UNIVERSITY OF CALIFORNIA SAN DIEGO

Efficient and Secure Learning across Memory Hierarchy

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

in

Computer Science (Computer Engineering)

by

Saransh Gupta

2021

DEDICATION

To my parents

For their endless love, support, and encouragement to go on and complete this journey

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to first thank my advisor, Professor Tajana S. Rosing for her guidance, support, and appreciation during my Ph.D. I am extremely grateful for her understanding in all aspects of my life. Her enthusiasm and willingness to explore new domains have always encouraged me to look at problems from different perspectives. I would also like to thank my dissertation committee members, Professors Chung-Kuan Cheng, Ryan Kastner, Farinaz Koushanfar, and Jishen Zhao for their feedback and discussions related to my Ph.D. work. I would like to thank all my colleagues in SEELab for their active collaboration, continuous support, and the lovely memories. I would like to give special thanks to Mohsen Imani, Yeseong Kim, and Behnam Khaleghi. I want to sincerely thank Rosario Cammarota for all the support and guidance he provided during the joint project. My research was made possible by funding from the National Science Foundation (NSF) Grant 1527034, 1730158, 1826967, 1911095, 2003279, CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and SRC-Global Research Collaboration grant.

Most importantly, I owe so much to my parents and all my family for their love, understanding, and patience that helped me navigate through the difficult moments during the last years. Finally, I thank all my friends whose constant support and liveliness kept me going everyday.

The material in this dissertation is based on the following publications.

Chapter 2, in part, is a reprint of the material as it appears in S. Gupta, M. Imani, and T. Rosing, “FELIX: Fast and Energy-Efficient Logic in Memory,” *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2018. The dissertation author was the primary investigator and author of this material.

Chapter 2, in part, is currently being prepared for submission for publication of the material. S. Gupta, J. Morris, X. Shen, M. Imani, B. Aksanli, and T. Rosing, “Tri-HD: Train, Re-train, and Infer with Hyperdimensional Computing in Memory.” The dissertation author was the primary investigator and author of this material.

Chapter 3, in part, is a reprint of the material as it appears in S. Gupta, J. Morris, M. Imani, R. Ramkumar, J. Yu, A. Tiwari, B. Aksanli, and T. Rosing, “THRIFTY: Training with Hyperdimensional Computing across Flash Hierarchy,” *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2020. The dissertation author was the primary investigator and author of this material.

Chapter 3, in part, is currently being prepared for submission for publication of the material. S. Gupta, B. Khaleghi, S. Salamat, J. Morris, R. Ramkumar, J. Yu, A. Tiwari, M. Imani, B. Aksanli, and T. Rosing, “Store-n-Learn: Classification and Clustering with Hyperdimensional Computing across Flash Hierarchy.” The dissertation author was the primary investigator and author of this material.

Chapter 4, in part, has been submitted for publication of the material as it may appear in S. Gupta and T. Rosing, “Accelerating Fully Homomorphic Encryption with Processing in Memory,” *Design Automation Conference (DAC)*, 2021. The dissertation author was the primary investigator and author of this material.

Chapter 4, in part, is currently being prepared for submission for publication of the material. S. Gupta, R. Cammarota, and T. Rosing, “MemFHE: End-to-End Computing with Fully Homomorphic Encryption in Memory.” The dissertation author was the primary investigator and author of this material.

My co-authors (Baris Aksanli, Rosario Cammarota, Mohsen Imani, Behnam Khaleghi, Justin Morris, Ranganathan Ramkumar, Prof. Tajana S. Rosing, Sahand Salamat, Xincheng Shen, Aniket Tiwari, and Jeffrey Yu, listed in alphabetical order) have all kindly approved the inclusion of the aforementioned publications in my dissertation.

VITA

2016	Bachelor of Engineering (Honors) in Electrical and Electronics Engineering, Birla Institute of Technology and Science (BITS) Pilani, K. K. Birla Goa Campus, India
2018	Master of Science in Electrical and Computer Engineering, University of California San Diego, USA
2021	Doctor of Philosophy in Computer Science (Computer Engineering), University of California San Diego, USA

PUBLICATIONS

Saransh Gupta, and Tajana Simunic Rosing, "Accelerating Fully Homomorphic Encryption with Processing in Memory," in *the Design Automation Conference (DAC), 2021.*

Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Simunic Rosing, "Digital-based Processing In-Memory for Acceleration of Unsupervised Learning," in *GOMACTech Conference, 2021.*

Minxuan Zhou, Mohsen Imani, Yeseong Kim, Saransh Gupta, and Tajana Simunic Rosing, "DP-Sim: A Full-stack Simulation Infrastructure for Digital Processing In-Memory Architectures," in *26th Asia and South Pacific Design Automation Conference (ASP-DAC), 2021.*

Mohsen Imani, Saikishan Pampana, Saransh Gupta, Minxuan Zhou, Yeseong Kim, and Tajana Simunic Rosing, "DUAL: Acceleration of Clustering Algorithms using Digital-based Processing In-Memory," in *IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020.*

Saransh Gupta, Justin Morris, Mohsen Imani, Ranganathan Ramkumar, Jeffrey Yu, Aniket Tiwari, Baris Aksanli, and Tajana Simunic Rosing, "THRIFTY: Training with Hyperdimensional Computing across Flash Hierarchy," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2020.*

Hamid Nejatollahi*, Saransh Gupta*, Mohsen Imani, Rosario Cammarota, Tajana Simunic Rosing, and Nikil Dutt, "CryptoPIM: In-Memory Acceleration for RLWE Lattice-based Cryptography," in *Design Automation Conference (DAC), 2020.*

Saransh Gupta, Mohsen Imani, Behnam Khaleghi, Niema Moshiri, and Tajana Simunic Rosing, "RAPIDx: A ReRAM Processing in-Memory Architecture for DNA Short Read Alignment," in *70th Virtual Meeting of The American Society of Human Genetics (ASHG), 2020.*

Saransh Gupta, Mohsen Imani, Hengyu Zhao, Fan Wu, Jishen Zhao, and Tajana Simunic Rosing, "Implementing Binary Neural Networks in Memory with Approximate Accumulation," in

ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2020.

Mohsen Imani, Mohammad Samragh, Yeseong Kim, Saransh Gupta, Farinaz Koushanfar, and Tajana Simunic Rosing, "Deep Learning Acceleration with Neuron-to-Memory Transformation," in *Proceeding of the IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2020.*

Saransh Gupta, Mohsen Imani, Joonseop Sim, Andrew Huang, Fan Wu, M. Hassan Najafi, and Tajana Simunic Rosing, "SCRIMP: A General Stochastic Computing Architecture using ReRAM in-Memory Processing," in *Proceeding of the Design, Automation & Test in Europe (DATE), 2020.*

Justin Morris, Yilun Hao, Saransh Gupta, Ranganathan Ramkumar, Jeffrey Yu, Mohsen Imani, Baris Aksanli, and Tajana Simunic Rosing, "Multi-label HD Classification in 3D Flash," in *Proceedings of IFIP/IEEE International Conference on VLSI and System-on-Chip (VLSI-SoC), 2020.*

Rosario Cammarota, Matthias Schunter, Anand Rajan, Fabian Boemer, Ágnes Kiss, Amos Treiber, Christian Weinert, Thomas Schneider, Emmanuel Stapf, Ahmad-Reza Sadeghi, Daniel Demmler, Huili Chen, Siam Umar Hussain, Sadegh Riazi, Farinaz Koushanfar, Saransh Gupta, Tajana Simunic Rosing, Kamalika Chaudhuri, Hamid Nejatollahi, Nikil Dutt, Mohsen Imani, Kim Laine, Anuj Dubey, Aydin Aysu, Fateme Sadat Hosseini, Chengmo Yang, Eric Wallace, and Pamela Norton, "Trustworthy AI Inference Systems: An Industry Research View," *arXiv preprint arXiv:2008.04449, 2020.*

Saransh Gupta, Mohsen Imani, Harveen Kaur, and Tajana Simunic Rosing, "NNPIM: A Processing In-Memory Architecture for Neural Network Acceleration," in *IEEE Transactions on Computers, 2019.*

Saransh Gupta, Mohsen Imani, Behnam Khaleghi, Venkatesh Ramesh Kumar, and Tajana Simunic Rosing, "RAPID: A ReRAM Processing in-Memory Architecture for DNA Sequence Alignment," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED), 2019.*

Mohsen Imani, Saransh Gupta, Yeseong Kim, Tajana Simunic Rosing, "FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA), 2019.*

Mohsen Imani, Saransh Gupta, and Tajana Simunic Rosing, "Digital-based Processing In-Memory: A Highly-Parallel Accelerator for Data Intensive Applications," in *ACM International Symposium on Memory Systems (MEMSYS), 2019.*

Minxuan Zhou, Mohsen Imani, Saransh Gupta, Tajana Simunic Rosing, "Thermal-Aware Design and Management for Search-based In-Memory Acceleration," in *Proceeding of the Design Automation Conference (DAC), 2019.*

Saransh Gupta, Mohsen Imani, and Tajana Simunic Rosing, “Exploring Processing In-Memory for Different Memory Technologies,” in *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI), 2019*.

Joonseop Sim*, Saransh Gupta*, Mohsen Imani, and Tajana Simunic Rosing, “UPIM : Unipolar Switching Logic for High Density Processing-In-Memory Applications,” in *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI), 2019*.

Mohsen Imani, Ricardo Garcia, Saransh Gupta, and Tajana Simunic Rosing, “Hardware- Software Co-design to Accelerate Neural Network Applications,” in *Journal on Emerging Technologies in Computing Systems (JETC), 2019*.

Mohsen Imani, Xunzhao Yin, John Messerly, Saransh Gupta, Michael Niemier, Xiaobo Sharon Hu, and Tajana Simunic Rosing, “SearcHD: A Memory-Centric Hyperdimensional Computing with Stochastic Training,” in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2019*.

Mohsen Imani, Saransh Gupta, Yeseong Kim, Minxuan Zhou, and Tajana Simunic Rosing, “DigitalPIM: Digital-based Processing In-Memory for Big Data Acceleration,” in *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI), 2019*.

Mohsen Imani, Yeseong Kim, Thomas Worley, Saransh Gupta, Tajana Simunic Rosing, “HDCluster: An Accurate Clustering Using Brain-Inspired High-Dimensional Computing,” in *Proceedings of the IEEE Design, Automation & Test in Europe (DATE), 2019*.

Joonseop Sim, Minsu Kim, Yeseong Kim, Saransh Gupta, Behnam Khaleghi and Tajana Simunic Rosing, “MAPIM: Mat Parallelism for High Performance Processing in Non-volatile Memory Architecture,” in *Proceedings of the IEEE International Symposium on Quality Electronic Design (ISQED), 2019*.

Mohsen Imani, Yeseong Kim, Saransh Gupta, Daniel Peroni, and Tajana Simunic Rosing, “In-Memory Acceleration of Deep Neural Network,” in *GOMACTech, 2019*.

Minxuan Zhou, Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Simunic Rosing, “GRAM: Graph Processing in a ReRAM-based Computational Memory,” in *Proceedings of the IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2019*.

Mohsen Imani, Sahand Salamat, Jiani Huang, Saransh Gupta, Tajana Simunic Rosing, “FACH: FPGA-based Acceleration of Hyperdimensional Computing by Reducing Computational Complexity,” in *Proceedings of the IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2019*.

Saransh Gupta, “Processing in Memory using Emerging Memory Technologies,” Master’s Thesis, UC San Diego, 2018.

