UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Fast and Energy Efficient Big Data Processing on FPGAs

Permalink
https://escholarship.org/uc/item/1wv8x2d2

Author
Salamat, Sahand

Publication Date
2021

Peer reviewed|Thesis/dissertation

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

UNIVERSITY OF CALIFORNIA SAN DIEGO

Fast and Energy Efficient Big Data Processing on FPGAs

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Sahand Salamat

Committee in charge:

Professor Tajana Rosing, Chair
Professor Chung-Kuan Cheng
Professor Ryan Kastner
Professor Farinaz Koushanfar
Professor Jishen Zhao

2021

Copyright

Sahand Salamat, 2021

All rights reserved.

The Dissertation of Sahand Salamat is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2021

DEDICATION

To my family and friends
for their unconditional love and support.

EPIGRAPH

Everything will turn out right, the world is built on that.

Mikhail Bulgakov, The Master and Margarita

Dead yesterdays and unborn tomorrows,
why fret about it, if today be sweet.

Omar Khayyâm

TABLE OF CONTENTS

Dissertation Approval Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures viii

List of Tables xi

Acknowledgements xii

Vita xiv

Abstract of the Dissertation xvii

Introduction 1

Chapter 1 FPGA Acceleration of DNNs 6
- 1.1 Background and Related Work 8
- 1.2 Proposed Residue-Net 10
  - 1.2.1 RNS Operations 11
  - 1.2.2 Residue-Net Architecture 15
- 1.3 Experimental Result 18
  - 1.3.1 Experimental Setup 18
  - 1.3.2 System Evaluation 19
  - 1.3.3 Operation Evaluation 21
- 1.4 Conclusion 24

Chapter 2 Accelerating HD computing on FPGAs 26
- 2.1 Background and Related Work 27
  - 2.1.1 Hyperdimensional Computing 27
  - 2.1.2 Related Studies 32
- 2.2 HD2FPGA Framework Overview 34
- 2.3 HD2FPGA Architecture 35
  - 2.3.1 HD2FPGA software program 42
- 2.4 Experimental Results 44
  - 2.4.1 HD Classification 46
  - 2.4.2 HD Clustering 52
- 2.5 Conclusion 56

Chapter 3 Efficiency of ML on Multi-FPGAs 58

3.1 Related Work 59
3.2 Motivational Analysis 63
3.3 Proposed Method 66
3.3.1 Workload Prediction 68
3.3.2 Frequency Scaling Flow 70
3.3.3 Voltage Scaling Flow 71
3.4 Proposed Architecture 72
3.5 Experimental Results 74
3.5.1 General Setup 74
3.5.2 Results 76
3.6 Conclusion 78

Chapter 4 Summary and Future Work 79
4.1 Thesis Summary 79
4.2 Future Directions 81

Bibliography 83

LIST OF FIGURES

Figure 1.1. Resource utilization of MAC operation in conventional binary, RNS, and Residue-Net on FPGA. 7

Figure 1.2. The RNS multiplication unit of Residue-Net. 14

Figure 1.3. (a) The proposed architecture for Residue-Net. (b) A convolutional kernel that performs computations on the inputs and weights R_4. (c) Residue-Net proposed comparator module. (d) A fully connected neuron working in inputs and weights R_4. 15

Figure 1.4. Comparing the normalized values of power, throughput and energy consumption of the FPGA baseline, Residue-Net (area-opt), and Residue-Net (performance-opt) w.r.t the FPGA baseline for different DNNs. 21

Figure 1.5. Comparing the LUT and BRAM utilization of the FPGA baseline, Residue-Net (area-opt), and Residue-Net (performance-opt), as well as the normalized values of power, throughput and energy consumption w.r.t the FPGA baseline for AlexNet. 22

Figure 1.6. LUT utilization and power consumption of Residue-Net building blocks in DNNs. 23

Figure 2.1. Overview of hyperdimensional learning and inference. 30

Figure 2.2. Overview of the proposed framework, HD2FPGA. 34

Figure 2.3. The architecture of the InputStream module. 36

Figure 2.4. (a) The matrix multiplication of the projection hypervectors and the input feature vector. (b) HD2FPGA architecture based on random-projection encoding 39

Figure 2.5. The partial matrix multiplication example when $D_{hv} = 16,\ D_{iv} = 8,\ C = 2$, and $R = 4$. 41

Figure 2.6. The proposed architecture of the Search unit for (a) HD Classification and (b) HD clustering. 41

Figure 2.7. Encoding time of CPU and HD2FPGA. 47

Figure 2.8. Retraining time of CPU and FPGA for 50 epochs. 48

Figure 2.9. End-to-end Training (encoding + 50 epochs retraining) time of CPU and HD2FPGA. 49

Figure 2.10. End-to-end Training (encoding + 50 epochs retraining) energy consumption of CPU and HD2FPGA. 50

Figure 2.11. Inference (encoding + associative search) time of CPU and HD2FPGA. . . 51

Figure 2.12. Inference (encoding + associative search) energy consumption of CPU and HD2FPGA. 52

Figure 2.13. Accuracy and latency (number of cycles) of HD2FPGA versus DNN accelerator of [1]. 53

Figure 2.14. End-to-end latency and performance improvement of HD2FPGA versus DNN accelerator of [1]. 53

Figure 2.15. Energy comparison of HD2FPGA versus DNN accelerator of [1]. 54

Figure 2.16. Inference (encoding + associative search) time of CPU and HD2FPGA. . . 55

Figure 2.17. Inference (encoding + associative search) energy consumption of CPU and HD2FPGA. 55

Figure 2.18. End-to-end performance improvement of HD2FPGA versus Kmeans accelerator of [2]. 56

Figure 2.19. Energy comparison of HD2FPGA versus Kmeans accelerator of [2]. 57

Figure 3.1. Delay of FPGA resources versus voltage 61

Figure 3.2. Dynamic power of FPGA resources versus voltage. 62

Figure 3.3. Static power of FPGA resources versus voltage. 62

Figure 3.4. Comparing DVFS techniques in different workloads. 65

Figure 3.5. Comparing DVFS techniques in different critical paths. 66

Figure 3.6. Comparing DVFS techniques in different BRAM power rates. 67

Figure 3.7. Overview of an FPGA-based datacenter platform. 68

Figure 3.8. Example of Markov chain for workload prediction. 69

Figure 3.9. (a) the architecture of the proposed energy-efficient multi-FPGA platform. The details of the (b) central controller, and (c) the FPGA instances. 71

Figure 3.10. Comparing the efficiency of different voltage scaling techniques under a varying workload for Tabla framework. 75

Figure 3.11. Voltage adjustment in different voltage scaling techniques under the varying workload for Tabla framework. 76

Figure 3.12. Power efficiency of the proposed technique in different acceleration frameworks. 77

LIST OF TABLES

Table 1.1. Accuracy of the Residue-Net as compared to the full precision and quantized to 6 bits networks. 19

Table 1.2. LUT overhead of using Residue-Net in executing DNNs. 24

Table 2.1. Comparing the accuracy and memory requirement of each encoding method in various edge sensing applications. 30

Table 3.1. Post place and route resource utilization and timing of the benchmarks. . . . 75

Table 3.2. Comparison of power efficiency of different approaches. 77

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Tajana Rosing. During my Ph.D., Tajana has been an extremely supportive advisor, insightful teacher, and considerate friend. Without her guidance and support, my Ph.D. journey would not be as successful and memorable as it is. I would like to thank the rest of my doctoral committee members, Prof. Chung-Kuan Cheng, Prof. Farinaz Koushanfar, Prof. Ryan Kastner, and Prof. Jishen Zhao for their support, thoughtful comments, and invaluable guidance that have improved the quality of this work.

I would also like to express my sincere appreciation to Dr. Yang Seok Ki from Samsung Semiconductor and my colleagues Dr. Armin Haj Aboutalebi, Dr. Joo Hwan Lee, Dr. Hui Zhang for their keen knowledge and insights. I am grateful to all my lab-mates who are amazing colleagues and real friends: Dr. Mohsem Imani, Dr. Yeseong Kim, Dr. Saransh Gupta, Anthony Thomas, Behnam Khaleghi, Derek Jones, Fatemeh Asgarinejad, Haichao Yang, Jaeyoung Kang, Jason Ma, Justin Morris, Kazim Ergun, Michael Ostertag, Minxuan Zhou, Onat Gungor, Rishikanth Chandrasekaran, Weihong Xu, Xiaofan Yu, and Uday Mallappa. I would like to especially thank Mohsen and Behnam for their collaboration and help in many of my projects.

I am additionally thankful to my dearest friends, who will always have a special place in my heart, for their kindness and support in my life: Amir, Amirali, Niloofar, Mohammad, Siavash, Behrooz, Sina, Elham, Elahe, Kazem, Bahram, Ali, Ashkan, Saina, Ehsan, Mohsen, and Reza. Beyond all, I want to express my most sincere appreciation to my family, Saeed, Roshanak, Ronak, and Misagh for their unconditional love and support.

My research was made possible by funding from the National Science Foundation (NSF) Grant 2003279, 2028040, 1527034, 1619261, 1730158, 1826967, 1911095, DARPA, CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA. The material in this dissertation is, in part, based on the following papers that are published or are under review.

Chapter 1 contains material from “Residue-Net: Multiplication-free Neural Network

by In-situ No-loss Migration to Residue Number Systems", by Sahand Salamat, Sumiran Shubhi, Behnam Khaleghi, and Tajana S. Rosing, which appears in IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2021 [3]. The dissertation author was the primary investigator and author of this paper.

Chapter 2 contains material from "HD2FPGA: Automated Framework for Accelerating Hyperdimensional Computing on FPGAs", by Sahand Salamat, Behnam Khaleghi, and Tajana S. Rosing, which is still in preparation. The dissertation author was the primary investigator and author of this paper.

Chapter 3 contains material from "Workload-Aware Opportunistic Energy Efficiency in Multi-FPGA Platforms", by Sahand Salamat, Behnam Khaleghi, Mohsen Imani, and Tajana S. Rosing, which appears in IEEE/ACM International Conference On Computer Aided Design (ICCAD), 2019 [4]. The dissertation author was the primary investigator and author of this paper.

VITA

2017	B. Sc. in Electrical Engineering, University of Tehran, Iran
2017-2021	Graduate Research Assistant, University of California, San Diego
2018	Graduate Teaching Assistant, University of California, San Diego
2021	Ph. D. in Computer Science and Engineering, University of California, San Diego

PUBLICATIONS

S. Salamat, N. Moshiri, T. Rosing ”FPGA Acceleration of Pairwise Distance Calculation for Viral Transmission Clustering,” The 2021 IEEE Biomedical Circuits and Systems Conference (BIOCAS), 2021.

S. Salamat, H. Zhang, Y. S. Ki, T. Rosing “NASCENT2: Generic Near-storage Sort Accelerator for Data Analytics on SmartSSD,” ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2021.

S. Salamat, A. H. Aboutalebi, B. Khaleghi, J. H. Lee, Y. S. Ki, T. Rosing ”NASCENT: Near-Storage Acceleration of Database Sort on SmartSSD,” The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2021.

M. Imani, Z. Zou, S. Bosch, S. A. Rao, S. Salamat, V. Kumar, Y. Kim, and T. Rosing. ”Revisiting hyperdimensional learning for fpga and low-power architectures,” IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2021.

S. Salamat, J. Kang, Y. Kim, M. Imani, N. Moshiri, T. Rosing ”FPGA Acceleration of Protein Back-Translation and Alignment,” Design, Automation Test in Europe Conference Exhibition (DATE). IEEE, 2021.

Y. Guo, M. Imani, J. Kang, S. Salamat, J. Morris, B. Aksanli, Y. Kim, T. Rosing. ”HyperRec: Efficient Recommender Systems with Hyperdimensional Computing,” 26th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 2021.

S. Salamat, S. Shubhi, B. Khaleghi, T. Rosing ”Residue-Net: Multiplication-free Neural Network by In-situ No-loss Migration to Residue Number Systems,” 26th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 2021.

B. Khaleghi, S. Salamat, T. Rosing ”Revisiting FPGA Routing under Varying Operating Conditions,” International Conference on Field-Programmable Technology (ICFPT). IEEE, 2020.

B. Khaleghi, S. Salamat, A. Thomas, F. Asgarinejad, Y. Kim, T. Rosing ”SHEARer: highly-efficient hyperdimensional computing by software-hardware enabled multifold approximation,” Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design. 2020.

S. Salamat, M. Imani, T. Rosing ”Accelerating hyperdimensional computing on fpgas by exploiting computational reuse,” IEEE Transactions on Computers 69.8 (2020): 1159-1171.

M. Imani, S. Bosch, S. Datta, S. Ramakrishna, S. Salamat, J. M. Rabaey, T. Rosing ”Quanthd: A quantization framework for hyperdimensional computing,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39.10 (2019): 2268-2278.

B. Khaleghi, S. Salamat, M. Imani, T. Rosing ”Fpga energy efficiency by leveraging thermal margin,” 2019 IEEE 37th International Conference on Computer Design (ICCD). IEEE, 2019.

S. Salamat, B. Khaleghi, M. Imani, T. Rosing ”Workload-aware opportunistic energy efficiency in multi-fpga platforms,” 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2019.

M. Imani, S. Salamat, B. Khaleghi, M. Samragh, F. Koushanfar, T. Rosing, ”Sparsehd: Algorithm-hardware co-optimization for efficient high-dimensional computing,” 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2019.

S. Salamat, M. Imani, B. Khaleghi, T. Rosing, ”F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing,” Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2019.

M. Imani, S. Salamat, S. Gupta, J. Huang, T. Rosing, ”Fach: Fpga-based acceleration of hyperdimensional computing by reducing computational complexity,” Proceedings of the 24th Asia and South Pacific Design Automation Conference. 2019.

S. Salamat, M. Imani, S. Gupta, T. Rosing, ”Rnsnet: In-memory neural network acceleration using residue number system,” 2018 IEEE International Conference on Rebooting Computing (ICRC). IEEE, 2018.

M. Ahmadi, S. Salamat, B. Alizadeh. ”A dynamic timing error avoidance technique using prediction logic in high-performance designs,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems 27.3 (2018), 734-737.

S. Salamat, M.R. Azarbad, B. Alizadeh. ”High-level Synthesis of Non-Rectangular Multi-Dimensional Nested Loops using Reshaping and Vectorization,” 2018 IEEE International Conference on Rebooting Computing (ICRC). IEEE, 2018.

S. Salamat, M. Ahmadi, B. Alizadeh, M. Fujita, ”Systematic approximate logic optimization using don’t care conditions,” 18th International Symposium on Quality Electronic Design (ISQED). IEEE, 2017.

ABSTRACT OF THE DISSERTATION

Fast and Energy Efficient Big Data Processing on FPGAs

by

Sahand Salamat

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2021

Professor Tajana Rosing, Chair

With the rapid development of the Internet of things (IoT), networks, software, and computing platforms, the size of the generated data is dramatically increasing, bringing the dawn of *big data* era. These ever-increasing data volumes and complexity require new algorithms and hardware platforms to deliver sufficient performance. Data from sensors, such as images, video, and text, contributed to 2.5 quintillions (2.5×10^{18}) bytes generated every day in 2020. The rate of generating data is outpacing the computational capabilities of conventional computing platforms and algorithms. CPU performance improvement has been stagnating in recent years, which is one of the causes of the rise of application-specific accelerators that process big data applications. FPGAs are also more commonly used for accelerating big data algorithms, such as

machine learning.

In this work, we develop and optimize both the hardware implementation, and also algorithms for FPGA-based accelerators to increase the performance of machine learning applications. We leverage Residue Number System (RNS) to optimize the deep neural networks (DNNs) execution and develop an FPGA-based accelerator, called Residue-Net, to entirely execute DNNs using RNS on FPGAs. Residue-Net improves the DNNs throughput by 2.8$\times$ compared to the FPGA-based baseline. Even though running DNNs on FPGAs provides higher performance compared to running on general-purpose processors, due to their intrinsic computation complexity, it is challenging to deliver high performance and low energy consumption using FPGAs, especially for the edge devices. Less complex and more hardware-friendly machine learning algorithms are needed in order to revolutionize the performance at and beyond the edge.

Hyperdimensional computing (HD) is a great example of a very efficient paradigm for machine learning. HD is intrinsically parallelizable with significantly fewer operations than DNNs, and thus can easily be accelerated in hardware. We develop an automated tool to generate an FPGA-based accelerator, called HD2FPGA, for classification and clustering applications, with accuracy that is comparable to the state of the art machine learning algorithms, but orders of magnitude higher efficiency. HD2FPGA achieves 578$\times$ speedup and 1500$\times$ energy reduction in end-to-end execution of HD classification compared to the CPU baseline. HD2FPGA, compared to state-of-the-art DNN running on an FPGA, delivers 277$\times$ speedup and 172$\times$ energy reduction.

As the volume of data increases, a single FPGA is not enough to get the desired performance. Thus, many cloud service providers offer multi-FPGA platforms. The size of the data centers workloads varies dramatically over time, leading to significant underutilization of computing resources such as FPGAs, while consuming a large amount of power, which is a critical contributor to data center inefficiency. We propose an efficient framework to throttle the power consumption of multi-FPGA platforms by dynamically scaling the voltage and hereby frequency at run time according to *prediction* of, and *adjustment* to the workload level, while maintaining the desired Quality of Service (QoS). Our evaluations by implementing state-of-the-art deep

neural network accelerators revealed that, providing an average power reduction of 4.0×, the proposed framework surpasses the previous works by 33.6% (up to 83%).

Introduction

Direct communication between the smart devices, sensors, and computational platforms in the IoT era has revolutionized the data generation and data processing tasks. IoT devices generate an unprecedented volume of data from heterogeneous sources [5, 6]. The size of the yearly generated data is growing exponentially, thanks to the development of IoT sensors and computation platforms. In 2020, 64.2 zettabytes of data was generated globally, which is almost three times the data generated in 2018 [7]. Processing this amount of data requires more advanced algorithms, as well as fast and energy-efficient computation platforms. Machine learning algorithms have significantly improved processing and extracting meaningful knowledge from the data [8]. However, running machine learning algorithms requires a tremendous amount of computation. The lack of significant advancement in current general processing units (CPUs and GPUs) motivates the ever-increasing adoption of application-specific accelerators in both edge devices and cloud platforms for running machine learning algorithms [9, 10, 11].

Machine learning algorithms have been widely used to process a wide range of applications [12] such as image and video processing [13, 14, 15, 16, 17], voice recognition [18, 19], robotics [20, 21, 22], and bioinformatics data [23, 24], to name a few. Deep Neural Networks (DNNs), thanks to their ever-increasing accuracy, are widely exploited for such data analysis. However, the accuracy of DNNs is highly dependent on the complexity of the model. Certain applications can afford to sacrifice the classification accuracy to achieve higher performance or reduce energy consumption whereas plenty of applications, e.g., autonomous driving workloads, require the highest accuracy to avoid fatal consequences of misprediction. Even in these applications, as DNNs are running on resource-constrained platforms, the energy usage and

performance to process hundreds of inputs per second is of the utmost importance.

Quantizing the floating-point weights and activations (i.e., outputs of network layers) to fixed-point numbers with lower precision is a widely applied technique to lower the memory footprint and computational complexity of DNN inference [25, 26, 27]. Previous works have shown that DNNs can be quantized, depending on the application, without affecting the accuracy of the results [28, 29]. Quantizing the network weights and activations simplifies the computation of the multiplication and accumulation (MAC) operations, which contribute to more than 99% of operations of DNN inference [30]. However, exploiting network quantization requires flexibility in processing cores to execute operations with customized data widths.

FPGAs have become an attractive platform to accelerate DNNs as they provide high flexibility in design, performance, and energy efficiency [31]. Several works have developed customized FPGA-based accelerators for DNNs with various hardware and software optimizations, including quantization [26, 32, 33]. The main advantage of network quantization results when the multiplication operations are replaced with more hardware friendly operations (e.g. XOR, shift, addition). Even though quantizing the network weights and activations to binary or ternary values simplifies the multiplications [26], these networks have significant accuracy degradation in many applications [34, 28].

In chapter 1, instead of quantizing the network to binary or ternary, we leverage a modified number representation, Residue Number System (RNS), that simplifies the MAC operations without significant impact on the accuracy of the networks. We additionally propose an FPGA-based accelerator, called Residue-Net, to fully utilize the advantage of using RNS to accelerate DNN inference. RNS is an unorthodox number representation developed based on the Chinese Remainder Theorem (CRT) where each number is represented with residues of dividing the number by a set of predefined modulis[35]. Residue-Net quantizes DNN weights and activations to 6 bits, which has been shown to be sufficient to achieve good accuracy in many application [28]. It then uses this six-bit RNS representation to execute DNNs and deliver the excellent accuracy while significantly simplifying the DNN operations.

As DNNs are computationally complex in nature, to achieve real-time learning on resource-constrained devices, we need to utilize more lightweight and hardware-friendly machine learning algorithms. Hyperdimensional (HD) computing is a novel computational approach that builds upon imitating the brain functionality in performing cognitive tasks [36, 37]. It has been shown that the brain computes with high dimensional and sparse patterns of neural activity, which can be realized as points in a hyperdimensional space, called hypervectors. By leveraging a non-complex and parallel set of operations on such ultra-wide vectors, HD affords promising capabilities in learning various types of applications such as classification, clustering, regression, recommendation systems, and reinforcement learning, to name a few [38, 39, 40, 41, 42, 43, 44, 45]. In addition to its inclusive cognitive application space and comparatively simpler computation model than other learning paradigms [46], HD computing is inherently robust against failures as the information in a hypervector is distributed over all of its comprising dimensions [36].

The first step in HD classification and clustering is *encoding* the input data to hypervectors. Encoding data into the hyperdimensional space provides simple classification and clustering schemes. In HD classification, each class is represented by adding all the hypervectors belonging to the class. Similarly, in HD clustering, all the encoded hypervectors in a cluster are aggregated to represent the centroid hypervector. Inference (classification or clustering) in HD computing is analogous; the encoded hypervector passes through an associative search (a.k.a similarity check) with the representative hypervectors to identify the associated class/cluster [36]. The encoding, training, and inference stages of HD computing require a substantial number of bit-level addition and multiplication operations, which can be effectively parallelized [47]. Such characteristics of HD computing inimitably matches with the intrinsic capabilities of FPGAs, making these devices a unique solution for accelerating these applications. However, implementing applications on FPGAs is a time consuming process [46].

In chapter 2 we propose HD2FPGA, an automated tool that generates an FPGA-based accelerator for HD classification and clustering to hide the implementation complexities from

the user and reduce the associated time invested in designing the hardware. HD2FGPA generates a synthesizable Verilog implementation of HD accelerator while taking the high-level user and target FPGA parameters into account. It is a flexible and highly optimized tool capable of generating a fast and energy-efficient accelerator considering the user's constraints (viz., performance, and power). HD2FPGA supports end-to-end training, retraining, and inference for both HD classification and clustering while delivering significantly higher performance compared to state-of-the-art DNNs running on FPGA.

With the emergence and prevalence of big data, the majority of large businesses are running machine learning algorithms on cloud services [48]. In 2010, data centers accounted for 1.1-1.5% of the world's total electricity consumption [49], with a spike to 4% in 2014 [50] raised by the move of localized computing to cloud facilities. The energy consumption of data centers is expected to double every five years [51]. The huge power consumption of cloud data centers has led to the wide deployment of FPGAs in data centers due to their high performance and energy efficiency [52, 53, 54]. Cloud service providers offer FPGAs as Infrastructure as a Service (IaaS) or use them to provide Software as a Service (SaaS). Cloud service providers offer multi-FPGA platforms for cloud users to implement their applications. They also provide applications as a service, e.g., convolutional neural networks [55], search engines [56], text analysis [57], etc. running on multi-FPGA platforms.

In spite of the benefits offered by FPGAs, underutilization of computing resources is still the main contributor to the energy loss in data centers. Data centers are expected to provide the required Quality of Service (QoS) of users while the size of the incoming workload varies temporally. Typically, the size of the workload is less than 30% of the users' expected maximum, directly translating to the fact that servers run at less than 30% of their maximum capacity [58]. A straightforward technique to lower the energy costs is to adjust the operating frequency in tandem with workload variation where all nodes are still responsible for processing a portion of the input data. This reduces the dynamic power consumption proportional to the workload, and resolves the problem of wake-up and reconfiguration time that come with power gating of

nodes. Nonetheless, as all nodes are active, the static power remains a challenge, especially in elevated temperatures near FPGA boards in data centers [56] due to exponential increase the leakage current with temperature.

In chapter 3, we optimize the energy consumption of multi-FPGA data center platforms, accounting for the fact that the workload is often considerably less than the maximum anticipated. We leverage this opportunity to use the available resources while dynamically scaling the voltage and frequency of the entire system so that the projected throughput (i.e., QoS) is delivered. We utilize a light-weight predictor for proactive estimation of the incoming workload and incorporate it to our power-aware timing analysis framework that adjusts the frequency and finds *optimal* voltages, keeping the process transparent to users.

The rest of the dissertation is organized as follows, in chapter 1 we propose and FPGA-based accelerator for DNNs utilizing RNS. In chapter 2, we develop an automated tool to automatically generate highly optimized FPGA-based accelerator for HD classification and clustering. In chapter 3, we optimize the energy efficiency of multi-FPGA platforms while delivering the user's desired QoS with specific focus on the machine learning tasks. Chapter 4 summarizes the dissertation and discuss the possible next steps.

Chapter 1

FPGA Acceleration of DNNs

Deep neural networks (DNNs) are used to solve a wide range of problems from edge-sensing applications to autonomous driving. The accuracy of these networks is usually proportional to their complexity. One strategy to improve their performance and lower the energy cost is to use quantization of model parameters (i.e., weights) and/or activations. General purpose processors cannot benefit from custom data types as their computation core is designed and optimized for certain data types (e.g. 32-bit, 16-bit, or 8-bit integers). FPGAs can efficiently execute quantized networks as they provide flexibility in design and execution of customized data types. Network quantization simplifies DNN operations, the majority of which are MACs, by reducing the bit-width of the operands. A number of publications sacrifice accuracy to get higher performance and energy efficiency by quantizing the network weights and activations to binary [46] or ternary values [59, 60, 61] to omit multiplications.

Figure 1.1 shows the efficacy of quantization in reducing the complexity of DNN inference (particularly MAC operations) on FPGA. Quantizing 32-bit fixed-point weights to six bits reduces the number of Look-up Tables (LUTs) by 97%, without adverse impact on the accuracy of the network after re-training. LUT count for 32-bit operands is not shown due to differences in scale, though the $\propto n^2$ trend can be discerned from the figure. Although reducing the bit-width below six bits saves area, its results in significantly lower accuracy that is not affordable for the majority of real-world applications [28].

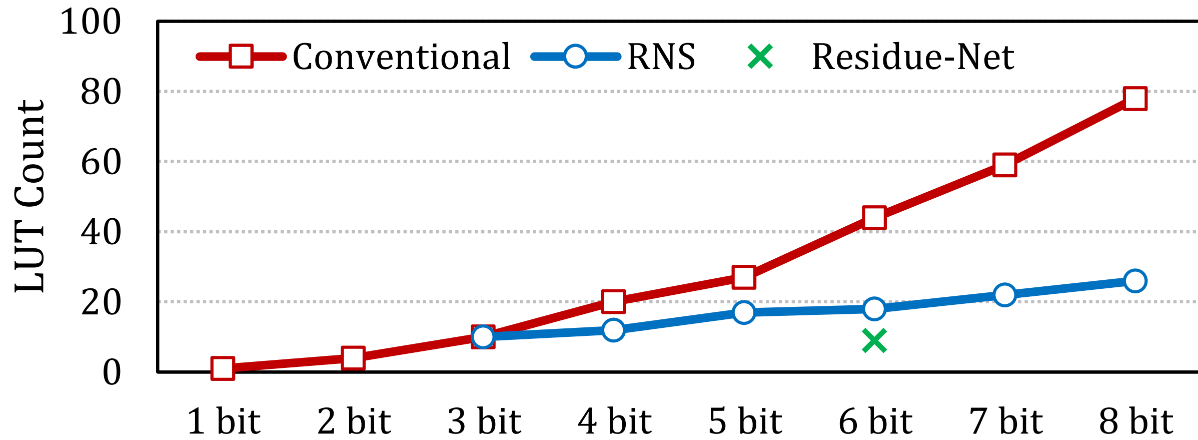

Figure 1.1. Resource utilization of MAC operation in conventional binary, RNS, and Residue-Net on FPGA.

In this chapter, we propose Residue-Net, a multiplication-free accelerator for neural networks that uses Residue Number System (RNS) to achieve substantial energy reduction. RNS is an unorthodox number representations that breakdown large numbers into multiple smaller numbers, resulting in replacing arithmetic operations with multiple simpler operations by reducing the operands' bit widths. Each residue (remainder) is always smaller than the corresponding modulus. Therefore, for representing each residue, fewer bits are needed. RNS simplifies the operations by breaking them down to the same operations on the residues with smaller bit-width. Consequently, it needs simpler arithmetic for implementation. Figure 1.1 shows the area of a MAC unit in RNS format for different bit-widths. The area is the total area of the residue MAC units. For instance, for six-bit binary numbers that are converted to RNS with $\{3,4,5\}$ as moduli set, the area of the MAC unit is shown as six-bit RNS MAC. Six-bit RNS MAC reduces the area by 2.4$\times$ as compared to the binary MAC unit (18 LUTs versus 44).

Residue-Net uses six-bit RNS with $\{3,4,5\}$ moduli set to exploit the benefits provided by RNS to represent six-bit network weights with two 2-bit and one 3-bit numbers. Considering the $\{3,4,5\}$ moduli set, in Residue-Net, the residues can only be within the set of $\{0,1,2,3,4\}$, as the residues are always less than the modulis. Therefore, Residue-Net replaces costly multiplications with non-complex, energy-efficient shift and add operations to further simplify the computational

complexity of neural networks as shown in section 1.2.1. As it is shown in Figure 1.1, the six-bit MAC unit of Residue-Net, by eliminating the multiplication operations, requires 50.0% fewer LUTs than the baseline RNS MAC unit, and 79.5% less than the original binary unit.

