: Hardware details of ETHZ-HACC.

CPU	8× Intel Xeon W-2200 @ 3.2GHz
Memory	128 GB
Switch	100 GbE Cisco Nexus 9336c FX2
OS	Ubuntu 20.04 LTS
Kernel	5.4.0-144-generic
FPGA Cards	Alveo U250, Alveo U280

addProject, deleteProject, worksOn, addEmployee, and query. The addProject, deleteProject, and worksOn methods are part of the same *synchronization group*, and worksOn relies on addProject and addEmployee due to the foreign-key constraint. The movie class comprises four methods, namely, addCustomer,deleteCustomer, addMovie, and deleteMovie, which operate on two separate relations, thus forming two synchronization groups. There is no dependency present in this class. The Courseware class consists of five methods, including addCourse, deleteCourse, enroll, registerStudent, and query. Conflict analysis revealed a single synchronization group, which comprises addCourse, deleteCourse, and enroll. The enroll method is dependent on both addCourse and registerStudent methods.

4.6.5 Results

In this Section we present the throughput and response time results for our implementation and compare them with message-passing CRDTs, and two RDMA-enabled

Table 4.3: Alveo accelerator cards specifications.

	Alveo U280	Alveo U250
INT8 Peak Throughput	24.5 TOPS	33.3 TOPS
HBM2 Capacity	8 GB	N/A
HBM2 Bandwidth	460 GB/s	N/A
DDR Capacity	32 GB	64 GB
DDR Bandwidth	77 GB/s	77 GB/s
Look-Up Tables	1,304k	1,341k
DSP Slices	9,024	12,288
Internal SRAM	43 MB	57 MB
Network Interface	2x QSFP28 (100GbE)	2x QSFP28

replication protocols, namely Hamband [43] and Mu [3]. We show how a node failure (leader or follower) impacts the performance. Furthermore, we show how solving the log allocation optimizations discussed in Section 4.5.5 provides us with a memory layout that results in optimal throughput and response time. Finally, we compare the energy consumption of our design with Hamband and Mu.

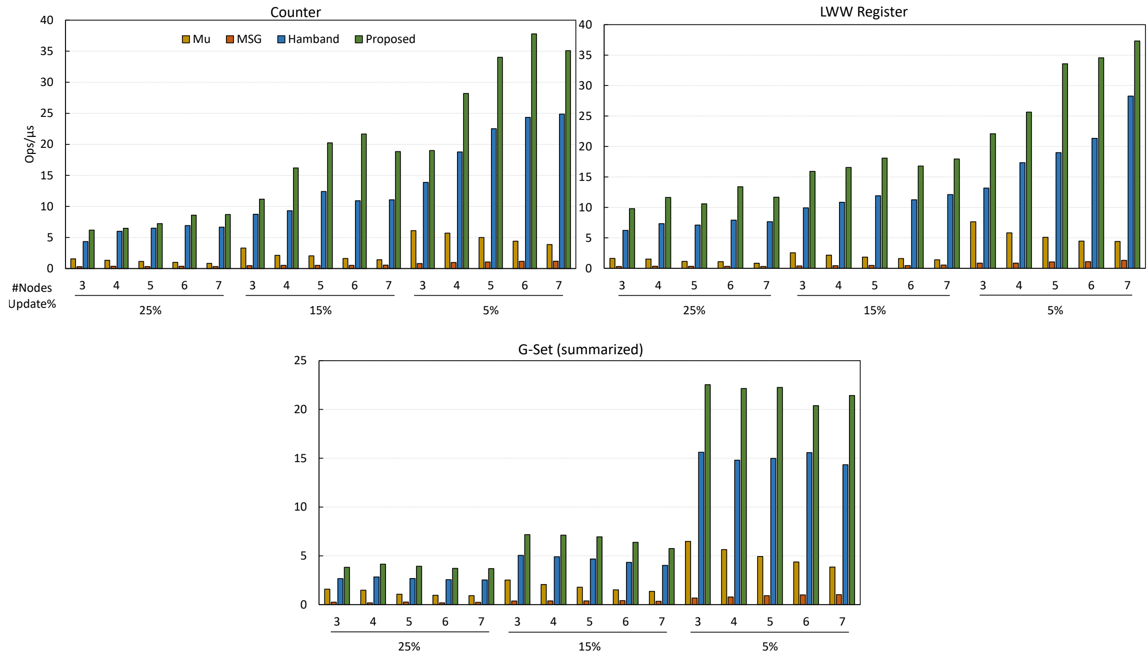

Figure 4.9: Comparison of throughput for reducible methods.

Throughput and Latency

We compare the throughput and response time for four different mix of categories: (1) use-cases containing only reducible methods, (2) use-cases with irreducible conflict-free methods, (3) Movie benchmark whose methods form two distinct synchronization groups, and (4) the project management benchmark which has methods in all three categories. Our experiments investigate the performance for 5%, 15%, and 25% update call ratios and various number of nodes (from 3 to 7).