Saransh Gupta, Mohsen Imani, and Tajana Simunic Rosing, "FELIX: Fast and Energy-Efficient Logic in Memory," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2018.*

Sahand Salamat, Mohsen Imani, Saransh Gupta, and Tajana Simunic Rosing, "RNSnet: In-Memory Neural Network Acceleration Using Residue Number System," in *Proceedings of the IEEE International Conference on Rebooting Computing (ICRC), 2018.*

Mohsen Imani, Ricardo Garcia, Saransh Gupta, Tajana Simunic Rosing, "RMAC: Runtime Configurable Floating Point Multiplier for Approximate Computing," *Proceedings of the IEEE/ ACM International Symposium on Low Power Electronics and Design, 2018.*

Minxuan Zhou, Mohsen Imani, Saransh Gupta, Tajana Simunic Rosing, "GAS: A Heterogeneous Memory Acceleration for Graph Processing," *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design, 2018.*

Mohsen Imani, Saransh Gupta, Sahil Sharma, Tajana Simunic Rosing, "NVQuery: Efficient Query Processing in Non-Volatile Memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018.*

Mohsen Imani, Saransh Gupta, Tajana Simunic Rosing, "GenPIM: Generalized Processing In-Memory to Accelerate Data Intensive Applications," *Proceedings of the IEEE/ACM Design Automation and Test in Europe Conference, 2018.*

Mohsen Imani, Saransh Gupta, Atl Arredondo, Tajana Simunic Rosing, "Efficient Query Processing in Crossbar Memory," *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design, 2017.*

Mohsen Imani, Saransh Gupta, Tajana Simunic Rosing, "Ultra-Efficient Processing In-Memory for Data Intensive Applications," *Proceedings of the IEEE/ACM Design Automation Conference, 2017.*

Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsky, "Logic Design Within Memristive Memories Using Memristor-Aided loGIC (MAGIC)," in *IEEE Transactions on Nanotechnology, 2016.*

ABSTRACT OF THE DISSERTATION

Efficient and Secure Learning across Memory Hierarchy

by

Saransh Gupta

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2021

Professor Tajana Šimunić Rosing, Chair

Recent years have witnessed a rapid growth in the amount of generated data. Learning algorithms, like hyperdimensional (HD) computing, promise to reduce the computation complexity of processing such a huge amount of data. However, traditional computing systems are highly inefficient for such algorithms, mainly due to the limited cache capacity and memory bandwidth. Processing in-memory (PIM) is an emerging paradigm which tries to address these issues by using memories as computing units. In this dissertation, we propose a PIM-based HD computing architecture that accelerates all phases of the HD computing pipeline namely, encoding, training, retraining, and inference. Our architecture is enabled by fast and energy-efficient in-memory logic operations, combined with a hardware-friendly distance metric. However, the improvements from PIM decrease as the size of the dataset increases beyond the memory capacity. Hence, we also design an in-storage computing (ISC) solution. Our ISC design includes on-flash-chip acceleration of HD encoding, which encodes multiple data points in parallel across different flash chips, exploiting the high parallelism provided by the flash hierarchy. This is supported by a controller-level accelerator that performs HD training, retraining, inference, and clustering. Our proposed PIM and ISC solutions provide 434× and 222× speedup as compared to the state-of-the-art HD computing implementations on CPU.

Many applications, most notably in healthcare, finance, and defense, rely on cloud computing for learning tasks and demand privacy which today's solutions cannot fully provide. Fully homomorphic encryption (FHE) elevates the bar of today's solutions by adding confidentiality of data during processing, while introducing noticeable data size expansion - the ciphertext is 5000× bigger than the aggregate of native data types. In this dissertation, we present a design of the first PIM-based accelerator of both client and server using the latest Ring-GSW based homomorphic encryption schemes. Our design supports various security levels and provides on average 2007× higher throughput than CPU while running FHE-enabled neural networks. This improvement comes from a significant reduction in total data-transfers and the high number of processing in-memory cores, which enables higher parallelism and deeper pipelining in our design.

Chapter 1

Introduction

There has been an unprecedented increase in the number of interconnected devices and smart systems that generate big data. Drawing meaningful results from this data requires advanced learning techniques, involving complex computations. Hence, they are either run on a local multi-core system or sent to the cloud and run on large servers [4, 5]. The rise of big data, the ever increasing demand for new complex applications, and the slowdown of Moore's Law have together pushed the current processing systems to their limits. Running data intensive workloads with large datasets on traditional cores results in high energy consumption and slow processing speed. Hence, there is a need to explore new computation models and algorithms.

Hyperdimensional Computing: Brain-inspired hyperdimensional (HD) computing is a computation paradigm which represents data in terms of extremely large vectors, called *hypervectors*. These hypervectors may have 10s of thousands of dimensions and present data in the form of a pattern of signals instead of numbers. By representing data in high-dimension space, HDC reduces the complexity of operations required to process data. HDC builds upon a well-defined set of operations with random HDC vectors, making HDC extremely robust in the presence of failures, and offers a complete computational paradigm that is easily applied to learning problems [6]. Prior work has shown the suitability of HDC for various applications like activity recognition, face detection, language recognition, image classification, etc [7, 8, 9, 10, 11, 12, 13].

Memory-Centric Hardware for HD Computing: Even though HD computing reduces the computation complexity, it has large energy and latency overheads when run on conventional computing systems. This is caused by the large amount of data movement between processing units and memory, owing to limited cache capacity and on-chip bandwidth [14, 15]. Processing in-memory (PIM) tries to address this issue by processing part of data in-place, eliminating the need to transfer all data to the processing unit. PIM has recently become an active area of research. Most of it is driven by the emergence of new non-volatile memory technologies, like resistive RAM (ReRAM) or memristors. They have fast switching speed, low switching energy, and high scalability, making them suitable for dense and fast PIM solutions [16, 17, 18, 19, 20]. A number of recent publications have exploited memristors to enable PIM [21, 22, 23, 24, 25]. We exploit these techniques to design, for the first time, a ReRAM PIM architecture that can accelerate all the phases of HD pipeline namely, encoding, training, retraining, and inference. Apart from the typical data movement reductions that PIM designs achieve, we show how our PIM functions can provide large vector-wide parallelism.

Going beyond Memory – In-Storage Computing: The huge improvements over traditional systems provided by our PIM solution for HD start to diminish as the size of data becomes too huge to fit in the memory. This is due to the data transfer overhead between memory and storage. Recent work has introduced computing capabilities to solid-state disks (SSDs) to process data in storage [26, 27, 28, 1]. This not only reduces the computation load from the processing cores but also processes raw data where it is stored. However, the state-of-the-art in-storage computing (ISC) solutions are not able to efficiently leverage its hierarchical design. Instead, we propose an HD computing system that spans multiple levels of the storage hierarchy and computes at flash chips as well as the top-level controller. Flash chips performs HD computing operations over multiple data samples in parallel which are then used for learning in the top-level controller.

Cloud Computing and Privacy of Data: The existing PIM and ISC solutions assume the data to be present in the raw form. However, there may be instances when that is not the case.

In critical areas, like healthcare, finance, insurance, etc, which deal with extremely sensitive information, user data is usually encrypted. Moreover, majority of the applications in these areas rely on cloud for computing needs due to the involvement of complex algorithms, very large and dynamic computation and learning models, proprietary algorithms, or the need to simultaneously learn from multiple users [29, 30, 31, 32]. Traditional cryptography schemes like AES, SHA-256, etc encrypt user data but decrypt them in the cloud before processing them using user-specific keys. This makes user keys and data prone to attacks.

On the contrary, fully homomorphic encryption (FHE) allows us to apply functions of arbitrary complexity on the encrypted data (ciphertext) without the need to decrypt it. This eliminates the need for private key exchanges and decrypting data at the server, raising the bar on security and privacy. However, computing on encrypted data comes at a huge data and computation cost, resulting in large performance and memory overheads. For example, encrypting an integer in homomorphic domain may explode its size from meagre 4B to more than 20KB. Moreover, homomorphically multiplying two FHE encrypted integers may require 10s of millions of operations. PIM is an excellent match for the FHE since it provides extensive parallelism, bit-level granularity, and an extensive library of compatible operations which dramatically improving both performance and energy efficiency [33, 34, 35, 36]. It addresses the issue of large data movement by processing data in memory where it is stored. We utilize PIM to present the first end-to-end acceleration of latest generation FHE cryptosystem based on [37]. Unlike previous HE proposals, which supported a library of functions, our accelerator allows computing arbitrary functions on encrypted data.

The rest of the introduction gives an overview of the contributions of this dissertation.

1.1 Learning with HD Computing in Memory

Chapter 2 presents a detailed PIM implementation of hyperdimensional (HD) computing, called Tri-HD. We, for the first time, design a ReRAM Digital-PIM architecture that can accelerate all the phases of HD pipeline namely, encoding, training, retraining, and inference. We present a new approximate distance metric which is PIM-compatible, unlike the traditionally used metrics. While this metric enables us to complete the HD pipeline in PIM, it comes at no loss in accuracy due to the robustness of HD computing. We also propose low latency PIM functions and discuss them from the perspective of vector operations. Apart from the typical data movement reductions that PIM designs achieve, we show how our PIM functions can provide large vector-wide parallelism. Moreover, in contrast to GPUs which are limited to 1000s of cores, our PIM designs can offer much greater compute capability by making every memory array in the PIM chip a computing core.

Tri-HD's architecture is enabled by novel single cycle PIM operations. While state-of-the-art ReRAM PIM implementations use NOR as the building block for all their logic operations, we propose a purely in-memory implementation of fast and energy efficient logic. It extends the functionality of in-memory operations by implementing *single cycle* NOR, NOT, NAND, minority (Min), and OR directly in crossbar memory. We used these low latency functions to implement functions like XOR and addition $2\times$ faster than MAGIC [33]. Our design further increased the amount of in-memory parallelism by using in-block switches.

Our evaluation shows that for all applications tested using HD, Tri-HD provides on average $434\times$ ($2170\times$) speedup and consumes $4114\times$ ($26019\times$) less energy as compared to the CPU while running end-to-end HD training (inference).

1.2 Accelerating HD Computing in Storage

To alleviate the data transfer overhead incurred by PIM for extremely large datasets, Chapter 3 proposes Store-n-Learn, an in-storage computing (ISC) based HD computing system that spans multiple levels of the storage hierarchy. We exploit the internal bandwidth and hierarchical structure of SSDs to perform HDC operations over multiple data samples in parallel. Store-n-Learn is a novel ISC architecture for HDC which performs HDC classification and clustering completely in storage. It enables computing at multiple levels of SSD hierarchy, allowing for highly-parallel ISC. Our hierarchical design provides parallelism and hides a significant part of the performance cost of ISC in the storage read/write operations.

We introduce the concept of batching in HDC and utilize it to make our ISC implementation more efficient. During training, we batch together multiple data samples encoded in the HDC domain in storage. This allows us to partially process data without accessing all encoded hypervectors. Batching enables us to have a minimal aggregation hardware requirement. Batching also reduces the amount of data sent out of storage. Store-n-Learn utilizes die-level accelerators to convert raw data into hypervectors locally in all the flash planes in parallel. Unlike previous work [2], our accelerator is simpler and hides its computation latency by the long read times of raw data from flash arrays. Our die-level accelerators can perform both batched and non-batched encoding efficiently in flash planes. For batched encoding, the accelerator processes multiple inputs in a page in parallel. It generates multiple dimensions corresponding to an input in parallel during non-batched encoding. This flexibility is enabled by our innovative adder tree design. We present a top-level SSD accelerator, which aggregates the data from different flash dies. This accelerator is implemented on an FPGA-based device controller. We implement new and efficient FPGA designs for HDC training, retraining, inference, and clustering. While HDC training provides sufficiently accurate initial models, retraining significantly improves the accuracy of the models by iterating over training data and updating the models multiple times. Store-n-Learn inference allows the users to directly obtain the classification result from the storage drive without sending the entire model to the host. Store-n-Learn clustering leverages the FPGA already present in storage and iteratively processes the datasets multiple times to generate high quality cluster centers. We also present host-side and drive-side primitives to enable the FPGA to work seamlessly with the die-level accelerators.

We evaluate Store-n-Learn over ten popular classification and clustering datasets. Our experimental results show that Store-n-Learn is on average 222× (543×) faster than CPU and 10.6× (7.3×) faster than the state-of-the-art ISC solution, INSIDER for HDC classification (clustering).