Residue-Net trains and quantizes the DNN model using the approach proposed in [62], and transforms the weights and activations into RNS to carry out the exact computations on operands with fewer bits. Since Residue-Net performs all the computations in exact mode, the accuracy of the network remains the same. Residue-Net, not only breaks down the computations to operations with fewer bits, but it is also able to replace the costly multiplications with simpler add operations to further reduce the computation complexity. The main contribution of this chapter is as follows.

- We propose Residue-Net, a novel FPGA-based accelerator based on RNS which maps all activations and weights of a neural network to RNS, which breaks down the six-bit operations into two 2-bit and a 3-bit operation.
- We further simplify the multiplications into add and shift operations, which further simplifies the computations of the neural network inference.
- The results of implementing four popular neural networks, LeNet, AlexNet, VGG16, and ResNet-50 on FPGA leveraging the our innovations reveal 2.8× better performance with no impact on accuracy.

1.1 Background and Related Work

Neural networks have been extensively used in many applications and have been accelerated on various platforms [63, 31]. Quantization has been explored to compress and accelerate the neural network in literature [28, 64, 65, 61]. The work in [66] developed a tool that enables both software simulation and hardware realization of DNNs using different data representations and approximate computing blocks. The work in [28] proposes an automated DNN inference

accelerator generator that first quantizes the network and then retrains it to retrieve the accuracy drop. The work [64] uses reinforcement learning for hardware-aware layer-wise weight quantization. The above works use static quantization so that the weight bit-width is set fixed either for the entire network or on a layer-by-layer basis. However, the studies in [65, 67] quantize the weights based on the complexity of the input such that difficult-to-predict inputs run through a more precise network. For over-simplifying the execution of the neural network, studies [46] quantize the network weights down to binary, $\{-1,+1\}$, which replaces the multiplications with XOR operations. Although binarized neural networks are significantly faster than fixed-point alternatives, they are practically unappealing for real-world applications due to their low accuracy [68, 69]. Ternary weight representation is proposed [59, 60, 61] to ameliorate the accuracy drop by using $\{-1,0,+1\}$ or $\{-W,0,+W\}$, even though ternary NNs still cannot provide the same accuracy as NNs with fixed-point weights.

On an orthogonal research path, unorthodox number representations have been studied to achieve better accuracy or performance than the conventional fixed-point or floating-point networks [70, 71, 72, 73]. The study in [70] proposes narrow-precision floating-point representations instead of 32-bit floating-point numbers to simplify the operations. The posit number system is introduced in [71] as an alternative representation of IEEE floating-point number to represent real numbers. Posit has a larger dynamic range and higher accuracy as compared to floating-point numbers which make it suitable for DNNs to get the same accuracy as 32-bit floating-point weights with less than eight-bit posit weights [74]. While these studies convert a trained network to posit representation, [75] uses posit numbers to train the networks. Posit operations, however, are more complicated than the floating-point ones [76].

Residue Number System: RNS has been used in compute-intensive applications such as digital signal processing [77] and neural networks [78, 79, 72] to accelerate them by reducing the bit-width of operands. A number in RNS is represented by the remainders (residues) of dividing it by a set of numbers, called moduli. Moduli set is a set of numbers $M_i \in$ Moduli Set$\{M_1, M_2, ..., M_k\}$ where any pair M_i and M_j are co-prime. The process of divid-

ing a number by the moduli set and representing the number by the residues is called *forward conversion*. The process of converting the RNS numbers back to the binary system is called *backward conversion*. Several studies such as [80] investigate the impact of different moduli sets on the computation complexity of the forward conversion step. Multiplication and add operations, the two most common operation in DNNs, can be directly performed in the RNS domain [72]. Being the remainder of the division by a modulus, each number in RNS is smaller than the modulus, thereby RNS represents a binary number with multiple smaller numbers with fewer bits. Several recent studies have focused on optimizing the architecture of RNS operations [81]. By decreasing the bit width of the operands at the cost of increasing the number of operations, RNS simplifies the hardware implementation that can be translated to a higher parallelism.

The work in [72] uses RNS to implement the DNNs with resistive processing-in-memory (PIM) technology. They utilize RNS to simplify the operation such that all the DNN operations can be run in RRAM crossbar memory. The work in [82] focuses on using RNS to improve multipliers for neural network applications on FPGA. Works in [78] and [83] use nested RNS to reduce the bit width of numbers so that it maps all the NN operations in FPGA LUTs to increase the efficiency of computations. In our work, we take the advantage of RNS to breakdown the weights and activations so that the costly multiplications can be performed with energy-efficient shift and add operations whilst keeping the accuracy intact.

1.2 Proposed Residue-Net

DNNs consist of convolutional (CNV), activation function (AF), fully connected (FC), and pooling layers. Residue-Net first converts the incoming inputs to RNS. The rest of the intermediate computations are carried out in RNS format. The moduli set for Residue-Net is $\{2^t-1, 2^t, 2^t+1\}$; for t equal to 2, the moduli set becomes $\{3,4,5\}$. Considering the RNS with the selected moduli set, valid values for R_3 (residue of dividing by 3) are $\{0,1,2\}$, and valid values for R_4 and R_5 are $\{0,1,2,3\}$ and $\{0,1,2,3,4\}$, respectively. To represent R_3 and R_4,

Residue-Net requires two bits, while to represent R_5 it requires three bits. Therefore, Residue-Net converts six-bit numbers and operations into two two-bit and one three-bit numbers and operations. In the following, we first introduce the required operations involved in neural networks, and then we propose the Residue-Net architecture to accelerate DNNs.

1.2.1 RNS Operations

RNS can uniquely represent binary numbers in the range of $D = \prod_{i=1}^{k} M_i$ (as each residue R_i can take i different values). Therefore, any binary number $\in [0, D)$ can be represented uniquely as $\{R_{M_1}, R_{M_2}, ..., R_{M_k}\}$ where R_{M_k} is $|X|_{M_k}$, i.e., X mod M_k. A six-bit binary number $x[5:0]$ can be expanded as $x = x[5:4] \times 2^4 + x[3:2] \times 2^2 + x[1:0] \times 2^0$. We therefore can obtain residues of R_3 following Equation (1.1).

$$\begin{aligned} R_3 = |x|_3 &= |x[5:4] \times 16 + x[3:2] \times 4 + x[1:0] \times 1|_3 \\ &= x[5:4] \times |16|_3 + x[3:2] \times |4|_3 + x[1:0] \times |1|_3 \\ &= x[5:4] + x[3:2] + x[1:0] \end{aligned} \tag{1.1}$$

Analogously, for R_4, and R_5 we have:

$$R_4 = |x|_4 = x[1:0] \tag{1.2a}$$

$$R_5 = |x|_5 = x[5:4] - x[3:2] + x[1:0] \tag{1.2b}$$

Note that to calculate $R_3 = x[5:4] + x[3:2] + x[1:0]$, if the result of the sum is greater than the modulus 3, we subtract the modulus from the result of the addition. R_4 is equal to the two least significant bits $x[1:0]$, and R_5 is calculated in a similar fashion of calculating the R_3, though the middle addition is replaced by subtraction. Therefore, Residue-Net requires two three-port adders for the forward conversion. To convert the RNS numbers back to the binary system, we

use a look-up table that maps every RNS number to its corresponding binary number.

Addition and Multiplication: Upon converting binary numbers to RNS, Residue-Net executes the neural network operations in RNS, where the majority of operations in convolution and fully-connected layers are addition and multiplication. Multiplication and addition in RNS are similar to the binary operations. To multiply (add) two RNS numbers, their corresponding residues are multiplied (added); if the result is greater than the modulus, the final residue is computed by subtracting the modulus, as represented by Equation (1.3) and (1.4).

$$
\begin{aligned}
&(A+B)_{RNS} = \{|A+B|_3, |A+B|_4, |A+B|_5\} \\
&= \{|A|_3 + |B|_3, |A|_4 + |B|_4, |A|_5 + |B|_5\} \\
&= \{|R_{3,A} + R_{3,B}|_3, |R_{4,A} + R_{4,B}|_4, |R_{5,A} + R_{5,B}|_5\}
\end{aligned}
\tag{1.3}
$$

$$
\begin{aligned}
&(A \times B)_{RNS} = \{|A \times B|_3, |A \times B|_4, |A \times B|_5\} \\
&= \{|A|_3 \times |B|_3, |A|_4 \times |B|_4, |A|_5 \times |B|_5\} \\
&= \{|R_1^A \times R_1^B|_3, |R_2^A \times R_2^B|_4, |R_3^A \times R_3^B|_5\}
\end{aligned}
\tag{1.4}
$$

As alluded before, considering the selected moduli set, RNS numbers will be in the set $\{[0,1],[0,1,2],[0,1,2,3]\}$. Thus, Residue-Net simplifies the multiplication to shift and/or add operation. Multiplication by 0 results in constant 0 output. In multiplying by 1, the output simply gets the input value. Multiplication by 2 and 4 are realized by shift operations. Residue-Net computes multiplication by 3 by adding the input with to its left-shifted (i.e., $3x = 2x + x$).

Figure 1.2 demonstrates how a Residue-Net multiplication module works. First, if the inputs are not in RNS, Residue-Net converts them to RNS using the forward conversion (FWC) formulated earlier. Each multiplexer covers all possible scenarios of the weight parameter (which is also in RNS format) and inputs/activation values. As shown in the figure, Residue-

Net only utilizes a two-bit and a three-bit adder plus three multiplexers to implement a six-bit multiplication. The output of multiplications in RNS format can also be larger than the moduli set, which needs to be converted back to the right range. For this, at the end of computations, Residue-Net uses a *Ranging Block* (RB) to calculate the residue in case the output of multiplexers are greater than the corresponding modulus.

The example of Figure 1.2 computes 7×4 in RNS. Residue-Net first transforms the input 7 to $\{R_3 = 1, R_4 = 3, R_5 = 2\}$ (if the input is an activation, i.e., outputs of intermediate layers, it will be already in RNS format). All weight parameters of the model are converted to RNS format offline for once. Here, the $w = 4$ is represented as $\{R_3 = 1, R_4 = 0, R_5 = 4\}$. Residues of the weights are connected to the *select* port of the multiplexers. The output of the multiplexers are, expectedly, $\{R_3 = 1 \times 1 = 1, R_4 = 3 \times 0 = 0, R_5 = 2 \times 4 = 8\}$. Since $R_5 = 8$ is greater than 5, the ranging blocks eventually convert the output to $\{1, 0, 3\}$, which matches with representing $7 \times 4 = 28$ in RNS format: $|28|_3 = 1$, $|28|_4 = 0$, and $|28|_5 = 3$. We note that the ranging blocks in Figure 1.2 are just for demonstration purposes. As explained in the following subsection, we share a single RB block for the entire adder-tree (that sums up the output of several multiplications) by deferring RB operations after the summations.

Comparison: Comparison is another operation required to execute DNNs, mainly used in pooling and activation layers. Comparison cannot be directly performed in RNS format by simply comparing the residues [84]. Instead of converting the RNS numbers back to the binary number system to perform the comparison, Residue-Net uses mathematical attributes of RNS for this purpose. For a number in our RNS representation with residues $\{R_3, R_4, R_5\}$, we define α as the smallest number that has the same residue for R_3 and R_5. For instance, for an arbitrary number with $R_3 = 0$ and $R_5 = 1$, we see $\alpha = 6$ as $|6|_3 = 0 = R_3$, and $|6|_5 = 1 = R_5$. Any number smaller than $\alpha = 6$ does not hold this property. Such a number (α) will have an arbitrary R_4 of β, i.e., $|\alpha|_4 = \beta$. That being said, if we multiply $x = \{R_3, R_4, R_5\}_{\text{RNS}}$ by $M_1 \times M_3 = 3 \times 5 = 15$, we observe $|15x|_3 = 0$, $|15x|_4 = |-x|_4 = -R_4$, and $|15x|_5 = 0$ as well. With these insights, we

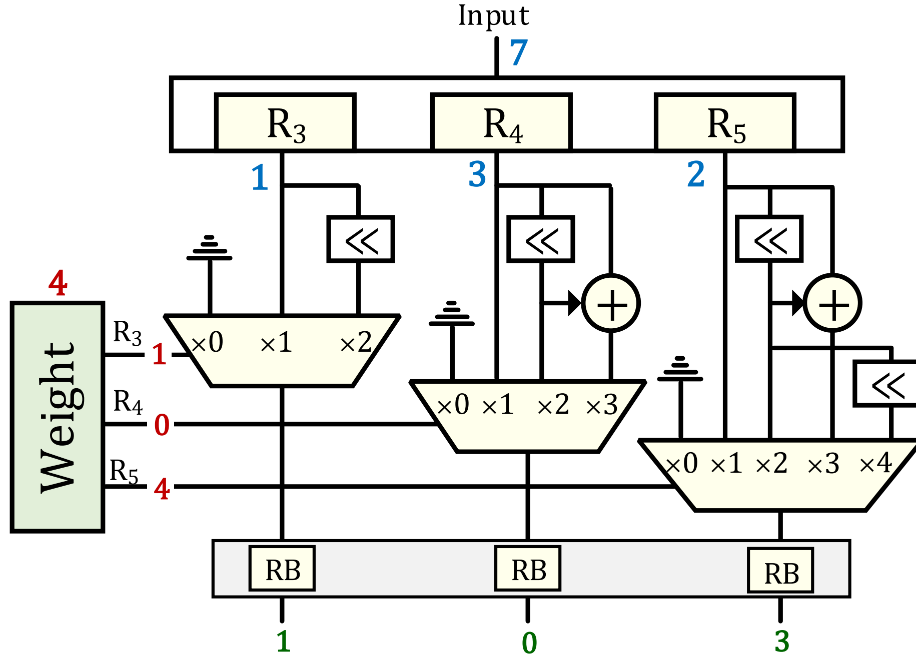

Figure 1.2. The RNS multiplication unit of Residue-Net.

can represent $x = \{R_3, R_4, R_5\}$ in RNS as follows.

$$x \equiv |\beta - R_4| \times 15 + \alpha \tag{1.5a}$$

$$\rightarrow |x|_3 = |(|\beta - R_4| \times 15)|_3 + |\alpha|_3 = 0 + R_3 = R_3 \tag{1.5b}$$

$$\rightarrow |x|_4 = |(|\beta - R_4| \times 15)|_4 + |\alpha|_4 = (R_4 - \beta) + \beta = R_4 \tag{1.5c}$$

$$\rightarrow |x|_5 = |(|\beta - R_4| \times 15)|_5 + |\alpha|_5 = 0 + R_5 = R_5 \tag{1.5d}$$

We use this property to compare two binary numbers $x' = |\beta_x - R_{4,x}| \times 15 + \alpha_x$ and $y' = |\beta_y - R_{4,y}| \times 15 + \alpha_y$ that have the same RNS representation of x and y: we can simply compare $|\beta_x - R_{4,x}|$ and $|\beta_y - R_{4,y}|$ to compare the actual values x and y. If these two terms are

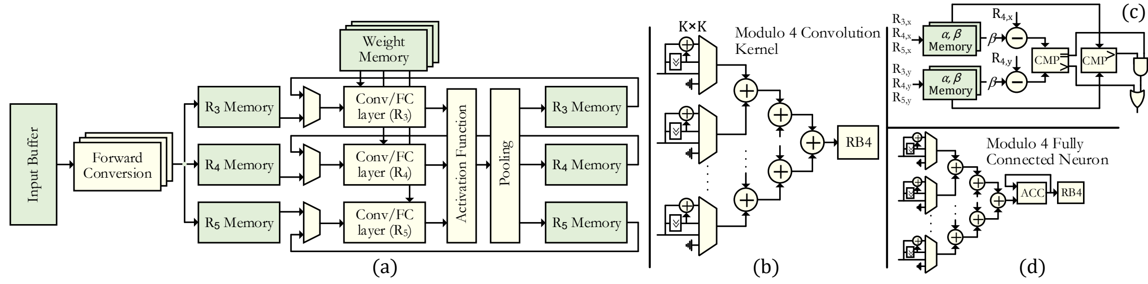

Figure 1.3. (a) The proposed architecture for Residue-Net. (b) A convolutional kernel that performs computations on the inputs and weights R_4. (c) Residue-Net proposed comparator module. (d) A fully connected neuron working in inputs and weights R_4.

equal, we then use the comparison of α_x and α_y. To avoid computational complexity, for every possible value of x, Residue-Net stores α_x and β_x in a memory. Note that binary x has six bits, so the memory footprint is trivial. The $R_{4,x}$ (i.e., $|x|_4$) is ready beforehand as the operands passed through the layers are all in RNS format.

We showed that Residue-Net can execute the neural network operations entirely in RNS representation. After performing all the computations, Residue-Net converts the final results back to the binary system using the backward conversion module explained above for the final soft-max operations. Since the binary number has a limited bit-width of six, Residue-Net transforms each RNS number to the corresponding binary number using a small lookup table.

1.2.2 Residue-Net Architecture

The architecture and dataflow of Residue-Net is illustrated in Figure 3.9(a). The network is trained and weights are quantized to 6-bit binary values offline. The weights are converted to RNS and transferred to FPGA DRAM. Since the computation in RNS is performed on each residue independently, Residue-Net stores the weights residues in three separate memory blocks to simplify the memory access pattern of different residues. During the execution of the network, Residue-Net transfers the weights to FPGA local BRAMs which have considerably faster access latency than the off-chip memory (DRAM). Residue-Net uses a weight stationary dataflow to load the weights. Weights of each kernel remain stationary in the FPGA BRAM to maximize the

kernel reuse. Once a convolutional kernel weights are fetched into BRAMs, all the computations that use those weights are executed.

Residue-Net transfers the primary inputs to the *Input Buffer* which is connected to forward conversion (*FWC*) modules. FWC modules calculate the residues as explained in Equation (1.1) and (1.2) for multiple input features in parallel. R_1, R_2, and R_3 of the converted inputs are stored in three separate memory blocks, one for each residue. Residue-Net comprises the required modules for DNN inference, convolutional, fully connected, ReLu activation function, pooling layers, memory blocks, scheduler, and controller. Multiplication and add operations can be carried out independently for each of the residues, however, an RNS comparison demands all the three representing residues. Thus, Residue-Net executes the computations of CNV and FC layers for each residue in parallel while ensuring that results of all residues become ready simultaneously to pass to the next layer. Residue-Net makes three instances of both convolution and fully connected layer, whereby each instance performs computations on the corresponding residue. As executing activation function and pooling layers require all the three residues, only one instance of these layers is present in the architecture. Note that, in Figure 3.9 (b)/(d), only the convolutional kernel and fully connected neuron of the residue R_4 is illustrated. Computations on the other two residues are performed in a module with the same architecture but slightly different in the primary arithmetic modules (see Figure 1.2).

The scheduler of Residue-Net moves primary input data to the input buffer as well as fetches the required weights into the weight memory blocks. Thereafter, based on the network architecture, it initiates different layers sequentially. Convolutional layers might consist of different numbers of convolutional kernels with different shapes. Thus, the scheduler applies the layer configurations to the CNV layers in the runtime. Since each layer may not be able to accomplish the computations in a single cycle, the controller generates the memory access addresses for the layer's inputs and weights. For the first layer, the controller reads the RNS inputs from the first array of residue memories and writes the intermediate results into the second array of the residue memories. The controller also issues hand-shaking signals when the

computations of a layer are accomplished. To execute the next layer, instead of reading the inputs from the first array, the controller reads the layer's inputs from the second memory array and writes the results into the first array. That is, the controller switches the memories alternatively for every other layer; a layer reads from one memory and writes to the other memory.

Convolutional layers consist of single or multiple convolutional kernels that multiply the inputs by the kernel weights and accumulate the results. The convolutional kernels move along the inputs to generate outputs. Figure 3.9(b) depicts the convolutional kernel of Residue-Net that operates on the R_4 residues. The convolutional of the kernel weights and the inputs are carried out in *Residue-Net multiplier* (see Figure 1.2). Since the values of R_4s are smaller than 4, Residue-Net simplifies the multiplication to selecting among the pre-calculated multiplied values. The number of multiplications in a $K \times K$ convolutional kernel is equal to the size of the kernel, i.e., $K \times K$. Residue-Net aggregates the multiplication outputs in a pipelined adder-tree. The result of the accumulation of the multiplied inputs and weights may be greater than the modulus. Therefore, as mentioned earlier, Residue-Net uses a ranging block to convert the output back to valid RNS representation. In the RNS multiplier shown in Figure 1.2, a ranging block is used immediately after the multiplication. However, in intermediate computations, temporary variables do not need to be in a *valid* RNS representation Therefore, Residue-Net utilizes only a single ranging block (here, mod 4) after computing the final value of convolutional kernel rather than using a ranging block after every operation. To fully utilize the available FPGA resources, Residue-Net instantiates multiple convolutional kernels within the convolutional layer.

In DNNs, the output of the layers passes through a non-linear activation function. Rectified Linear Unit (ReLU) activation function is the most widely used activation function layer. To implement the ReLU activation function, Residue-Net needs to perform the comparison using the procedure explained in Section 1.2.1. Figure 3.9(c) shows the Residue-Net comparator. To compare two RNS numbers, R_3, and R_5 of both inputs are connected to a lookup table to the values of α and β. Residue-Net comparator calculates the difference of R_4 of each input and their corresponding β; the one with the larger difference is greater. For equal differences, the number

with the greater α is larger. To implement the ReLU activation function, Residue-Net uses the implemented comparator with one input fixed. Residue-Net also uses the RNS comparator to implement max/min pooling layers.

In fully connected layers, similar to convolutional layers, multiple neurons are executing simultaneously. The architecture of each neuron is illustrated in Figure 3.9(d). Since the number of inputs in different fully connected layers varies, Residue-Net uses a generic architecture, which parallelizes the computation by instantiating multiple neurons, where each neuron multiplies and accumulates multiple inputs. Each neuron multiplies the outputs of the previous layer by the corresponding weights and accumulates the results. A series of Residue-Net multipliers multiply the weights by the inputs. The results are aggregated in a pipelined adder-tree and an accumulator adds up the intermediate results of consecutive iterations. The controller issues the memory access signals and reads all the inputs of the layer (P inputs per cycle), and at the end, converts the accumulated RNS result to *valid* RNS representation.

1.3 Experimental Result

1.3.1 Experimental Setup

Residue-Net migrates the neural network execution to the RNS domain to reduce the complexity of DNNs without losing the accuracy of the quantized network. To evaluate the efficiency of Residue-Net, we implemented four common network architectures, LeNet, Alexnet, VGG16, and ResNet50 on FPGA. Table 1.1 compares the Top-1 accuracy of Residue-Net with that of the 32-bit fixed-point, and networks quantized to 6-bit over four popular DNNs classifying various popular datasets (LeNet on MNIST, AlexNet on ImageNet, VGG16 and ResNet-50 on CIFAR-10). To train and quantize the networks we used the approach proposed in [62]. As presented in the table, Residue-Net provides almost the same accuracy as the full-precision network.

Residue-Net host code is written in OpenCL to convert the quantized weights to RNS.

Table 1.1. Accuracy of the Residue-Net as compared to the full precision and quantized to 6 bits networks.

	LeNet (MNIST)	AlexNet (ImageNet)	VGG16 (CIFAR-10)	ResNet50 (CIFAR-10)
32-bit Fixed-point	99.3%	47.95%	92.44%	93.62%
6-bit Fixed-Point	99.3%	47.95%	92.41%	93.58%
Residue-Net	99.3%	47.94%	92.41%	93.59%

The host code uses the Xilinx Vitis software platform to transfer the primary inputs and RNS weights to FPGA DRAM and invoke the Residue-Net kernel. Residue-Net FPGA kernel is implemented in SystemVerilog HDL and synthesized using Xilinx Vivado Design Suite. The timing and the functionality of the Residue-Net are also verified using Vivado Design Suite. We implemented Residue-Net on Kintex-7 FPGA KC705 Evaluation kit. To estimate the device power we used the built-in Xilinx Power Estimator tool in Vivado. To evaluate the efficiency, we compared Residue-Net with the baseline FPGA implementation. Since the dynamic range of RNS used in Residue-Net can only uniquely represent 6-bit binary numbers, we implemented the network quantized to 6-bit weights and activations in the binary system on FPGA with the same architecture. We used the same architecture for the baseline and Residue-Net to minimize the impact of the accelerator architecture on the evaluations and emphasize the effectiveness of migrating to RNS from the binary system and multiplication-free execution of DNNs. We also compared the efficiency of the building blocks of DNNs to represent the efficiencies and overheads of Residue-Net.

1.3.2 System Evaluation

The baseline FPGA implementation is designed to utilize ~90% of the FPGA available LUTs which are the bottleneck of increasing the parallelism. We took two approaches in selecting the parallelism level of Residue-Net: i) Residue-Net *(area-opt)* targets minimizing the area and power consumption of the network. It provides the same performance as the baseline while reducing the power and area of the network. ii) Residue-Net *(performance-*

opt) targets to fully utilize (∼90%) the FPGA LUTs to improve the performance and energy consumption of the network. Figure 1.4 shows the throughput improvement and energy reduction values of all the four DNNs as compared to the baseline for both *area-opt* and *performance-opt* implementations. Residue-Net (area-opt) provides the same performance as the baseline with less area and consequently lower power and energy consumption. Residue-Net (area-opt) shows 24% energy reduction on average as compared to the baseline. Residue-Net (performance-opt), by increasing the parallelism provides 2.8× higher throughput, with a close area and power consumption to the baseline; thus increasing the energy efficiency by 2.7×. The main efficiency of the Residue-Net comes from the optimized MAC unit, therefore, Residue-Net shows higher performance improvement in networks with a higher number of MAC operations. As the number of MAC operations in LeNet is significantly less than those of the ResNet-50 the performance improvement in LeNet is less than that in ResNet-50 (2.1× as compared to 3.1×). Although Residue-Net requires more memory to store the weights (7 bits as compared to 6 bits in the baseline), since all of Residue-Net implementations are compute-bound, the memory and communication overhead is negligible.

Figure 1.5 shows the LUT, and BRAM utilization of the FPGA (divided by the total available resources), as well as the power, throughput, and energy consumption of both Residue-Net (area-opt) and Residue-Net (performance-opt) as compared to the baseline in AlexNet. Residue-Net (area-opt) shows 33% reduction in the number of required LUTs as compared to the baseline. Although Residue-Net increases the BRAM utilization because of using multiple memory instances to store the RNS weights and activations, this overhead does not affect the performance as BRAMs are not the bottleneck of the design. Residue-Net (area-opt) in AlexNet, is able to reduce the power consumption and consequently the energy consumption of classifying an input by 17%, while delivering the same performance as the baseline. Residue-Net (area-opt) on average, reduces the resource utilization by 36% and power consumption by 21%. Residue-Net (performance-opt), by fully utilizing the FPGA resources, is able to increase the throughput and energy efficiency by 2.9× as compared to the AlexNet quantized to 6 bits. Since CNV and

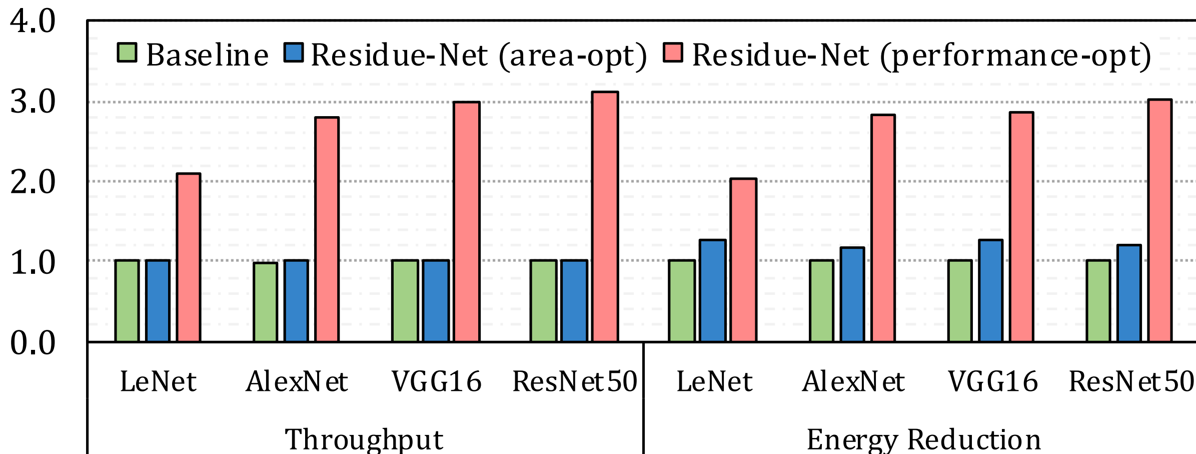

Figure 1.4. Comparing the normalized values of power, throughput and energy consumption of the FPGA baseline, Residue-Net (area-opt), and Residue-Net (performance-opt) w.r.t the FPGA baseline for different DNNs.

FC layers in DNNs contribute to the majority of execution time, Residue-Net (performance-opt) employs the saved resources to increase the parallelism in CNV and FC layers, thereby increasing the performance. Residue-Net (performance-opt) provides higher speed-up and energy reduction due to the efficiency of Residue-Net in executing MAC operations. In the following subsection, we evaluate the efficiency of DNN sub-modules to justify where the efficiency of the system comes from.

1.3.3 Operation Evaluation

Residue-Net tackles the computation complexity of MAC operation and particularly the complexity of the multiplication operation. It has been shown that convolutional layers consume 80% of the execution time [85]. To show the effectiveness of Residue-Net in breaking down the computation complexity of the multiplication operation we evaluate the efficiency of Residue-Net in executing the building blocks of convolutional, fully connected, activation function and pooling layer separately.