Figure 4.9 shows the throughput for three reducible methods: Counter, Last Writer Win Register, and Grow Only Set (G-Set). These methods are summarized locally on each node and then propagated using a single RDMA WRITE. As it can be inferred from the

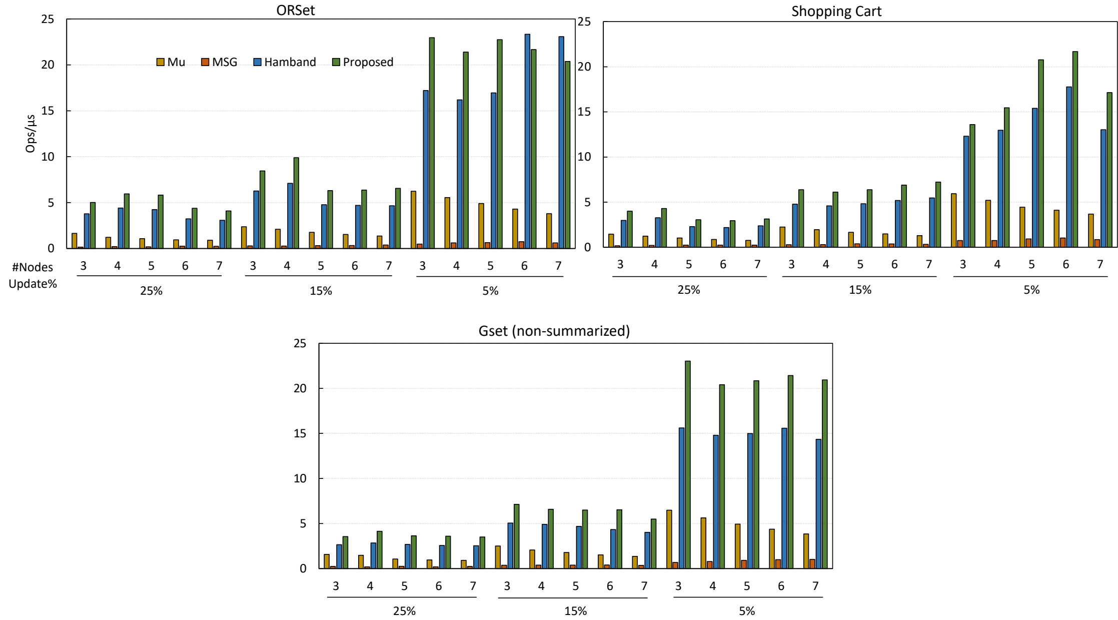

Figure 4.10: Comparison of throughput for irreducible conflict-free methods.

figure, our design is highly salable. As the number of nodes or the update increases, our design shows an increasing trend. Our design outperforms Mu, message-passing (MSG), and Hamband's throughput by 6.62, 27.67, and 1.48×, and improve their response time by 2.14, 32.43, and 1.49×, respectively. Storing the summary cell locally on the BRAM, allows the RDMA stack to access the payload with negligible latency and hence improving the throughput.

Figures 4.10 and 4.11 compares the throughput and latency of irreducible conflict-free methods, namely, ORSet, Shopping Cart, and a variation of G-Set that uses buffers instead of summaries. These methods are not reducible because they are either not summarizeable or have dependencies. Our implementation shows 3.95, 22.28, and 1.33× improvement in the throughput, and 37.93, 2.04, and 1.54× reduction in the response time compared to

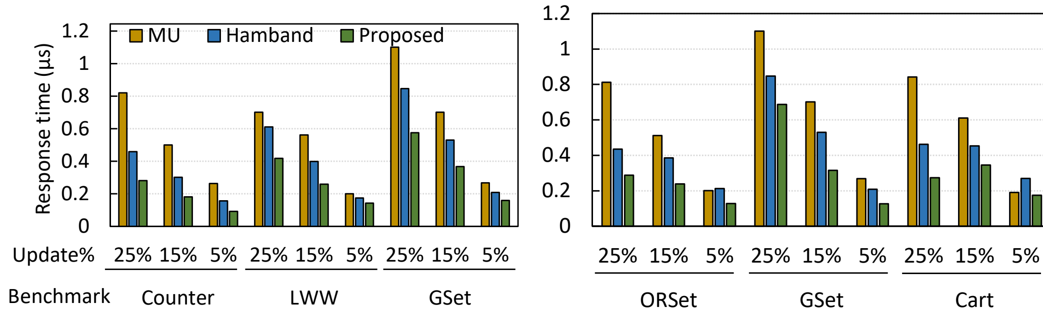

Figure 4.11: Comparison of response time for reducible and irreducible conflict-free methods.

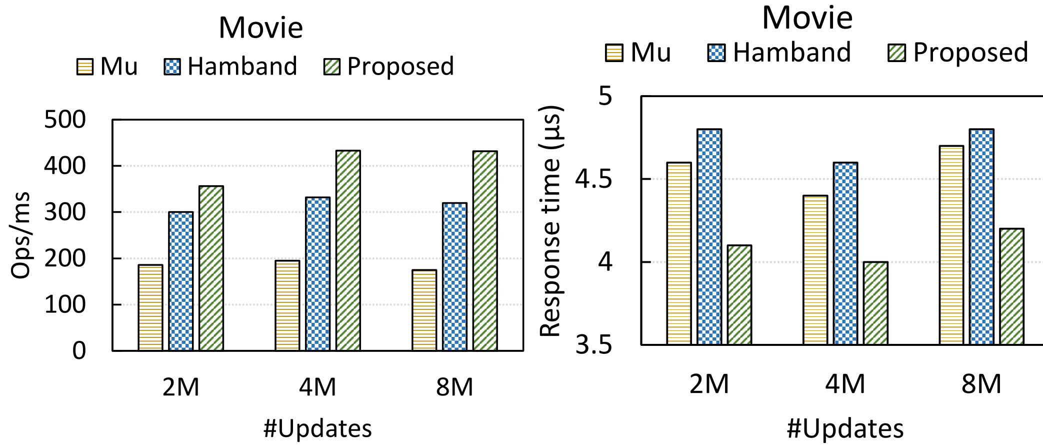

Figure 4.12: Comparison of throughput and response time for Movie benchmark.

Mu, MSG, and Hamband, respectively. Similar to Hamband, our implementation avoids synchronization for irreducible conflict-free methods which leads to higher throughput and lower response time compared to Mu.