1.3 Secure and Privacy-Preserving Computing with FHE in Memory

This dissertation goes beyond learning at the edge to make end-to-end data privacy, one of the most desired components of cloud computing, feasible. Chapter 4 presents the first end-to-end acceleration of latest generation FHE cryptosystem based on [37]. Unlike previous HE proposals, which supported a library of functions, the latest RGSW-based cryptosystem allows computing arbitrary functions on encrypted data. Our proposed MemFHE has two main components, the client and the server PIM accelerators. The client PIM accelerator runs ultra-efficient in-memory operations to not only encode and decode data but also enables ring learning with errors (RLWE) to encrypt and decrypt data. The encrypted data (ciphertext), along with an encrypted version of secret key, are sent to the server PIM accelerator for processing. Server PIM receives the ciphertext from multiple clients and performs operations on ciphertext to generate output. To enable this, server PIM uses PIM-enabled bootstrapping which keeps the accumulated noise low so that the output ciphertext can be decrypted by the intended client. This ciphertext is sent back to the client. In MemFHE, only the client has the means to decrypt the output ciphertext and access the unencrypted data.

While individual PIM operations are slower than in CPU, MemFHE employs ciphertext-level and operation level parallelism combined with operation-level pipelining to achieve orders of magnitude of performance improvement over the traditional systems. Our server PIM design includes fast bootstrapping, key switching, and modulus switching in memory. It distributes the key memory units to reduce the instances of data contention. It sequentially processes different inputs in different pipeline stages for the best processing throughput. We accelerate the bottleneck process of bootstrapping by using a highly pipelined architecture. Our bootstrapping introduces parallel accumulation units, which supports two different types of bootstrapping techniques.

We propose a novel implementation for the core bootstrapping operation, NTT. Unlike existing works, our NTT doesn't require any special interconnect structure. Moreover, it is flexible and can process many NTT stages without needing extra hardware. Our client PIM design includes encryption and decryption. MemFHE enables encryption efficiently in memory by exploiting bit-level access and accelerates dot product with a new in-memory implementation.

We evaluate MemFHE for various security-levels and compare it with state-of-the-art CPU implementations for Ring-GSW based FHE. MemFHE is up to $20k\times$ faster than CPU for FHE arithmetic operations and provides on average $2007\times$ higher throughput than [3] while implementing neural networks with FHE.

Chapter 2

Learning with Hyperdimensional Computing in Memory

Hyperdimensional computing reduces the complexity of computation. However, it has large energy and latency overhead when run on conventional computing systems. This is caused by the large amount of data movement between processing units and memory, owing to limited cache capacity and on-chip bandwidth [14, 15]. Processing in-memory (PIM) tries to address this issue by processing part of the data in-place, significantly reducing the need to transfer all the data to the processing unit.

In this chapter, we exploit single cycle operations to present a detailed PIM implementation of hyperdimensional (HD) computing, called Tri-HD. For the first time, we design a ReRAM PIM architecture that can accelerate all the phases of HD pipeline namely, encoding, training, retraining, and inference. We present a new approximate distance metric, which is PIM-compatible unlike the traditionally used metrics. While this metric enables us to complete the HD pipeline in PIM, it comes at no loss in accuracy. We also propose low latency PIM functions and discuss them from the perspective of vector operations. Apart from the typical data movement reductions that PIM designs achieve, we show how our PIM functions can provide large vector-wide parallelism. Moreover, in contrast to GPUs which are limited to some 1000s of cores, our PIM designs can offer much greater compute capability by making every memory array in the PIM chip a computing core. Our evaluation shows that for all applications tested using HD, Tri-HD provides on average 434$\times$ (2170$\times$) speedup and consumes 4114$\times$ (26019$\times$) less energy as compared to the CPU implementation while running end-to-end HD training (inference).

2.1 Background on Processing in Memory with ReRAM

A number of recent publications have exploited ReRAM (memristors) to enable PIM [21, 22, 23, 24, 25]. Some compute logic at the periphery of the memory by modifying the memory sense amplifiers [22, 24, 38]. They read the stored data from the memory, use transistor based circuits to process data, and store the results back to the memory. In these designs, the amount of data that can be processed in parallel is limited by the amount of sense circuitry present at the periphery of the memory. Other work exploits the bipolar switching behaviour of memristors to implement logic in-memory [33, 21, 23, 39, 40]. Some of them implement logic purely in memory such as stateful implication logic [39, 41], and Memristor Aided loGIC (MAGIC) [33].These designs depend on application of voltage at various memory cells with no change in the sense amplifiers. They are purely in-memory operations which do not need to read out data but are restricted by the limited functionality they can implement. For example, MAGIC [33] only supports NOR directly in crossbar memory. All other functions are implemented by repeated multiple NOR cycles.

Logic execution with MAGIC is fully compatible with the usual crossbar design, requires a lower number of voltages, and supports NOR which can be used to implement any Boolean logic. Also, it is non-destructive, unlike implication logic based designs like IMPLY [39] which destroy one of the inputs. These properties of MAGIC make it a preferred logic family for resistive crossbar memories. MAGIC uses voltage threshold based memristors which switch whenever the voltage difference between the two terminals of the memory device exceeds a threshold. However, they don't fully utilize the threshold based switching of memristors and only implement NOR in crossbar memory. All other functions are derived using NOR, which results in unnecessary latency overheads. In contrast, in this chapter, we propose a purely in-memory implementation of fast and energy efficient logic. It extends the functionality of in-memory operations by implementing *single cycle* NOR, NOT, NAND, minority (Min), and OR directly in crossbar memory. We use these low latency functions to implement functions like XOR and addition $2\times$ faster than MAGIC [33].

2.2 Hyperdimensional Computing

Brain-inspired HyperDimensional (HD) computing is a computing paradigm which works based on understanding the fact that brains compute with *patterns of neural activity* [6, 42, 43, 8], where such neural activity patterns can only be modeled with points of high-dimensional space (e.g., D=10,000). Classification is one of the most important supervised learning algorithms. Figure 2.1a shows the overview of HD computing architecture for a classification problem consisting of an encoder module and an associative memory. The goal of the encoder is to map an input data to a single hypervector with D dimensions and then combine these hypervectors for all of the images in a class to generate a unique hypervector representing each class. Each class hypervector is a long vector with D dimension, where each dimension can have binary (0, 1) elements. Associative memory stores the trained hypervectors for all classes. In test mode, HD classifies an unknown input by encoding the input image to a hypervector using the same encoder used for training. The query hypervector has binary elements and the same D dimension as the class hypervectors. Next, associative memory checks the similarity of the query hypervector to all classes and classifies it to a class which has the closest similarity.

2.2.1 Encoding Module

Figure 2.1b shows the encoding module in HD computing. Let's assume each data point in original space can be represented using a features vector $\{v_1, \ldots, v_n\}$. The goal of encoding module is to map this feature vector to high-dimensional space while keeping all its information in a high-dimensional vector. Each feature vector stores two types of information: the value of

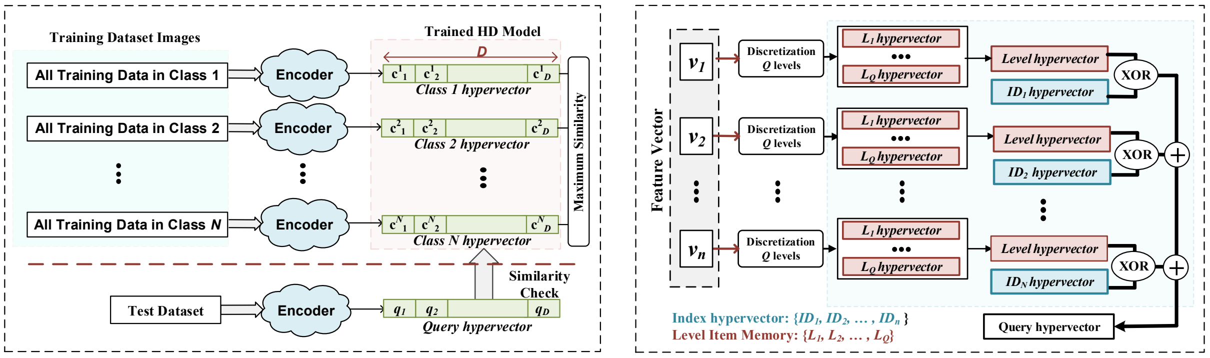


Figure 2.1. (a) The overview of HD computing architecture for classification task. (b) The encoding module of HD computing mapping a feature vector with n elements to high dimensional space using pre-generated identity and level hypervectors.

signal and the index of each feature.

Feature values: In order to consider the impact of feature values, our design first identifies the minimum and maximum value that signal can take in all dimensions $\{v_{min} \& v_{max}\}$. Then, it quantizes the feature values into Q levels were v_{min} and v_{max} are the first and last levels respectively. HD assigns a single hypervector with D dimension to each of the quantized levels $\mathbf{L} = \{L_1, L_2, \dots, L_Q\}$ where $L_i \in \{0,1\}^D$ and L_1 and $L-Q$ correspond to the v_{min} and v_{max} respectively. The generation of the level hypervector is similar to work [42], such that the level hypervectors have similar values if the corresponding original data are closer, while L_1 and L_Q will be nearly orthogonal.

Feature index: To specify the impact of each feature index on encoded hypervector, HD generates a set of random identification hypervector $\{ID_1, \dots, ID_n\}$, where $ID_i \in \{0,1\}^D$ represents a hypervector corresponding to i^{th} feature index. Due to random generation, the ID hypervectors are semi-orthogonal, meaning that:

$$\delta(ID_i,\, ID_j) \simeq D/2 \qquad \text{for } i \neq j \tag{2.1}$$

Depending on feature values, each feature maps to one of the Q generated hypervectors. Hypervectors are combined together using element-wise XOR of the level and ID hypervector, and then summing the resulting hypervectors over all features:

$$H = \overline{L}_1 \oplus ID_1 \; + \; \overline{L}_2 \oplus ID_2 \; + \cdots + \; \overline{L}_n \oplus ID_n \quad f \in [1, m] \tag{2.2}$$

where $\overline{L}_i$ is the (binary) hypervector corresponding to the i-th feature of vector v.

2.2.2 HD Training and Retraining

The simplicity of HD training makes it distinguished from conventional learning algorithms. Consider hypervector H_i as the encoded hypervector of input i with the procedure explained above. Each input i belongs to a class j, so we further annotate H_i^j to show the class j of input i, as well. HD training simply adds all hypervectors of the same class to generate the final model hypervector. Therefore, the class hypervector of label j, denoted by C^j, is:

$$C^j = H_0^j + H_1^j + \cdots = \sum_i H_i{}^j \tag{2.3}$$

Meaning that we simply accumulate the encoded hypervectors for which their original input belongs to class j.

Another advantage of HD over DNNs is HD supports efficient one-pass training, i.e., visiting each input just once and adding the H_is to create the model yields acceptable accuracy, while DNN training requires hundreds of iterations over the whole data set to converge to the final accuracy. HD accuracy can also be improved by *retraining* the model. During retraining, the encoded hypervector of each input is created again, and its similarity with the existing class (model) hypervectors is checked. If a *misprediction* is observed, say that encoded H^j belonging to class C^j is predicted as class C^k, the model is updated as follows, which means the information of H^j causing (mis)-similarity to C^k is discarded. The parameter α is the learning rate of the model.

$$\begin{aligned} C^j &= C^j + H^j \alpha \\ C^k &= C^k - H^j \alpha \end{aligned} \tag{2.4}$$

2.2.3 HD Inference

The inference step as well as the retraining step need to find out the most similar class hypervector to the encoded one. Most commonly, this is performed by cosine similarity while other metrics (e.g. Hamming distance) could be appropriate depending on the problem.

$$cos(\vec{H},\vec{C}^j) = \frac{\vec{H}\cdot\vec{C}^j}{\|\vec{H}\|\cdot\|\vec{C}^j\|} \tag{2.5}$$

Equation (2.5) shows the similarity checking of encoded hypervector H with class hypervector C^j. Since classes are constant, $\|\vec{C}^j\|$ can be pre-calculated. $\|\vec{H}\|$ can be factored out as it is common for all candidate classes to be compared with H. Hence, cosine similarity reduces to a simple dot-product between H and C^js. These vectors are *not* in binary, they are the results of accumulating several other binary vectors.

2.2.4 Existing HD Computing Work in PIM

Recent work has proposed ways to use PIM to accelerate HD computing. The associative memory data structure of HD computing is generally regarded as the most suitable candidate for acceleration by PIM. The work in [44] was the first to recognize this, and proposed an HD computing accelerator based on resistive CAMs. The design achieved 746× reduction in energy-delay product (EDP) as compared to the CMOS-based ASIC proposed in [45]. The authors in [46] tried to alleviate the thermal challenges related to PIM implementation of associative memory. They proposed memory block selection and activation schemes to reduce the overall chip temperature, resulting 57.2% memory lifetime improvement and 17.6% performance gain.