Efficiency: The efficiency of DNN building modules is illustrated in Figure 1.6. Figure 1.6(a) shows the LUT utilization, $\frac{LUTs}{Totlal\ LUTs}$, of each module in FPGA. Utilization values not

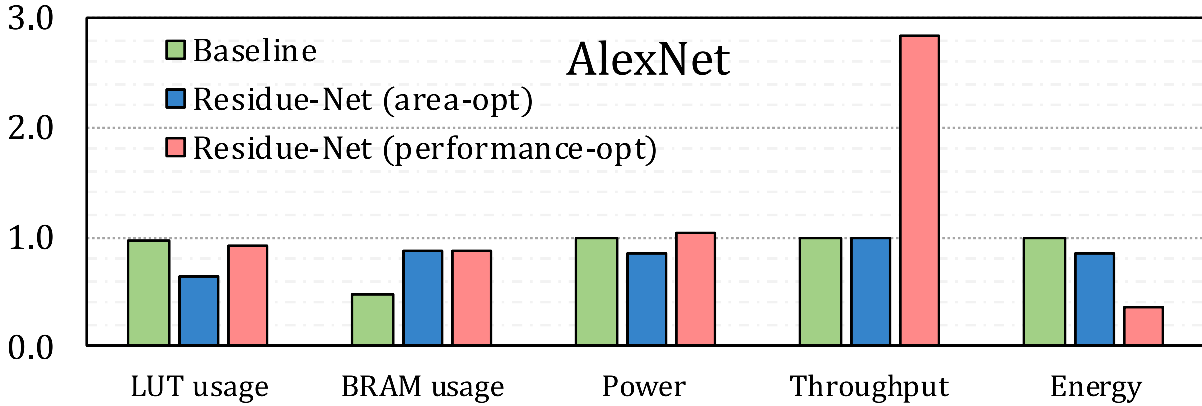

Figure 1.5. Comparing the LUT and BRAM utilization of the FPGA baseline, Residue-Net (area-opt), and Residue-Net (performance-opt), as well as the normalized values of power, throughput and energy consumption w.r.t the FPGA baseline for AlexNet.

only demonstrate the reduction in the area achieved by Residue-Net but also show the relative size of each module as compared to the available FPGA resources. These building modules can be integrated into any DNN accelerator to reduce the area of the accelerators. Residue-Net shows 79.5% area reduction in implementing MAC operations which comprise more than 99% of DNN operations [30]. Residue-Net MAC module also shows 80% power reduction as compared to 6-bit binary MAC operation. In figure 1.6 power and performance are calculated for 100 MAC modules since due to the small size of the MAC module, the power consumption of a single MAC module would be close to 0 and hard to represent. We also evaluate the efficiency of four common convolutional kernels (CNV 3×3, CNV 4×4, CNV 5×5, CNV 11×11). Residue-Net convolutional kernels, on average, show 54.5% area reduction and 59% power reduction as compared to the baseline modules. Consuming most of the execution time of the network, convolutional layers can be considerably accelerated in Residue-Net.

In fully connected layers, a neuron multiplies its inputs by weights and accumulate the results. In Figure 1.6, 16 FC, 64 FC and 128 FC represent a neuron that has 16, 64, and 128 inputs respectively. For instance, 64 FC multiplies 64 inputs by weights and accumulate the results in an adder-tree with 64 inputs. Note that if the inputs of the layer are more than 64, the module calculates the result in multiple iterations. In fully connected neurons, Residue-Net

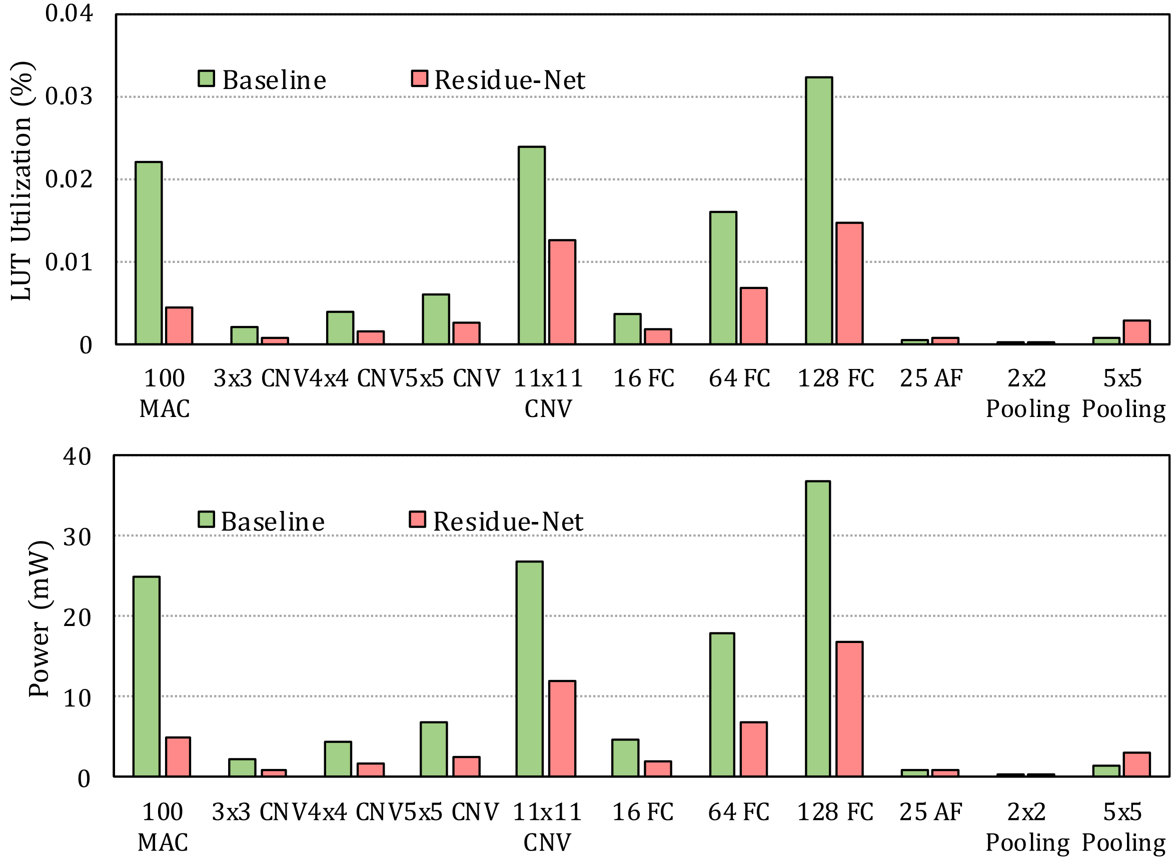

Figure 1.6. LUT utilization and power consumption of Residue-Net building blocks in DNNs.

reduces the area by 53% and power consumption by 57% as compared to the baseline.

Residue-Net shows a significant reduction in computation complexity of both convolutional and fully connected layers which are the most time-consuming parts of DNN execution. However, due to higher complexity in comparison operations in RNS, Residue-Net activation function and pooling layers require more resources than the baseline layers. Residue-Net requires 75% more resources to implement the activation function for 25 inputs than the baseline activation function, 215% more resources for implementing the pooling layer. However, these two sub-modules require significantly fewer resources than the FC and CNV modules, and the Residue-Net area overhead in these two layers will not considerably impact the effectiveness of Residue-Net.

Overhead: Table 1.2 shows the LUT overhead of Residue-Net as compared to the

Table 1.2. LUT overhead of using Residue-Net in executing DNNs.

Forward Conversion	Backward Conversion	Ranging Block	Comparator
5	12	9	15

baseline. First, Residue-Net needs to convert inputs to RNS using forward conversion modules, each requires 5 LUTs. Also, after executing the network, Residue-Net converts the results back to the binary system using backward conversion modules. The backward conversion module is a table with all the possible binary numbers and maps RNS numbers to a unique binary number, which requires 12 LUTs to implement. After performing MAC operations, Residue-Net should calculate the residues using a ranging block that needs 9 LUTs. AF and pooling layers require comparison operators, each of which needs 15 more LUTs in Residue-Net as compared to the same operation in the binary system. Considering the limited number of each of these modules in the entire system, the overhead of Residue-Net will be negligible in the efficiency of the accelerator.

1.4 Conclusion

In this chapter, we proposed Residue-Net, a multiplication-free DNN accelerator that uses the residue number system. Residue-Net transforms a pretrained and 6-bit quantized DNN model to RNS representation using $\{3, 4, 5\}$ as the moduli set. It breaks down the 6-bit operations into smaller operations. It also replaces the complex multiplications with energy-efficient shift and addition operations. Residue-Net on average, shows 36% and 21% reduction in area and power respectively as compared to the baseline FPGA implementation with the same throughput when executing widely-used DNNs (LeNet, AlexNet, VGG16, and ResNet50). Residue-Net, with the same area as the baseline implementation, shows $2.8\times$ speedup on average (up to $3.1\times$ in ResNet-50), while delivering the same accuracy as the 6-bit quantized network.

Even though implementing DNNs on FPGAs and replacing multiplication operations with simpler operations increases the performance, many applications can be processed with

much simpler machine learning algorithms at significantly higher performance and lower energy consumption. In the next chapter we introduce hyperdimensional (HD) computing as an alternative light-weight and hardware-friendly solution for many machine learning applications. It provides comparable accuracy as the conventional machine learning algorithms while being highly parallizable in hardware.

This chapter contains material from “Residue-Net: Multiplication-free Neural Network by In-situ No-loss Migration to Residue Number Systems”, by Sahand Salamat, Sumiran Shubhi, Behnam Khaleghi, and Tajana S. Rosing, which appears in IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2021 [3]. The dissertation author was the primary investigator and author of this paper.

Chapter 2

Accelerating HD computing on FPGAs

Hyperdimensional (HD) computing is a novel computational paradigm that emulates the brain functionality in performing cognitive tasks. HD provides close or even better accuracy (as shown in table 1.1) in a wide range of applications compared to conventional machine learning algorithms with significantly lower computational complexity. The underlying computation of HD involves a substantial number of element-wise operations (e.g., addition and multiplications) on ultra-wide hypervectors, in the granularities of as small as a single bit, which can be effectively parallelized. In addition, though different HD applications might vary in terms of number of input features and output classes (labels), they generally follow the same computation flow. Such characteristics of HD computing inimitably matches with the intrinsic capabilities of FPGAs, making these devices a unique solution for accelerating these applications.

In this chapter, we propose HD2FPGA, an automated tool to generate fast and highly efficient FPGA-based accelerator for HD classification and clustering. It eliminates the arduous task of handcrafted designing of hardware accelerators by automatically generating an FPGA implementation of HD accelerator leveraging a template of optimized processing elements, according to the applications specification and user's constraint. HD2FPGA generates a synthesizable Verilog implementation of HD accelerator while taking the high-level user and target FPGA parameters into account. It is flexible and highly optimized to deliver fast and energy efficient accelerator according to the user-specified constraints (viz., performance, and power). It

supports end-to-end training, retraining and inference for both HD classification and clustering. Specifically, this chapter makes the following contributions:

- Develops a publicly available tool [86] that generates FPGA-based synthesizable architectures for accelerating training, retraining, inference for HD classification and clustering.
- Utilizes random projection encoding to deliver high accuracy while removing the dependency of the encoding hardware to the number of input feature levels. We show that HD2FPGA is able to deliver high accuracy for a wide variety of applications.
- Supports training, retraining, inference, and online model refinement using the developed template-based framework.
- Enables easy access for researchers and industry to implement classification and clustering on FPGAs with orders of magnitude speed-up and energy reduction, compared to CPU, with a push of a button by using HD2FPGA's Graphical User Interface.
- Evaluates the efficiency of HD2FPGA classification and clustering using different benchmarks compared to running the CPU-based baseline on an Intel Core-I7 CPU.

2.1 Background and Related Work

In this section, we first articulate the operations behind HD computing, including encoding, training, inference, and retraining for classification and clustering. Afterward, we review the previous work regarding the utilization and implementation of the HD computing.

2.1.1 Hyperdimensional Computing

HD computing builds on the fact that the cognitive tasks of the human brain can be explained by mathematical operations on ultra-wide hypervectors [36]. In other words, the brain computes with patterns of neural activity, which can be better represented by hypervectors rather than scalar numbers. A hypervector comprises $\mathscr{D}_{hv}$, e.g., 10,000 bits, independent components

(dimensions) whereby the enclosed information is distributed uniformly among all $\mathscr{D}_{hv}$ dimensions. This makes hypervectors robust to failure as the system remains functional under a certain number of component failings, and as degradation of information does not depend on the position of the failing components [87, 38, 88].

Encoding: Several encoding methods have been proposed to map input data into hypervectors. Random projection [89], permutation-based [47, 90, 91], n-gram [92], and ID-vector [91] encoding methods have been used in literature, among which the random projection method works well on wider variety of data sets and can be efficiently implemented on FPGA. Therefore, in HD2FPGA we utilize the random projection encoding for classification and clustering. Suppose the input data is $X = \{x_1, x_2, \ldots, x_{D_{iv}}\}$, where D_{iv} is the number of input features in the input vector and each feature can be quantized to L unique levels. Table 2.1 shows the accuracy of each encoding method on various edge sensing applications. We quantify the number of multiplication and accumulation (MAC) operations as well as the size of the required memory for encoding an input with D_{iv} input features quantized to L levels. Additionally, we compare the accuracy of HD computing with deep neural networks (DNNs) multi-layer perceptron (MLP) and random forest as conventional machine learning algorithms. As illustrated in the table, the random projection encoding provides a comparable accuracy compared to other conventional machine learning algorithms for a wide range of applications. The accuracy of the random projection, ID-level and permutation-based encoding are relatively close. However, implementing ID-level and permutation-based encodings on hardware is impossible as they require a huge amount of memory when full-precision (32-bit) inputs. Implementing these two encoding methods on FPGA is possible when the input features are quantized to 8 levels (3-bit) [47]; nevertheless, quantizing the input features to 8 levels drops the accuracy of the ID-level and permutation-based encodings for 9.1% and 10.2% respectively. Equation 2.1 shows the random projection encoding.

$$\vec{H} = \sum_{i=1}^{j=d_{iv}} \vec{P}_i \times x_i \tag{2.1}$$

In this encoding, the input vector is multiplied by a projection matrix consists of a set of D_{iv} binary vectors P_i. The P matrix consists of D_{iv} columns of projection hypervectors (P_i). Each P_i consists of D_{hv} randomly generated binary values.

$$\mathrm{P}_i = \{p_j\}_{j=1}^{j=D_{hv}},\ p_j \in \{0,1\} \tag{2.2}$$

HD encoding maps inputs to hypervectors and involves the following steps. First, it initializes projection hypervectors, each of which corresponds to a specific input feature level. Indeed, input of the HD algorithm is a feature vector $\vec{V}_{iv}$ with D_{iv} dimensions (elements) wherein each dimension represents a feature value. According to Table 2.1, the size of the required memory in the ID-Level is proportional to the number of features and number of quantization levels ($\mathscr{O}(D_{iv}L)$). The permutation-based encoding, used in [47], memory requirement is in the order of $\mathscr{O}(L)$, while the memory requirement in the random projection encoding is $\mathscr{O}(D_{iv}$. Input features can be quantized to simplify the encoding step. The ID-Level and Permutation-based encoding are more susceptible to the number of input features levels (i.e. input bit-width), as there should be a unique hypervector for each level. However, in the random projection encoding the input bit-width only affects the operations bit-width. In many applications, reducing the quantization levels significantly affect the accuracy, as shown in table 2.1. Therefore, unlike the random projection encoding, both ID-Level and permutation-based encoding methods are unable to support 32-bit input features, which is needed in some applications to provide the utmost accuracy on FPGA due to the limited on-chip memory needed to store the base hypervectors. Moreover, in HD2FPGA we reduce the memory requirement to $\mathscr{O}(1)$ by generating the required projection hypervectors on the fly.

After loading the projection hypervectors, the projection matrix is getting multiplied by the input features to generate the encoded hypervector. Each dimension of the encoded hypervector can be generated independent of the others. The element j^{th} of the encoded hypervector can be generated by multiplying the jth row of the projection matrix (D_{iv} element)

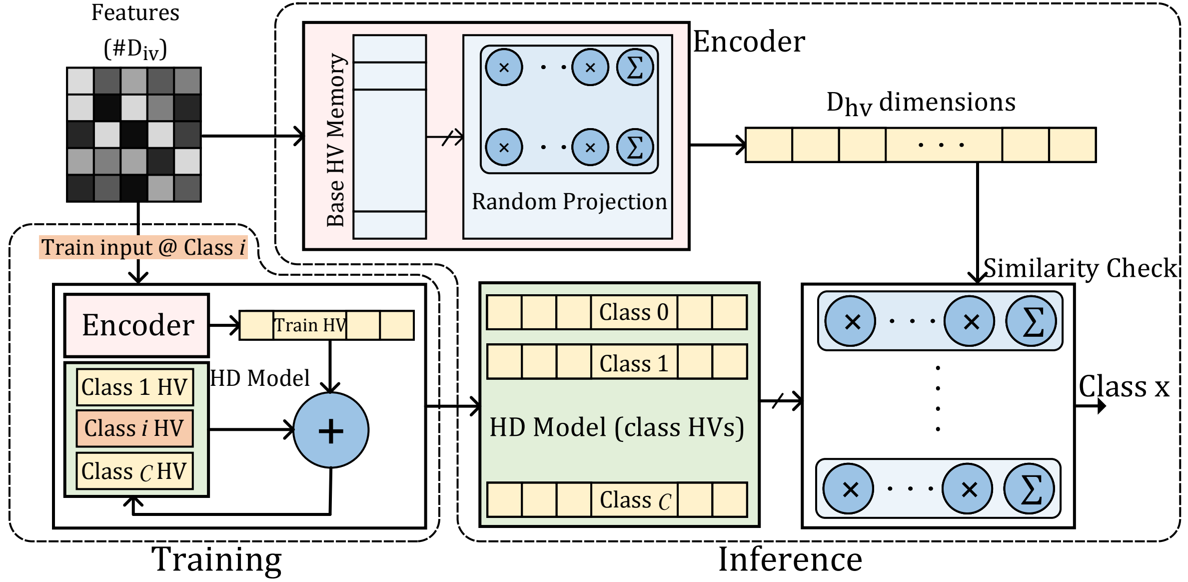

Figure 2.1. Overview of hyperdimensional learning and inference.

Table 2.1. Comparing the accuracy and memory requirement of each encoding method in various edge sensing applications.

	Cardio	Face	ISOLET	UCIHAR	MNIST	Page	PAMAP2	Average	Memory
Random Projection	84.9%	95.3%	93.5%	94.1%	93.5%	96.2%	90%	92.5%	O(Div)
ID-Level	88.1%	95.6%	93.2%	94.6%	89.4%	91.6%	94.6%	92.4%	O(DivL)
ID-Level 3-bit	84.3%	95.6%	93.2%	94.1%	86.6%	91.4%	37.9%	83.3%	O(DivL)
Permutation-based 32-bit	88.2%	96.1%	93.5%	94.7%	89.3%	91.7%	95.8%	92.7%	O(DivL)
Permutation-based 3-bit	84.6%	95.8%	93.2%	94.4%	86.4%	91.4%	31.9%	82.2%	O(DivL)
MLP	86.4%	95.5%	95%	94.6%	96.7%	96.5%	92.9%	93.9%	
Linear Regression	86.4%	92.3%	94%	93.1%	92.6%	96%	92.9%	92.4%	
Random Forest	95.3%	92.5%	92.2%	95.6%	96%	97.4%	95.6%	94.9%	

by the input vector. Therefore, generating different dimensions of the encoded hypervector can be parallelized which makes FPGAs with intrinsically high parallelism a great platform for executing HD encoding.

Training: After mapping each input $\vec{V}_{iv}$ to hypervector $\vec{H}$ as above, all hypervectors belonging to the same class (label) are simply summed to form the final representative hypervectors. Thus, assuming $\vec{H}^l = \langle h_0, h_1, \cdots, h_{d_{hv}} \rangle^l$ denotes a generated class hypervector for an input data with label l, the final (representative) class hypervectors are obtained as Equation 2.3, in which each dimension c_j is obtained through dimension-wise addition of all h^l_js, and $\mathscr{K}$ is the number

of input data with label l.

$$\vec{C}_l = \langle c_0, c_1, \cdots, c_{d_{hv}} \rangle = \sum_{k=0}^{\mathscr{K}} \vec{H}_k^l \tag{2.3}$$

Inference: The first steps of inference in HD computing is similar to training; an input feature vector is encoded to $\mathscr{D}_{hv}-$dimension query hypervector, as explained in the encoding step. This is followed by a similarity check between the query hypervector $\vec{\mathscr{H}}$ and all representative class hypervectors, $\vec{\mathscr{C}_l}$. The similarity is defined as calculating the cosine similarity, which is obtained by multiplying each dimension in the query vector to the corresponding dimension of the class hypervectors, and adding up the partial products:

$$\text{similarity}(\vec{\mathscr{H}}, \vec{\mathscr{C}_l}) = \sum_{j=0}^{\mathscr{D}_{hv}} h_k \cdot c_k \tag{2.4}$$

The class with the highest similarity with the query hypervector indicates the classification result.

Retraining: Retraining might be used to enhance the model accuracy by calibrating it either via new training data or by multiple iterations on the same training data. Retraining is basically done by removing the mispredicted query hypervectors from the mispredicted class and adding it to the right class. Thus, for a new input feature vector $\vec{V}_{in}$ with query hypervector $\vec{\mathscr{H}}$ belonging actually to class with hypervector $\vec{\mathscr{C}_l}$, if the current model predicts the class $\mathscr{C}_{l'}$ where $\mathscr{C}_{l'} \neq \mathscr{C}_l$, the model updates itself as follows:

$$\begin{aligned} \vec{\mathscr{C}_l} &= \vec{\mathscr{C}_l} + \vec{\mathscr{H}} \\ \vec{\mathscr{C}_{l'}} &= \vec{\mathscr{C}_{l'}} - \vec{\mathscr{H}} \end{aligned} \tag{2.5}$$

This, indeed, reduces the similarity between $\vec{\mathscr{H}}$ and mispredicted class $\mathscr{C}_{l'}$, and adds $\vec{\mathscr{H}}$ to the correct class $\mathscr{C}_l$ to increase their similarity and the model will be able to correctly classify such query hypervectors.

Clustering: Similar to HD classification, in HD clustering every input needs to be encoded first. In HD clustering, the centroid hypervectors are initialized by hypervector of

randomly selected inputs. HD clustering keeps 2 copies of HD model (centroid hypervectors). It uses one for finding the closest centroid to the current input, and use the other model to update the clustering centroids for the next iteration. During each iteration of clustering, it finds the similarity between the encoded hypervector and every centroid and the cluster with the maximum similarity is the result of clustering (similar to classification). After finding the closest cluster, the input is used to update the copy of HD model. The encoded input is added to the centroid hypervector representing the cluster. After processing all the input in the dataset, the HD model is being replaced by the updated model for the next iteration. The number of iterations is a parameter defined by the user.

2.1.2 Related Studies

HD computing is gaining traction as an alternative solution to perform cognitive tasks in a light-weight fashion that uses significantly simpler operations compared to conventional machine learning techniques that deal with complex learning procedures with substantial number of costly operations. So far, HD computing has been widely used for a wide range of classification [93], clustering [94], recommendation systems [45] to name a few. Language identification [95], DNA sequencing [96], physical activity prediction [40, 97], speech recognition [41, 98], gesture recognition [99, 100], EEG signal classification [101, 102], robotics [103, 104] are a few example of HD applications.

On par with studies investigating the HD applications, several studies have attempted to propose application-specific accelerators (ASIC [105, 106, 100, 107], FPGA [108, 87, 90, 91, 47, 109, 98]) and algorithmic solutions to enhance the efficacy of HD computing. The study in [87] proposes logical operations to generate the hypervector corresponding to each feature on the fly, in order to reduce the costly BRAM accesses. They also propose an approximate majority gate to compose the binary class hypervectors without requiring to hold the summation on hypervector components in a multi-bit format in the course of training. This is, however, limited to low-accuracy binarized HD computing wherein each dimension of the query and class hypervectors

is one bit. The authors of [110] propose a hierarchical HD computing solution that consists of the main stage with multiple classifiers each can trade between efficiency and accuracy. There is also a decider stage that learns and selects the appropriate encoder within the main stage based on a so-called difficulty metric of the input data. The work in [90] exploits computational reuse to reduce the computation complexity of HD classification. It proposed an FPGA-based architecture to execute HD classification on FPGA. It reuses the previously encoded hypervector to encode the current input to simplify the encoding step. It additionally, clusters class hypervectors dimensions to reduce the number of multiplications in the associative search step. The work in [91] proposes approximate encoding modules to significantly accelerate the encoding step. They approximate and optimize the encoding operations based on the characteristics of FPGA resources.

Multiple works leverage advances of emerging technologies in HD computing [92, 111, 112, 113, 114, 115]. The work in [92] presented a complete in-memory platform for executing both encoding and associative steps of HD computing on memristive devices. the developed platform supports hypervectors with different dimensionality and input datasets with different numbers of input features and classes. Thanks to the intrinsic robustness of HD computing to noise, the work in [92] approximates the mathematical operations to further accelerate HD computing. In [111], the authors leverage CNT-FET and Resistive RAM to fabricate an end-to-end HD computing solution. They exploit the variations in RRAM resistance and CNT-FET drives current to project the input features to query hypervectors as well as propose an approximate accumulation circuit using gradual RRAM reset operation. The work in [112] demonstrates HD computing with 3D vertical RRAM in-memory kernels capable of performing multiplication, addition, and permutation by analog operations on RRAM cells. The work in [114] exploits stochastic computing to execute HD computing operations in memory. The proposed architecture introduces stochastic operation on HD vectors which can be easily parallelized in memory.

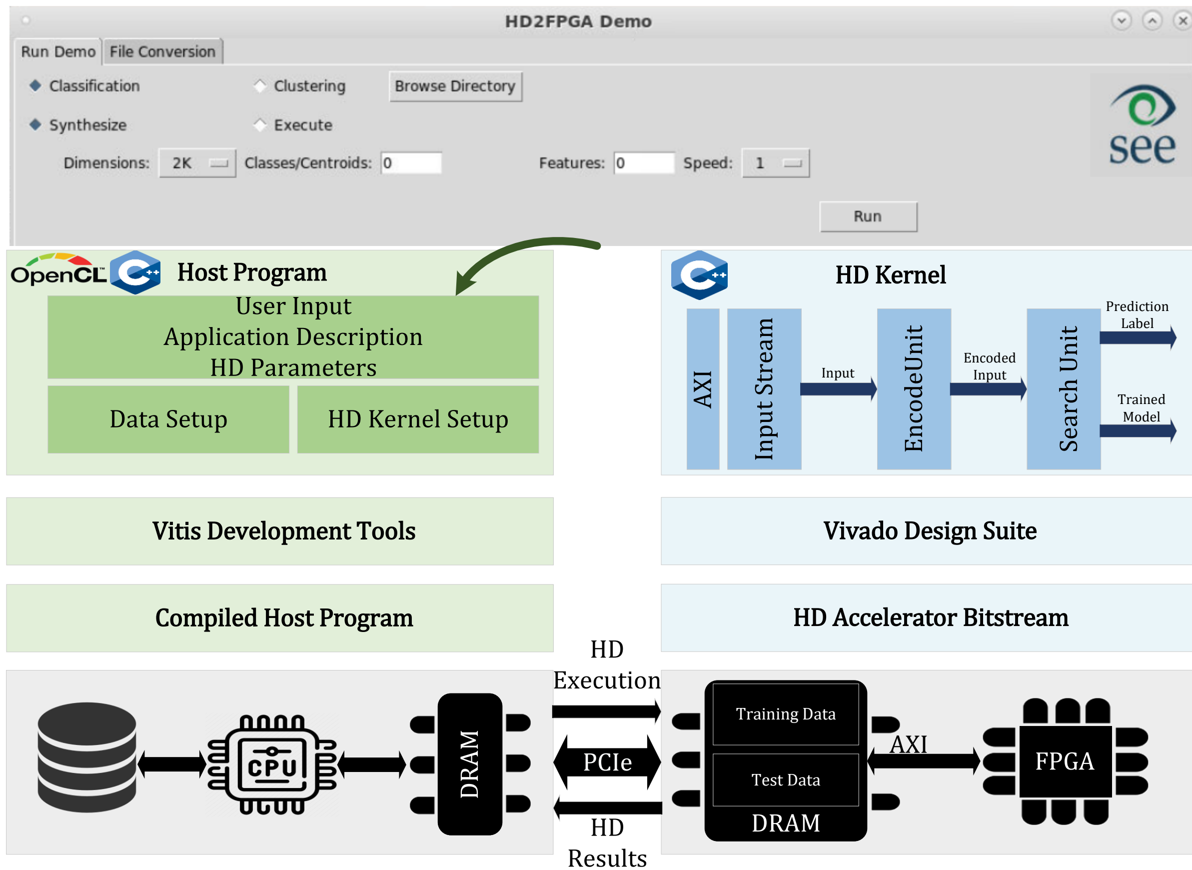

Figure 2.2. Overview of the proposed framework, HD2FPGA.

2.2 HD2FPGA Framework Overview

HD2FPGA aims to abstract away the complexities behind employing FPGAs for accelerating AI applications. HD2FPGA is an automated tool to accelerate the design time of an FPGA-based HD classification and clustering considering the user-specified criteria, e.g., power budget, performance-accuracy trade-off, and FPGA model (available resources). The overview of HD2FPGA is illustrated in Figure 2.2. HD2FPGA can be split into two parts: the host program and the HD kernel. The host program runs on the host CPU and is responsible for transferring data to the FPGA off-chip memory, initiate the HD kernel, and read the results from the FPGA. It, first, loads the dataset from the storage. A part of the dataset is for training

and retraining the HD model while the rest is used to evaluate the accuracy of the model. It uses the training data to train the HD model. As explained in the previous section, multiple iterations of retraining is needed to adjust and increase the accuracy of the HD model. After training the HD model, HD2FPGA model can be used to classify/cluster unseen data. HD2FPGA gets the application description and HD parameters from the user and automatically generates an FPGA-based accelerator, called HD kernel, for the user's application. The HD kernel runs on FPGA and supports end-to-end HD execution, including encoding, training, retraining, and inference of HD classification and clustering.