Figure 4.12 shows throughput and latency of the Movie use-case for 2, 4, and 8M update operations. This benchmark only contains conflicting call which require synchronization. Our design improves the throughput by 1.28 and 2.20× compared to Hamband and Mu and decreases the response time by 1.15, and 1.11×, respectively. While our implementation runs a similar consensus protocol to Mu, by storing the replication state locally on the FPGA and

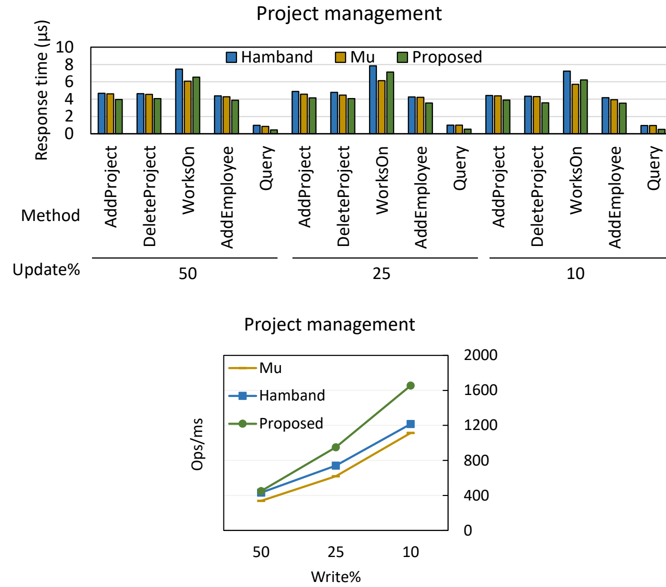

Figure 4.13: Project management throughput and response time.

as close as possible to the replication engine and the RDMA stack, our design outperforms Mu in terms of throughput and latency.

Finally, figure 4.13 shows the throughput and response time (per method) for the project management benchmark for different update call ratios. This benchmark contains methods in all three categories. Our implementation improves the throughput by 1.22 and 1.45× compared to Hamband and Mu. In terms of response time, our design cuts the query latency almost in half as it is as cheap as a signle access to local BRAM memories. The response time for *WorksOn* calls is higher compared to Mu since they are dependent on *addProject* and *addEmployee* calls and have to wait for them to be delivered.

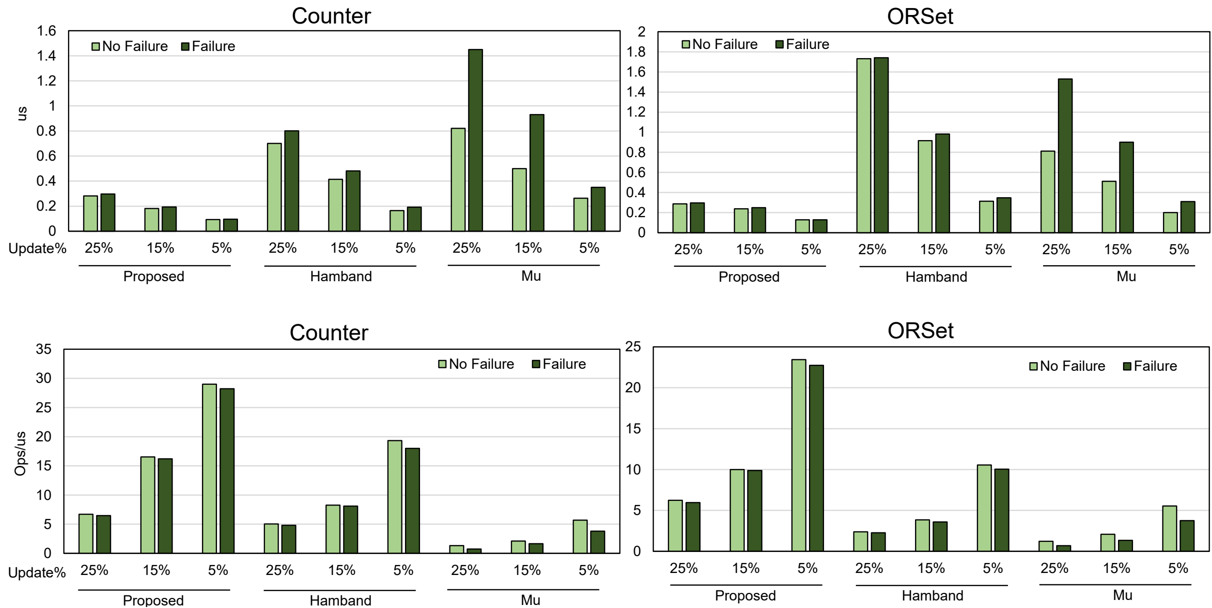

Figure 4.14: Effect of failure on the Counter and ORSet use-cases.

Fault Tolerance

In this section we investigate the effect of node failure on throughput and response time. Node failure is simulated by suspending the node's heartbeat, which causes other nodes to suspect it. After a failure, all requests to the failed node are redirected to the next available node, and in the case of leader failure, conflicting calls must wait for the leader-change protocol to elect a new leader and hand down the write permissions to the new leader. The experiments in this section are executed on 4 nodes with 4M operations. We analyze the node failure effect on two CRDTs which contain no conflicting methods and use either reliable broadcast protocol or single RDMA writes. Furthermore, we present the results for the Courseware use-case which contains methods in all three categories.