The design in [47] extended the idea of the work in [44] and supplemented it with a digital HD mapper and encoder to accelerate complete HD algorithm. However, the encoding schemes used by them do not provide state-of-the-art results. The work in [48, 49] presented an HD-chip which implemented both encoding and search operations using a combination of carbon nanotube field-effect transistors (CNFETs) and ReRAM cells to achieve low EDP. However, they implement only inference and supported only few specialized applications. The work in [50] proposed crossbar memory based encoding and associative search. The encoding module used analog computing primitives to perform bitwise logical AND of hypervector dimensions. The demonstrated chip provided 6.6× energy and 3.8× area improvement. However, approximately 92% of the area and 89% of energy were consumed by memory peripherals.

Another set of designs in [51, 52] take a different approach where instead of implementing associative memory using PIM, they used large ReRAM PIM-based XOR operations to compare different hypervectors. In both the implementations, HD encoder used ReRAM PIM bitwise operations to generate hypervectors. A similar work in [53] breaks down all HD operations into dot product and bitwise operations and implements them using analog computing techniques. Finally, recent research in [54] presented a PIM implementation of both HD training and inference. To maintain high training accuracy while achieving the efficient bitwise computation, the work proposed the use of stochastic training, which generated multiple binary hypervectors for each class. It increased the baseline accuracy, which was still 4-9% less than what HD could achieve.

2.2.5 Challenges with the Existing Work

Most of prior work tried to accelerate HD by speeding up the computation in associative memory [44]. In this work, we use HD computing for practical classification problems such as speech recognition [55], face recognition [56], activity recognition [57], and physical monitoring [58]. We observe that in most tested applications, the data point in original data is a long feature vector. Encoding such feature vector to high dimensional space is extremely costly. Table 2.1 compares the energy consumption and execution time of HD computing for several applications including language recognition, speech recognition, face recognition, and activity recognition. The experiments have been performed on the Intel i7 CPU with 16GB memory. Our evaluation shows that for practical problems, the encoding module is a dominant part of energy consumption and execution time. For example, for four tested applications the encoding module

Table 2.1. Energy consumption of HD encoding module and associative memory for different applications

	Encoding Module	*Associative Memory*
Speech Recognition	8.18 mJ	8.78 mJ
Face Recognition	7.85 mJ	1.43 mJ
Activity Recognition	7.01 mJ	3.87 mJ
Physical Monitoring	0.23 mJ	2.25 mJ

takes around 60% of total energy.

2.3 Tri-HD Acceleration in Memory

In this section, we present PIM implementation of the various stages in HD computing pipeline. Tri-HD also demonstrates the benefits of PIM for large vectors since the entire HD computing pipeline operates these large vectors. Hence, implementation of HD computing on PIM allows us to evaluate its vector-wide parallelism benefits, while providing a look into the potential applicability of PIM for big data algorithms.

2.3.1 Tri-HD HD Encoding

Here, we discuss how HD is mapped to memory and its acceleration using Tri-HD. The HD efficiency depends on the amount of parallelism which we can apply to encoding module. The most area efficient method is to store all ID and level hypervectors in a single memory partition and perform XOR operation between each ID and corresponding level in series. This method serially processes the features and its performance is directly related to the number of features. Our design parallelizes the encoding module by partitioning the memory block.

Let us assume that a feature vector has n elements and Q corresponding Ls (Levels). In HD, this results in n IDs. In total we have $n + Q$ vectors with D=10,000 dimensions. All these vectors are stored in a memory block. Each vector is mapped to a row of the memory, where each row has a capacity of 10,000 bits as shown in Figure 2.2. As such huge memory blocks are hard to achieve, a row is usually split among multiple smaller-width memory blocks.

Unless otherwise mentioned, we consider a block size of 1024×1024 in our implementation. In HD encoding, as discussed in the previous section, first each ID is XORed with one of the Ls depending upon the values of the feature. The results of the XOR operations are then added together dimensionwise. We map all these operations in Tri-HD-enabled memory.

We perform n Tri-HD XORs (one for each ID) and generate n outputs. For the first n iterations, we select one ID and XOR it with one of the Ls in every iteration. For a pair of ID and L, Tri-HD XOR can be computed in parallel for all dimensions since all dimensions of a vector are stored in the same row. Each XOR operation requires one additional memory cell to store the output of the result. This requires 10,000 additional cells, equivalent to a row of the memory, to store the output. We then count the number of 1s in all XOR results for each dimension. We execute this by Tri-HD addition. In order to perform addition for all the dimensions in parallel, we store the output of addition vertically in a column, instead of a row, as shown in Figure 2.2. We add n elements serially, three bits at a time. If $X_1,\ X_2, \dots\ X_n$ are the vectors to be added together, we first add X_1, X_2, and X_3 to generate S_1 and C_1. We then add X_4, X_5, and $\{C_1,\ S_1\}$ to generate S_2 and C_2, and so on till we have added all XOR results. The addition of n 1-bit elements results in an output with $p = \lceil log_2 n \rceil$ bits, requiring p rows to store the output of addition. In addition, Tri-HD also requires $p+1$ rows to store the intermediate results of addition like C_1, S_1, S_2, C_2, etc.

As described above, we add a maximum of three XOR results at a time. Hence, instead of calculating all XOR results first and then adding them, we calculate them two at a time, except the first step when we calculate three, and add them. This reduces the memory requirement for XOR results from n rows to just 3. Apart from the 3 rows for XOR results and p rows required for the addition results, we require $p+1$ rows to store the intermediate addition results and 2 rows for the intermediate Tri-HD stages, as shown in Figure 2.2.

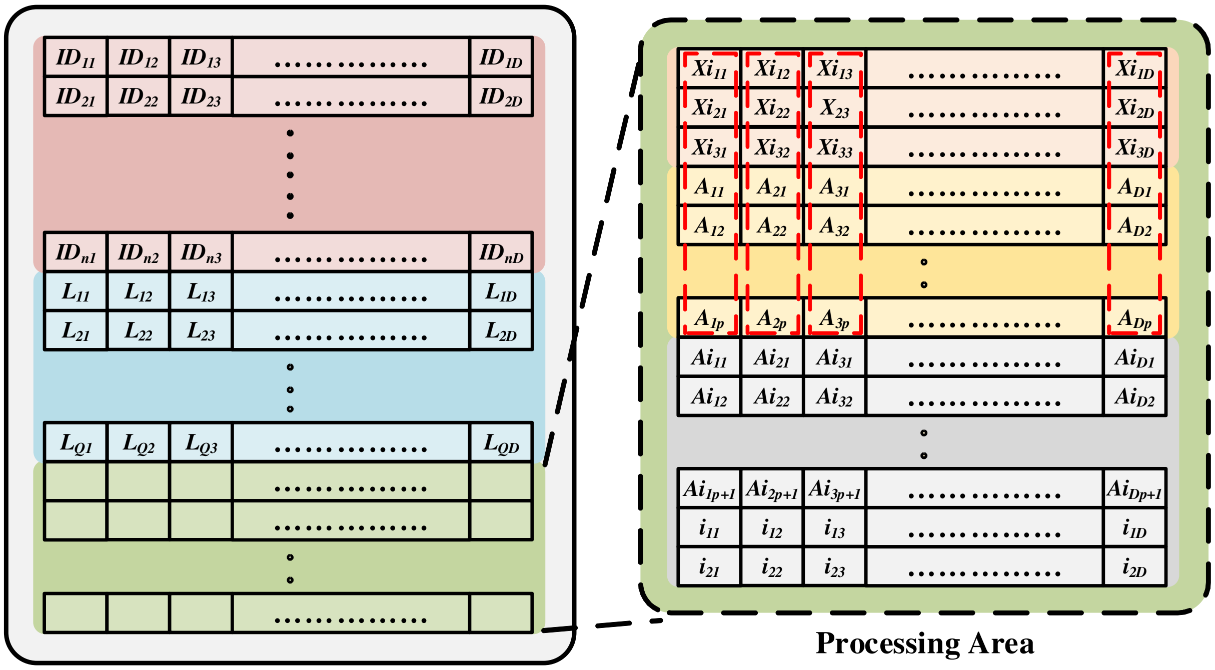


Figure 2.2. Organization of data in Tri-HD-enabled HD accelerator.

2.3.2 Parallelized Tri-HD HD Encoding

We accelerate the baseline Tri-HD HD by using the parallelism technique proposed in Section 2.5. We observe that the *n* XOR operations in encoding are independent of each other. Also, the addition of XOR results is not affected by the order of operands. Hence, all XORs and majority of the addition operations can be paralleized. We divide our block into *k* smaller partitions using transistor switches described in Section 2.5. Each partition stores *n/k* IDs and all *Q* Ls. The transistors are first switched off. This makes each smaller partition to work similar to the memory block in the baseline implementation. Since, the transistors physically segment the bitlines, all the partitions operate independently in parallel. This reduces the number of iterations required for executing XOR from n to n/k. Moreover, the size of addition in each partition reduces from $\lceil log_2 n \rceil$ to $\lceil log_2(n/k) \rceil$.

The results of addition in different partitions are then aggregated. To implement this, the transistors are switched on, allowing the block to behave as a single partition. We use Tri-HD addition to perform aggregation in parallel for all the dimensions. This aggregation leads to

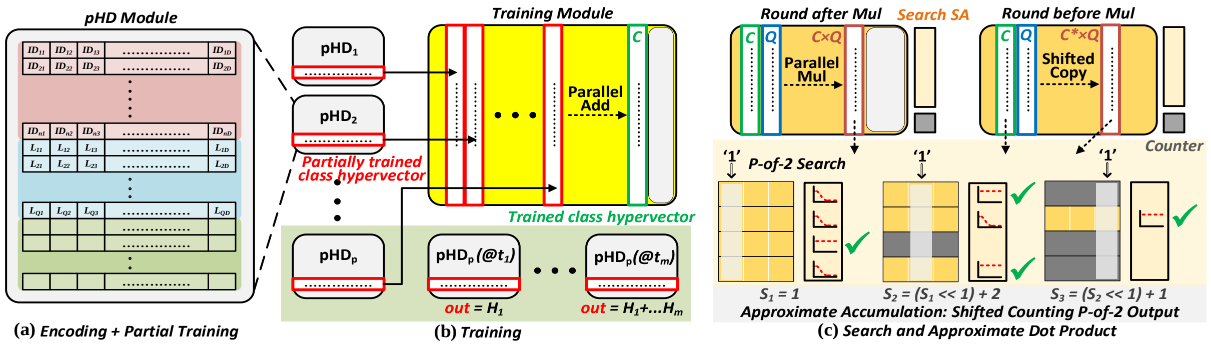


Figure 2.3. Tri-HD Overview. (a) Encoding and partial training combined in a single module, (b) training on partially trained hypervectors, and (c) approximate dot product and power-of-2 (P-of-2) search in Tri-HD. C* represents C with P-of-2 applied to it.

additional latency overhead. Moreover, the effective memory requirement increases, since each partition needs to replicate the level hypervectors and assign rows to perform computation and store the XOR and addition result.

2.3.3 Tri-HD HD Training

As discussed in Section 2.2 HD training involves class-wise addition of the hypervectors generated by encoding. This addition is similar to that involved in encoding but instead of adding 1-bit-element vectors (X_i in encoding) to integer vectors, here we need to add multiple integer vectors. Moreover, the implementation in the encoding module is sequential. The simple training algorithm of HD allows us to split training into multiple modules, where each module *independently* performs (partial) training over a subset of the data and then the partially trained class hypervectors are added together.