2.3 HD2FPGA Architecture

In this section, we articulate the contributions of HD2FPGA in more detail. We begin by elaborating on the proposed encoding scheme that reduces BRAM usage. Afterward, we illustrate the architecture overview and detail the functionality and structure of the building blocks in the course of training, retraining, and inference of both classification and clustering. We also formulate the required resources of each module, based on the HD parameters.

HD2FPGA automatically generates the FPGA-based accelerator for user's HD classification and clustering applications. HD2FPGA gets the HD parameters (e.g., hypervector lengths, number of input features, number of classes/clusters) and generates an accelerator based on users' inputs. HD2FPGA consists of three main modules (InputStream, EncodeUnit, and SearchUnit) to execute all the operations in both HD classification and clustering. Details of each module are disccussed in the following.

InputStream: It reads the inputs from the FPGA memory through AXI bus interfaces and writes them into parallel buffers as illustrated in Figure 2.3. The AXI interface bit width is set to 32 and since the host program quantizes the dataset to 32-bit numbers, the HD kernel reads one feature per transaction. The number of features for each input is a parameter (D_{iv}) set by the user, during the synthesis. The InputStream module continuously reads the D_{iv} input features

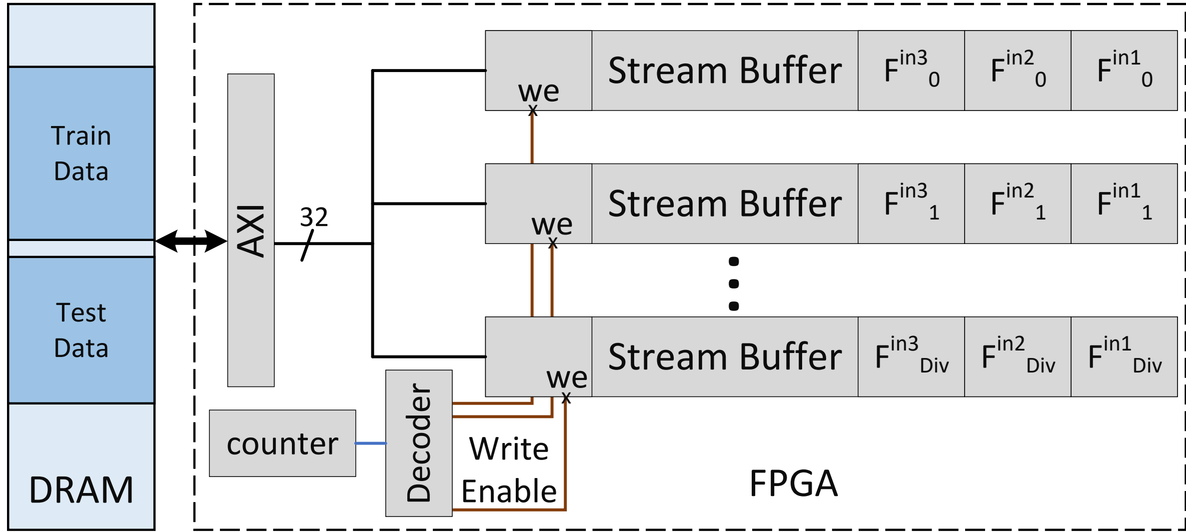

Figure 2.3. The architecture of the InputStream module.

and writes them into parallel stream buffers, implemented as First-In-First-Out (FIFO) buffers, as shown in Figure 2.3. Features belonging to an input vector can be read in parallel as they are written in different stream buffers. F_i^j, in the figure, represents the ith feature of the jth input. The encoding module needs to read multiple input features in each cycle, therefore, parallel buffers provide simultaneous access to all input features.

EncodeUnit: It reads the input features from the stream buffers and encodes the input to a hypervector with D_{hv} dimensions (defined by the user during the runtime). HD2FPGA utilizes random projection encoding, as it is more efficient for edge devices and provides higher accuracy in most of the applications. The random projection encoding is shown by the following equation.

$$H=\sum_{k=1}^{D_{iv}} P_k \times v_k \tag{2.6}$$

Here $V_{iv} =< v1, , v_{D_{iv}} >$ is the input vector and v_k shows a quantized feature element, while P_k is the kth projection hypervector. The projection hypervectors are randomly generated hypervectors for each input feature. Projection hypervectors are frequently being used to generate the encoded hypervectors. Thus, they should be stored in FPGA on-chip memory (BRAMs

and/or URAMs) to maximize performance. However, the size of the FPGA on-chip memory is limited, and storing a hypervector for each input feature requires multiple megabytes of on-chip memory which is higher than what is available on small FPGAs. To make HD2FPGA compatible with more FPGA family types and specifically smaller FPGAs used in edge devices, the HD kernel only stores a single seed hypervector (SeedHV), and by applying permutations, it creates the projection hypervectors on the fly. To permute the seed hypervector, HD2FPGA uses hardware-friendly shift operation as shown in the following equation to generate the projection hypervectors.

$$\mathrm{P}_k = SeedHV << k \tag{2.7}$$

To generate the projection hypervector corresponding to the kth feature, HD2FPGA applies k rotational shifts on the seed hypervector. As the encoding operations are regular, HD2FPGA does not need to store the projection hypervector, so it generates the required dimensions of the projection hypervectors on the fly. The random projection encoding by using the seed hypervector is shown in the equation below.

$$\mathrm{H}{=}\sum_{k=1}^{D_{iv}} (seedHV << k) \times v_k \tag{2.8}$$

Figure 2.4(a) shows the matrix multiplication operations for the random projection encoding. The projection hypervector matrix in the figure shows how the projection hypervectors are achieved from the seed hypervector. B_i is the ith dimension of the seed hypervector and column P_i is the projection hypervector of the ith feature. P_1 directly uses the seed hypervector, and the other projection hypervectors are conducted by permuting P_1. HD2FPGA breaks down the encoding operations to generate the encoded hypervector in multiple cycles to be compatible with a wider range of FPGA devices with different amounts of resources. HD2FPGA is able

to increase or decrease the parallelization based on the users' desired performance and the available FPGA resources. It accelerates HD operations by parallelizing the HD operations at the dimension level. Each dimension of the encoded hypervector can be generated independent of the others. The similarity metrics between dimensions of the encoded hypervector and class hypervectors can also be calculated independent of the other dimensions. Therefore, there is a trade-off between the parallelization and consequently the resource utilization of HD2FPGA and the performance of HD2FPGA. HD2FPGA uses two parameters to adjust the parallelization of HD. It breaks down the matrix multiplication computation using a sliding window that covers C elements of the input features and generates R dimensions of the encoded hypervector in $\frac{D_{iv}}{C}$ cycles in parallel.

At each cycle, HD2FPGA multiplies a window (i.e. several columns of a few rows) of the projection binary vectors with the corresponding portion of the features column. Thus, each vector-vector multiplication generates a portion of a dimension of the encoded hypervector. In the next cycle, the window slides to the next portion of the same rows to generate the remaining partials of the encoding dimensions. As the window slides over rows, it also slides on the features column so we are always multiplying elements of column k with the element k of the feature column. This procedure accumulates the partial results of the same encoding dimensions over several cycles, without moving the partial accumulation results. Thus, we have a fixed architecture consisting of several pipelined tree-adders for vector-vector multiplications in which we just need to feed the proper dimensions (bits) of the projection hypervectors and feature values to tree-adders, in a deterministic sequence, as shown in Figure 2.4(b).

The EncodeUnit, instead of storing all the projection hypervectors, generates the dimensions of the projection hypervectors that are needed in every clock cycle by reading a window of the seed hypervector and performing shift operations on the sliding window. Since shift operations in hardware are implemented by simply modifying the wires connections, The EncodeUnit generates the required dimensions of the projections hypervectors, on the fly, by reading a window with fixed wirings that handle shift operations. To explain how the EncodeUnit

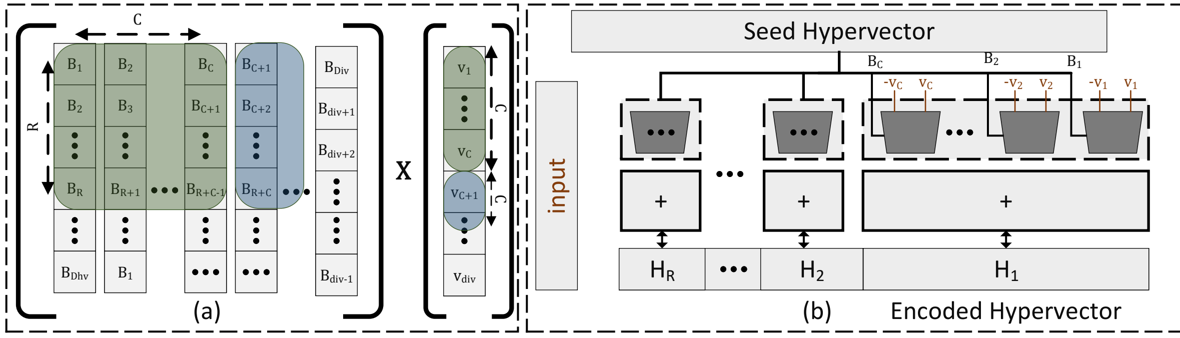

Figure 2.4. (a) The matrix multiplication of the projection hypervectors and the input feature vector. (b) HD2FPGA architecture based on random-projection encoding

generates the dimensions of the projection hypervectors needed in the matrix multiplication, Figure 2.5 shows a simplified example of the matrix multiplication where $D_{hv} = 16$ (dimensionality of hypervectors), $D_{iv} = 8$ (number of input features), $C = 2$, and $R = 4$. As illustrated in the figure, every $\frac{D_{iv}}{C} = \frac{8}{2} = 4$ cycles 4 dimensions of the encoded hypervector is generated. The encodeUnit, in the first cycle, multiplies the $[[B_0, B_1], [B_1, B_2], [B_2, B_3], [B_3, B_4]]$ matrix by the input vector $[v_0, v_1]$ to generate the intermediate values for the first 4 dimensions of the encoded hypervector. Therefore, $\{B_0, B_1, B_2, B_3, B_4\}$ elements of the seed hypervector (B_i is the ith element of the seed hypervector) are required to execute the intermediate values for $\{H_0, H_1, H_2, H_3\}$ dimensions (H_i is the ith dimension of the encoded hypervector). In the second cycle, $\{B_2, B_3, B_4, B_5, B_6\}$ and $\{F_1, F_2\}$ are required to update the values of $\{H_0, H_1, H_2, H_3\}$. All the execution steps of the encoding is illustrated in Figure 2.5. At the end of the fourth cycle, H_0, H_1, H_2, and H_3 are generated.

Generally, in each cycle, $C + R - 1$ elements of the seed hypervector are needed for the partial matrix multiplication. However, to simplify the accesses to the seed hypervector memory, the EncodeUnit reads a window of $2 \times R$ elements of the seed hypervector. After each cycle, it shifts the window for R elements. In cycles 1 through cycle 4 the following set of seed hypervector elements are needed:

$clk_1 = \{B_0, B_1, B_2, B_3, B_4\}$

$clk_2 = \{B_2,\ B_3,\ B_4,\ B_5,\ B_6\}$

$clk_3 = \{B_4,\ B_5,\ B_6,\ B_7,\ B_8\}$

$clk_4 = \{B_6,\ B_7,\ B_8,\ B_9,\ B_{10}\}$

Note that at the end of the fourth cycle, the dimensions $\{H_0, H_1, H_2, H_3\}$ of the encoded hypervector have been generated. The EncodeUnit reads the first 8 elements of the seed hypervector in the first cycle, then, in the second cycle, it moves the window for two steps to cover elements $\{B_2,\ B_3,\ \ldots,\ B_8\}$. In every cycle, the window has the required elements for the next $\frac{R}{C}$ cycles. This prefetching mechanism helps the synthesis tool to schedule the operations more efficiently. The generated dimensions of the encoded hypervector are written into a memory, which is used for the following HD steps, including training and associative search. The HD kernel stores the entire encoded hypervector for the retraining step in the off-chip memory to avoid re-encoding each input during the multiple retraining epochs. During the retraining steps, the encoded hypervectors are fetched to adjust the HD model for any misprediction. Thus, in the retraining steps, encoding is not executed from scratch anymore.

SearchUnit: It executes HD training, inference, and retraining. If the user loads a trained HD model, SearchUnit loads the trained class hypervectors from the FPGA memory. Otherwise, the class hypervectors are initialized with zeros, and the SearchUnit updates the HD model during the training. To train the model, it adds the encoded hypervectors to the class hypervector they belong to. The training is usually followed by multiple retraining iterations. The number of retraining iterations is empirically set by the user during the runtime such that the difference between the accuracy of the HD model between two consecutive iterations is less than the user's desired threshold (the model converges). In the retraining step, the predicted label is compared to the original label of the data. In case of misprediction, SearchUnit adds the encoded hypervector to the class hypervector it belongs to and subtracts the encoded hypervector from the predicted class. At the end of retraining, SearchUnit writes the trained HD model into the FPGA memory. To perform HD inference, SearchUnit calculates the cosine similarity between the encoded hypervector and the class hypervectors. It multiplies the generated dimensions of the

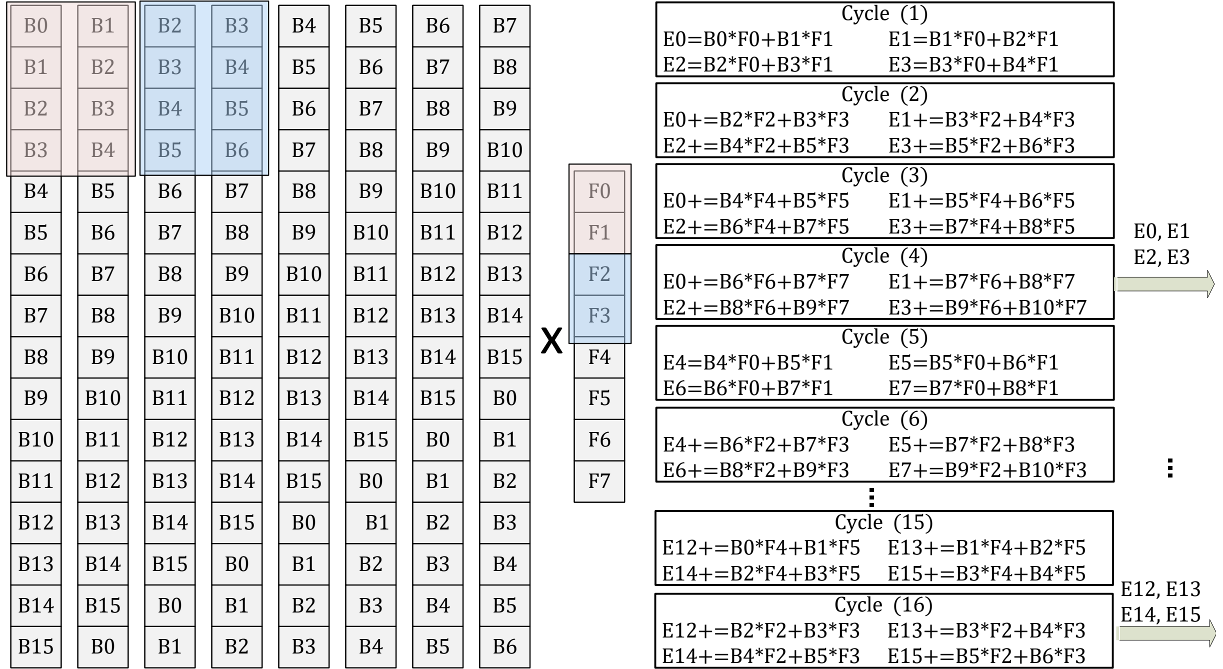

Figure 2.5. The partial matrix multiplication example when $D_{hv} = 16$, $D_{iv} = 8$, $C = 2$, and $R = 4$.

encoded hypervector by the corresponding elements of the class hypervectors and accumulates the multiplied results. After processing all the dimensions of the encoded hypervector, the class with the highest similarity metric is the prediction result. During the inference step, the predicted label is written into the FPGA off-chip memory.

HD2FPGA classification and clustering kernels share the same EncodeUnit and InputStream modules. The only difference is in the SearchUnit, as illustrated in Figure 2.6. In retraining the classification model, the HD kernel performs the associative search and finds the

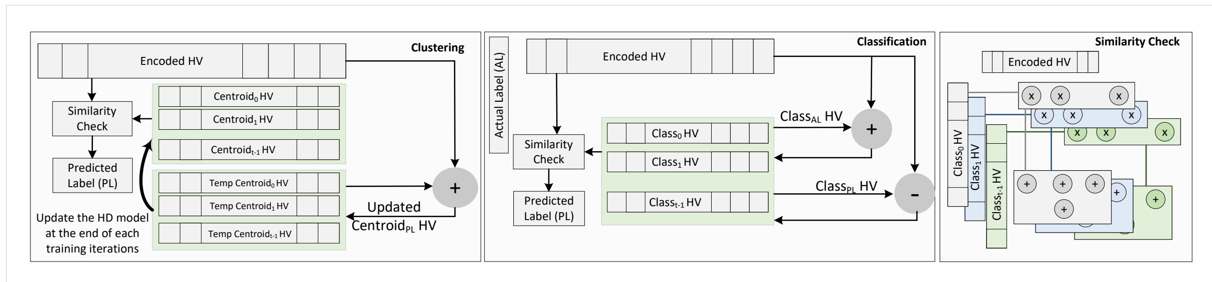

Figure 2.6. The proposed architecture of the Search unit for (a) HD Classification and (b) HD clustering.

most similar class to the encoded hypervector and compares the predicted label with the actual label, and in case of misprediction, it adds the encoded hypervector to the class hypervector of the actual label and subtracts the encoded hypervector from the mispredicted class. However, in training the clustering HD model, we do not use the actual labels. The HD clustering kernel initializes each centroid hypervector with the encoded hypervector of a randomly selected input. Thereafter, it calculates the similarity between each encoded hypervector and all the centroid hypervectors and finds the most similar cluster (predicted centroid). The clustering SearchUnit also keeps a copy of the clustering HD model (temp Centroids), and it adds the encoded hypervectors to the predicted centroid of the temporary model. At the end of each retraining iteration, the SearchUnit replaces the HD clustering model with the temporary model. HD2FPGA clustering kernel, similar to the classification kernel, encodes the input in the first iteration of clustering and stores the encoded hypervectors into the FPGA off-chip DRAM. In the next iterations, it only reads the encoded hypervectors from FPGA DRAM to avoid the costly encoding step, thereby increasing the performance of clustering.

2.3.1 HD2FPGA software program

HD2FPGA is equipped with a graphical user interface (GUI) for users to enter the application specification and HD characteristics which have been shown in Figure 2.2. The GUI gets the user parameters and passes them to the host program. The host program has been developed based on the Vitis platform [116], which is the Xilinx software development environment. The Vitis software platform integrates FPGA-based accelerators with the software applications thanks to the Vitis core development kit and Xilinx Runtime (XRT) library [117]. HD2FPGA utilizes the Vitis development flow to develop the FPGA-based accelerator for HD classification and clustering using standard programming languages for both software and hardware components. HD2FPGA is compatible with Intel FPGAs as the Intel Quartus toolchain provides a similar platform to synthesize C++ kernels on Intel FPGAs, and a software development kit to communicate with the FPGA kernels in the host program [118].

As shown in Figure 2.2, execution of HD2FPGA is split between a host program and the HD kernel with a communication channel between them. The host program, written in C++ and using OpenCL APIs, runs on the host processor, while the HD kernel runs on the FPGA. The API calls, managed by XRT, are used to process transactions between the host program and the HD kernel through the PCIe bus between the host CPU and the FPGA. These communications include transferring the control signals to/from the HD kernel as well as transferring the dataset from the host CPU to the FPGA global memory (DRAM). The host memory is only accessible to the host CPU while the FPGA global memory is accessible by both the host processor and FPGA kernels, therefore, the host is responsible for transferring the dataset to the FPGA DRAM and read the results from the FPGA DRAM upon kernel completion. Since the host CPU is responsible for orchestrating the data transfer and kernel initiation, HD2FPGA has no limitation in the size of the dataset. For instance, if the dataset size is greater than the available FPGA's off-chip memory, the host splits the data into the chunks that fit into the FPGA DRAM, sends the commands to the FPGA to process the chunk of the data, and then transfer the other chunks. This is possible since HD training, retraining, and inference are independent of the other chunks. Execution of HD2FPGA host program can be divided into three steps:

Data setup: the host program reads the dataset in the compressed format, quantizes the features if necessary, allocates the corresponding space in the FPGA global memory, and transfers the data to the FPGA global memory through the PCIe bus. The host program also generates the seed projection hypervector which is a randomly generated binary hypervector with the length of D_{hv}. The seed hypervector will be used in the HD kernel for the encoding step to produce all other projection hypervectors on the fly. A user can also load an already trained HD model, for which the host program can read the trained class hypervectors from a file specified by the user and execute the inference with the loaded classes.

HD kernel setup: The host program sets up the kernel with its input parameters as well as pointers to the data in the FPGA memory. The input parameters to the HD kernel are the

number of data inputs, the length of the hypervectors, and the HD task (training, retraining, and inference).

HD execution: The host program triggers the execution of the HD kernel on the FPGA. The HD kernel performs the required computations while reading the input data from the FPGA memory and writes the results back to the FPGA memory. When the HD kernel trains the HD model, it also writes the trained class hypervectors to global memory. In HD training and retraining, the FPGA kernel reads the training data labels, while in inference, the HD kernel writes the prediction results.

HD results: The HD kernel, upon compilation, notifies the host that it has completed the task. The host program measures the execution time from initiating the kernel to the task completion. The host program reads the resulting data (e.g. prediction results and/or trained class hypervectors) back from FPGA global memory into the host memory.

2.4 Experimental Results

HD2FPGA is a flexible framework for efficient implementation of different HD computing applications in FPGA hardware, respecting the application specifications and user's requirements. HD2FPGA is equipped with a GUI written in `Python` which gets the input parameters from the user and generates the header files for both host program and HD kernel. the host program has been implemented in `OpenCL` and executes on CPU. HD2FPGA GUI also synthesize and generates the FPGA bitstream for the HD kernel of the user's application. The HD kernel is written in `C++` and optimized to deliver high performance. HD kernel is synthesized using the Vivado High-Level Synthesis tool (HLS) and integrated with the host code using Xilinx Vitis Accel 2019.2 and is running on U280 FPGA board. To measure the performance of the end-to-end execution of HD classification and clustering on FPGA, we used OpenCL event profiling. We report end-to-end execution times, including reading the data from the FPGA DRAM, executing HD on FPGA and writing the results back to the FPGA DRAM. To evaluate

the energy efficiency of HD2FPGA, we measure the power consumption of the FPGA, including its off-chip DRAM using the API provided by Xilinx XRT.

We compare the performance and energy efficiency of HD2FPGA running on FPGA with Intel i7 7600 CPU with 16GB memory. We evaluate HD on a wide range of benchmark datasets. These datasets cover a broad spectrum of signal classification tasks commonly encountered in edge-sensing applications ranging from human activity detection to text recognition. The publicly available datasets we use are:

Medical Diagnosis (CARDIO): This dataset provides medical diagnosis based on information about each patient. The training and testing datasets are taken from the Cardiotocography dataset.

Gesture Recognition (EMG): Here we try to recognize five different hand gestures (rested hand, closed hand, open hand, 2-finger pinch, and point index) using EMG data. The gestures were sampled at 500Hz with the use of an elastic band containing four EMG sensors. We used the data collected from five different subjects. The data was collected by each subject performing 10 repetitions of each gesture for three seconds each with a three second resting period in between. Therefore, each sample contains 6,000 data points.

Face Recognition (FACE): We exploit Caltech's 10,000 web faces dataset. Negative training images, i.e., non-face images, are selected from CIFAR-100 and Pascal VOS 2012 datasets.

Speech Recognition (ISOLET): Recognize voice audio of the 26 letters of the English alphabet. The training and testing datasets are taken from the Isolet dataset. This dataset consists of 150 subjects speaking each letter of the alphabet twice. The speakers are grouped into sets of 30 speakers. The training of hypervectors is performed on Isolet 1,2,3,4, and tested on Isolet 5.

Activity Recognition (UCIHAR): Detect human activity based on 3-axial linear acceleration and 3-axial angular velocity that has been captured at a constant rate of 50Hz. The training and testing datasets are taken from the Human Activity Recognition dataset. This dataset contains 10,299 samples each with 561 attributes.

We evaluate HD clustering on the ISOLET dataset and FCPS, the fundamental clustering problem suite, which have been widely used in the literature. FCPS offers a variety of clustering problems

that any algorithm shall be able to handle when facing real world data. We also evaluate HD clustering on the speech and the pattern recognition datasets. The publicly available datasets we use are:

FCPS Hepta: The Hepta data set, which is part of the FCPS, is used to illustrate the general problems with quality measures (QMs) and projections from the perspective of structure preservation. The three-dimensional Hepta data set consists of seven clusters that are clearly separated by distance, one of which has a much higher density.

FCPS Tetra: The Tetra data set, which is part of the FCPS, consists of 400 data points in four clusters in R2 that have large intra-cluster distances. The clusters are nearly touching each other, resulting in low inter-cluster distances. FCPS TwoDiamonds: The data consists of two clusters of two-dimensional points. Inside each "diamond" the values for each data point were drawn independently from uniform distributions". The clusters contain 300 points each.

FCPS WingNut: The Wing Nut dataset (FCPS) consists of two symmetric data subsets of 500 points each. Each of these subsets is an overlay of equally spaced points with a lattice distance of 0.2 and random points with a growing density in one corner. The data sets are mirrored and shifted such that the gap between the subsets is larger than 0.3.

Pattern Recognition (Iris): The data set consists of 50 samples from each of three species of Iris flower. Four features were measured from each sample: the length and the width of the sepals and petals. One class is linearly separable from the other 2; the latter are not linearly separable from each other.

2.4.1 HD Classification

Encoding

Encoding module is used in both training and inference. The encoder module works in a pipeline stage with the SearchUnit module. Thus, the more generated dimensions by the encoding module, the more throughput HD2FPGA can achieve. To evaluate the effectiveness

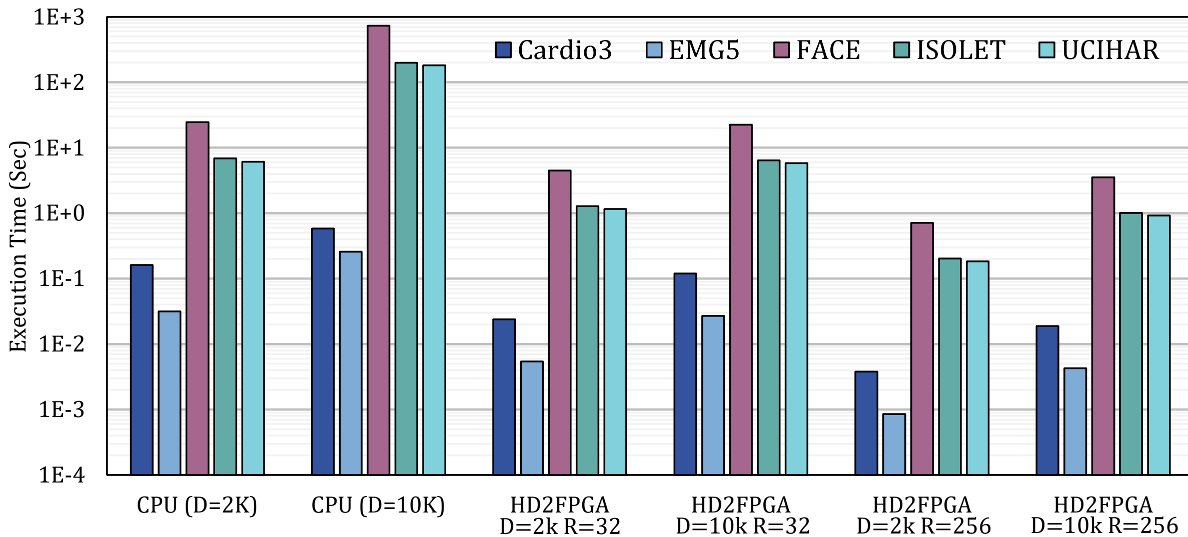

Figure 2.7. Encoding time of CPU and HD2FPGA.

of our proposed EncodeUnit module, we compare the hardware implementation of HD2FPGA encoding with a baseline HD computing encoding running on CPU. Figure 2.7 compares the execution time of encoding the entire datasets on CPU and FPGA. The complexity of the encoding step is propotional to the number of input features, the length of hypervectors (D_{hv}) and the size of the dataset (number of samples). We show results for $D_{hv} = 2K$ and $D_{hv} = 10K$ with $C = 32$ and two different parallelization levels ($R = 32$ and $R = 256$). HD2FPGA encoding's performance is linearly proportional to D_{hv} since the number of generated dimensions of the encoded hypervector is fixed. However, as illustrated in the figure, the encoding time in CPU is not linearly proportional with the dimensionality. For instance, in ISOLET, encoding with $D_{hv} = 10K$ is 29× slower on CPU relative to $D_{hv} = 2K$. According to Figure 2.7, for $D_{hv} = 2K$, the FPGA implementation with $R = 32$ and $R = 256$ parallel rows improve the performance by 16.5×, and 127.9×, respectively. For $D_{hv} = 10K$, these numbers increase to 76.2× ($R = 32$) and 564.5× ($R = 256$) as the CPU's performance diminishes significantly for larger D_{hv}.