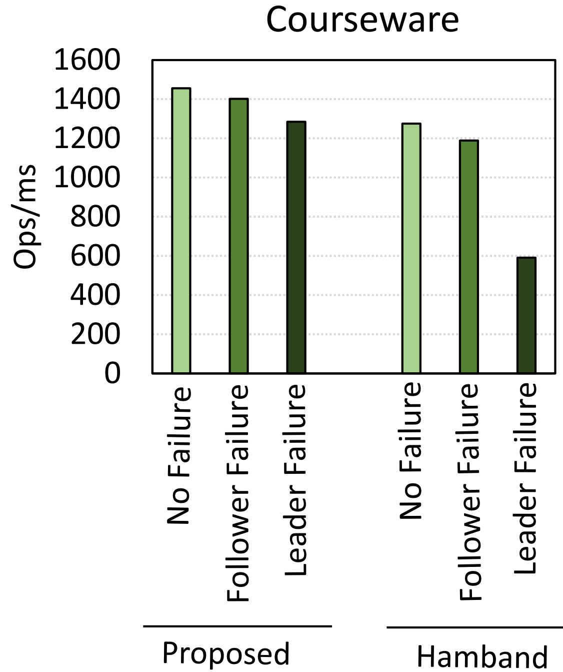

Figure 4.15: The effect of failure on the courseware throughput.

Figure 4.14 shows the throughput and latency of Counter and ORSet benchmarks with and without failure for Mu, Hamband, and our proposed design. In the failure scenario, *Counter* and ORSet show only 2.5% and 3% decrease in the throughput and 5.3% and 2.6% increase in the response time compared to the no failure scenario. The results indicate that our implementation can withstand failures smoothly for conflict-free use-cases, with a small decrease in throughput and a moderate increase in response time. Hamband, on the other hand, suffers from a 5% decrease in the throughput for both use-cases and 15% and 6% increase in the response time for the counter and ORSet, respectively. The leader failure detection thread might get de-scheduled by the OS in Hamband which tampers the performance; however, in our design we offload the thread to the FPGA and ensure that it is executed periodically. Mu's throughput is decreased by 72% and its response time is

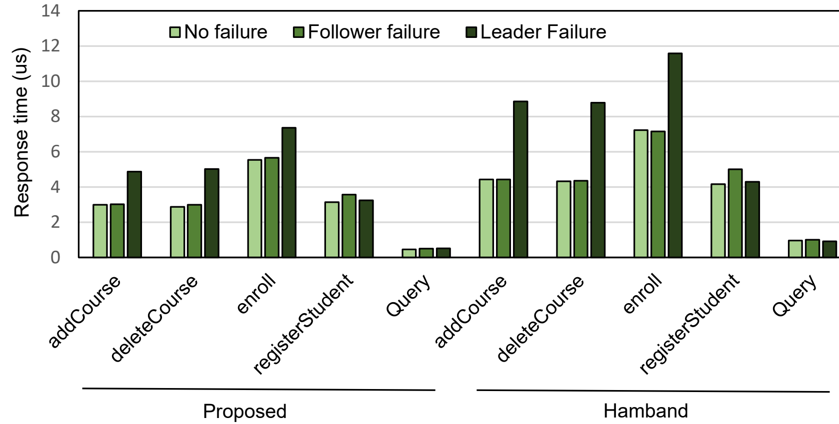

Figure 4.16: The effect of failure on the courseware response time per method.

increased by 34% in case of the leader failure. The reason behind this huge performance decrease is the fact that Mu synchronizes the calls for these two benchmarks; although, they can be executed without synchronization and the leader failure causes a massive delay in the system.

Figure 4.15 shows the throughput and Figure 4.16 shows the latency of Courseware use-case for three different scenarios: no failure, follower failure, and leader failure. The follower failure impacts the throughput by 3.7% and the leader failure decreases the throughput by 11.7%. Hamband suffers up to 53% reduction in the throughput in the case of leader failure [43]. By offloading the heartbeat scan thread to the FPGA and storing all the Queue Pair metadata locally on the FPGA, our implementation mitigates the throughput loss drastically. The *registerStudent* method, which is conflict-free, shows little to no change in response time even in the event of leader failure. This is because this method does not require synchronization by the leader, and thus can be easily redirected to follower nodes without much impact on its response time. However, the response time of conflicting methods, such

as *addCourse*, *deleteCourse*, and *enroll*, suffers from a 35% increase. This is because these methods need to wait for the leader-change protocol to elect a new leader, resulting in a significant delay in their response time.

Log Allocation Optimization

In order to investigate the effect of log allocation optimizations discussed in Section 4.5.5, we take the project management benchmark, assign call frequency and priority to each method and feed it to the SMT solver. The SMT solver gives us the amount of memory for each method log, which memory to allocate it in, and the processing quota for each log. The project management class encompasses five methods, namely, addProject, deleteProject, worksOn, addEmployee, and query. The addProject, deleteProject, and worksOn methods are part of the same synchronization group and will be allocated a single memory log. We assume that these three methods are called with a frequency of 300K calls per second, the addEmployee with a frequency of 100K and the query method with a frequency of 600K calls per second. We assign priority values of 0.6, 0.3, and 0.1 to addEmployee, methods in the synchronization group, and the query.

Figure 4.17 shows the throughput and response time for different allocation scenarios. Each data point in the plot is represented as a 6-tuple (s_{BRAM}, s_{DDR}, s_{HBM}, q_{query}, $q_{addEmployee}$, q_{synch_group}). The s_{mem} shows the amount of logs allocated in that memory type in Megabytes, and q_{method} represents the processing quota allocated to that method log. The dark green diamond on the far right side of the plot is the memory allocation based on the SMT solver's output and the rest of the data points are randomly generated.