To implement this in memory, we fuse HD encoding and partial training to form a partial HD (*p-HD*) module, as shown in Figure 2.3a. For each input data, we perform encoding as explained in Section 2.3.2. However, instead of creating a new encoded hypervector for each input, we keep on accumulating the generated encoding outputs to a single hypervector. Hence, at any point during the computation, we don't have individual encoded data points but only a partially trained class hypervector. After encoding all the data, *p-HD* modules send their class hypervectors to the training memory block for final accumulation, as shown in Figure 2.3b. While sending data from *p-HD* to training block, each dimension which was column-wise stored in *p-HD* is now stored row-wise. This is achieved by reading the hypervectors, one-row at a time. As discussed in Section 2.3.2 each row of the final hypervector stores one bit of each dimension. The j_{th} element of the i_{th} row read from *p-HD* is stored as the i_{th} bit of the j_th dimension of the hypervector when stored in the training block. This way, each class hypervector is now stored column-wise, with each dimension stored in a single row, as shown in Figure 2.3b, making the computation of similarity metric in the future easier.

2.3.4 Tri-HD HD Inference

Tri-HD HD inference uses the Tri-HD encoding and similarity blocks. The encoding block converts input data into hypervectors. The encoded vectors are then sent to the similarity block to find its closest class hypervector. In the next section, we present a novel approach to implement the similarity block. The output of the similarity check is the closest class and the corresponding label is the output of HD inference.

2.3.5 Tri-HD HD Re-training

Similarly, re-training involves HD inference, followed by two-part training. First, the input data is sent to Tri-HD inference block. The output of the inference is compared to the actual true label of the input. If the inference output is same as the label, we bypass the remaining stages of retraining. Otherwise, the encoded hypervector is sent to the Tri-HD training *p-HD* corresponding to the true class. In addition, the hypervector is also sent to training *p-HD* corresponding to the inferred class. However, for the inferred class, *p-HD* perform hypervector subtraction instead of addition. This process is carried on for a maximum number of iterations, provided by the application parameters, and the model corresponding to the best accuracy is chosen as the final trained model.

2.4 Approximation of Similarity Metric

2.4.1 Dot Product Similarity Metric

A key operation in the HD pipeline is the dot product, which is used as the similarity metric during retraining and inference. The dot product of two vectors can be broken down into their element-wise multiplication and the accumulation of the generated product vector. To implement element-wise multiplication, the two vectors are stored column-wise so that a row of the memory contains one element from each vector, corresponding to the same dimension. All the elements of a vector share the same columns in memory. Then, we implement row-parallel in-memory multiplication for the entire vector using the switching techniques presented in [59, 40]. This gives us the product vector. The accumulation of this vector involves adding D elements together in memory, which is a slow and serial operation due to the large dimensionality of the hypervectors. Instead, we implement an approximate version of the accumulation. To achieve this, we quantize the elements of the hypervector to the maximum power of 2 number that is less than the each element's value.

The best time for approximation is just before the final dot product accumulation. This would, in theory, result in the least quality loss. However, this would still require fixed-point multiplication over thousands of dimensions, which is both slow and energy-consuming. Hence, we further evaluate the quality loss of HD if we instead perform approximation before multiplication. Figure 2.11 shows the accuracy of HD classification over five datasets with approximation before and after multiplication and without approximation.

Because HD computing is robust to computational errors, both rounding after and before multiplication result in similar accuracy to the reference with minimal rounding errors. This property of HD computing comes from the high dimensionality of the data. Distributing information equally across all of the dimensions leads to a representation with data redundancy. Therefore, even in the event of various dimensions having errors, the redundancy in the data results in a close approximation of the final similarity. This robustness to noise naturally scales with the dimensionality of the model. We can also see this in Figure 2.11. At a lower dimensionality, the accuracy difference between rounding and the reference grows. Therefore, Tri-HD uses a dimensionality of $D = 10,000$ to achieve the highest robustness to the rounding, resulting in a model that maintains accuracy to the reference. Due to the highly parallel nature of digital processing in memory, there is no real cost to increasing the dimensionality of the model as long as the hypervertors still fit in memory.

2.4.2 Implementation in Tri-HD

Both before and after approximations described above involve quantizing the elements of a vector to the maximum power of 2 number that is *less* than the each element's value. In hardware, this is equivalent to finding the leading (trailing) one for positive (negative) numbers. This is implemented using the associative operations presented in [60, 61]. We utilize the exact search operations to implement this power-of-2 (P-of-2) search, as shown in Figure 2.3c. Search is done in parallel over all the elements of the vector. Each column of the vector stores one bit from all the elements of the vector. We perform column-wise search on the vector. For each column, whenever a '1' is detected, the corresponding rows are deactivated from further search operations. We implement a counter at the periphery which counts the number of rows that are selected for each bit-column. At the end, the values in the counter are multiplied by the corresponding power of 2 and added. The result is an approximate accumulation of the vector.

For approximation after the product, Tri-HD simply quantizes the product vector. In the case of approximation before the product, we independently quantize just the class hypervectors. Then, we use the value of the quantized class hypervector (C*) to shift the input hypervector. Since all the elements of C* are power of 2, this shifting is equivalent to multiplication of input with C*. The output of this shift is again quantized with P-of-2 to perform the final accumulation. This puts forth a latency, energy consumption, and accuracy trade-off. The before (after) approximation performs quantization of two (one) hypervectors, while using zero (one) element-wise multiplication step. Hence, before approximation avoids a vector-wide multiplication by introducing an extra but faster (and less efficient) quantization step. We evaluate the corresponding performance and energy consumption in Section 2.7.5 and show that approximating before multiplication is 3.4× faster and 2.3× more energy-efficient as compared to approximating after multiplication while providing on average 0.16% better accuracy.

2.5 Parallelism in Tri-HD

In order to increase in-memory parallelism, we split the array into smaller partitions such that the achievable parallelism directly depends upon the number of partitions of the memory block. These partitions are created by the transistor switches which divide the bitlines into smaller segments. This enables Tri-HD to independently implement multiple operations simultaneously. Consider a memory array with a 64-bit wordline and capacity of 1024 words. Now, a bitwise OR operation needs to be carried out between 10 pairs of words and outputs be stored in the memory. All these steps are independent of each other and can happen in parallel if the memory supports it. Tri-HD can execute it in a single cycle if the memory has 10 or more partitions. In contrast, the conventional design would execute it in 10 steps with each step implementing 64 parallel single cycle OR operations between a pair of words [23].

Figure 2.4 shows how Tri-HD behaves in a memory with no partition when operations are parallelized across rows and columns simultaneously. The currents from different operations interfere with each other, as shown by {Op1 Op3} and {Op2 Op4} in Figure 2.4. It effectively results in a single operation with more inputs and multiple copies of output. Transistors physically split the bitlines while keeping them logically the same. Figure 2.5 shows how Tri-HD behaves when a memory is divided into two partitions using transistors. The transistors, when switched off, prevent the currents belonging to different operations from merging. This enables Tri-HD to parallelize operations in rows and columns simultaneously.

Ideally, we would like to have as many partitions as possible. However, increasing the number of partitions comes with additional overhead. First, more partitions lead to reduced mem-

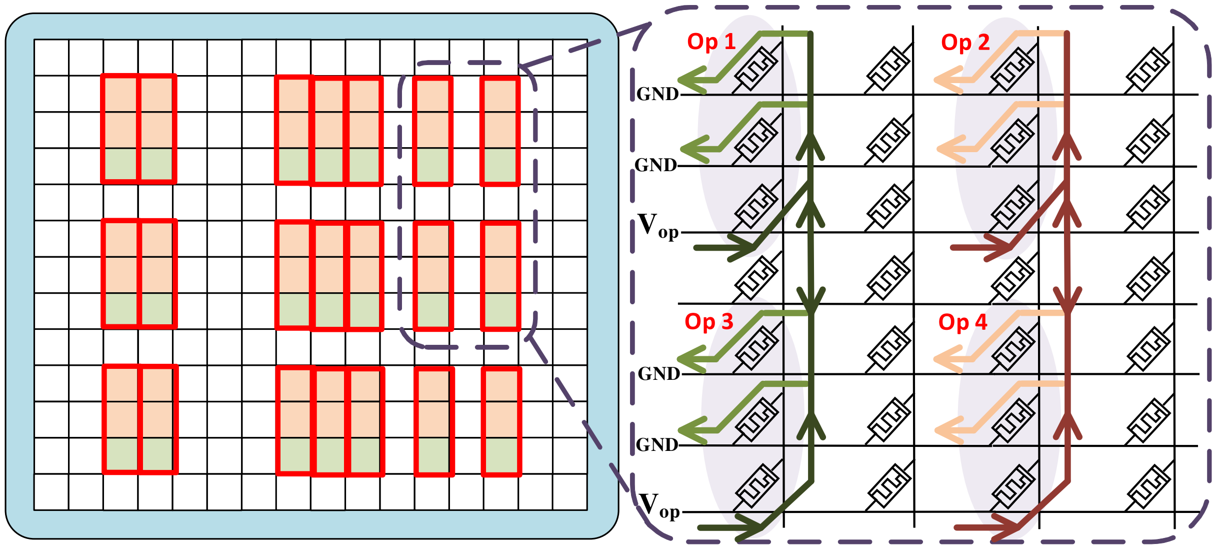


Figure 2.4. Limitation of the memory in implementing operations in a row and column simultaneously. The crossbar structure does not distinguish the currents from different operations.

ory utilization. Tri-HD requires some additional devices or processing elements for executing logic. These elements store the intermediate states involved in achieving the final output. Since each partition needs its own processing elements, increasing the number of partitions linearly increases the number of devices required. These processing elements cannot be used for storing logic because they are used by Tri-HD to implement operations. When the memory size is fixed, increasing the number of processing elements directly reduces the amount of memory usable for storage. Second, more partitions require a higher number of transistors to segment the bitlines, leading an increased area overhead. Figure 2.6 shows the change in memory utilization and area overhead for a 1024$\times$64 memory block as the number of partitions increases.

2.6 Basic Operations in Tri-HD

Tri-HD uses purely in-memory implementations of Boolean functions. It executes NOR, NOT, Min, NAND, and OR in a single cycle in crossbar memory. Tri-HD uses a variable voltage based execution scheme, where the applied voltage defines the operation to be performed. In addition, instead of relying only on resetting behavior of memristors, we exploit it's two-way

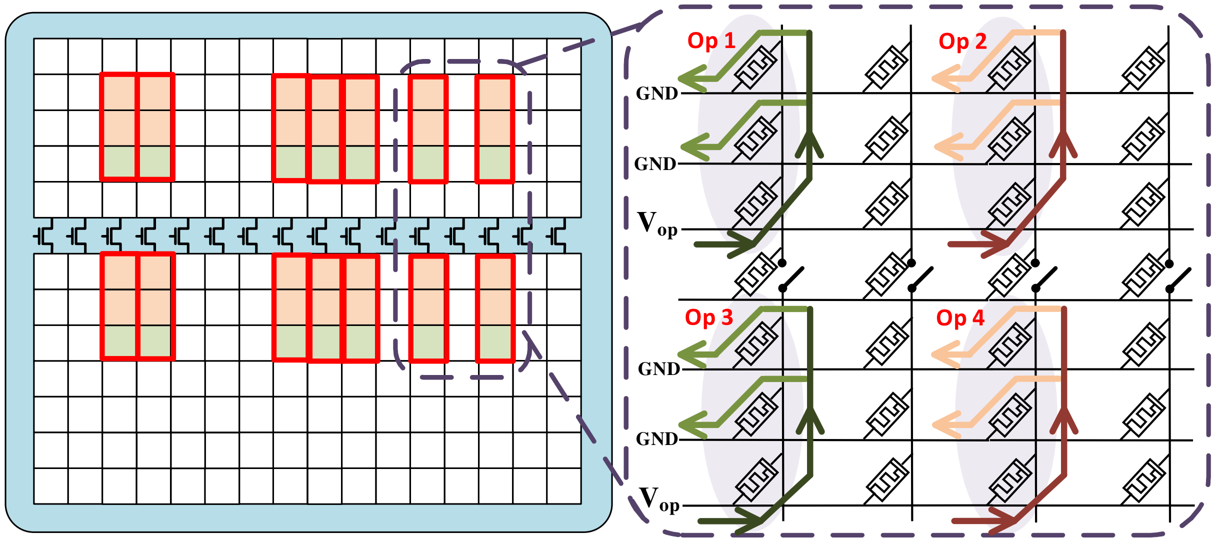


Figure 2.5. Tri-HD with transistors and the resultant memory crossbar. The currents for the four operations do not interfere with each other.

switching to extend the capabilities of PIM. When the voltage $V_{pn} > |v_{on}|$, a memristor switches from a high resistive state (R_{OFF}) to a low resistive state (R_{ON}). On the other hand, when $V_{np} > v_{off}$, it switches from R_{ON} to R_{OFF}. Here, $|v_{on}|$ and v_{off} are the device dependent voltage thresholds and V_{pn} is the voltage difference between terminals p and n.