Retraining

We store and reuse the encoded hypervectors during retraining to increase efficiency in both CPU and FPGA-based implementations. Similar to encoding, the performance of FPGA is linearly

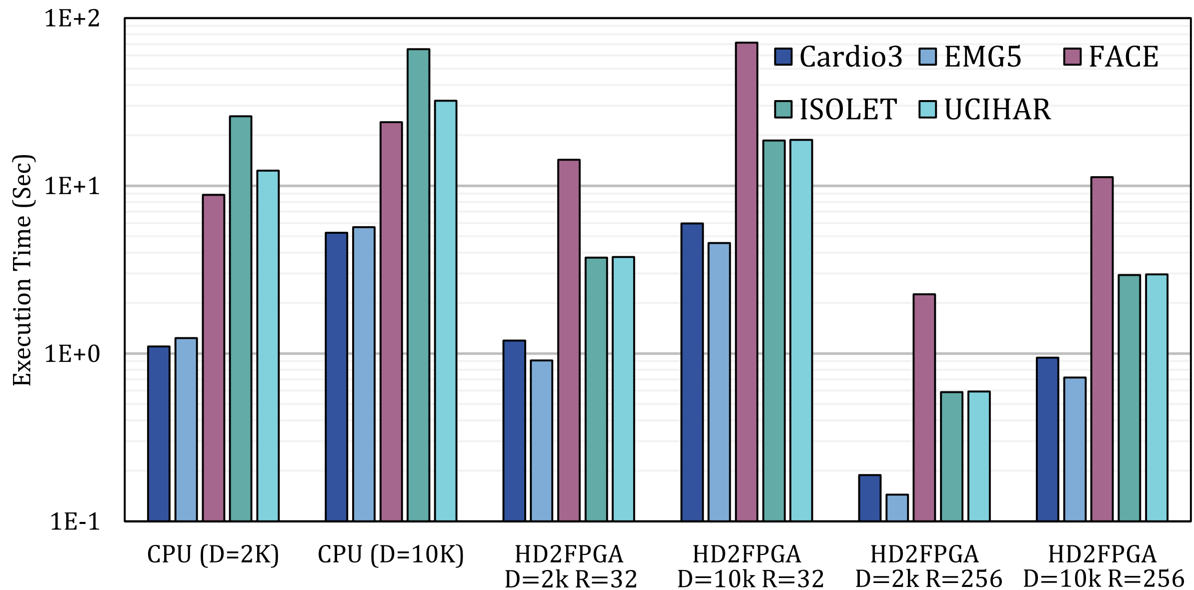

Figure 2.8. Retraining time of CPU and FPGA for 50 epochs.

dependent on the dimensionality. Figure 2.8 shows the retraining time of HD2FPGA compared to CPU baseline for 50 epochs. Based on the figure, for $D_{hv} = 2K$, FPGA's retraining time is 1.2× and 8.8× faster than CPU with $R = 32$ and $R = 256$, respectively. However, unlike encoding, the CPU's relative performance improves for larger dimensionality as the time of fetching the encoded hypervectors is amortized by parallel search among the classes as well as having higher DRAM bandwidth compared to FPGA resulting in relatively lower improvements to FPGA's relative performance is 0.8× ($R = 32$) and 5.6× ($R = 256$) for $D_{hv} = 10K$.

End-to-End Training

In Figure 2.9, we combine the encoding, initial training step and the retraining times of CPU and FPGA to obtain the total training time (including the encoding and 50 retraining epochs). Accordingly, we observe that HD2FPGA provides 1.8× (with $R = 32$) and 12.9× (with $R = 256$) speed-up for hypervectors with $D_{hv} = 2K$ dimensions. For $D_{hv} = 10K$, we achieve 4.2× ($R = 32$) and 30.5× ($R = 256$) speed-up compared to CPU baseline. By moving from $D_{hv} = 2K$ to $D_{hv} = 10K$, the relative efficiency of FPGA in encoding increases but it decreases for retraining. However, since the encoding time dominates the retraining time of the CPU, in $D_{hv} = 10K$

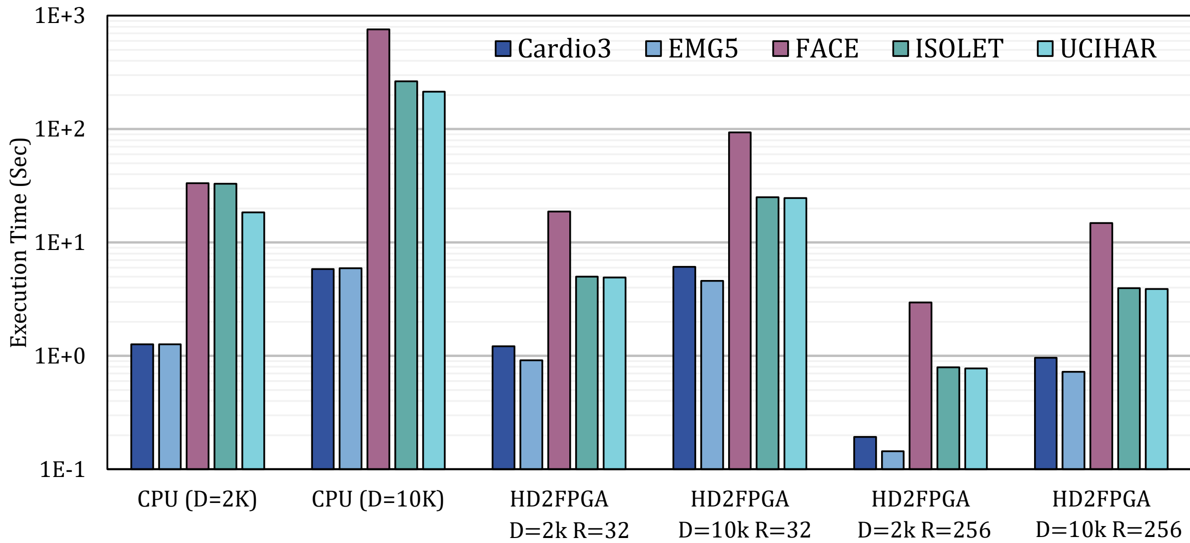

Figure 2.9. End-to-end Training (encoding + 50 epochs retraining) time of CPU and HD2FPGA.

HD2FPGA achieves higher improvement for the training time. To evaluate the energy efficiency of HD2FPGA compared to the CPU baseline, we measure the power consumption of HD2FPGA and CPU during the runtime. Regardless of the benchmark, we observe a power of $\sim 65W$ for CPU encoding, and $\sim 40W$ for its retraining. Figure 2.10 shows the training energy for CPU and FPGA, comprising the sum of encoding and 50 epochs of retraining energy. HD2FPGA achieves $4.3\times$ $(R = 32)$ and $25.0\times$ $(R = 256)$ training energy efficiency for $D_{hv} = 2K$. With $D_{hv} = 10K$, the energy efficiency increases to $12.8\times$ $(R = 32)$ and $73.9\times$ $(R = 256)$. In ISOLET dataset, HD2FPGA consumes $\sim 20W$ for $R = 32$ configuration, and $\sim 25W$ for $R = 256$. The difference between the power consumption of these two configurations is not proportional to their resource utilization due to the high static power consumption of the U280 FPGA board.

Figure 36. Training energy of CPU and FPGA. D is the dimensionality, and R is the number of parallel rows in the window of the FPGA's matrix-vector multiplication unit.

Inference

Figure 2.11 compares the performance of HD inference in HD2FPGA compared to CPU. The inference includes the encoding and search steps for HD classification. The encoding step dominates the execution time, so we expect a similar improvement we observed for encoding.

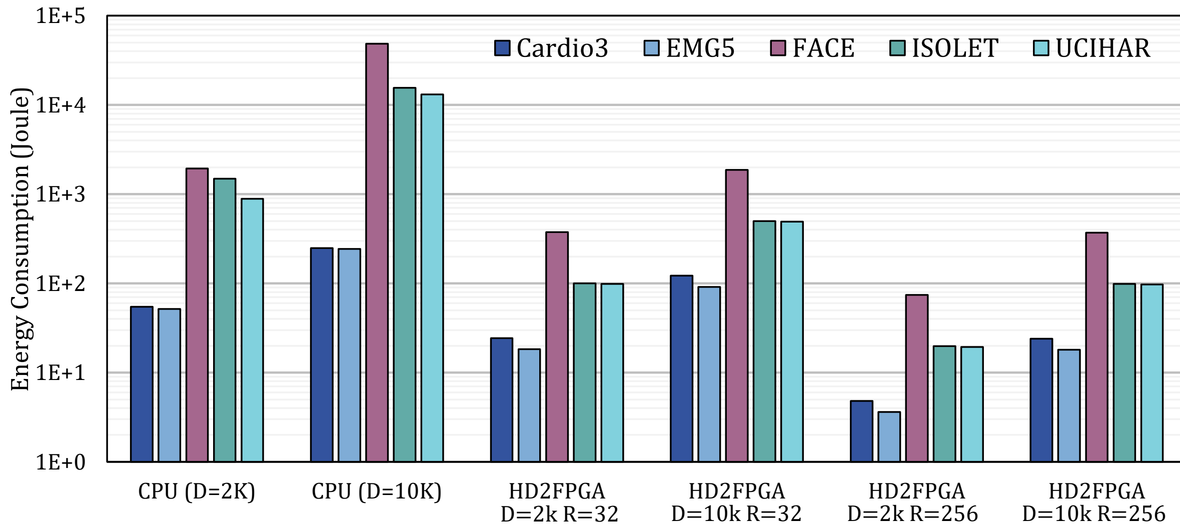

Figure 2.10. End-to-end Training (encoding + 50 epochs retraining) energy consumption of CPU and HD2FPGA.

However, the SearchUnit module of HD2FPGA is fully pipelined with The EncodeUnit module, meaning that in CPU the total inference time is the execution time of the encoding in addition to the associative search, while in HD2FPGA the execution time is the maximum of the execution time of the encoding and associative search as they are executing simultaneously in a pipeline fashion. Depending on the application parameters (number of features per input and number of classes), either of the encoding or associative search steps can become bottleneck. In Cardio and EMG the search module is bottleneck, as they have relatively low number of input features and consequently less complex encoding. Nevertheless, in the FACE, ISOLET, and UCIHAR, the encoding is more computationally complex than the associative search. For $D_{hv} = 2K$, HD2FPGA achieves 19.5$\times$ (with $R = 32$) and 140.8$\times$ ($R = 256$) speed-up over the CPU baseline. For $D_{hv} = 10K$, HD2FPGA inference is 80.0$\times$ faster when $R = 32$ and the speed-up further increases to 578.3$\times$ when the matrix-vector multiplication is configured to use 256 parallel rows ($R = 256$). Figure 2.12 compares the energy consumption of HD2FPGA compared to the CPU baseline. In inference, the encoding step is usually the computational bottleneck and hence, it dominates the energy consumption. As HD2FPGA significantly outperformed the CPU

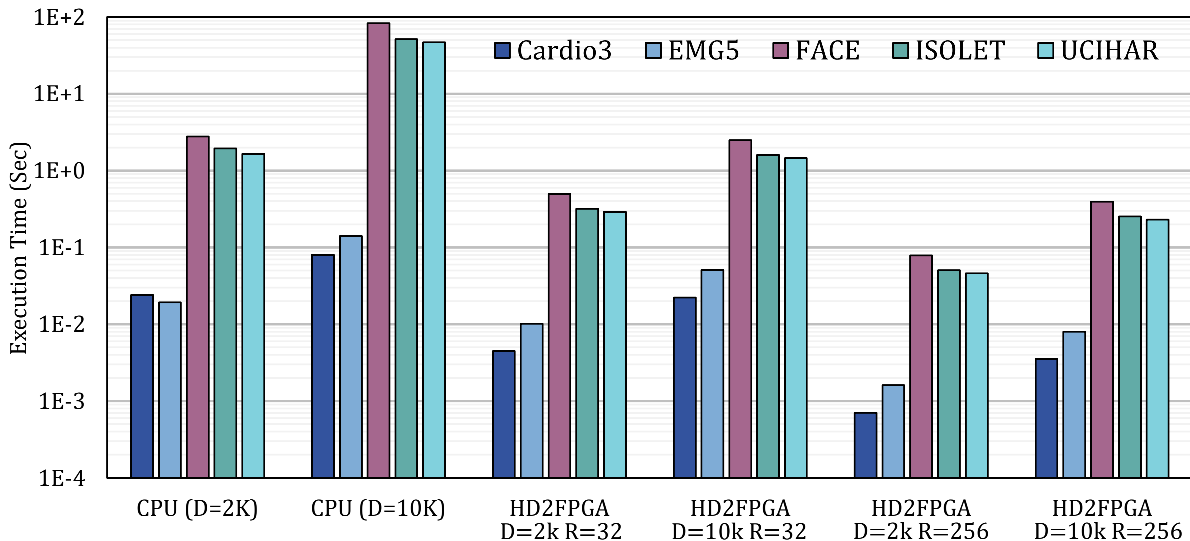

Figure 2.11. Inference (encoding + associative search) time of CPU and HD2FPGA.

baseline in encoding, we expect higher energy efficiency for inference. According to Figure 2.12, HD2FPGA is 63.3$\times$ ($R = 32$) and 366.0$\times$ ($R = 256$) more energy efficient than the CPU baseline for $D_{hv} = 2K$. When $D_{hv} = 10K$ is used, the energy reduction increases to 260.1$\times$ and 1503.7$\times$ respectively.

Comparison with DNN accelerator:

We evaluate the efficiency of HD2FPGA versus a state-of-the-art neural network implementation in FPGA using the ISOLET dataset [119]. For neural network baseline, we leverage the optimized 617×512×512×26 topology utilized in [1]. Figure 2.13 compares the number of cycles of HD2FPGA and DNN accelerator developed in [1]. The x-axis shows different number of HD cycles, which belong to different dimensionality (D_{hv}) and the corresponding accuracy for that particular D_{hv} is also shown. The DNN takes exactly 9014 cycles. For $D_{hv} = 2k$, HD2FPGA achieves 143$\times$ cycle reduction with an accuracy of $\sim 92\%$. Note that HD runs at frequency of 200 MHz while DNN is twice slower, at 100 MHz (i.e., 90,146 nano-second latency). In Figure 2.14, we show the end-to-end latency of HD (nano-second) and compare with DNN for different HD dimensions. For $D_{hv} = 2k$, HD2FPGA shows 277$\times$ speedup compared to DNN accelerator in [1]. Figure 2.15 compares the energy consumption of HD2FPGA and DNN for classifying

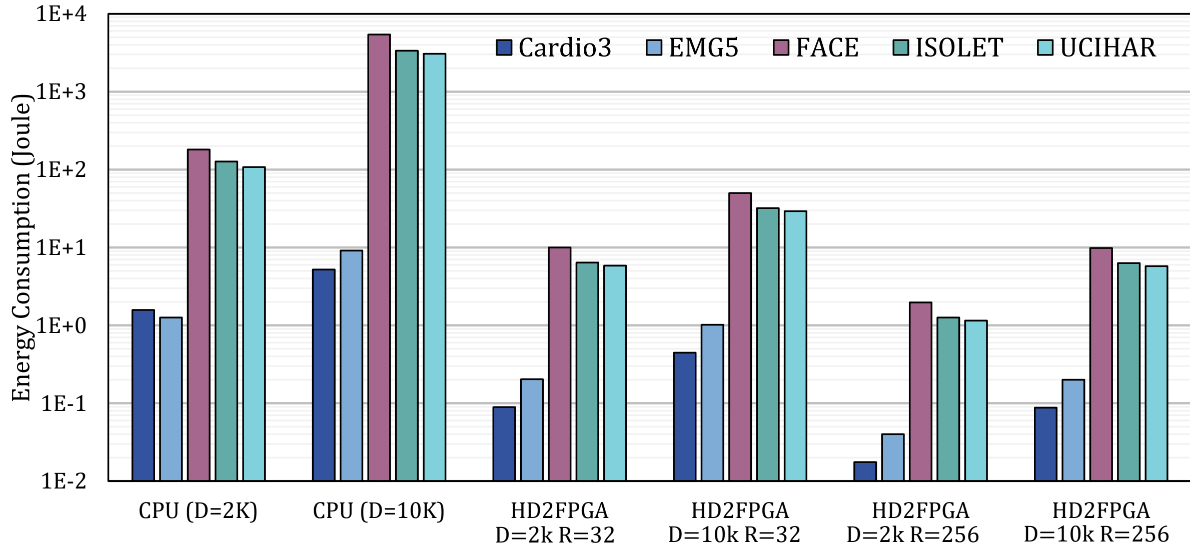

Figure 2.12. Inference (encoding + associative search) energy consumption of CPU and HD2FPGA.

an input. For $D_{hv} = 2k$ and $D_{hv} = 4k$, HD improves the energy consumption for $172\times$ and $87\times$, respectively.

2.4.2 HD Clustering

HD clustering algorithm involves encoding the input data, selecting a subset of the encoded hypervectors as centroids, assigning the similar encoded hypervectors to the same centroids, and bundling the assigned hypervectors to create the new centroids. It is similar to the HD encoding and retraining algorithms, where the retraining step finds the similar class (centroid) and in case of misprediction, performs vector-wise addition. Figure 2.16 shows the execution time for one epoch of clustering. Since clustering repeatedly performs encoding and search, the execution time linearly increases with the number of training epochs. In all datasets but ISOLET, the search step is the bottleneck of the FPGA pipeline as the datasets contain a small number of features per input, making the encoding comparatively faster than search among the centroids. ISOLET has significantly higher number of input features and thus, the encoding step in ISOLET is more complicated than the other datasets. With $D_{hv} = 2K$, HD2FPGA performance outperforms the CPU by $3.5\times$ for $R = 32$, and $22.1\times$ for $R = 256$. With $D_{hv} = 10K$, HD2FPGA

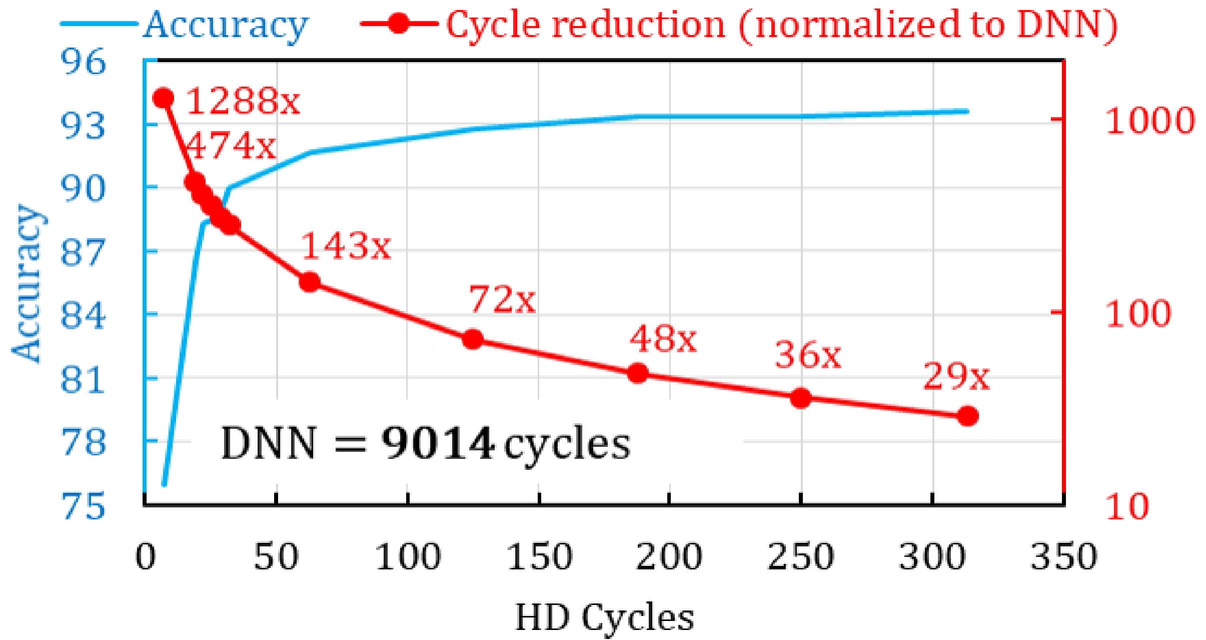

Figure 2.13. Accuracy and latency (number of cycles) of HD2FPGA versus DNN accelerator of [1].

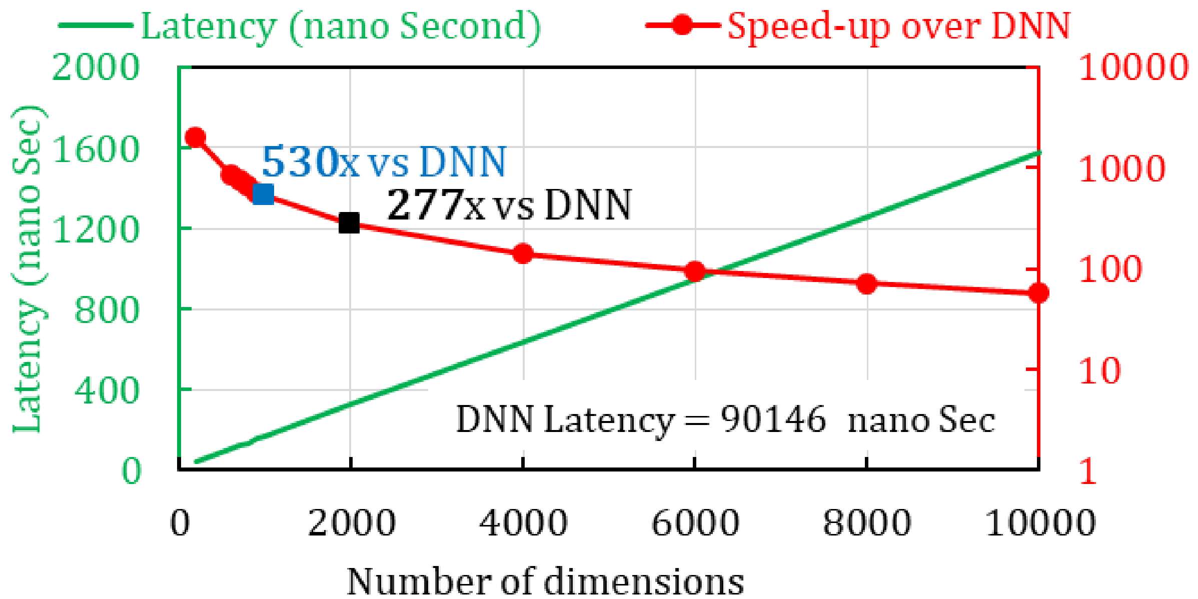

Figure 2.14. End-to-end latency and performance improvement of HD2FPGA versus DNN accelerator of [1].

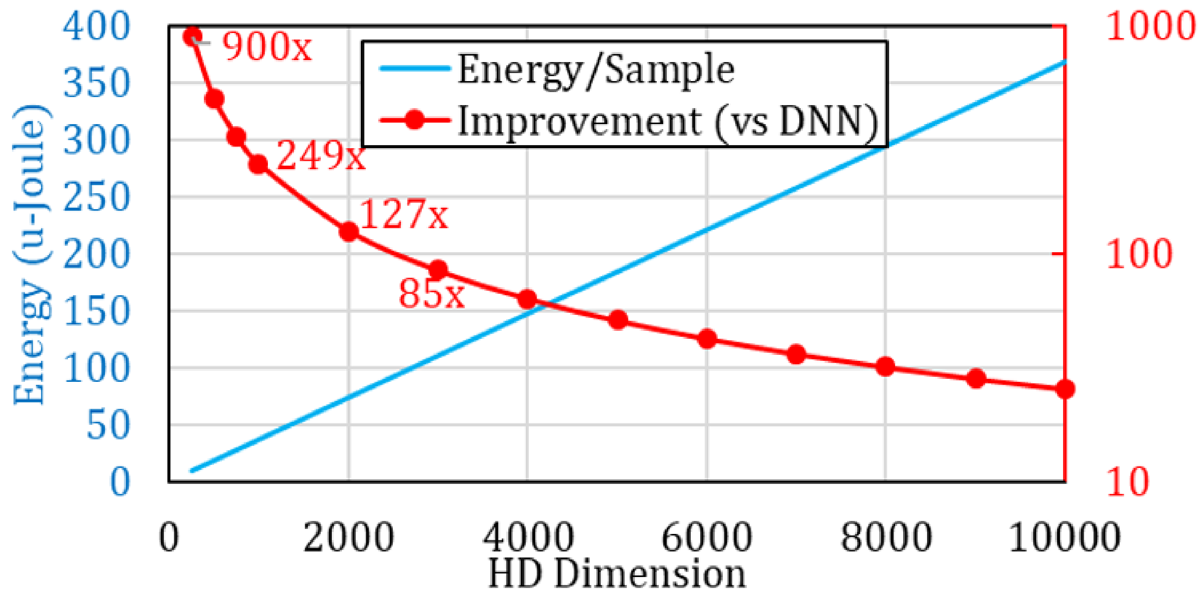

Figure 2.15. Energy comparison of HD2FPGA versus DNN accelerator of [1].

is 9× faster when $R = 32$, and 57× faster when $R = 256$.

Figure 2.17 shows the energy consumption for the CPU baseline and HD2FPGA clustering. The encoding and search steps consume the same amount of power in the classification step as the algorithm (hence the architecture) is the same. With 2K dimensionality, the HD2FPGA consumes 10.2× and 51.6× for $R = 32$ and $R = 256$, respectively. For $D_{hv} = 10K$, HD2FPGA energy efficiency increases to 27.7× (for $R = 32$) and 140.5× (for $R = 256$). AS mentioned earlier, the execution time of HD2FPGA increases linearly with D_{hv}, while in CPU we observe larger increase (e.g., the ISOLET encoding runtime increases by 29.7× when moving from $D_{hv} = 2K$ to $D_{hv} = 10K$). Thus, by moving to larger dimensionalities, HD2FPGA's energy efficiency further raises.

Comparison with Kmeans accelerator:

We evaluate the efficiency of HD2FPGA versus state-of-the-art implementation of Kmeans on FPGA developed by Xilinx [2]. The Kmeans developed in [2] is written in C++, synthesized and implemented on FPGA using Vitis Development tool, in a process similar to HD2FPGA, running on the same FPGA as HD2FPGA. Figure 2.18 compares the clustering

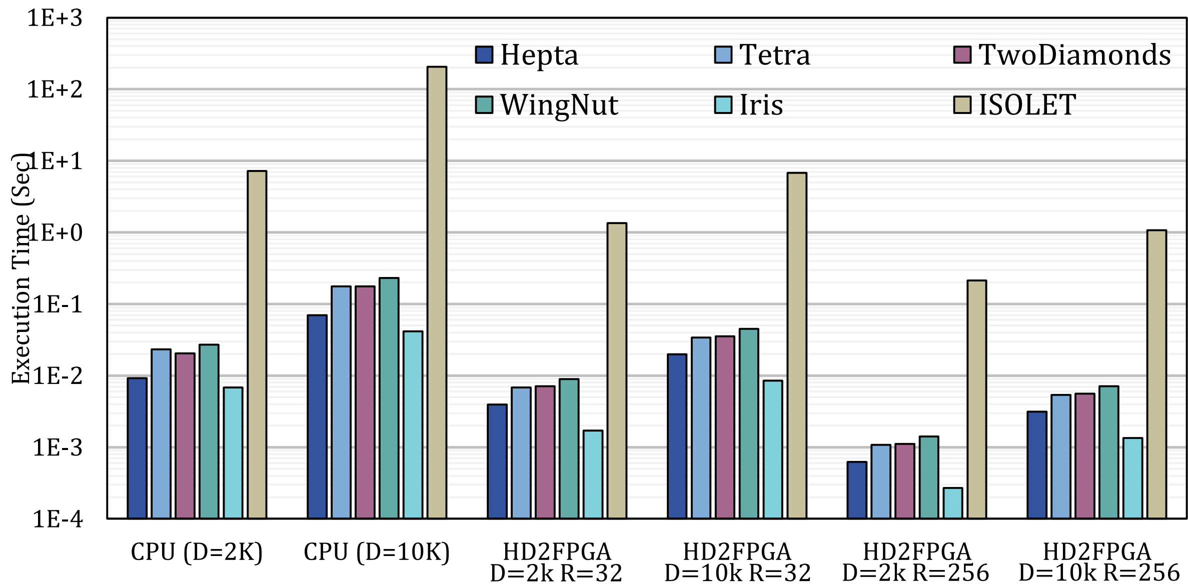

Figure 2.16. Inference (encoding + associative search) time of CPU and HD2FPGA.

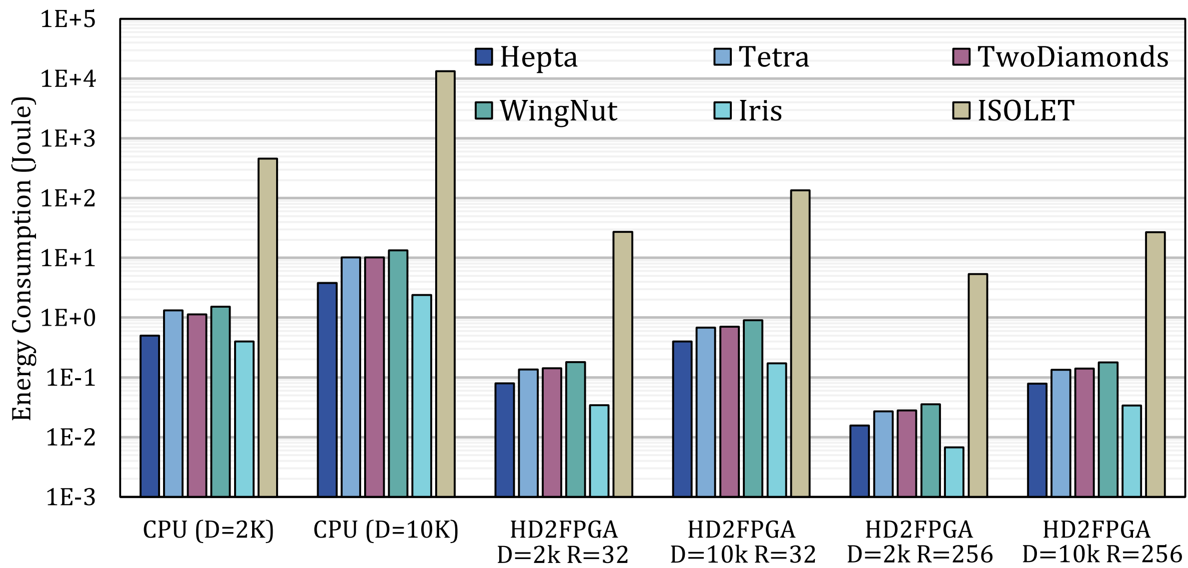

Figure 2.17. Inference (encoding + associative search) energy consumption of CPU and HD2FPGA.