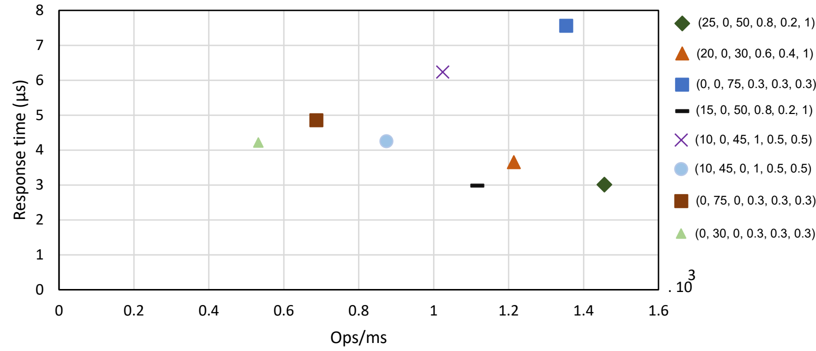

Figure 4.17: Effect of log allocation optimization.

The SMT solver assigns query and addEmployee calls to the BRAM with processing quota of 0.8 and 0.2, and allocates the method calls in the synchronization group in HBM. This is consistent with the frequency call and the priority assigned to the calls. Query has the highest frequency call so it would benefit from being as close as possible to the Application Engine. Next, the solver has to decide whether to put addEmployee or the three conflicting methods in the BRAM, since all of them would not fit. It seems like based on the priority of addEmployee, the solver allocates it in the BRAM and assigns the conflicting methods to HBM. The solver's output has the highest throughput (1.456 Ops/ms) and lowest response time(3 /µs) compared to other configurations.

The data point (15, 0, 50, 0, 0.8, 0.2, 1) shows the same allocation as the sovler's output but with less memory allocated in the BRAM for the replication log. This data point has the same response time as the solver's output; however, the throughput in this allocation scheme is 26% less than the optimum allocation. We also tried increasing the log size in the HBM and did not notice any improvements in the performance which shows that the solver's output is the optimal value.

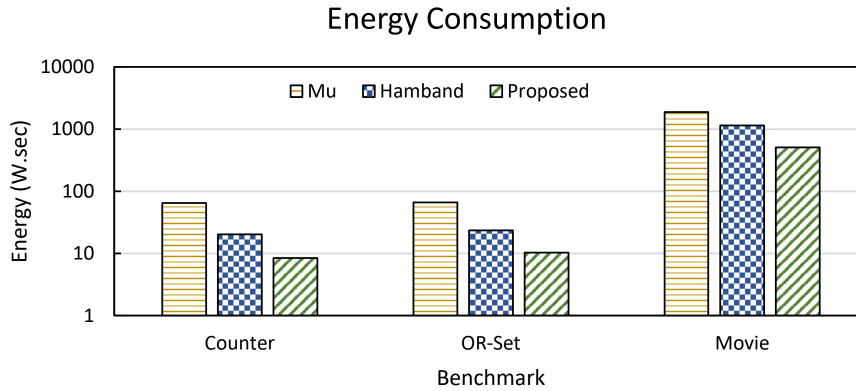

Figure 4.18: Comparison of energy consumption for Mu, Hamband, and our proposed work. (note the logarithmic scale on the vertical axis).

Energy Consumption

Figure 4.18 compares the energy consumption in units of Watt-seconds (joule) for Mu, Hamband, and our FPGA implementation. The total energy was calculated by integrating the power as a function of time that each hardware platform consumes for 4M operations. We run the experiments for 4 nodes and report the energy consumption of a single node. As shown in Figure 4.18, the energy gap between the FPGA and Mu/Hamband increases drastically with the complexity of the benchmark (note the logarithmic scale on the vertical axis). On average, Hamband and Mu consume 2.31 and 5.94 times more energy compared to our FPGA implementation.

Chapter 5

Conclusion

5.1 FA-LAMP Concluding Remarks

Chapter three 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.

5.2 FPGA-Based Consensus Concluding Remarks

Chapter four proposed an FPGA architecture which is capable of replicating user requests with low latency and fail-overing the system within microseconds. We implemented our design on Alveo U280 and Alveo U250 FPGAs and executed experiments in various scenarios (no failure, leader failure, follower failure).

Our design improved the throughput by 1.48× and 6.62× compared to Hamband and Mu for CRDTs with reducible methods and reduced the response time by 2.14× and 1.49×, respectively. For a use-case containing method calls in all three categories of semantics (reducible, irreducible conflict-free, and conflicting), our design improved the throughput by 1.22× and 1.45× compared to Hamband and Mu.

5.3 Future Work for FA-LAMP

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.

5.4 Future Work for FPGA-based Consensus

The current existing RDMA stack (based on StRoM) lacks several RDMA semantics such as Completion Queue, Memory Regions, and Queue Pair configurations such as timeout and retry count. We would like to extend the network stack with these capabilities to have a complete RDMA system that closely corresponds to the software RDMA library (*libibverb)*. We would also like to test our design with larger use-cases which represent real world applications such as TPC-C and TPC-E. There is also significant opportunity to explore and optimize the Mu's consensus protocol implementation on the FPGA or explore other alternatives for synchronizing conflicting method calls.

Bibliography

[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. 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*, pages 411–418. IEEE Computer Society, 2018.

[2] Alireza Abdoli, Sara Alaee, Shima Imani, Amy Murillo, Alec Gerry, Leslie Hickle, and Eamonn Keogh. 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)*, page 3328–3336. ACM, 2020.

[3] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[4] Richard Allen, Holly Brown, Margaret Hellweg, Oleg Khainovski, Peter Lombard, and Doug Neuhauser. Real-time earthquake detection and hazard assessment by ElarmS across California. *Geophysical Research Letters - GEOPHYS RES LETT*, 36, 2009.