Table 2.2 compares the execution of different boolean logic functions in Tri-HD with previously proposed PIM techniques. The latencies in the table and the discussion exclude the first initialization cycle which is common to all designs. The numbers in brackets represent the properties of the area conservative designs [23]. It shows that Tri-HD performs either the same as or significantly better than the fastest state-of-the-art technique [33, 23]. For example, for addition, Tri-HD is 2$\times$ faster, has 2$\times$ better energy efficiency, and 3$\times$ lower memory size. In the following subsections, we outline the implementation of basic boolean operations in Tri-HD and how they can be used to perform vector-wide operations that form the basis of Tri-HD architecture.

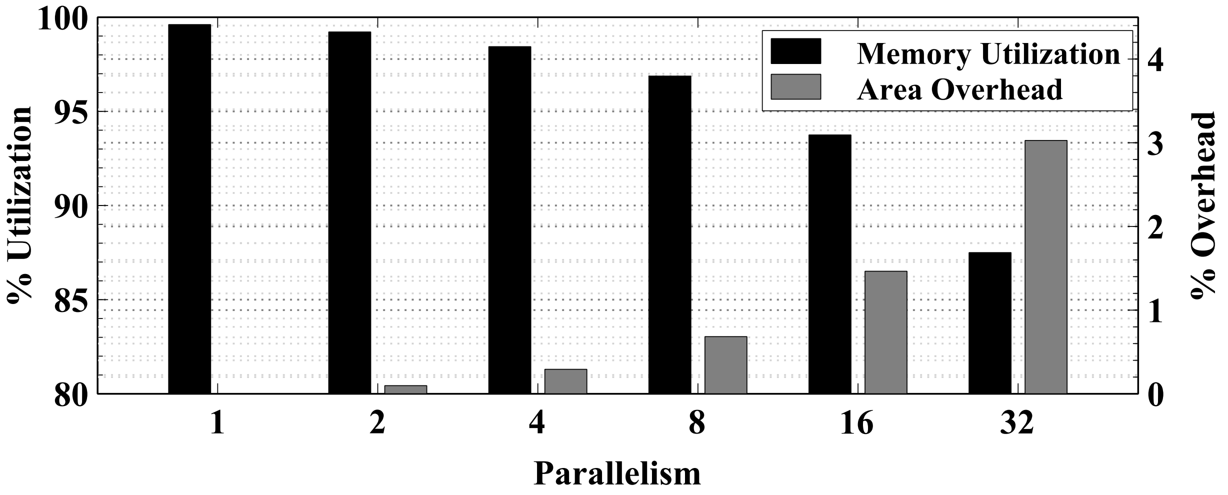


Figure 2.6. Change in memory utilization and area overhead due to transistors with increase in parallelism in Tri-HD.

2.6.1 Single-Cycle Operations

NOR: Figure 2.7 shows how NOR is implemented in memristor-based crossbar array [33]. The output memristor is initialized to R_{ON} in the beginning. To execute NOR in a row, an execution voltage, V_0, is applied at the p terminals of the inputs and the p terminal of the output memristor is grounded, as shown in Figure 2.7a. When NOR is executed in a column, the n terminals of the inputs are grounded while V_0 is applied to the n terminal of the output, as shown in Figure 2.7b. The motive behind both the executions is to switch the output memristor from R_{ON} to R_{OFF} whenever the NOR output is '0'. Tri-HD is as fast as MAGIC [33] for NOR.

Assume two vectors A and B with 100 1-bit elements each are stored in the memory such that ith element of a vector is present in the ith row of the memory. Moreover, all the elements of vector A (B) occupy the ath (bth) column. We call this way of storing vectors as column-wise storage. To perform a NOR over the ith elements of the two vectors, we implement the 2-input row implementation of NOR. As discussed before, we apply voltage V_0 to bitlines a and b, while ground the output bitline, say c. However, we notice that this voltage application remain the same irrespective of the value of i. Hence, the NOR operation discussed above provides vector-wide parallelism, where an operation over all the elements of a vector takes the same time as the

Table 2.2. Comparison of Tri-HD with state-of-the-art PIM technique designed for highest performance.

Property	Design	NOR3	NAND3	Min3	OR3	Maj3	AND3	XOR2	1-bit ADD
Latency (Cycles)	**MAGIC**	1	5 (6)	5 (6)	2	4	4	5 (7)	12 (14)
	Tri-HD	1	1	1	1	2	2	2	6
	Improv.	1×	5 (6)×	5 (6)×	2×	2×	2×	2.5 (3.5)×	2 (2.33)×
Memory (# of Cells)	**MAGIC**	1	5 (4)	5 (4)	2	4	4	5 (3)	12 (6)
	Tri-HD	1	1	1	1	2	2	1	4
	Improv.	1×	5 (4)×	5 (4)×	2×	2×	2×	5 (3)×	3 (1.5)×
Energy (fJ)	**MAGIC**	24.11	120.17	120.38	48.12	96.17	96.15	120.29	288.82
	Tri-HD	24.11	49.24	41.64	9.53	65.65	73.26	34.97	135.60
	Improv.	1×	2.44×	2.89×	5.05×	1.47×	1.31×	3.44×	2.13×

operation over single element of the vector. This discussion can be similarly extended to the case when the vectors are stored row-wise and we use NOR in a column implementation.

NAND and Min: Tri-HD does not directly depend upon the inputs but the voltage developed across the n and p terminals of the output device. This enables it to implement minority and NAND in memory. Consider the case of a 3-input NOR using a V_0 of 1V and v_{off} of 0.5V. The voltage developed across the output memristor is equal to 0V, 0.5V, 0.67V, and 0.75V when the inputs are '000,' '001,' '011,' and '111' respectively. Here, the output memristor switches in all cases except the first one. Now, if V_0 is changed to 0.75V, developed across the output memristor changes to 0V, 0.38V, 0.5V, and 0.56V when the inputs are '000,' '001,' '011,' and '111' respectively. In this case, the output switches to R_{OFF} only when there are at least two '1's in the input. The output is effectively the 3-bit minority function (Min3). As V_0 is further decreased to 0.67V, output changes only when the inputs are '111.' In other words, the output is '0' only when all the inputs are '1.' This is equivalent to a 3-bit NAND operation. The above logic can be extended to N-bit minority and NAND functions. The execution voltage, V_0, required to implement these functions is given by Equation 2.6, where N is the number of inputs.

$$\left(\frac{v_{off}}{R_{ON}}\right)\cdot\left\{R_{ON}+\left(\frac{R_{OFF}}{N-(n+1)}\right)||\left(\frac{R_{ON}}{n+1}\right)\right\} < V_0 < \left(\frac{v_{off}}{R_{ON}}\right)\cdot\left\{R_{ON}+\left(\frac{R_{OFF}}{N-n}\right)||\left(\frac{R_{ON}}{n}\right)\right\}, \quad (2.6a)$$

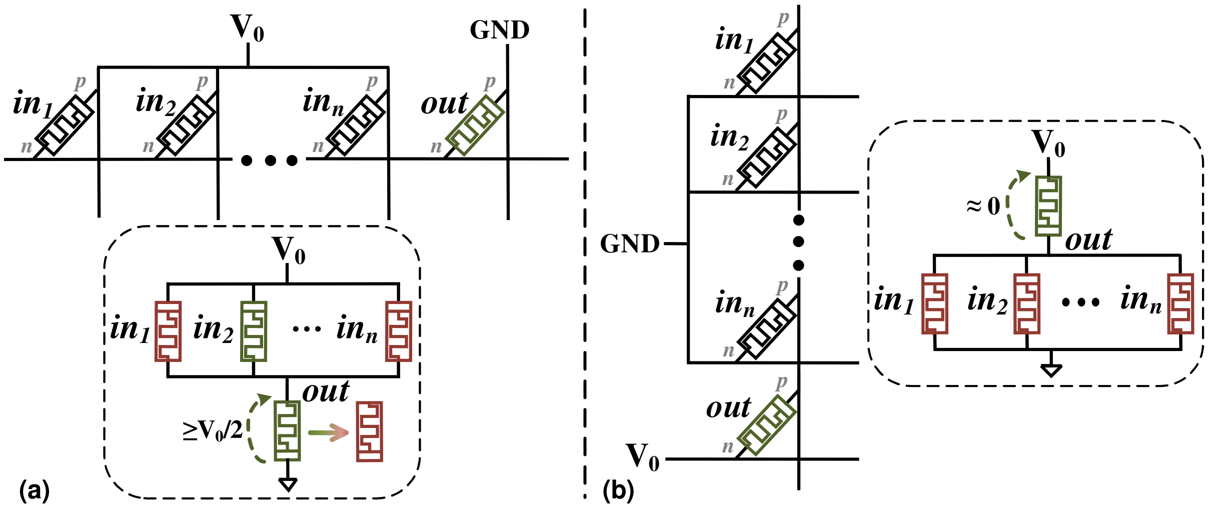


Figure 2.7. n-input NOR implementation in (a) a row and (b) a column.

$$\left(\frac{n+2}{n+1}\right) \cdot v_{off} < V_0 < \left(\frac{n+1}{n}\right) \cdot v_{off} \tag{2.6b}$$

The value of n is defined by the operation to be executed. For Min, $n = \lceil N/2 \rceil$ and for NAND, $n = N$. Equation 2.6b is an approximation of Equation 2.6a under the assumption that $R_{OFF} >> R_{ON}$.

Hence, in addition to NOR and NOT, Tri-HD supports a single cycle MinN and NAND. In theory this technique can be extended for any n. However, the non-availability of different voltage levels challenges its practical feasibility for large values of n. For example, Tri-HD requires a V_0 of 0.58V to implement a 6-bit NAND. It changes to 0.57V and 0.56V in case of a 7-bit and 8-bit NAND respectively. It is difficult to reliably generate these different and closely valued levels of voltages. Hence, to keep the implementations practical, we restrict Tri-HD to 2-bit and 3-bit NAND and Min. The vector-wide parallelism provided by these and future operations can be derived from the discussion on NOR operations.

OR: Tri-HD reduces the latency of OR operation to one cycle by exploiting the setting behavior of the memristor device. As discussed in Section 2, a device can be switched from R_{OFF} to R_{ON} by applying a voltage greater than the threshold, $|v_{on}|$. On the other hand, since MAGIC relies just on the resetting behavior of memristors, implementing OR in crossbar memory using MAGIC involves two NOR cycles. Figure 2.8 shows the voltages division for different possible inputs. Ground and V_0 terminals are opposite of those used for MinN. When all the input and output memristors are R_{OFF}, the voltage across the output memristor is much less than V_0. On the other hand, if one or more inputs are R_{ON}, the voltage across the output is approximately V_0. If V_0 is greater than v_{on}, then the output memristor switches to R_{ON}.

The above behavior is exploited to implement OR in memristive memory. The output memristor is first initialized to R_{OFF}. To execute OR in a row, the p terminals of the input memristors are grounded while V_0 is applied at the p terminal of the output. In case of OR in a column, V_0 is applied at the n terminals of the inputs the n terminal of the output is grounded (show p and n terminals in a figure). If the logical 'high' and 'low' states are represented by R_{ON} and R_{OFF} states of memristor, the result of OR operation corresponds to R_{OFF} when all the input bits are low and R_{ON} otherwise. The execution voltage, V_0, required to implement OR is given by,

$$\frac{|v_{on}|}{R_{OFF}} \cdot \left\{ R_{OFF} + (R_{ON}) \,||\, \left(\frac{R_{OFF}}{N-1}\right) \right\} < V_0 < \frac{|v_{on}|}{R_{OFF}} \cdot \left\{ R_{OFF} + \left(\frac{R_{OFF}}{N}\right) \right\}, \tag{2.7a}$$

$$\left(1 + \frac{R_{ON}}{R_{OFF}}\right) \cdot |v_{on}| < V_0 < \left(\frac{N+1}{N}\right) \cdot |v_{on}|, \tag{2.7b}$$

where N is the number of inputs. Equation 2.7b is an approximation of Equation 2.7a under the assumption that $R_{OFF} >> R_{ON}$.