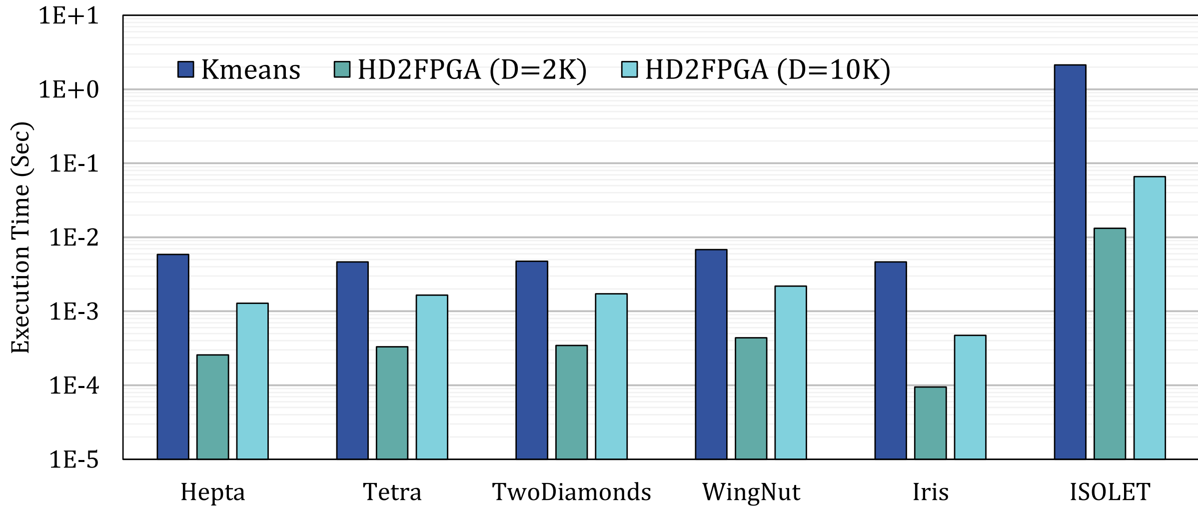

Figure 2.18. End-to-end performance improvement of HD2FPGA versus Kmeans accelerator of [2].

execution time of Kmeans and HD2FPGA for $R = 256$. According to the figure, the performance of HD2FPGA is 46× higher than Kmeans when $D_{hv} = 2K$. When $D_{hv} = 10K$, due to higher computation complexity of HD with longer dimensionality, the performance improvement drops 9.2×. Figure 2.19 shows the energy consumption of HD2FPGA and Kmeans for clustering the entire dataset. According to the figure, the energy consumption of HD2FPGA is 47.8× less than Kmeans for $D_{hv} = 2K$. When $D_{hv} = 10K$, HD2FPGA improves the energy efficiency for 9.6× compared to the Kmeans accelerator [2].

2.5 Conclusion

In this chapter, we proposed HD2FPGA, an automated framework for FPGA-based acceleration of HD classification and clustering. HD2FPGA abstracts away the complexities associated with the design of hardware accelerators from the user. The proposed framework enables the user to provide the HD application specifications (e.g., the number of input features, classes and training data) as well as the application task (classification or clustering) and then automatically generates a customized FPGA implementation. HD2FPGA supports end-to-end training and inference of HD classification and end-to-end execution of HD clustering on FPGA.

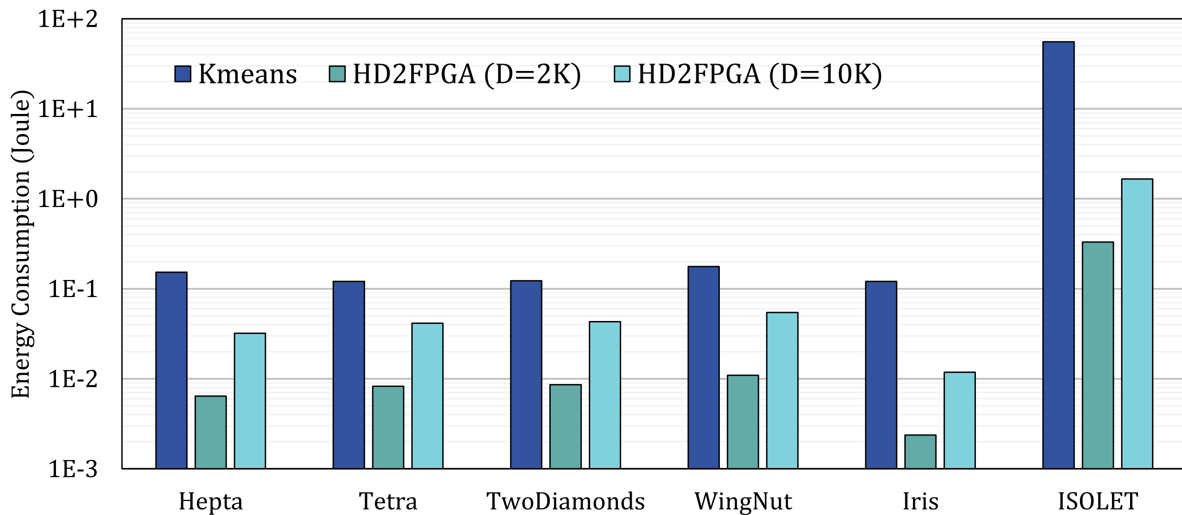

Figure 2.19. Energy comparison of HD2FPGA versus Kmeans accelerator of [2].

We evaluated the efficiency of HD2FPGA extensively, thereby showing 30.5× speedup and 73.9× efficiency in HD training. HD2FPGA shows 578.3× and 57× performance improvement in HD classification and clustering, respectively. It moreover, increases the energy efficiency of HD classification and clustering by 1503.7× and 140.5×, respectively. HD2FPGA, in classification, shows, 277× speedup and 172× energy reduction compared to state-of-the-art DNN running on FPGA. It also shows 46× speedup and 47.8× energy reduction compared to state-of-the-art FPGA-based K-Means clustering accelerator. While HD2FPGA improves the performance and energy efficiency of classification and clustering on a single FPGA, as many users utilize cloud FPGAs, the energy efficiency of runnign multiple FPGAs is also crucial. The next chapter introduces an approach to minimize the energy consumption of multi-FPGA platforms commonly used for cloud-based processing.

This chapter contains material from “HD2FPGA: Automated Framework for Accelerating Hyperdimensional Computing on FPGAs”, by Sahand Salamat, Behnam Khaleghi, and Tajana S. Rosing, which is still in preparation. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Efficiency of ML on Multi-FPGAs

In the previous chapters we proposed two highly optimized FPGA-based accelerators for DNNs and HD. Even though these accelerators increase the efficiency of running DNNs and HD and thereby enabling processing more data, as the majority of the users are using cloud services optimizing the energy efficiency of cloud computation platforms is crucial for efficient big data processing. The continuous growth of big data applications with high computational and scalability demands has resulted in increasing popularity of cloud computing. Optimizing the performance and power consumption of cloud resources is therefore crucial to relieve the rising costs of data centers.

In recent years, multi-FPGA platforms have gained traction in data centers with their low-cost and high-performance when used as acceleration engines, thanks to the high degree of parallelism they provide. Several cloud service providers offer multi-FPGA platforms as ”Infrastructure as a Service” where users can implement their own applications. They also offer machine learning software as a service running on multi-FPGA platforms. Nonetheless, the size of data centers workloads varies during service time, leading to significant underutilization of computing resources while consuming a large amount of power, which is a key factor contributing to data center inefficiency, regardless of the underlying hardware structure.

In this chapter, we propose an efficient framework to throttle the power consumption of multi-FPGA platforms by dynamically scaling the voltage and hereby frequency during runtime

according to *prediction* of, and *adjustment* to the workload level, while maintaining the desired Quality of Service (QoS). This is in contrast to, and more efficient than, conventional approaches that merely scale (i.e., power-gate) the computing nodes or frequency. The proposed framework carefully exploits a pre-characterized library of delay-voltage, and power-voltage information of FPGA resources, which we show is indispensable to obtain the efficient operating point due to the different sensitivity of resources w.r.t. voltage scaling, particularly considering multiple power rails residing in these devices. Our evaluations by implementing state-of-the-art deep neural network accelerators revealed that, providing an average power reduction of 4.0×, the proposed framework surpasses the previous works by 33.6% (up to 83%).

3.1 Related Work

The use of FPGAs in modern data centers have been gained attention recently as a response to rapid evolution pace of data center services in tandem with the inflexibility of application-specific accelerators and unaffordable power requirement of GPUs [120, 57]. Data center FPGAs are offered in various ways, Infrastructure as a Service for FPGA rental, Platform as a Service to offer acceleration services, and Software as a service to offer accelerated vendor services/software [121]. Though primary works deploy FPGAs as tightly-coupled server addendum, recent works provision FPGAs as an ordinary standalone network-connected server-class node with memory, computation and networking capabilities [121, 57]. Various ways of utilizing FPGA devices in data centers have been well elaborated in [120].

FPGA data centers, in parts, address the problem of programmability with comparatively less power consumption than GPUs. Nonetheless, the significant resource underutilization in non-peak workload yet wastes a high amount of data centers energy. FPGA virtualization attempted to resolve this issue by splitting the FPGA fabric into multiple chunks and implementing applications in the so-called virtual FPGAs. Yazdanshenas *et al.* have quantified the cost of FPGA virtualization in [120], revealing up to 46% performance degradation with 2.6× increase

in wire length of the *shell*, i.e., the static region responsible to connect the virtual FPGAs to external resources such as PCI and DDR. This hinders the routability of the shell as the number of virtual FPGAs increase. These overheads excluded the area overhead of the shell itself, which occupies up to 44% of FPGA area. FPGA virtualization is also not practical for large data center applications such as deep neural networks that occupy a whole or multiple devices [55].

Another foray for FPGA power optimization includes approaches that exploit dynamic frequency and/or voltage scaling. The main goal of these studies is to utilize the available timing headroom conservatively considered for worst-case temperature, aging, variation, etc. and scale the frequency for performance boosting, or voltage reduction without performance degradation, though a few of them consider workload. Chow *et al.* [122] propose a dynamic voltage scaling scheme that exploits a ring-oscillator based logic delay measurement circuit to mimic the timing behavior of application critical path and adjust the voltage accordingly. However, the inaccuracy of path monitor circuitries in FPGAs and even ASICs has been well elaborated [123, 124, 125, 126]. Levine *et al.* employ timing error detectors inserted as capture registers with a phase-shifted clock at the end of critical paths to find out the timing slack of FPGA-mapped designs through a gradual reduction of voltage [123]. Their approach adds extra area and power overhead, cannot be implemented in paths heading to hard blocks such as memories, and assumes the corresponding paths will be exercised at runtime. Zhao *et al.* propose an elaborated two-step approach by extracting the critical paths of the design using the static timing analysis tool and sequentially mapping into the FPGA [124]. Thence, they vary the FPGA core voltage to obtain the voltage-delay ($V_{core} - D$) relation of the paths for online adjustment during the operation time. It requires analyzing a huge number of paths, especially originally non-critical paths might become critical when the voltage changes. Salami *et al.* evaluate the impact of block RAM (BRAM) voltage (V_{bram}) scaling on the power and accuracy of a neural network application [127]. They observed that V_{bram} can be reduced by 39% of the nominal value, which saves the BRAM dynamic power by one order of magnitude, with a negligible error at the output. Their approach is intuitive and does not examine timing violation, i.e., it is

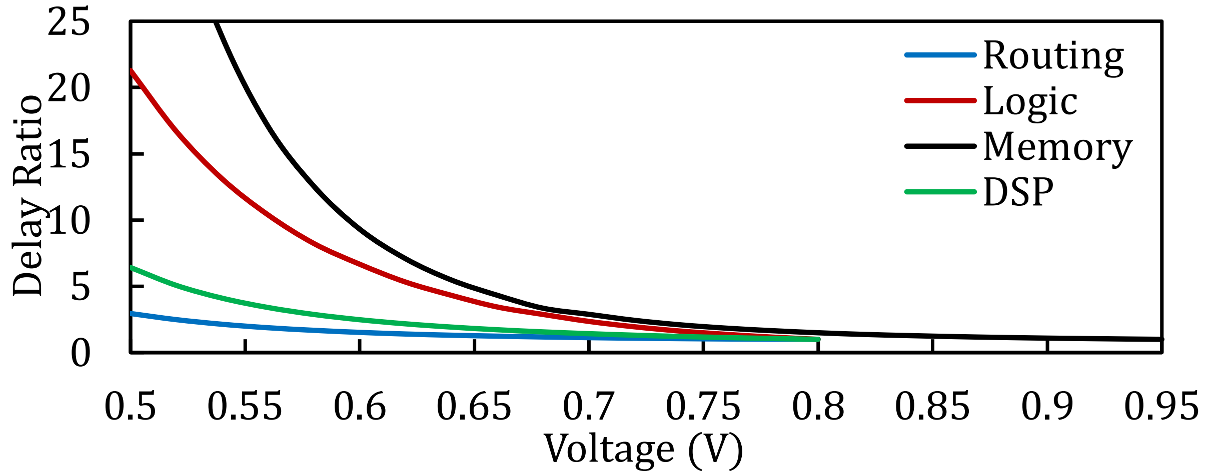

Figure 3.1. Delay of FPGA resources versus voltage

not known if the timing will not be eventually violated in a particular voltage level. Similarly, Khaleghi *et al.* leverage the thermal margin of FPGAs for frequency boosting though they integrate it in the conventional flow of FPGA using the pre-characterizition of resources [52]. Eventually, Jones *et al.* propose a workload-aware frequency scaling approach that temporarily allows over-clocking of applications when the temperature is safe enough, i.e., the workload is not bursty [128]. They assume the design has inherently sufficient slack to tolerate the frequency boosting without overscaling the voltage.

As mentioned earlier, the primary goal of the latter studies is to leverage the pessimistic timing headrooms for efficiency, while they struggle in guaranteeing timing safety. More importantly, the utmost effort of previous works is to satisfy the timing of critical or near-critical paths under either (and mainly) V_{core} scaling, or V_{bram} scaling. Nevertheless, unlike single voltage scaling where there is only one minimal voltage level for a target frequency, for simultaneous scaling of V_{core} and V_{bram}, numerous 'V_{core}, V_{bram}' pairs will minimally yield the target frequency while only one pair of this solution space has the minimum power dissipation. Therefore, accurate timing *and power* analysis under multiple voltage scaling is inevitable.

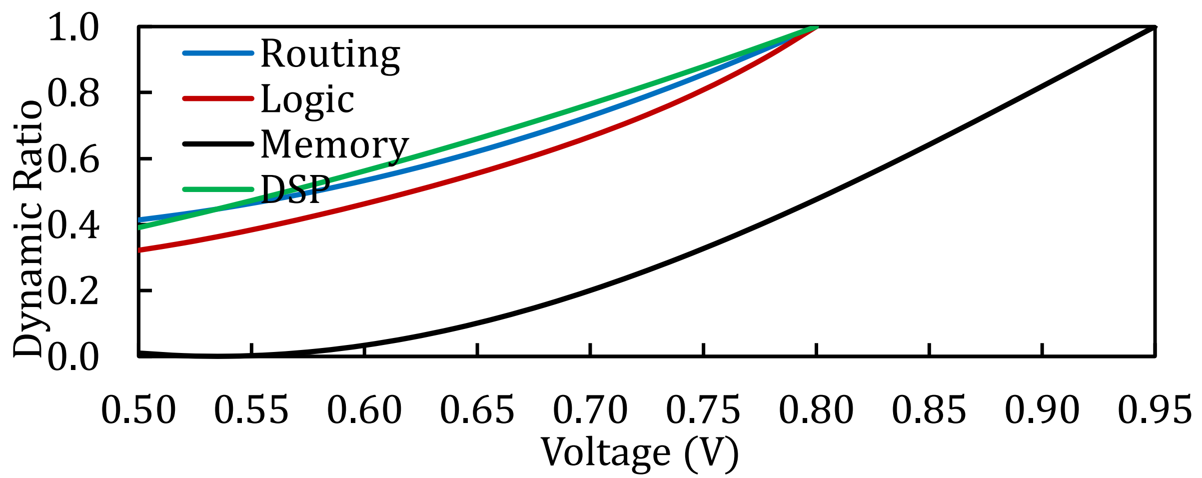

Figure 3.2. Dynamic power of FPGA resources versus voltage.

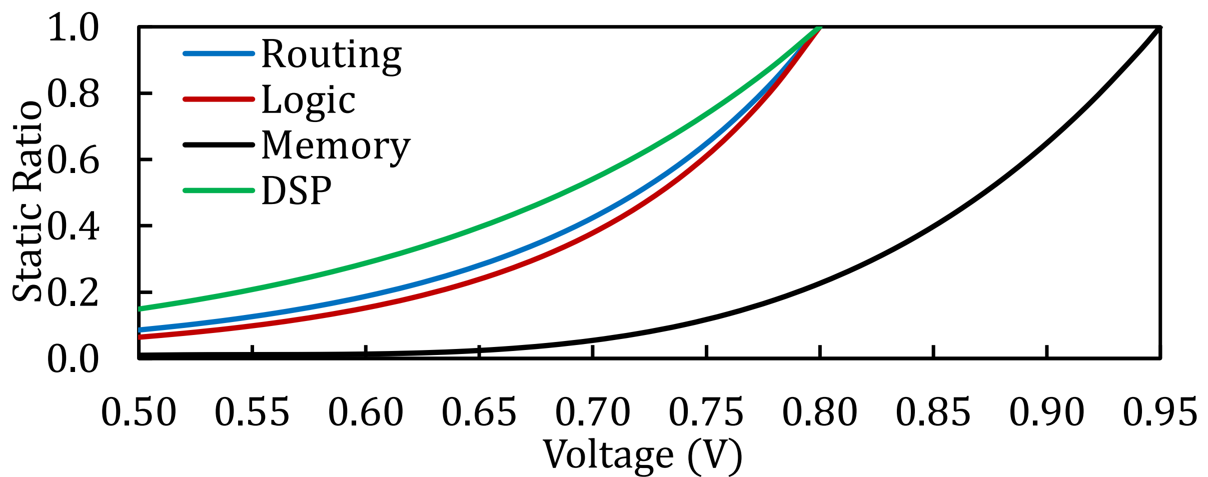

Figure 3.3. Static power of FPGA resources versus voltage.

3.2 Motivational Analysis

In this section, we use a simplified example to justify the necessity of the proposed scheme and how it surpasses the conventional approaches in power efficiency. Figures 3.1, 3.2, and 3.3 shows the relation of delay and power consumption of FPGA resources when voltage scales down. Experimental results will be elaborated in Section 3.5, but concisely, routing and logic delay and power indicate the average delay and power of individual routing resources (e.g., switch boxes and connection block multiplexers) and logic resources (e.g., LUTs). Memory stands for the on-chip BRAMs, and DSP is the digital signal processing hard macro block. Except memory blocks, the other resources share the same V_{core} power rail. Since FPGA memories incorporate high-threshold process technology, they utilize a V_{bram} voltage that is initially higher than nominal core voltage V_{core} to enhance the performance [129]. We assumed a nominal memory and core voltage of 0.95V and 0.8V, respectively [129].

The different sensitivity of resources' delay and power with respect to voltage scaling implies cautious considerations when scaling the voltage. For instance, by comparing Figure 3.1 and Figure 3.3, we can understand that reducing the memory voltage from 0.95V down to 0.80V has a relatively small effect on its delay, while its static power decreases by more than 75%. Then we see a spike in memory delay with trivial improvement of its power, meaning that it is not beneficial to scale V_{bram} anymore. Similarly, routing resources show good delay tolerance versus voltage scaling. It is mainly because of their simple two-level pass-transistor based structure with boosted configuration SRAM voltage that alleviates the drop of drain voltages [130]. Notice that we assume a separate power rail for configuration SRAM cells and do not change their voltages as they are made up of thick high-threshold transistors that have already throttled their leakage current by two orders of magnitude though have a crucial impact on FPGA performance. Nor we do scale the auxiliary voltage of I/O rails to facile standard interfacing. While low sensitivity of routing resources against voltage implied V_{core} is a prosperous candidate in interconnection-bound designs, the large increase of logic delay with voltage scaling hinders

V_{core} scaling when the critical path consists of mostly LUTs. In the following we show how varying parameters of workload, critical path(s), and application affect optimum 'V_{core}, V_{bram}' point and energy saving.

Let us consider the critical path delay of an arbitrary application as Equation (3.1).

$$d_{cp} = d_{l0} \cdot \mathscr{D}_l(V_{core}) + d_{m0} \cdot \mathscr{D}_m(V_{bram}) \tag{3.1}$$

Where d_{l0} stands for the initial delay of the logic and routing part of the critical path, and $\mathscr{D}_l(V_{core})$ denotes the voltage scaling factor, i.e., information of Figure 3.1. Analogously, d_{m0} and $\mathscr{D}_m(V_{bram})$ are the memory counterparts. The original delay of the application is $d_{l0} + d_{m0}$, which can be stretched by $(d_{l0} + d_{m0}) \times S_w$ where $S_w \geq 1$ indicates the workload factor, meaning that in an 80% workload, the delay of all nodes can be increased up to $S_w = \frac{1}{0.8} = 1.25\times$. Defining $\alpha = \frac{d_{m0}}{d_{l0}}$ as the relative delay of memory block(s) in the critical path to logic/routing resources, the applications need to meet the following:

$$d_{cp} \propto \mathscr{D}_l(V_{core}) + \alpha \cdot \mathscr{D}_m(V_{bram}) \leq (1+\alpha) \cdot S_w \tag{3.2}$$

We can derive a similar model for power consumption as a function of V_{core} and V_{bram} shown by Equation (3.3).

$$p_{cir} \propto \mathscr{P}_l(V_{core}, d_{cp}) + \beta \cdot \mathscr{P}_m(V_{bram}, d_{cp}) \tag{3.3}$$

where $\mathscr{P}_l(V_{core}, d_{cp})$ is for the total power drawn from the core rail by logic, routing, and DSP resources as a function of voltage V_{core} and frequency (delay) d_{cp}, and β is an application-dependent factor to determine the contribution of BRAM power. In the following, we initially assume $\alpha = 0.2$ (i.e., BRAM contributes to $\frac{0.2}{1+0.2}$ of critical path delay [130]) and $\beta = 0.4$ (i.e., BRAM power initially is $\sim 25\%$ of device total power [127]).

Figures 3.4, 3.5, and 3.6 demonstrates the efficiency of different voltage scaling schemes

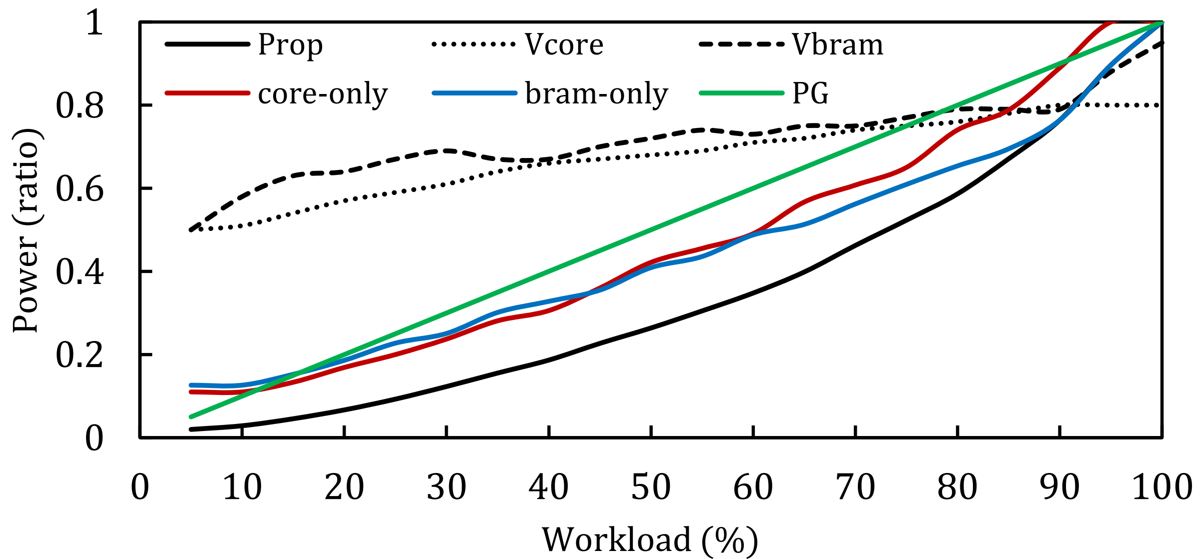

Figure 3.4. Comparing DVFS techniques in different workloads.

under varying workloads, applications' critical paths ('α's), and applications' power characteristics (i.e., β, the ratio of memory to chip power). *Prop* means the proposed approach that simultaneously determines V_{core} and V_{bram}, core-only is the technique that only scales V_{core} [124, 123], and bram-only is similar to [127]. Dashed lines of *Vcore* and *Vbram* in the figures show the *magnitude* of the V_{core} and V_{bram} in the proposed approach, *Prop* (for the sake of clarity, we do not show voltages of the other methods). According to Figure 3.4, in high workloads ($> 90\%$, or $S_w < 1.1$), our proposed approach mostly reduces the V_{bram} voltage because slight reduction of the memory power in high voltages significantly improves the power efficiency, especially because the contribution of memory delay in the critical path is small ($\alpha = 0.2$), leaving room for V_{bram} scaling. For the same reason, core-only scheme has small gains there. The Figure also reveals the sophisticated relation of the minimum voltage points and the size of workload; each workload level requires re-estimation of 'V_{core}, V_{bram}'. In all cases, the proposed approach yields the lowest power consumption. It is noteworthy that the conventional power-gating approach (denoted by *PG* in Figure 3.4) scales the number of *computing nodes* linearly with workload, though, the other approaches scale both frequency and voltage, leading to twofold power saving. In very low workloads, power-gating works better than the other two

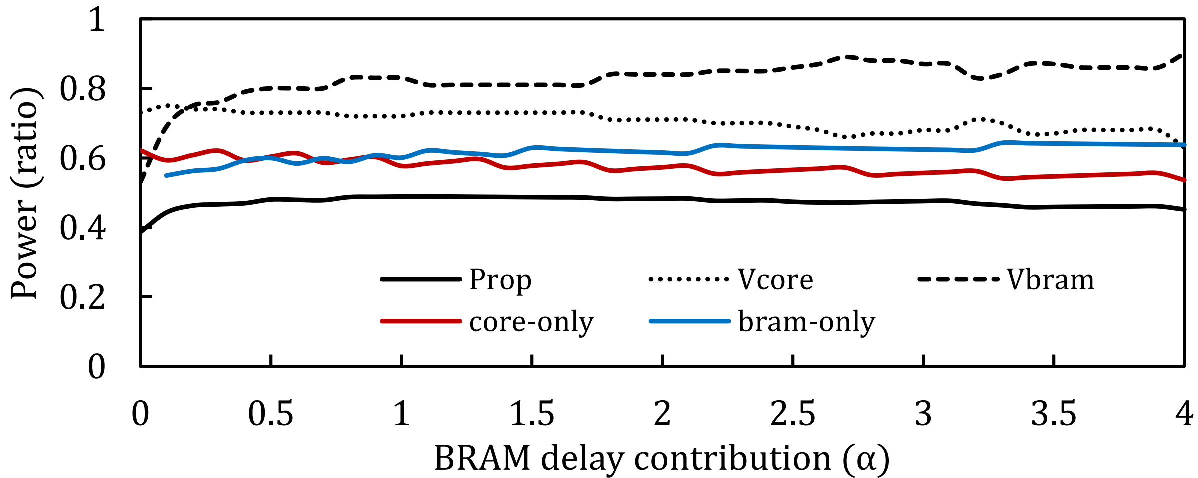

Figure 3.5. Comparing DVFS techniques in different critical paths.

approaches because the crash voltage ($\sim 0.50V$) prevents further power reduction.

Similar insights can also be grasped from Figure 3.5 and 3.6. A constant workload of 50% is assumed here while α and β parameters change. When the contribution of BRAM delay in total reduces, the proposed approach tends to scale the V_{bram}. For $\alpha = 0$ highest power saving is achieved as the proposed method can scale the voltage to the minimum possible, i.e., the crash voltage. Analogously in Figure 3.6, the effectiveness of the core-only (bram-only) method degrades (improves) when BRAM contributes to a significant ratio of total power, while our proposed method can adjust both voltages cautiously to provide minimum power consumption. It is worth to note that the efficiency of the proposed method increases in high BRAM powers because in these scenarios a minor reduction of BRAM power saves huge power with a small increase of delay (compare Figure 3.4 and 3.6).