[5] Gustavo Alonso, Carsten Binnig, Ippokratis Pandis, Kenneth Salem, Jan Skrzypczak, Ryan Stutsman, Lasse Thostrup, Tianzheng Wang, Zeke Wang, and Tobias Ziegler. Dpi: The data processing interface for modern networks. In *9th Biennial Conference on Innovative Data Systems Research*, 2019.

[6] Kota Ando, Kodai Ueyoshi, Yuka Oba, Kazutoshi Hirose, Ryota Uematsu, Takumi Kudo, Masayuki Ikebe, Tetsuya Asai, Shinya Takamaeda-Yamazaki, and Masato Motomura. Dither NN: an accurate neural network with dithering for low bit-precision hardware. In *International Conference on Field-Programmable Technology, FPT 2018, Naha, Okinawa, Japan, December 10-14, 2018*, pages 6–13. IEEE, 2018.

[7] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. An OpenCL™ deep learning accelerator on Arria 10. In *Proceedings of the 2017*

ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), page 55–64. ACM, 2017.

[8] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno M. Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In Laurent Réveillère, Tim Harris, and Maurice Herlihy, editors, *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 6:1–6:16. ACM, 2015.

[9] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Towards fast invariant preservation in geo-replicated systems. *SIGOPS Oper. Syst. Rev.*, 49(1):121–125, jan 2015.

[10] Valter Balegas, Nuno M. Preguiça, Sérgio Duarte, Carla Ferreira, and Rodrigo Rodrigues. IPA: invariant-preserving applications for weakly-consistent replicated databases. *CoRR*, abs/1802.08474, 2018.

[11] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

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

[13] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. Prism: Rethinking the rdma interface for distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 228–242, New York, NY, USA, 2021. Association for Computing Machinery.

[14] 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. TVM: an automated end-to-end optimizing compiler for deep learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association, 2018.

[15] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 73–82. ACM, 2019.

[16] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Using dataflow to optimize energy efficiency of deep neural network accelerators. *IEEE Micro*, 37(3):12–21, 2017.

[17] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko,

Reto Achermann, Gustavo Alonso, and Timothy Roscoe. Enzian: An open, general, cpu/fpga platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 434–451, New York, NY, USA, 2022. Association for Computing Machinery.

[18] Philip Colangelo, Nasibeh Nasiri, Eriko Nurvitadhi, Asit Mishra, Martin Margala, and Kevin Nealis. Exploration of low numeric precision deep learning inference using Intel FPGAs. In *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 73–80, 2018.

[19] Jason Cong, Hui Huang, and Xin Yuan. Technology mapping and architecture evalution for k/m-macrocell-based fpgas. *ACM Trans. Design Autom. Electr. Syst.*, 10(1):3–23, 2005.

[20] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems - concepts and design (2. ed.)*. International computer science series. Addison-Wesley, 1994.

[21] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[22] Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. Ecros: Building global scale systems from sequential code. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[23] Sergio Decherchi, Paolo Gastaldo, Alessio Leoncini, and Rodolfo Zunino. Efficient digital implementation of extreme learning machines for classification. *IEEE Trans. Circuits Syst. II Express Briefs*, 59-II(8):496–500, 2012.

[24] G. Deng, H. Zhou, G. Yu, Z. Yan, Y. Hu, and X. Xu. Scalable and parameterized dynamic time warping architecture for efficient vehicle re-identification. In *4th International Conference on Transportation Information and Safety (ICTIS)*, pages 48–53, 2017.

[25] ETH. Ethz heterogenous accelerated compute cluster. `https://github.com/fpgasystems/hacc`, 2023.

[26] 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. hls4ml:

An open-source codesign workflow to empower scientific low-power machine learning devices, 2021.

[27] Daniel Falbel. Keras, 2015.

[28] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association.

[29] 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. A configurable cloud-scale DNN processor for real-time AI. In Murali Annavaram, Timothy Mark Pinkston, and Babak Falsafi, editors, *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pages 1–14. IEEE Computer Society, 2018.

[30] Yao Fu. Deep learning with INT8 optimization on Xilinx devices, 2017.

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

[32] Vasilis Gavrielatos, Antonios Katsarakis, Vijay Nagarajan, Boris Grot, and Arpit Joshi. Kite: Efficient and available release consistency for the datacenter. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–16, 2020.

[33] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 371–384. ACM, 2016.

[34] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159, 2017.

[35] Rachid Guerraoui, Antoine Murat, and Athanasios Xygkis. Velos: One-sided paxos for rdma applications, 2021.

[36] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[37] Xushen Han, Dajiang Zhou, Shihao Wang, and Shinji Kimura. 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*, pages 320–327. IEEE Computer Society, 2016.

[38] Martin Hardieck, Martin Kumm, Konrad Möller, and Peter Zipf. Reconfigurable convolutional kernels for neural networks on FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 43–52. ACM, 2019.

[39] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990.

[40] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E. Grant, and Ron Brightwell. Spin: High-performance streaming processing in the network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Farzin Houshmand and Mohsen Lesani. Hamsaz: Replication coordination analysis and synthesis. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[42] Farzin Houshmand, Mohsen Lesani, and Keval Vora. Grafs: Declarative graph analytics. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

[43] Farzin Houshmand, Javad Saberlatibari, and Mohsen Lesani. Hamband: Rdma replicated data types. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 348–363, New York, NY, USA, 2022. Association for Computing Machinery.