2.6.2 Multi-Cycle Operations

The in-memory operations proposed in the above section can be combined to extend the functionality of the memory.

Maj and AND: Majority (MajN) and AND can be implemented by inverting MinN and NAND respectively. This results in 2-cycle MajN and AND in Tri-HD in contrast to four cycles

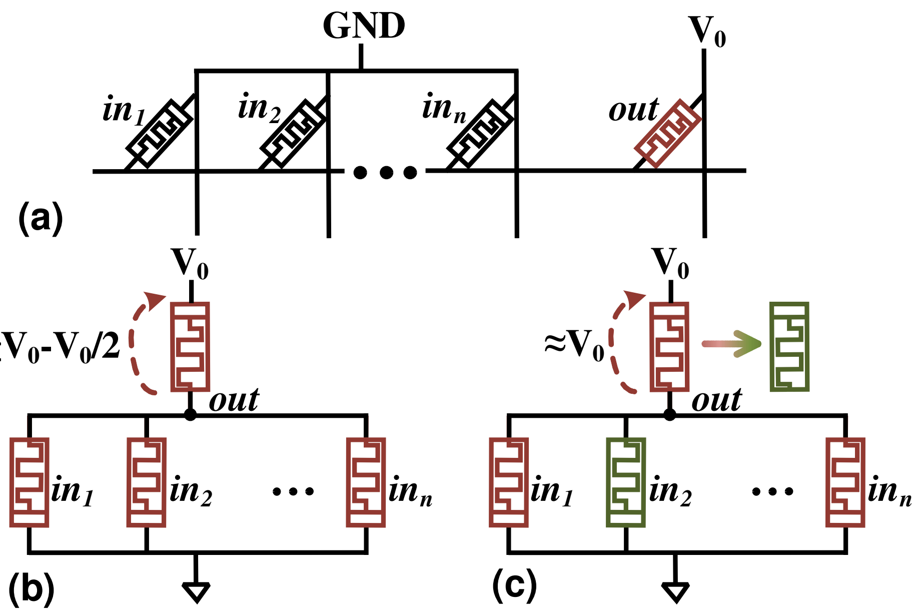


Figure 2.8. Voltage division for Tri-HD OR. (a) Application of voltage for OR, (b) output memristor remains R_{OFF} when all the inputs are R_{OFF} (red), and (c) output memristor switches to R_{ON} when one or more inputs are R_{ON} (green).

in MAGIC.

XOR: XOR ($\oplus$) can be expressed in terms of OR ($+$), AND (.), and NAND ($(.)'$) as follows:

$$A \oplus B = (A+B).(A.B)' \tag{2.8}$$

Figure 2.9 shows the in-memory implementation of Equation 2.8. Instead of calculating OR and NAND separately and then ANDing them, we first calculate OR and then use its output cell to implement NAND. In this way, we eliminate separate execution of AND operation. This logic just requires 2 Tri-HD cycles and one additional memristor device, which also acts as the output cell. In contrast, the state-of-the-art PIM technique proposed in [23] uses 5 cycles and 5 memristors for the fastest XOR implementation, while the most area conservative approach takes 7 cycles and 3 memristors. Hence, the proposed XOR implementation is both faster and smaller.

Addition: Tri-HD implements addition by combining XOR and MajN operations. A

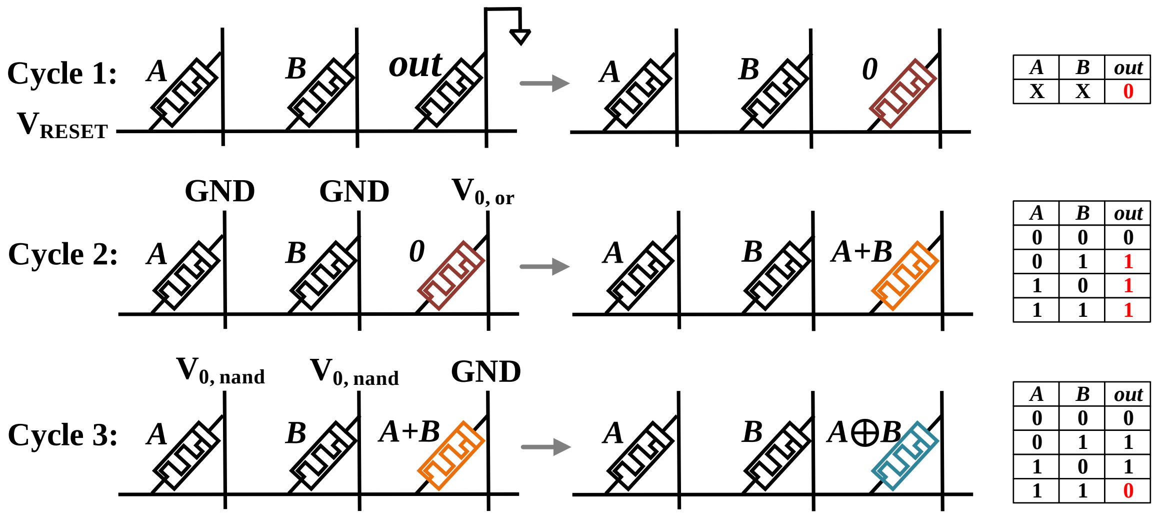


Figure 2.9. Different stages in implementing 2-bit XOR using Tri-HD

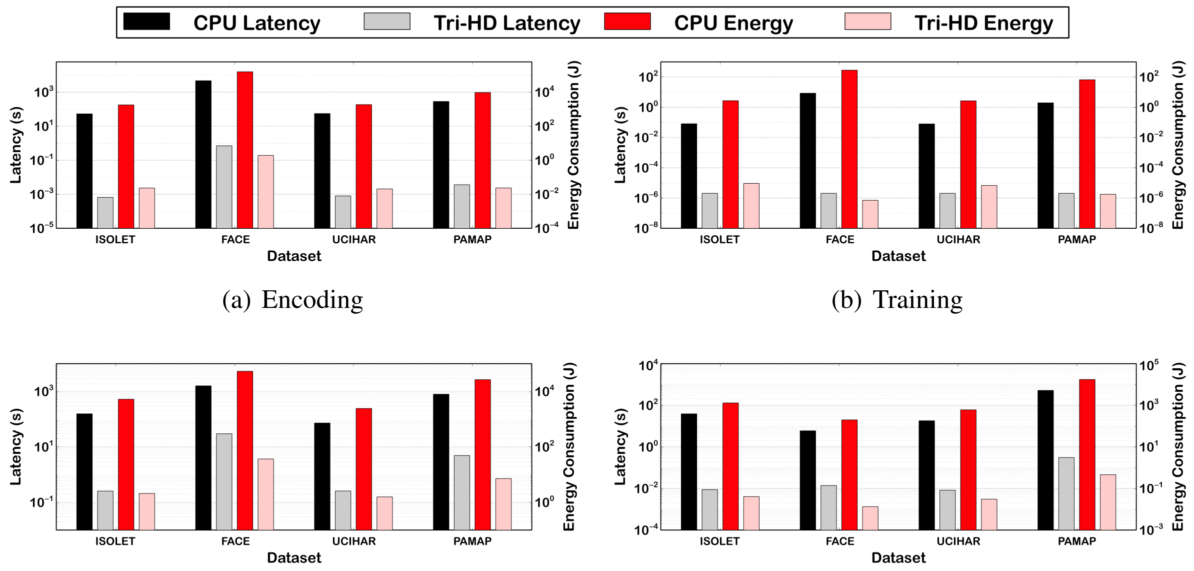
1-bit adder can be represented by,

$$S = A \oplus B \oplus C_{in}, \tag{2.9a}$$

$$C_{out} = A.B + B.C + C.A = MajN(A, B, C_{in}), \tag{2.9b}$$

where A, B, and C_{in} are 1-bit inputs while S and C_{out} are the generated sum and carry bits respectively. Here, S is implemented as two serial in-memory XOR operations. C_{out}, on the other hand, can be executed by inverting the output of MinN. Hence, S takes a total of 4 cycles and 2 additional memristors, while C_{out} needs 2 cycles and 2 additional memristors.

The previously proposed state-of-the-art processing in-memory techniques also support addition within the crossbar memory [23, 62]. These approaches break down an operation into a series of NOR operations. A typical addition implementation requires 12 NOR operations, resulting in 12 MAGIC NOR cycles [23] as compared to 6 in Tri-HD and 12 additional memristors as compared to 4 in Tri-HD.



(a) Encoding (b) Training

(c) Retraining (d) Inference

Figure 2.10. Latency and energy consumption of HD on CPU and the proposed Tri-HD-based architecture.

2.7 Results

2.7.1 Experimental Setup

We compare Tri-HD performance and energy efficiency with Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz (12 cores) with 16GB memory and 256GB SSD. All software support for application level evaluation including training and testing of HD model have been performed in CPU using python implementation. For hardware level evaluation we have designed a cycle accurate simulator which emulates the HD computing functionality at inference. Our simulator pre-store the randomly generated level and index hypervectors in memory and performs the encoding, training, retraining, and inference operations in-memory using the controller signals.

We extracted the circuit level characteristic of Tri-HD performing basic bitwise operations and give them as input to simulator. Performance and energy consumption of proposed hardware are obtained from circuit level simulations for a 45nm CMOS process using Cadence Virtuoso. We use VTEAM memristor model [63] for our memory design simulation with R_{ON} and R_{OFF}

of $10k\Omega$ and $10M\Omega$ respectively.

2.7.2 Workloads

We evaluate the efficiency of the proposed Tri-HD on four popular classification applications, as listed below:

Speech Recognition (ISOLET) [55]: The goal is to recognize voice audio of the 26 letters of the English alphabet. The training and testing datasets are taken from 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.

Face Recognition (FACE): We exploit Caltech dataset of 10,000 web faces [56]. Negative training images, i.e., non-face images, are selected from CIFAR-100 and Pascal VOS 2012 datasets [64]. We select 10% of images for the testing dataset which are completely separated from the training dataset. For the Histogram of Oriented Gradients (HOG) feature extraction, we divide a 32x32 image to (i) 2x2 regions for three color channels and (ii) 8x8 regions for gray-scale.

Activity Recognition (UCIHAR) [57]: The dataset includes signals collected from motion sensors for 8 subjects performing 19 different activities. The objective is to recognize the class of human activities.

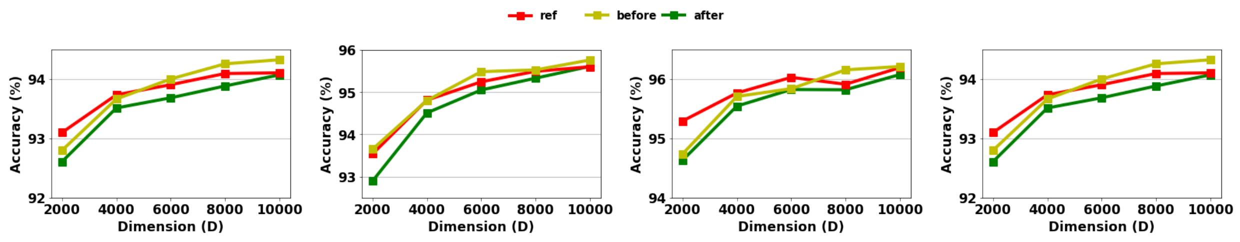
Physical Activity Monitoring (PAMAP) [58]: This dataset includes logs of 8 users and three 3D accelerometers positioned on arm, chest and ankle. They were collected over different human activities such as lying, walking and, ascending stairs, and each of them was corresponded to an activity ID. The goal is to recognize 12 different activities.

2.7.3 Tri-HD Results

Here we compare the efficiency of HD computing with D=10,000 for three different platforms: CPU, proposed in-memory Tri-HD architecture, and MAGIC [23].

Table 2.3. The energy efficiency, speedup and memory efficiency of Tri-HD as compared to MAGIC running different applications.