3.3 Proposed Method

In practice, the generated data from different users are processed in a centralized FPGA platform located in datacenters. The computing resources of the data centers are rarely completely idle and sporadically operate near their maximum capacity. In fact, most of the time the incoming workload is between 10% to 50% of the maximum nominal workload. Multiple FPGA instances

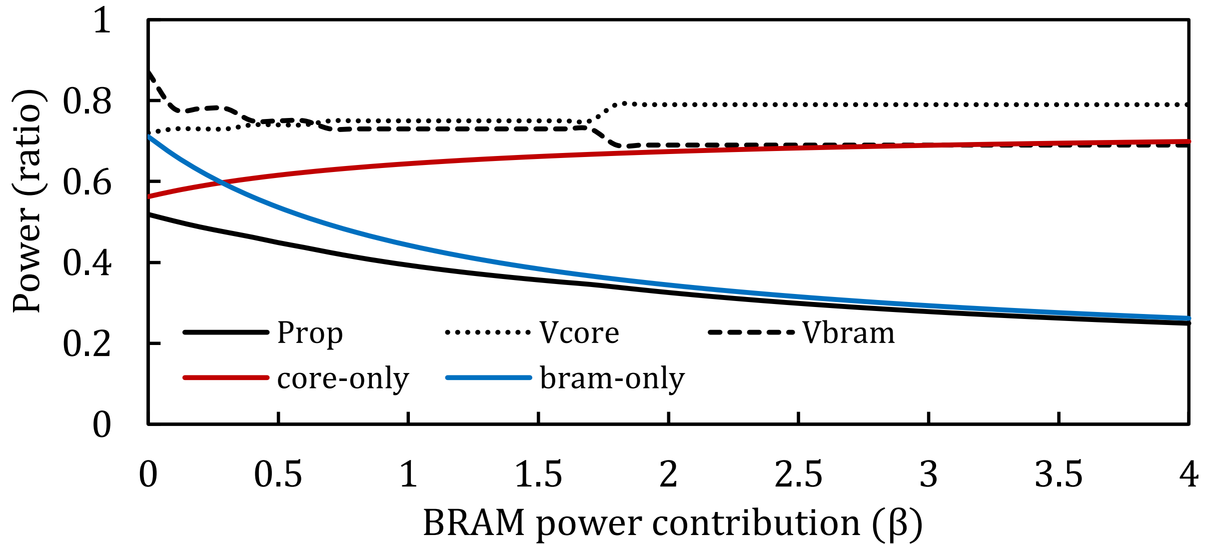

Figure 3.6. Comparing DVFS techniques in different BRAM power rates.

are designed to deliver the maximum nominal workload when running on the nominal frequency to provide the users' desired quality of service. However, since the incoming FPGA workloads are often lower than the maximum nominal workload, FPGA become underutilized. By scaling the operating frequency proportional to the incoming workload, the power dissipation will be reduced without violating the desired throughput. It is noteworthy that if an application has specific latency restrictions, it should be considered in the voltage and frequency scaling. The maximum operating frequency of the FPGA can be set depending on the delay of the critical path such that it guarantees the reliability and the correctness of the computation. By underscaling the frequency, i.e., stretching the clock period, delay of the critical path becomes less than the clock toggle rate. This extra timing room can be leveraged to underscale the voltage to minimize the energy consumption untill the critical path delay again reaches the clock delay.

Figure 3.7 abstracts an FPGA cloud platform consisting of n FPGA instances where all of them are processing the input data gathered from one or different users. FPGA instances are provided with the ability to modify their operating frequency and voltage. In the following we explain the workload prediction, dynamic frequency scaling and dynamic voltage scaling implementations.

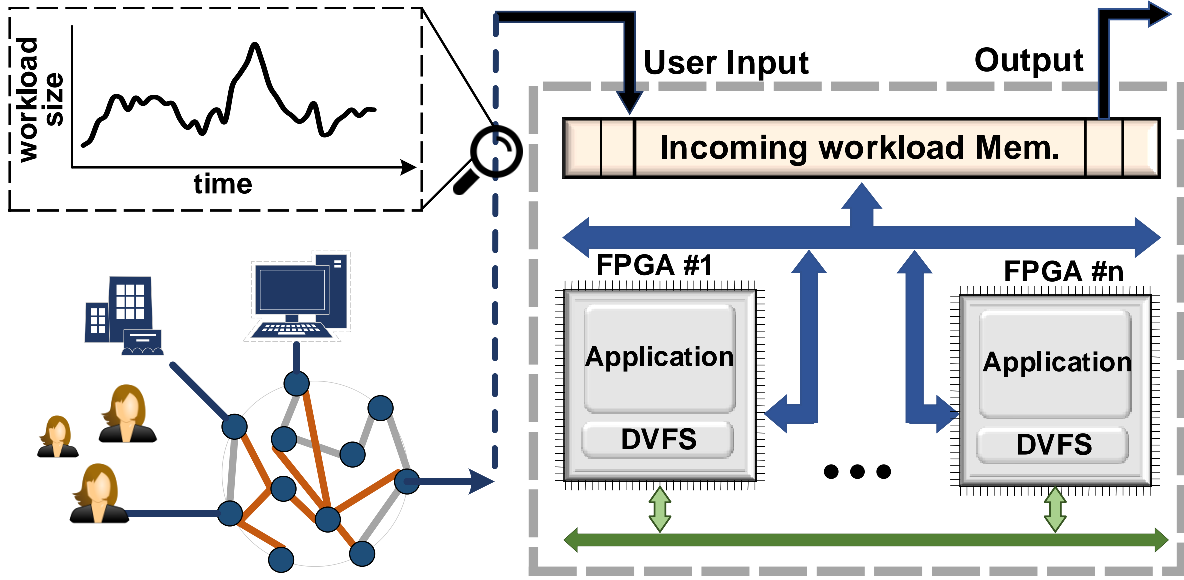

Figure 3.7. Overview of an FPGA-based datacenter platform.

3.3.1 Workload Prediction

We divide the FPGA execution time to steps with the length of τ, where the energy is minimized separately for each time step. At the ith time step (τ_{i-1}), our approach predicts the size of the workload for the $i+1$ time step. Accordingly, we set the working frequency of the platform such that it can complete the the predicted workload for the τ_i time step.

To provide the desired QoS as well as minimizing the FPGA idle time, the size of the incoming workload needs to be predicted at each time step. The operating voltage and frequency of the platform is set based on the predicted workload. Generally, to predict and allocate resources for dynamic workloads, two different approaches have been established: reactive, and proactive. In reactive approach, resources are allocated to the workload based on a predefined thresholds [131, 132], while in proactive approach, the future size of the workload is predicted and resources are allocated based on this prediction [133, 134, 135].

In this work, we use a light-weight online workload prediction method similar to the one proposed in [135] which is able to extract short-term features. In the cases the service provider knows the periodic signatures of the incoming workload, the predictor can be loaded with this

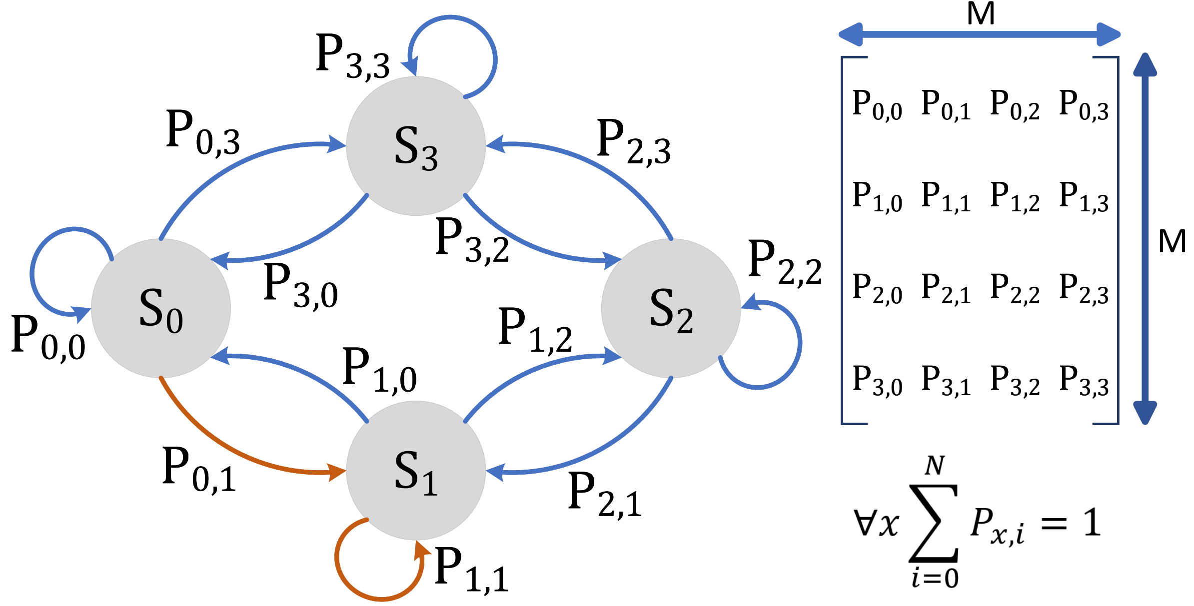

Figure 3.8. Example of Markov chain for workload prediction.

information. Workloads with repeating patterns are divided into time intervals which are repeated with the period. The average of the intervals represents a bias for the short-term prediction. For applications without repeating patterns, we use a discrete-time Markov chain with a finite number of states to represents the short-term characteristics of the incoming workload.

The size of the workload is discretized into M bins, each represented by a state in the Markov chain; all the states are connected through a directed edge. $P_{i,j}$ shows the transition probability from state i to state j. Therefore, there are $M \times M$ edges between states where each edge has a probability learned during the training steps to predict the size of the incoming workload. Figure 3.8 represents a Markov chain model with 4 states, $\{S_0, S_1, S_2, S_3\}$, in which a directed edge with label $P_{i,j}$ shows the transition from S_i to S_j which happens with the probability of $P_{i,j}$. Considering the The total probability of the outgoing edges of state S_i has to be 1 as probability of selecting the next state is one.

Starting from S_0 with probability of $P_{0,i}$ the next state will be S_i. In the next time step, the third state will be again S_1 with $P_{1,1}$ probability. If a pre-trained model of the workload is available, it can be loaded on FPGA, otherwise, the model needs to be trained during the runtime.

During system initialization, the platform runs with the maximum frequency and works with the nominal frequency for the first I time steps. In the training phase, the Markov model learns the patterns of the incoming workload and the probability of transitions between states are set during this phase.

After I time steps, the Markov model predicts the incoming input of the next time step and the frequency of the platform is selected accordingly, with a $t\%$ throughput margin to offset the likelihood of workload under-estimation as well as to preclude consecutive mispredictions. Mispredictions can be either under-estimations or over-estimations. In case of over-estimation, QoS is meet, however, some power is wasted as the frequency (and voltage) is set to a unnecessarily higher value. In case of workload under-estimation the desired QoS may be violated. The work in [135] tackles most of the underestimations by $t = 5\%$ margin.

3.3.2 Frequency Scaling Flow

To achieve high energy efficiency, the operating FPGA frequency needs to be adjusted according to the size of the incoming workload. To scale the frequency of FPGAs, Intel (Altera) FPGAs enable Phase-Locked Loop (PLL) hard-macros (Xilinx also provide a similar feature). Each PLL generates up to 10 output clock signals from a reference clock. Each clock signal can have an independent frequency and phase as compared to the reference clock. PLLs support runtime reconfiguration through a Reconfiguration Port (RP). The reconfiguration process is capable of updating most of the PLL specifications, including clock frequency parameters sets (e.g. frequency and phase). To update the PLL parameters, a state machine controls the RP signals to all the FPGA PLL modules.

PLL module has a *Lock* signal that represents when the output clock signal is stable. The lock signal activates whenever there is a change in PLL inputs or parameters. After stabling the PLL inputs and the output clock signal, the lock signal is asserted again. The lock signal is de-asserted during the PLL reprogramming and will be issued again in, at most, $100\mu Sec$. Each of the FPGA instances in the proposed DFS module has its own PLL modules to generate the

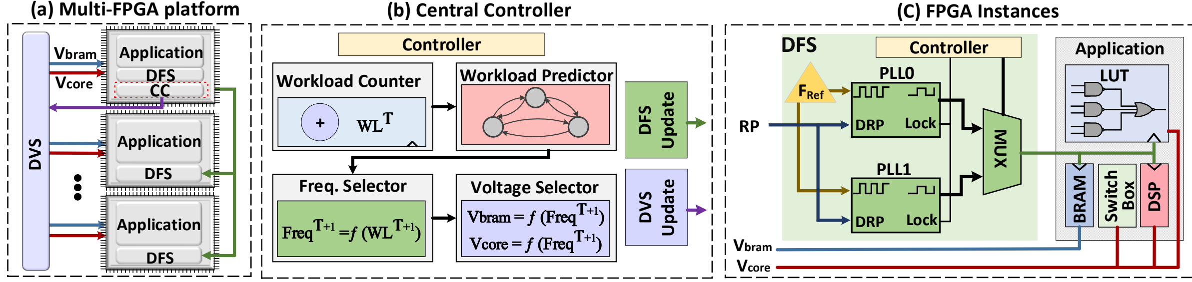

Figure 3.9. (a) the architecture of the proposed energy-efficient multi-FPGA platform. The details of the (b) central controller, and (c) the FPGA instances.

clock signal from the reference clock provided in the FPGA board. For simplicity of explanations, we assume the design works with one clock frequency, however, our design supports multiple clock signals with the same procedure. Each PLL generates one clock output, *CLK0*. At the start-up, the PLL is initialized to generate the output clock equal to the reference clock. When the platform modifies the clock frequency, at τ_i based on the predicted workload for τ_{i+1}, the PLL is reconfigured to generate the output clock that meets the QoS for τ_{i+1}.

3.3.3 Voltage Scaling Flow

To implement the dynamic voltage scaling for both V_{core} and V_{bram}, Texas Instruments (TI) PMBUS USB Adapter can be used [136] for different FPGA vendors. TI adapter provides a C-based Application Programming Interface (API), which eases adjusting the board voltage rails and reading the chip currents to measure the power consumption through Power Management Bus (PMBUS) standard. To scale the FPGA voltage rails, the required PMBUS commands are sent to the adapter to set the V_{bram} and V_{core} to certain values. This adopter is used as a proof of concept, while in industry fast DC-DC converters are used to change the voltage rails. The work in [137] has shown a latency of 3-5 nSec, and is able to generate voltages between 0.45V to 1V with 25mV resolution. As these converters are faster than the FPGAs clock frequency, we neglect the performance overhead of the DVS module in the rest of the chapter.

3.4 Proposed Architecture

Figure 3.9(a) demonstrates the architecture of the proposed energy efficient multi-FPGA platform. Our platform consists of n FPGAs where one of them is a central FPGA. The central FPGA has Central Controller (CC) and DFS blocks and is responsible to control the frequency and voltage of all other FPGAs. Figure 3.9(b) shows the details of the CC managing the voltage/frequency of all FPGA instances. The CC predicts the workload size and accordingly scales the voltage and frequency of all other FPGAs. A *Workload Counter* computes the number of incoming inputs in a central FPGA, assuming all other FPGAs have the similar input rate. The *Workload Predictor* module compares the counter value with the predicted workload at the previous time step. Based on the current state, the workload predictor estimates the workload size in the next time step. Next, *Freq. Selector* module determines the frequency of all FPGA instances depending on the workload size. Finally, the *Voltage Selector* module sets the working voltages of different blocks based on the clock frequency, design timing characteristics (e.g., critical paths), and FPGA resources characteristics. This voltage selection happens for logic elements, switch boxes, and DSP cores (V_{core}); as well as the operating voltage of BRAM cells (V_{bram}). The obtained voltages not only guarantee timing (which has a large solution space), but also minimizes the power as discussed in Section 3.2. The optimal operating voltage(s) of each frequency is calculated during the design synthesis stage and are stored in the memory, where the DVS module is programmed to fetch the voltage levels of FPGAs instances.

Misprediction Detection: In CC, the misprediction happens when the workload bin for time step i^{th} is not equal to the bin achieved by the workload counter. To detect mispredictions, the value of $t\%$ should be greater than $1/m$, where m is the number of bins. Therefore, the system discriminates each bin with the higher level bin. For example, if the size of the incoming workload is predicted to be in bin i^{th} while it actually belongs to $i+1^{th}$ bin, the system is able to process the workload with the size of $i+1^{th}$ bin. After each misprediction, the state of the Markov model is updated to the correct state. If the number of mispredictions exceeded a

threshold, the probabilities of the corresponding edges are updated.

PLL Overhead: The CC issues the required signals to reprogram the PLL blocks in each FPGA. To reprogram the PLL modules, the DVF reprogramming FSM issues the RP signal serially. After reprogramming the PLL module, the generated clock output is unreliable until the lock signal is issued, which takes no longer than 100 μSec. In the cases the framework changes the frequency and voltage very frequently, the overhead of stalling the FPGA instances for the stable output clock signal limits the performance and energy improvement. Therefore, we use two PLL modules to eliminate the overhead of frequency adjustion. In this platform, as shown in Figure 3.9(c), the outputs of two PLL modules pass through a multiplexer, one of them is generating the current clock frequency, while the other is being programmed to generate the clock for the next time step. Thus, in the next clock, the platform will not be halted waiting for a stable clock frequency.

In case of having one PLL, each time step with duration τ requires t_{lock} extra time for generating a stable clock signal. Therefore, using one PLL has t_{lock} set up overhead. Since $t_{lock} \ll \tau$, we assume the PLL overhead, t_{lock}, does not affect the frequency selection. The energy overhead of using one PLL is:

$$\underbrace{P_{Design} \times t_{lock}}_{\text{Design energy during } t_{lock}} + \underbrace{P_{PLL} \times (\tau + t_{lock})}_{\text{PLL energy}} \tag{3.4}$$

In case of using two PLLs, there is no performance overhead. The energy overhead would be equal to power consumption of two PLLs multiplied by τ. The performance overhead is negligible since $t_{lock} < 100\mu Sec \ll \tau$. Therefore, it is more efficient to use two PLLs when the following condition is hold:

$$P_{design} \times t_{lock} + P_{PLL} \times (\tau + t_{lock}) > 2 \times P_{PLL} \times \tau \tag{3.5}$$

Since $t_{lock} \ll \tau$, we should have $P_{design} \times t_{Lock} > P_{PLL} \times \tau$. Our evaluation shows that this

condition can be always satisfied over all our experiments. In practice, the fully utilized FPGA power consumption is around 20W while the PLL consumes about 0.1W, and $t_{lock} \simeq 10\mu Sec$. Therefore, when $\tau > 2mSec$, the overhead of using two PLL becomes less than using one PLL. In practice, τ is at least in order of seconds or minutes; thus it is always more beneficial to use two PLLs.

3.5 Experimental Results

3.5.1 General Setup

We evaluated the efficiency of the proposed method by implementing several state-of-the-art neural network acceleration frameworks on a commercial FPGA architecture. To generate and characterize the SPICE netlist of FPGA resources from delay and power perspectives, we used the latest version of COFFE [138] with 22nm predictive technology model (PTM) [139] and an architectural description file similar to Stratix IV devices due to their well-provided architectural details [140]. COFFE does not model DSPs, so we hand-crafted a Verilog HDL of Stratix IV DSPs [141] and characterized with Synopsys Design Compiler using NanGate 45nm Open-Cell Library [142] tailored for libraries with different voltages by the means of Synopsys SiliconSmart. Eventually we scaled the 45nm DSP characterization to 22nm following the scaling factors of a subset of combinational and sequential cells obtained through SPICE simulations.

We synthesized the benchmarks using Intel (Altera) Quartus II software targeting Stratix IV devices and converted the resulted VQM (Verilog Quartus Mapping) file format to Berkeley Logic Interchange Format (BLIF) format, recognizable by our placement and routing VTR (Verilog-to-Routing) toolset [140]. VTR gets a synthesized design in BLIF format along with the architectural description of the device (e.g., number of LUTs per slice, routing network information such as wire length, delays, etc.) and maps (i.e., performs place and routing) on the smallest possible FPGA device and simultaneously tries to minimize the delay. The

Table 3.1. Post place and route resource utilization and timing of the benchmarks.

Parameter	Tabla	DnnWeaver	DianNao	Stripes	Proteus
LAB	127	730	3430	12343	2702
DSP	0	1	112	16	144
M9K	47	166	30	15	15
M144K	1	13	2	1	1
I/O	567	1655	4659	8797	5033
Freq. (MHz)	113	99	83	40	70

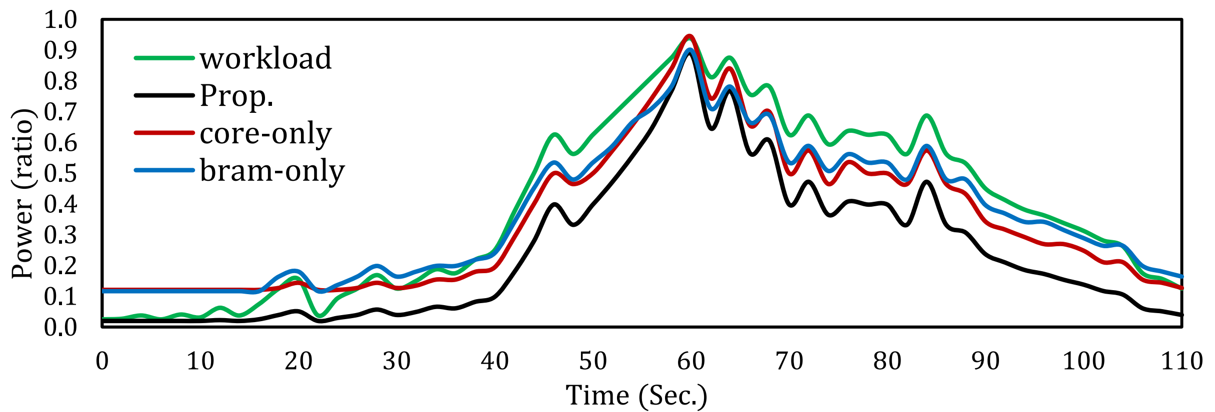

Figure 3.10. Comparing the efficiency of different voltage scaling techniques under a varying workload for Tabla framework.

only amendment we made in the device architecture was to increase the capacity of I/O pads from 2 to 4 as our benchmarks are heavily I/O bound. Our benchmarks include Tabla [53], DnnWeaver [54], DianNao [143], Stripes [144], and Proteus [145] which are general neural network acceleration frameworks capable of optimizing various objective functions through gradient descent by supporting huge parallelism. The last two networks provide serial and variable-precision acceleration for energy efficiency. Table 3.1 summarizes the resource usage and post place and route frequencies of the synthesized benchmarks. LAB stands for Logic Array Block and includes 10 6-input LUTs. M9K and M144K show the number of 9Kb and 144Kb memories.

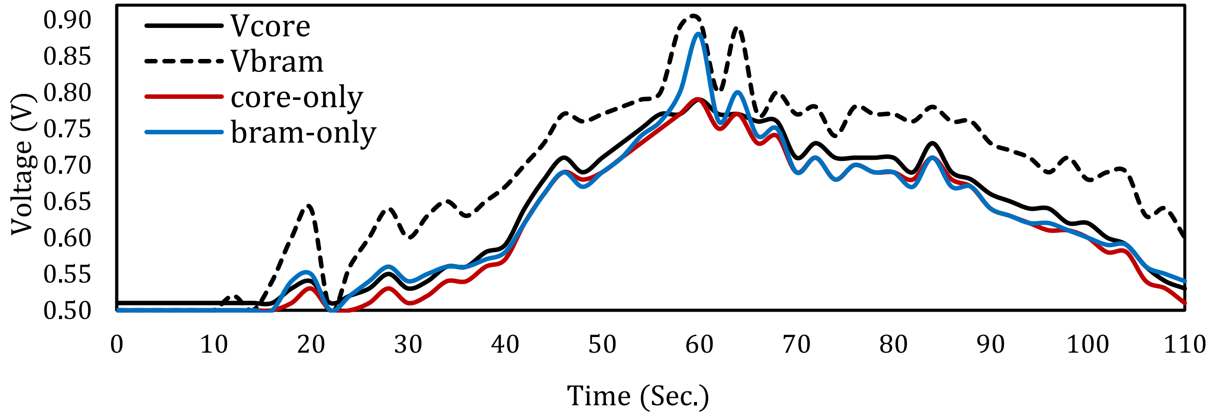

Figure 3.11. Voltage adjustment in different voltage scaling techniques under the varying workload for Tabla framework.

3.5.2 Results

Figure 3.10 compares the achieved power gain of different voltage scaling approaches implemented the Tabla acceleration framework under a varying workload. We considered a synthetic workload with 40% average load (of the maximum) from [146] with $\lambda = 1000$, $H = 0.76$ and $IDC = 500$ where λ, $0.5 < H \leq 1$ and IDC denote the average arrival rate of the whole process, Hurst exponent, and the index of dispersion, respectively. The workload also has been shown in the same figure (in green line) which is normalized to its expected peak load. We have showed the corresponding V_{core} and V_{bram} voltages of all approaches in Figure 3.11. Note that we have not showed V_{bram} (V_{core}) for the core-only (bram-only) techniques as it is fixed 0.95V (0.8V) in this approach. An average of 4.1× power reduction is achieved, while this is 2.9× and 2.7× for the core-only and bram-only approaches. This means that the proposed technique is 41% more efficient than the best approach, i.e., only considering the core voltage rails. An interesting point in Figure 3.11 is the reaction of bram-only approach with respect to workload variation. It follows a similar scaling trend (i.e., slope) as V_{bram} in our approach. However, our method also scales the V_{core} to find more efficient energy point, thus V_{bram} in our proposed approach is always greater than that of bram-only approach.

Figure 3.12 compares the power saving of all accelerator frameworks employing our

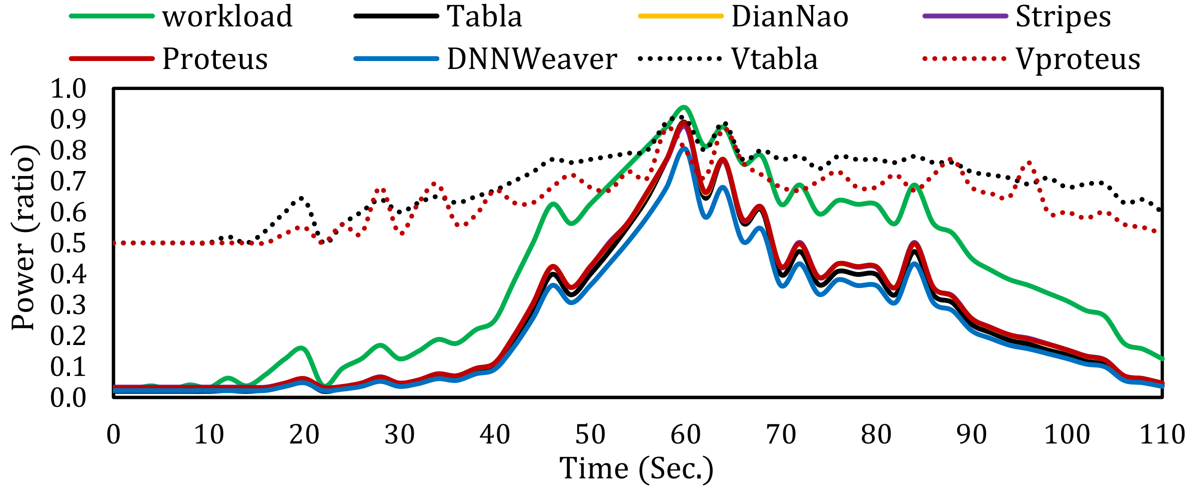

Figure 3.12. Power efficiency of the proposed technique in different acceleration frameworks.

Table 3.2. Comparison of power efficiency of different approaches.

Technique	**Tabla**	**DianNao**	**Stripes**	**Proteus**	**DNNWeav.**	**Average**
Core-only	2.9×	3.1×	3.1×	3.1×	2.9×	3.02×
Bram-only	2.7×	1.9×	1.8×	2.0×	2.9×	2.26×
The proposed	4.1×	3.9×	3.9×	3.8×	4.4×	4.02×
Efficiency	41-52%	26-105%	26-116%	23-90%	52%	33.6% – 83%

proposed method, where they follow a similar trend. This is due to the fact that the workload has considerably higher impact on the opportunity of power saving. We could also infer this from Figure 3.4 where the power efficiency is significantly affected by workload load rather than the application specifications (α and β parameters). In addition, we observed that BRAM delay contributes to a similar portion of critical path delay in all of our accelerators (i.e., α parameters are close). Lastly, the accelerators are heavily I/O-bound which are obliged to be mapped to a considerably larger device where static power of the unused resources is large enough to cover the difference in applications power characteristics. Nevertheless, we have also represented the BRAM voltages of the Table (V_{Tabla} in dashed black line, the same presented in Figure 3.11) and Proteus ($V_{Proteus}$) applications in 3.12. As we can see, although the power trends of these applications almost overlap, they have a noticeably different minimum V_{bram} points.

Table 3.2 summarizes the average power reduction of different voltage scaling schemes

over the aforementioned workload. On average, the proposed scheme reduces the power by 4.0×, which is 33.6% better than the previous core-only and 83% more effective than scaling the V_{bram}. As elaborated in Section 3.2, different power saving in applications (while having the same workload) arises from different factors including the distribution of resources in their critical path where each resource exhibits a different voltage-delay characteristics, as well as the relative utilization of logic/routing and memory resources that affect the optimum point in each approach.

3.6 Conclusion

In this chapter, we proposed an efficient framework to throttle the power consumption of multi-FPGA platforms by effectively scaling the voltage and frequency during runtime. We utilize a light-weight predictor for proactive estimation of the incoming workload and incorporate it to our power-aware timing analysis framework. The timing analysis framework then adjusts the frequency and finds optimal voltages according to the available workload margin, while maintaining the desired quality of service. We evaluated the efficiency of our framework by implementing the state-of-the-art deep neural network accelerators on a solid FPGA architecture. Experimental results signified the efficiency of the proposed method, where we observed 4.0× power improvement, which is 33.6% to 83% more effective than previous approaches that merely consider a single voltage rail. In the next chapter we summarize our work and provide some ideas for future research.

This chapter contains material from "Workload-Aware Opportunistic Energy Efficiency in Multi-FPGA Platforms", by Sahand Salamat, Behnam Khaleghi, Mohsen Imani, and Tajana S. Rosing, which appears in IEEE/ACM International Conference On Computer Aided Design (ICCAD), 2019 [4]. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Summary and Future Work

4.1 Thesis Summary

The rise of the IoT era has massively increased the size of the generated data. Machine learning algorithms have been extensively used to process this data. Executing these algorithms is challenging as require significant resource to provide sufficient accuracy. The goal of our research is to improve the performance and energy-efficiency of big data processing by modifying the software tools and optimizing the hardware platforms. In this dissertation we 1) optimize and accelerate the existing DNNs, 2) accelerate HD computing as a light-weight and hardware friendly alternative to DNNs on FPGAs, and 3) optimize the energy-efficiency of cloud FPGAs where most of the machine learning applications are running.