[44] Po-Yang Hsu, Ping-Chuan Lu, and Yi-Yu Liu. An efficient hybrid lut/sop reconfigurable architecture. In *Proceedings of 2010 International Symposium on VLSI Design, Automation and Test*, pages 173–176, 2010.

[45] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[46] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *OSDI*, pages 239–256, 2021.

[47] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 425–438, 2016.

[48] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *In Proceedings of the 22nd ACM international conference on Multimedia*, MM '14, page 675–678, New York, NY, USA, 2014. Association for Computing Machinery.

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

[50] Amin Kalantar, Zachary Zimmerman, and Philip Brisk. 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*, pages 40–49. IEEE, 2021.

[51] Kamalavasan Kamalakkannan, Gihan R. Mudalige, Istvan Z. Reguly, and Suhaib A. Fahmy. High-level FPGA accelerator design for structured-mesh-based explicit numerical solvers. *arXiv preprint arXiv:2101.01177*, 2021.

[52] S. Kang, J. Moon, and S. Jun. FPGA-accelerated time series mining on low-power IoT devices. In *IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 33–36, 2020.

[53] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 201–217, 2020.

[54] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR*, 2014.

[55] Ian Kuon and Jonathan Rose. *Quantifying and Exploring the Gap Between FPGAs and ASICs*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[56] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In Kia Bazargan and Stephen Neuendorffer, editors, *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*, pages 242–251. ACM, 2019.

[57] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[58] Andrew Lavin and Scott Gray. 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*, pages 4013–4021. IEEE Computer Society, 2016.

[59] Yann LeCun and Yoshua Bengio. *Convolutional Networks for Images, Speech, and Time Series*, page 255–258. MIT Press, 1998.

[60] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 137–152, New York, NY, USA, 2017. Association for Computing Machinery.

[61] Cheng Li, João Leitão, Allen Clement, Nuno M. Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 281–292. USENIX Association, 2014.

[62] Xiao Li, Farzin Houshmand, and Mohsen Lesani. Hampa: Solver-aided recency-aware replication. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 324–349, Cham, 2020. Springer International Publishing.

[63] Xiao Li, Farzin Houshmand, and Mohsen Lesani. Hamraz: Resilient partitioning and replication. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2267–2284, 2022.

[64] Google LLC. Coral toolkit. `https://coral.ai/`, 2020.

[65] Google LLC. Edge tpu. `https://cloud.google.com/edge-tpu`, 2021.

[66] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2017.

[67] Microsoft. tensorflow onnx. `https://github.com/onnx/tensorflow-onnx`, 2022.

[68] Sarah E. Minson, Men-Andrin Meier, Annemarie S. Baltay, Thomas C. Hanks, and Elizabeth S. Cochran. The limits of earthquake early warning: Timeliness of ground motion estimates. *Science Advances*, 4(3), 2018.

[69] Alex Montgomerie. Streaming architecture mapping optimiser. `https://github.com/AlexMontgomerie/samo`, 2022.

[70] Hiroki Nakahara, Zhiqiang Que, and Wayne Luk. High-throughput convolutional neural network on an FPGA by customized JPEG compression. In *28th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2020, Fayetteville, AR, USA, May 3-6, 2020*, pages 1–9. IEEE, 2020.

[71] NVIDIA. Mellanox innova-2 flex open programmable smart-nic, 2023.

[72] Emmanuel Oyekanlu. Predictive edge computing for time series of industrial iot and large scale critical infrastructure based on open-source software analytic of big data. In 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, editors, *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 1663–1669. IEEE Computer Society, 2017.

[73] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 4th Edition*. Springer, 2020.

[74] Alessandro Pappalardo. Xilinx/brevitas, 2021.

[75] Lucian Petrica, Tobias Alonso, Mairin Kroes, Nicholas Fraser, Sorin Cotofana, and Michaela Blott. Memory-efficient dataflow inference for deep CNNs on FPGA. *arXiv preprint arXiv:2011.07317*, 2020.

[76] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 107–118, New York, NY, USA, 2015. Association for Computing Machinery.

[77] `https://github.com/powerapi-ng/pyRAPL`, 2019.

[78] 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. 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)*, page 26–35. ACM, 2016.

[79] Zhiqiang Que, Hiroki Nakahara, Eriko Nurvitadhi, Hongxiang Fan, Chenglong Zeng, Jiuxi Meng, Xinyu Niu, and Wayne Luk. 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*, pages 10–18. IEEE, 2020.

[80] `https://powerapi-ng.github.io/rapl.html`, 2022.

[81] SeyedRamin Rasoulinezhad, Hao Zhou, Lingli Wang, and Philip H. W. Leong. 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*, pages 35–44. IEEE, 2019.

[82] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 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, pages 779–788. IEEE Computer Society, 2016.

[83] D. Sart, A. Mueen, W. Najjar, E. Keogh, and V. Niennattrakul. Accelerating dynamic time warping subsequence search with GPUs and FPGAs. In *IEEE International Conference on Data Mining*, pages 1001–1006, 2010.

[84] Claudio Satriano, Anthony Lomax, and Aldo Zollo. Real-time evolutionary earthquake location for seismic early warning. *Bulletin of the Seismological Society of America*, 98:1482–1494, 2008.

[85] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[86] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bull. EATCS*, 104:67–88, 2011.

[87] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 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*, pages 17:1–17:12. IEEE Computer Society, 2016.

[88] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[89] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2015.

[90] J. S. Tai, K. F. Li, and H. Elmiligi. Dynamic time warping algorithm: A hardware realization in VHDL. In *International Conference on IT Convergence and Security (ICITCS)*, pages 1–4, 2013.

[91] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition.* Pearson Education, 2007.