One of the main challenges in accelerating DNNs is quantizing the networks without loosing the accuracy. Quantizing the networks reduces both the network size and the computation complexity of DNN operations. However, quantizing networks after a certain point (6 bits as shown in [28]) lowers the accuracy to a level that is not practical for real-world applications. In chapter 1, we exploit Residue Number System (RNS), instead of the binary numbers, to simplify DNN operations thereby increasing their performance. Our proposed Residue-Net quantizes the networks to 6 bits and then converts the weights to RNS and entirely executes DNNs using RNS. It utilizes RNS to replace the multiplication operations with more hardware-friendly operations such as shift and addition. We also proposed and FPGA-based accelerator for Residue-Net that

provides 2.8$\times$ speedup on average (up to 3.1$\times$ in ResNet-50) compared to the FPGA baseline while delivering the same accuracy as the DNN quantized to six bits.

DNNs provide good accuracy for various types of applications, but at the cost of complexity, performance and relatively high energy consumption. Many tasks can be processed with more light-weight and hardware-friendly algorithms. Hyperdimensional computing provides a comparable accuracy in many tasks to conventional machine learning algorithms. In chapter 2, we developed an automated framework that generates FPGA-based accelerators for HD classification and clustering, called HD2FPGA. HD2FPGA is highly optimized and parallelized to execute the entire HD training, retraining, and inference for HD classification and clustering on the FPGA. Compared to state-of-the-art FPGA-based machine learning accelerators, HD2FPGA achieves 277$\times$ speedup and 172$\times$ energy reduction in classification as well as 46$\times$ speedup and 47.8$\times$ energy reduction in clustering.

With the ever-increasing demand for application-specific accelerators, many cloud service providers have deployed FPGAs in their cloud infrastructure. They offer machine learning services running on cloud FPGAs; additionally, they offer FPGAs as an infrastructure as a service where users can implement their applications. On average, the size of the incoming workload to the cloud services is $\sim$ 30% of the maximum expected workload. Therefore, most of the time the cloud FPGAs are underutilized which reduces the overall efficiency of these platforms. In chapter 3, we improve the energy efficiency of multi-FPGA platforms commonly used for cloud-based processing. We created a dynamic voltage and frequency scaling approach to minimize the energy consumption of multi-FPGA platforms. Our approach first predicts the size of the incoming workload and then adjusts the frequency and voltage of the FPGAs to deliver the user's desired quality of service while minimizing the energy consumption.

4.2 Future Directions

The constant evolution of IoT systems has led to exponential growth in the size of the annual generated data. As the size of the generated data increases, new problems will be introduced and the computational bottlenecks will change. Conventional machine learning algorithms are widely used in big data applications such as bioinformatics for the classification and clustering of datasets with massive sizes. Exponentially growing genomics data sets and the immense amount of computations required to process these datasets have motivated the researchers to utilize application-specific accelerators and emerging technologies to significantly accelerate bioinformatics applications.

Residue-Net and HD2FPGA are two FPGA-based accelerators that are highly optimized to maximize the performance and minimize the energy consumption of running DNNs and HD computing. These two accelerators and specifically HD2FPGA, thanks to its higher performance, can be utilized in bioinformatics datasets for classifying and clustering the genomics data. Designing an ASIC accelerator for HD classification and clustering can further improve the efficiency of executing HD. ASIC accelerators provide significantly higher flexibility in design and optimization. However, ASIC accelerators, unlike FPGAs, provide limited reconfigurability; therefore, the ASIC accelerator has to support HD with different parameters such as the number of input features, dimensionality, and the number of classes/clusters, to name a few. Using ASIC accelerators for HD computing will address the computational bottleneck issue but as the size of datasets increases, more data has to be transferred from/to the storage devices to/from the computation core.

Today memory and storage access are the computational bottlenecks for many applications. Increasing the performance of systems requires optimizing the underlying algorithms for more efficient memory and storage access. Moving the computations to where the data is stored reduces the data movement in the memory hierarchy (storage, memory, and on-chip memory of computation cores). Emerging computational paradigms, such as in-memory and

in-storage computing, are getting ever-increasing attention as they directly address the data movement issues. NVM memories provide the ability to directly execute operations in memory cells, which leads to high parallelism and high efficiency in executing operations on the stored data. Additionally, computational storage devices can be equipped with custom accelerators inside the SSD controller to execute operations on the stored data. The computational storage devices benefit from the higher internal bandwidth of SSDs in addition to the efficiency of custom accelerators. Executing HD in storage and in memory can improve the efficiency of HD computing by orders of magnitude.

In summary, going forward, it is important to develop an infrastructure that can learn efficiently on massive data sets. Some ideas on how to accomplish this include: 1) Using our already developed accelerators, Residue-Net and HD2FPGA, to process applications with massive data size. 2) Developing an ASIC accelerator for HD computing to increase our capacity to process more data with less energy. 3) Developing an in-storage or in-memory accelerator for HD computing to improve the efficiency of performing tasks (e.g. classification, and clustering) by orders of magnitude.

Bibliography

[1] M. Samragh, M. Ghasemzadeh, and F. Koushanfar, "Customizing neural networks for efficient fpga implementation," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 85–92, IEEE, 2017.

[2] "State-of-the-art fpga-based acceleratof for kmeans clustering." https://github.com/Xilinx/Vitis_Accel_Examples/tree/master/demo/kmeans.

[3] S. Salamat, S. Shubhi, B. Khaleghi, and T. Rosing, "Residue-net: Multiplication-free neural network by in-situ no-loss migration to residue number systems," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 222–228, IEEE, 2021.

[4] S. Salamat, B. Khaleghi, M. Imani, and T. Rosing, "Workload-aware opportunistic energy efficiency in multi-fpga platforms," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2019.

[5] M. Ge, H. Bangui, and B. Buhnova, "Big data for internet of things: a survey," *Future generation computer systems*, vol. 87, pp. 601–614, 2018.

[6] A. Oussous, F.-Z. Benjelloun, A. A. Lahcen, and S. Belfkih, "Big data technologies: A survey," *Journal of King Saud University-Computer and Information Sciences*, vol. 30, no. 4, pp. 431–448, 2018.

[7] "Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025." https://www.statista.com/statistics/871513/worldwide-data-created.

[8] E. Adi, A. Anwar, Z. Baig, and S. Zeadally, "Machine learning and data analytics for the iot," *Neural Computing and Applications*, vol. 32, no. 20, pp. 16205–16233, 2020.

[9] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey and benchmarking of machine learning accelerators," in *2019 IEEE high performance extreme computing conference (HPEC)*, pp. 1–9, IEEE, 2019.

[10] Z. Li, Y. Zhang, J. Wang, and J. Lai, "A survey of fpga design for ai era," *Journal of Semiconductors*, vol. 41, no. 2, p. 021402, 2020.

[11] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[dl] a survey of fpga-based neural network inference accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 1, pp. 1–26, 2019.

[12] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[13] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[14] A. Bashar, "Survey on evolving deep learning neural network architectures," *Journal of Artificial Intelligence*, vol. 1, no. 02, pp. 73–82, 2019.

[15] P. Druzhkov and V. Kustikova, "A survey of deep learning methods and software tools for image classification and object detection," *Pattern Recognition and Image Analysis*, vol. 26, no. 1, pp. 9–15, 2016.

[16] N. Ibrahim, P. Maurya, O. Jafari, and P. Nagarkar, "A survey of performance optimization in neural network-based video analytics systems," *arXiv preprint arXiv:2105.14195*, 2021.

[17] C. Morikawa, M. Kobayashi, M. Satoh, Y. Kuroda, T. Inomata, H. Matsuo, T. Miura, and M. Hilaga, "Image and video processing on mobile devices: a survey," *The Visual Computer*, pp. 1–19, 2021.

[18] M. Alam, M. D. Samad, L. Vidyaratne, A. Glandon, and K. M. Iftekharuddin, "Survey on deep neural networks in speech and vision systems," *Neurocomputing*, vol. 417, pp. 302–321, 2020.

[19] N. H. Tandel, H. B. Prajapati, and V. K. Dabhi, "Voice recognition and voice comparison using machine learning techniques: A survey," in *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pp. 459–465, IEEE, 2020.

[20] J. Ruiz-del Solar, P. Loncomilla, and N. Soto, "A survey on deep learning methods for robot vision," *arXiv preprint arXiv:1803.10862*, 2018.

[21] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[22] A. I. Károly, P. Galambos, J. Kuti, and I. J. Rudas, "Deep learning in robotics: Survey on model structures and training strategies," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 51, no. 1, pp. 266–279, 2020.

[23] H. Bhaskar, D. C. Hoyle, and S. Singh, "Machine learning in bioinformatics: A brief survey and recommendations for practitioners," *Computers in biology and medicine*, vol. 36, no. 10, pp. 1104–1125, 2006.

[24] K. Lan, D.-t. Wang, S. Fong, L.-s. Liu, K. K. Wong, and N. Dey, "A survey of data mining and deep learning in bioinformatics," *Journal of medical systems*, vol. 42, no. 8, pp. 1–20, 2018.

[25] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. G. Rapsang, S. K. Reinhardt, B. Darvish Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Y. Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[26] Y. Guo, "A survey on methods and theories of quantized neural networks," *arXiv preprint arXiv:1808.04752*, 2018.

[27] S. Migacz, "Nvidia 8-bit inference width tensorrt," in *GPU Technology Conference*, 2017.

[28] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, Y. Zhang, B. Zimmer, W. J. Dally, J. Emer, S. W. Keckler, Khailany, and Brucek, "Magnet: A modular accelerator generator for neural networks," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2019.

[29] A. Jain, P. Goel, S. Aggarwal, A. Fell, and S. Anand, "Symmetric k-means for deep neural network compression and hardware acceleration on fpgas," *IEEE Journal of Selected Topics in Signal Processing*, 2020.

[30] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 764–775, IEEE, 2018.

[31] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M. Martina, "An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks," *Future Internet*, vol. 12, no. 7, p. 113, 2020.

[32] R. Ayachi, Y. Said, and A. B. Abdelali, "Optimizing neural networks for efficient fpga implementation: A survey," *Archives of Computational Methods in Engineering*, pp. 1–11, 2021.

[33] A. G. Blaiech, K. B. Khalifa, C. Valderrama, M. A. Fernandes, and M. H. Bedoui, "A survey and taxonomy of fpga-based deep learning accelerators," *Journal of Systems Architecture*, vol. 98, pp. 331–345, 2019.

[34] T. Simons and D.-J. Lee, "A review of binarized neural networks," *Electronics*, vol. 8, no. 6, p. 661, 2019.

[35] A. R. Omondi and B. Premkumar, *Residue number systems: theory and implementation*. World Scientific, 2007.

[36] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, 2009.

[37] P. Kanerva, "Computing with 10,000-bit words," in *Communication, Control, and Computing (Allerton), 2014 52nd Annual Allerton Conference on*, pp. 304–310, IEEE, 2014.

[38] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 64–69, ACM, 2016.

[39] F. R. Najafabadi, A. Rahimi, P. Kanerva, and J. M. Rabaey, "Hyperdimensional computing for text classification," in *Design, Automation Test in Europe Conference Exhibition (DATE), University Booth*, pp. 1–1, 2016.

[40] O. J. Räsänen and J. P. Saarinen, "Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns," *IEEE transactions on neural networks and learning systems*, vol. 27, no. 9, pp. 1878–1889, 2016.

[41] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *Rebooting Computing (ICRC), 2017 IEEE International Conference on*, pp. 1–8, IEEE, 2017.

[42] F. Montagna, A. Rahimi, S. Benatti, D. Rossi, and L. Benini, "Pulp-hd: accelerating brain-inspired high-dimensional computing on a parallel ultra-low power platform," in *Proceedings of the 55th Annual Design Automation Conference*, p. 111, ACM, 2018.

[43] O. Rasanen and J. Saarinen, "Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, no. 99, pp. 1–12, 2015.

[44] A. Joshi, J. Halseth, and P. Kanerva, "Language geometry using random indexing," *Quantum Interaction 2016 Conference Proceedings*, In press.

[45] Y. Guo, M. Imani, J. Kang, S. Salamat, J. Morris, B. Aksanli, Y. Kim, and T. Rosing, "Hyperrec: Efficient recommender systems with hyperdimensional computing," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 384–389, IEEE, 2021.

[46] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, ACM, 2017.

[47] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, "F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 53–62, 2019.

[48] A. Bhattacherjee and S. C. Park, "Why end-users move to the cloud: a migration-theoretic analysis," *European Journal of Information Systems*, vol. 23, no. 3, pp. 357–372, 2014.

[49] M. Wahlroos, M. Pärssinen, S. Rinne, S. Syri, and J. Manner, "Future views on waste heat utilization–case of data centers in northern europe," *Renewable and Sustainable Energy Reviews*, vol. 82, pp. 1749–1764, 2018.

[50] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner, "United states data center energy usage report," 2016.

[51] H.-W. Tseng, T.-T. Yang, K.-C. Yang, and P.-S. Chen, "An energy efficient vm management scheme with power-law characteristic in video streaming data centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 297–311, 2018.

[52] B. Khaleghi and T. Š. Rosing, "Thermal-aware design and flow for fpga performance improvement," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 342–347, IEEE, 2019.

[53] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 14–26, IEEE, 2016.

[54] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[55] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, pp. 1–4, 2015.

[56] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Larus, James Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 13–24, IEEE, 2014.

[57] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached fpgas for data center applications," in *2016 International Conference on Field-Programmable Technology (FPT)*, pp. 36–43, IEEE, 2016.

[58] A. Altomare, E. Cesario, and A. Vinci, "Data analytics for energy-efficient clouds: design, implementation and evaluation," *International Journal of Parallel, Emergent and Distributed Systems*, pp. 1–16, 2018.

[59] W. Sung, S. Shin, and K. Hwang, "Resiliency of deep neural networks under quantization," *arXiv preprint arXiv:1511.06488*, 2015.

[60] K. Hwang and W. Sung, "Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1," in *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 1–6, IEEE, 2014.

[61] H. Wi, H. Kim, S. Choi, and L.-S. Kim, "Compressing sparse ternary weight convolutional neural networks for efficient hardware acceleration," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2019.

[62] D. Zhang, J. Yang, D. Ye, and G. Hua, "Lq-nets: Learned quantization for highly accurate and compact deep neural networks," in *Proceedings of the European conference on computer vision (ECCV)*, pp. 365–382, 2018.

[63] H. Yan, A. H. Aboutalebi, and L. Duan, "Efficient allocation and heterogeneous composition of nvm crossbar arrays for deep learning acceleration," in *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8, IEEE, 2018.

[64] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8612–8620, 2019.

[65] D. J. Pagliari, E. Macii, and M. Poncino, "Dynamic bit-width reconfiguration for energy-efficient deep learning hardware," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 1–6, 2018.

[66] M. Nazemi and M. Pedram, "Deploying customized data representation and approximate computing in machine learning applications," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 1–6, 2018.

[67] E. Park, D. Kim, S. Kim, Y.-D. Kim, G. Kim, S. Yoon, and S. Yoo, "Big/little deep neural network for ultra low power inference," in *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pp. 124–132, IEEE, 2015.

[68] R. Ding, Z. Liu, R. Shi, D. Marculescu, and R. Blanton, "Lightnn: Filling the gap between conventional deep neural networks and binarized networks," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pp. 35–40, 2017.

[69] N. Khoshavi, C. Broyles, and Y. Bi, "Compression or corruption? a study on the effects of transient faults on bnn inference accelerators," in *2020 21st International Symposium on Quality Electronic Design (ISQED)*, pp. 99–104, IEEE, 2020.

[70] P. Hill, B. Zamirai, S. Lu, Y.-W. Chao, M. Laurenzano, M. Samadi, M. Papaefthymiou, S. Mahlke, T. Wenisch, J. Deng, L. Tang, and J. Mars, "Rethinking numerical representations for deep neural networks," *arXiv preprint arXiv:1808.02513*, 2018.

[71] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017.

[72] S. Salamat, M. Imani, S. Gupta, and T. Rosing, "Rnsnet: In-memory neural network acceleration using residue number system," in *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–12, IEEE, 2018.

[73] Y. Gu, T. Wahl, M. Bayati, and M. Leeser, "Behavioral non-portability in scientific numeric computing," in *European conference on Parallel Processing*, pp. 558–569, Springer, 2015.

[74] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Deep positron: A deep neural network using the posit number system," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1421–1426, IEEE, 2019.

[75] J. Lu, S. Lu, Z. Wang, C. Fang, J. Lin, Z. Wang, and L. Du, "Training deep neural networks using posit number system," *arXiv preprint arXiv:1909.03831*, 2019.

[76] Y. Uguen, L. Forget, and F. de Dinechin, "Evaluating the hardware cost of the posit number system," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 106–113, IEEE, 2019.

[77] G. C. Cardarilli, A. Nannarelli, and M. Re, "Residue number system for low-power dsp applications," in *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*, pp. 1412–1416, IEEE, 2007.

[78] H. Nakahara and T. Sasao, "A deep convolutional neural network based on nested residue number system," in *FPL*, IEEE, 2015.

[79] N. Samimi, M. Kamal, A. Afzalli-Kusha, and M. Pedram, "Res-dnn: A residue number system-based dnn accelerator unit," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2019.

[80] K. Anitha, T. Arulananth, R. Karthik, and P. B. Reddy, "Design and implementation of modified sequential parallel rns forward converters," *International Journal of Applied Engineering Research*, vol. 12, no. 16, pp. 6159–6163, 2017.

[81] R. de Matos, R. Paludo, N. Chervyakov, P. A. Lyakhov, and H. Pettenghi, "Efficient implementation of modular multiplication by constants applied to rns reverse converters," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, IEEE, 2017.

[82] E. B. Olsen, "Rns hardware matrix multiplier for high precision neural network acceleration:" rns tpu"," in *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*, pp. 1–5, IEEE, 2018.

[83] H. Nakahara and T. Sasao, "A high-speed low-power deep neural network on an fpga based on the nested rns: Applied to an object detector," in *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*, pp. 1–5, IEEE, 2018.

[84] V. Krasnobayev, A. Yanko, and S. Koshman, "A method for arithmetic comparison of data represented in a residue number system," *Cybernetics and Systems Analysis*, vol. 52, no. 1, pp. 145–150, 2016.

[85] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[86] "Hd2fpga tool." https://github.com/seelab-ucsd/HD2FPGA.git.

[87] M. Schmuck, L. Benini, and A. Rahimi, "Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory," *arXiv preprint arXiv:1807.08583*, 2018.

[88] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, "Exploring hyperdimensional associative memory," in *2017 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 445–456, IEEE, 2017.

[89] M. Imani, J. Morris, J. Messerly, H. Shu, Y. Deng, and T. Rosing, "Bric: Locality-based encoding for energy-efficient brain-inspired hyperdimensional computing," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[90] S. Salamat, M. Imani, and T. Rosing, "Accelerating hyperdimensional computing on fpgas by exploiting computational reuse," *IEEE Transactions on Computers*, 2020.

[91] B. Khaleghi, S. Salamat, A. Thomas, F. Asgarinejad, Y. Kim, and T. Rosing, "Shear er: highly-efficient hyperdimensional computing by software-hardware enabled multifold approximation," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 241–246, 2020.

[92] G. Karunaratne, M. Le Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian, "In-memory hyperdimensional computing," *Nature Electronics*, vol. 3, no. 6, pp. 327–337, 2020.

[93] L. Ge and K. K. Parhi, "Classification using hyperdimensional computing: A review," *IEEE Circuits and Systems Magazine*, vol. 20, no. 2, pp. 30–47, 2020.

[94] M. Imani, Y. Kim, T. Worley, S. Gupta, and T. Rosing, "Hdcluster: An accurate clustering using brain-inspired high-dimensional computing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1591–1594, IEEE, 2019.

[95] M. Imani, J. Hwang, T. Rosing, A. Rahimi, and J. M. Rabaey, "Low-power sparse hyperdimensional encoder for language recognition," *IEEE Design & Test*, vol. 34, no. 6, pp. 94–101, 2017.

[96] M. Imani, T. Nassar, A. Rahimi, and T. Rosing, "Hdna: Energy-efficient dna sequencing using hyperdimensional computing," in *2018 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI)*, pp. 271–274, IEEE, 2018.

[97] Y. Kim, M. Imani, and T. S. Rosing, "Efficient human activity recognition using hyperdimensional computing," in *Proceedings of the 8th International Conference on the Internet of Things*, pp. 1–6, 2018.

[98] M. Imani, S. Salamat, S. Gupta, J. Huang, and T. Rosing, "Fach: Fpga-based acceleration of hyperdimensional computing by reducing computational complexity," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 493–498, 2019.

[99] A. Rahimi, S. Benatti, P. Kanerva, L. Benini, and J. M. Rabaey, "Hyperdimensional biosignal processing: A case study for emg-based hand gesture recognition," in *Rebooting Computing (ICRC), IEEE International Conference on*, pp. 1–8, IEEE, 2016.

[100] M. Imani, J. Messerly, F. Wu, W. Pi, and T. Rosing, "A binary learning framework for hyperdimensional computing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 126–131, IEEE, 2019.

[101] A. Rahimi, A. Tchouprina, P. Kanerva, J. d. R. Millán, and J. M. Rabaey, "Hyperdimensional computing for blind and one-shot classification of eeg error-related potentials," *Mobile Networks and Applications*, vol. 25, no. 5, pp. 1958–1969, 2020.

[102] F. Asgarinejad, A. Thomas, and T. Rosing, "Detection of epileptic seizures from surface eeg using hyperdimensional computing," in *2020 42nd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, pp. 536–540, IEEE, 2020.

[103] P. Neubert, S. Schubert, and P. Protzel, "An introduction to hyperdimensional computing for robotics," *KI-Künstliche Intelligenz*, vol. 33, no. 4, pp. 319–330, 2019.

[104] M. Hersche, E. M. Rella, A. Di Mauro, L. Benini, and A. Rahimi, "Integrating event-based dynamic vision sensors with sparse hyperdimensional computing: A low-power accelerator with online learning capability," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 169–174, 2020.

[105] B. Khaleghi, H. Xu, J. Morris, and T. Š. Rosing, "tiny-hd: Ultra-efficient hyperdimensional computing engine for iot applications," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 408–413, IEEE, 2021.

[106] A. Menon, A. Natarajan, R. Agashe, D. Sun, M. Aristio, H. Liew, Y. S. Shao, and J. M. Rabaey, "Efficient emotion recognition using hyperdimensional computing with combinatorial channel encoding and cellular automata," *arXiv preprint arXiv:2104.02804*, 2021.

[107] S. Datta, R. A. Antonio, A. R. Ison, and J. M. Rabaey, "A programmable hyper-dimensional processor architecture for human-centric iot," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 439–452, 2019.

[108] N. Karvonen, J. Nilsson, D. Kleyko, and L. L. Jiménez, "Low-power classification using fpga—an approach based on cellular automata, neural networks, and hyperdimensional computing," in *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pp. 370–375, IEEE, 2019.

[109] M. Imani, Z. Zou, S. Bosch, S. A. Rao, S. Salamat, V. Kumar, Y. Kim, and T. Rosing, "Revisiting hyperdimensional learning for fpga and low-power architectures," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 221–234, IEEE, 2021.

[110] M. Imani, C. Huang, D. Kong, and T. Rosing, "Hierarchical hyperdimensional computing for energy efficient classification," in *Proceedings of the 55th Annual Design Automation Conference*, p. 108, ACM, 2018.

[111] T. F. Wu, H. Li, P.-C. Huang, A. Rahimi, J. M. Rabaey, H.-S. P. Wong, M. M. Shulaker, and S. Mitra, "Brain-inspired computing exploiting carbon nanotube fets and resistive ram: Hyperdimensional computing case study," in *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International*, pp. 492–494, IEEE, 2018.

[112] H. Li, T. F. Wu, A. Rahimi, K.-S. Li, M. Rusch, C.-H. Lin, J.-L. Hsu, M. M. Sabry, S. B. Eryilmaz, J. Sohn, W.-C. Chiu, M.-C. Chen, T.-T. Wu, J.-M. Shieh, W.-K. Yeh, J. M. Rabaey, S. Mitra, and W. H.-S. Philip, "Hyperdimensional computing with 3d vrram in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition," in *2016 IEEE International Electron Devices Meeting (IEDM)*, pp. 16–1, IEEE, 2016.

[113] S. Gupta, M. Imani, and T. Rosing, "Felix: Fast and energy-efficient logic in memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7, IEEE, 2018.

[114] P. Poduval, Z. Zouψ, H. Najafi, H. Homayoun, and M. Imani, "Stochd: Stochastic hyperdimensional system for efficient and robust learning from raw data," in *IEEE/ACM DAC*, 2021.

[115] J. Morris, K. Ergun, B. Khaleghi, M. Imani, B. Aksanli, and T. Rosing, "Hydrea: Towards more robust and efficient machine learning systems with hyperdimensional computing," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 723–728, IEEE, 2021.

[116] "Vitis unified software platform." https://www.xilinx.com/products/design-tools/vitis.html.

[117] "Xilinx runtime library (xrt)." https://www.xilinx.com/products/design-tools/vitis/xrt.html.

[118] "Intel quartus prime software suite." https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html.

[119] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/ISOLET.

[120] S. Yazdanshenas and V. Betz, "Quantifying and mitigating the costs of fpga virtualization," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–7, IEEE, 2017.

[121] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling fpgas in hyperscale data centers," in *IEEE Intl Conf on Ubiquitous Intelligence and Computing (UIC-ATC-ScalCom)*, pp. 1078–1086, IEEE, 2015.

[122] C. T. Chow, L. S. M. Tsui, P. H. W. Leong, W. Luk, and S. J. Wilton, "Dynamic voltage scaling for commercial fpgas," in *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology, 2005.*, pp. 173–180, IEEE, 2005.

[123] J. M. Levine, E. Stott, and P. Y. Cheung, "Dynamic voltage & frequency scaling with online slack measurement," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pp. 65–74, ACM, 2014.

[124] S. Zhao, I. Ahmed, C. Lamoureux, A. Lotfi, V. Betz, and O. Trescases, "A universal self-calibrating dynamic voltage and frequency scaling (dvfs) scheme with thermal compensation for energy savings in fpgas," in *2016 IEEE Applied Power Electronics Conference and Exposition (APEC)*, pp. 1882–1887, IEEE, 2016.

[125] H. Amrouch, B. Khaleghi, A. Gerstlauer, and J. Henkel, "Reliability-aware design to suppress aging," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2016.

[126] M. Ahmadi, S. Salamat, and B. Alizadeh, "A dynamic timing error avoidance technique using prediction logic in high-performance designs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 3, pp. 734–737, 2018.

[127] B. Salami, O. S. Unsal, and A. C. Kestelman, "Comprehensive evaluation of supply voltage underscaling in fpga on-chip memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 724–736, IEEE, 2018.

[128] P. H. Jones, Y. H. Cho, and J. W. Lockwood, "Dynamically optimizing fpga applications by monitoring temperature and workloads," in *International Conference on VLSI Design (VLSID'07)*, pp. 391–400, IEEE, 2007.

[129] S. Yazdanshenas, K. Tatsumura, and V. Betz, "Don't forget the memory: Automatic block ram modelling, optimization, and architecture exploration," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 115–124, ACM, 2017.

[130] C. Chiasson and V. Betz, "Coffe: Fully-automated transistor sizing for fpgas," in *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 34–41, IEEE, 2013.

[131] N. Bonvin, T. G. Papaioannou, and K. Aberer, "Autonomic sla-driven provisioning for cloud applications," in *IEEE/ACM international symposium on cluster, cloud and grid computing*, pp. 434–443, IEEE Computer Society, 2011.

[132] Q. Zhu and G. Agrawal, "Resource provisioning with budget constraints for adaptive applications in cloud environments," in *ACM International Symposium on High Performance Distributed Computing*, pp. 304–307, ACM, 2010.

[133] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, 2012.

[134] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using arima model and its impact on cloud applications' qos," *IEEE Transactions on Cloud Computing*, vol. 3, no. 4, pp. 449–458, 2015.

[135] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *2010 International Conference on Network and Service Management*, pp. 9–16, Ieee, 2010.

[136] "Texas instruments (ti), "fusion digital power designer"." http://www.ti.com/tool/FUSION_DIGITAL_POWER_DESIGNER.

[137] R. Jain, B. M. Geuskens, S. T. Kim, M. M. Khellah, J. Kulkarni, J. W. Tschanz, and V. De, "A 0.45–1 v fully-integrated distributed switched capacitor dc-dc converter with high density mim capacitor in 22 nm tri-gate cmos," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 4, pp. 917–927, 2014.

[138] S. Yazdanshenas and V. Betz, "Coffe 2: Automatic modelling and optimization of complex and heterogeneous fpga architectures," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 1, p. 3, 2019.

[139] "Predictive technology model."

[140] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "Vtr 7.0: Next generation architecture and cad system for fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, pp. 1–30, 2014.

[141] "Stratix iv device handbook." Datasheet, September 2014.

[142] "Nangate open cell library."

[143] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” in *ACM Sigplan Notices*, vol. 49, pp. 269–284, ACM, 2014.

[144] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, “Stripes: Bit-serial deep neural network computing,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[145] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos, “Proteus: Exploiting numerical precision variability in deep neural networks,” in *Proceedings of the 2016 International Conference on Supercomputing*, pp. 1–12, 2016.

[146] J. Yin, X. Lu, X. Zhao, H. Chen, and X. Liu, “Burse: A bursty and self-similar workload generator for cloud computing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 668–680, 2014.