[92] Nicholas Gerard Timmons and Andrew Rice. Approximating activation functions. *arXiv preprint arXiv:2001.06370*, 2020.

[93] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 306–324, New York, NY, USA, 2017. Association for Computing Machinery.

[94] Wen-Chung Tsai, You-Jyun Shih, and Nien-Ting Huang. Hardware-accelerated, short-term processing voice and nonvoice sound recognitions for electric equipment control. *Electronics*, 8:924, 2019.

[95] Yaman Umuroglu. FINN: fast, scalable quantized neural network inference on fpgas. https://github.com/Xilinx/finn, 2022.

[96] Stylianos I. Venieris and Christos-Savvas Bouganis. 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*, pages 40–47. IEEE Computer Society, 2016.

[97] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions. *ACM Comput. Surv.*, 51(3), jun 2018.

[98] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides. LUTNet: Rethinking inference in FPGA soft logic. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 26–34, 2019.

[99] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. Deepburning: automatic generation of fpga-based learning accelerators for the neural network family. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, pages 110:1–110:6. ACM, 2016.

[100] Zhiguang Wang, Weizhong Yan, and Tim Oates. 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*, pages 1578–1585. IEEE, 2017.

[101] Zilong Wang, Sitao Huang, Lanjun Wang, Hao Li, Yu Wang, and Huazhong Yang. Accelerating subsequence similarity search based on dynamic time warping distance with FPGA. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, page 53–62, 2013.

[102] Denis S. Willett, Justin George, Nora S. Willett, Lukasz L. Stelinski, and Stephen L. Lapointe. Machine learning for characterization of insect vector feeding. *PLOS Computational Biology*, 12(11):1–14, 11 2016.

[103] Ephrem Wu, Xiaoqian Zhang, David Berman, Inkeun Cho, and John Thendean. Compute-efficient neural-network acceleration. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 191–200. ACM, 2019.

[104] Xilinx. DPU for convolutional neural network v3.0, DPU IP product guide, 2019.

[105] Xilinx. Xilinx vitis network example. https://github.com/Xilinx/xup_vitis_network_example, 2022.

[106] https://www.xilinx.com/products/design-tools/vivado.html, 2022.

[107] Jinwei Xu, Zhiqiang Liu, Jingfei Jiang, Yong Dou, and Shijie Li. CaFPGA: An automatic generation model for CNN accelerator. *Microprocessors and Microsystems*, 60, 2018.

[108] Pengfei Xu et al. AutoDNNchip: An automated DNN chip predictor and builder for both FPGAs and ASICs. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 40–50. ACM, 2020.

[109] X. Xu, F. Lin, W. Xu, X. Yao, Y. Shi, D. Zeng, and Y. Hu. MDA: a reconfigurable memristor-based distance accelerator for time series mining on data centers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):785–797, 2019.

[110] Zhilin Xu, Jincheng Yu, Chao Yu, Hao Shen, Yu Wang, and Huazhong Yang. 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*, pages 33–37. IEEE, 2020.

[111] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzynek, and Kurt Keutzer. Synetgy: Algorithm-hardware co-design for ConvNet accelerators on embedded FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 23–32. ACM, 2019.

[112] C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh. 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)*, pages 1317–1322, 2016.

[113] Yunxuan Yu, Tiandong Zhao, Kun Wang, and Lei He. 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)*, page 122–132. ACM, 2020.

[114] Hanqing Zeng, Ren Chen, Chi Zhang, and Viktor K. Prasanna. A framework for generating high throughput CNN implementations on FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 117–126. ACM, 2018.

[115] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks. In Frank Liu, editor, *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016*, pages 12:1–12:8. ACM, 2016.

[116] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 161–170. ACM, 2015.

[117] Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 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, pages 770–778. IEEE Computer Society, 2016.

[118] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen Mei Hwu, and Deming Chen. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design, (ICCAD)*, 2018.

[119] Y. Zhang, K. Adl, and J. Glass. Fast spoken query detection using lower-bound dynamic time warping on graphical processing units. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5173–5176, 2012.

[120] Ruizhe Zhao, Ho-Cheung Ng, Wayne Luk, and Xinyu Niu. Towards efficient convolutional neural network for domain-specific applications on FPGA. In *28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 147–1477, 2018.

[121] H. Zhou, X. Xu, Y. Hu, G. Yu, Z. Yan, F. Lin, and W. Xu. Energy-efficient pipelined DTW architecture on hybrid embedded platforms. In *Sixth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, 2015.

[122] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association.

[123] Chaoyang Zhu, Kejie Huang, Shuyuan Yang, Ziqi Zhu, Hejia Zhang, and Haibin Shen. An Efficient Hardware Accelerator for Structured Sparse Convolutional Neural Networks on FPGAs, 2020.

[124] 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. 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):949–979, 2020.

[125] Zachary Zimmerman. LAMP supporting webpage: `https://sites.google.com/view/lamp2019`. Accessed: 2021-01-14.

[126] Zachary Zimmerman, Kaveh Kamgar, Nader Shakibay Senobari, Brian Crites, Gareth J. Funning, Philip Brisk, and Eamonn J. Keogh. 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*, pages 74–86. ACM, 2019.

[127] Zachary Zimmerman, Nader Shakibay Senobari, Gareth J. Funning, Evangelos E. Papalexakis, Samet Oymak, Philip Brisk, and Eamonn J. Keogh. Matrix profile XVIII:

time series mining in the face of fast moving streams using a learned approximate matrix profile. In Jianyong Wang, Kyuseok Shim, and Xindong Wu, editors, *2019 IEEE International Conference on Data Mining, ICDM 2019, Beijing, China, November 8-11, 2019*, pages 936–945. IEEE, 2019.