	ISOLET	**FACE**	**UCIHAR**	**PAMAP**
Energy Improv.	2.20×	2.20×	2.21×	2.26×
Speedup	1.86×	1.86×	1.88×	1.87×
Memory Efficiency	1.61×	1.61×	1.61×	1.82×



(a) ISOLET (b) FACE (c) UCIHAR (d) PAMAP

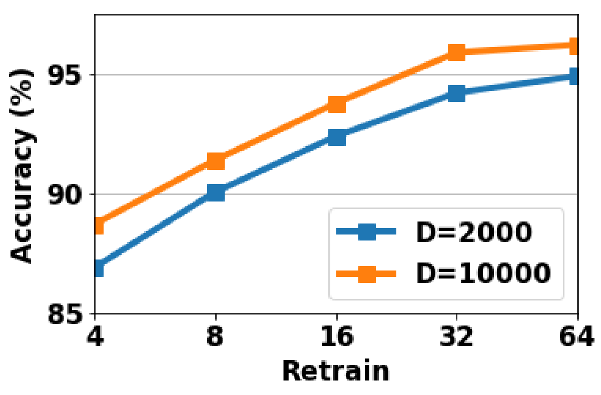
Figure 2.11. Impact of HD dimension reduction and rounding approaches on the classification accuracy for different datasets.

Table 2.3 compares the energy efficiency, speedup and memory efficiency of Tri-HD with MAGIC [23] while running different HD classification applications. The memory efficiency is defined as the number of processing cells required to execute in-memory operations. For a fair comparison, we use the proposed architecture to evaluate both Tri-HD and MAGIC. The results show that Tri-HD HD can achieve on an average 2.21× higher energy efficiency, 1.86× speedup, and 1.68× lower memory requirement as compared to MAGIC. Moreover, both FELIX [59] and MAGIC [33] enable dot product with parallel multiplication followed by a series of addition operations. In contrast, Tri-HD uses an approximate version of dot product which does not cause any inference accuracy loss for HD applications. Tri-HD's dot product with rounding before multiplication is 858× faster and 1.8× more energy efficient than dot product in FELIX. This is a direct result of replacing multiplication and serial addition operations with faster parallel in memory search operations.

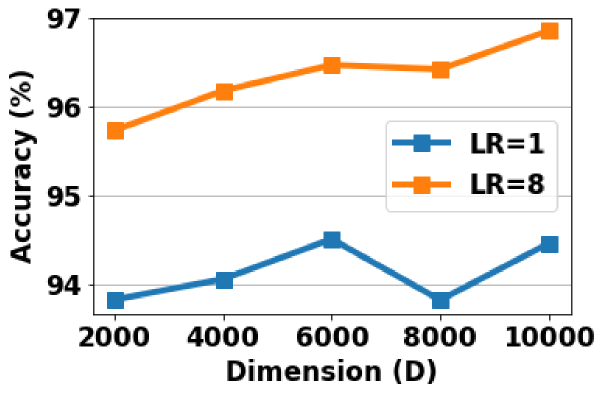
Figure 2.10 shows the energy consumption and execution time of HD encoding, training, retraining, and inference for different application on CPU and Tri-HD. Our evaluation shows that for all the applications tested, Tri-HD provides on average 434× and 2170× speedup and 4114× and 26019× lower energy consumption as compared to the CPU while running end-to-end HD training and inference respectively. End-to-end HD training involves encoding and training, followed by 64 iterations of retraining to achieve maximum accuracy. We observe that the latency does not depend much on the number of classes in a dataset in Tri-HD. Tri-HD exploits the fact that computation for each class is independent and executes them in parallel. This is specifically evident in the case of ISOLET and FACE datasets, where Tri-HD latency of end-to-end training mainly depends upon the number of training samples. Moreover, the iterative nature of retraining makes it the slowest operation in the HD pipeline for both Tri-HD and CPU. However, the extensive parallelism offered by Tri-HD makes in-memory retraining 276× faster than CPU. Here, we reported the result for Tri-HD with single memory partition. The higher efficiency of the Tri-HD comes from (i) memory compatible operations of HD which enables Tri-HD to parallelize the operations in different dimensions and (ii) lower data movement and higher locality of the data in-memory for Tri-HD computation. We see that applications with large number of features require more resources in order to perform the computation. Similarly, for each application Tri-HD efficiency can change depending on the number of partitions that each memory uses as shown in Section 2.5.

2.7.4 Tri-HD vs Previous Work

Here, we compare Tri-HD with existing PIM-based design for HD computing. It should be noted that Tri-HD is the first works that uses 32-bit dimensions for hypervector, in contrast with just 1-bit dimensions used by all other approaches. This increase both the algorithmic and hardware complexity in Tri-HD. However, owing to the higher bitwidth, Tri-HD is able to achieve much higher classification accuracy as compared to the existing work. For example, the model obtained from Tri-HD is 2.2% more accurate (Figure 2.11) as compared to RRAM-based design in [52]. At the same time, our end-to-end training with 20 retraining iterations is 48× faster and 3.4× more energy efficient. The speedup is a result of highly parallel and memory-compatible



(a) Retraining Iterations



(b) Learning Rate

Figure 2.12. Impact of number of retraining iterations and training learn rate on the application's classification accuracy on data set UCIHAR without rounding.

operations implemented in Tri-HD, whereas the approximate similarity metric of Tri-HD reduces the otherwise high energy consumed by more complex operations. While comparing both designs without retraining, Tri-HD is 289× faster and 5.2× more energy efficient.

Other PIM accelerators for HD [47, 48, 49, 50, 44, 54] implement fundamentally different encoding schemes. Owing to the simplicity of their encoders, these designs are faster and more energy-efficient. However, they suffer significantly in accuracy. For HDC inference, the accelerators proposed in [47] and [54] are on average 57× and 24× faster and consume 3× and 731× less energy than Tri-HD respectively. However, they achieve 12.7-18.2% and 4.9-9.1% less accuracy respectively as compared to Tri-HD. The remaining work [48, 49, 50, 44] do not support the classification tasks discussed in this chapter. Hence, a comparison with these work is beyond the scope of the dissertation.

We also compare Tri-HD with DNN running on FloatPIM [36] for ISOLET dataset. We observe that Tri-HD is 221× (2132×) faster and 1.9× (3.6×) more energy-efficient than FloatPIM with 32-bit fixed point (16-bit bfloat) data representation, while providing 1.7% inference accuracy loss. The speedup is the result of simpler operations in Tri-HD as compared to vector-matrix multiplications in FloatPIM.

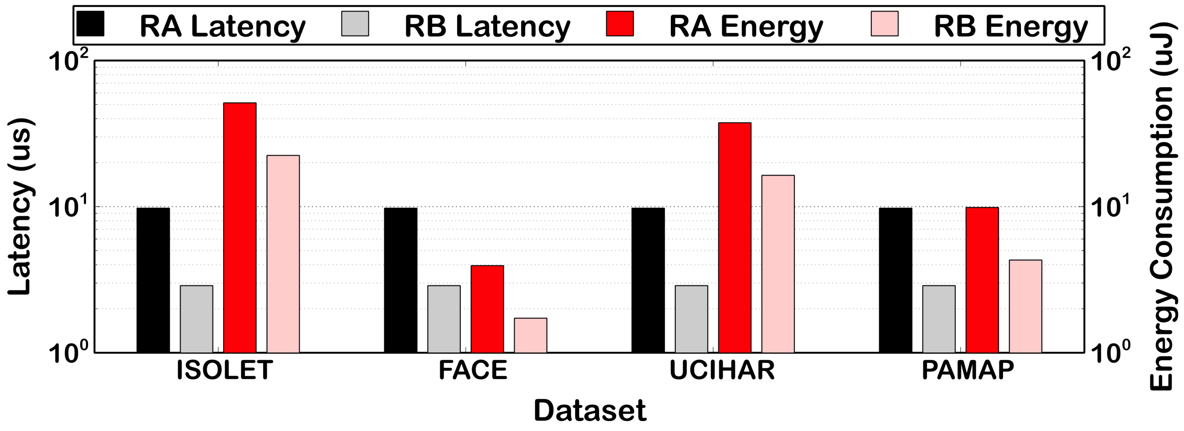


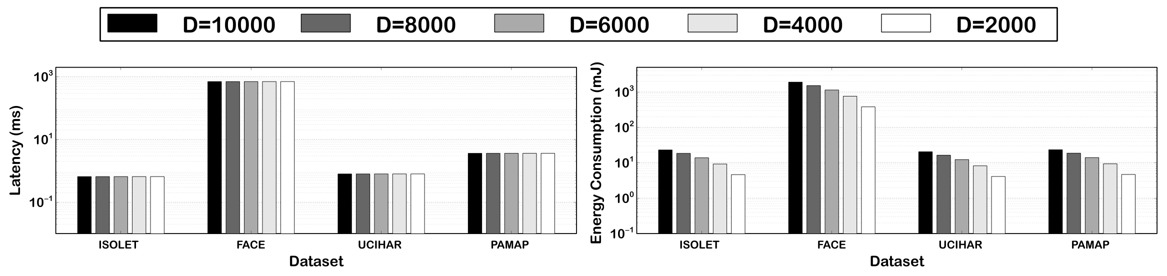
Figure 2.13. Latency and energy consumption per similarity check in Tri-HD with different rounding approaches. RA stands for rounding after multiplication and RB stands for rounding before multiplication.

2.7.5 Tri-HD with Approximate Similarity

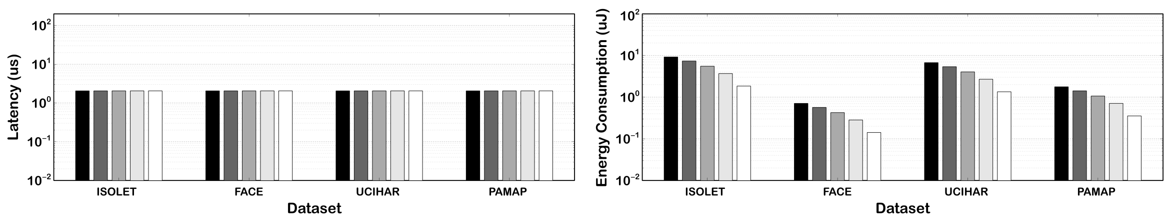
Figure 2.11 shows the impact of the quantization in Tri-HD on the classification accuracy of HD. We show three different policies: the reference, that is not rounding the products from the matrix multiplication, round before, and round after, as explained in Section 2.4.

Accuracy: Figure 2.11 shows that the prediction accuracy increases at a diminishing rate as the number of dimension increases. This is true for all five data sets we used, and for all three rounding policies we used. It is noticeable that both the rounding after policy and round before policy perform no worse than the reference policy. In fact, on average, Tri-HD using the before (after) policy, is able to achieve 0.52% (0.36%) better accuracy when using $D = 10,000$.

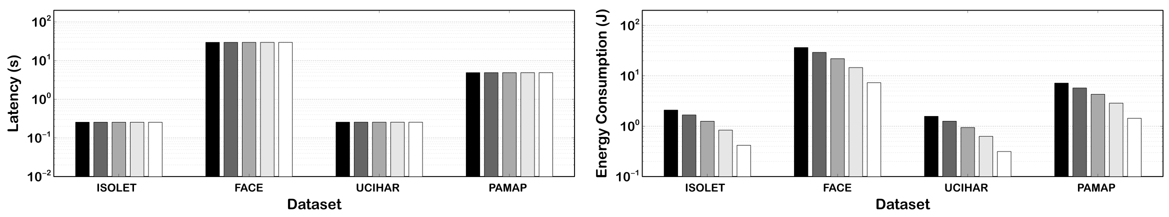
Figure 2.12 demonstrates the influence on the number of retrain iterations and learning rate on the classification accuracy of Tri-HD. The plot on the left shows the change in accuracy during retraining on the UCIHAR dataset. This demonstrates that even in the presence of inaccurate computations from quantizing the accumulation to powers of 2, HD is still able to improve in accuracy by a significant amount during retraining. This shows that HD is robust to noise. This property can further be exploited to increase the efficiency of the design as discussed in Section 2.7.6. The plot on the right demonstrates the difference in accuracy after retraining



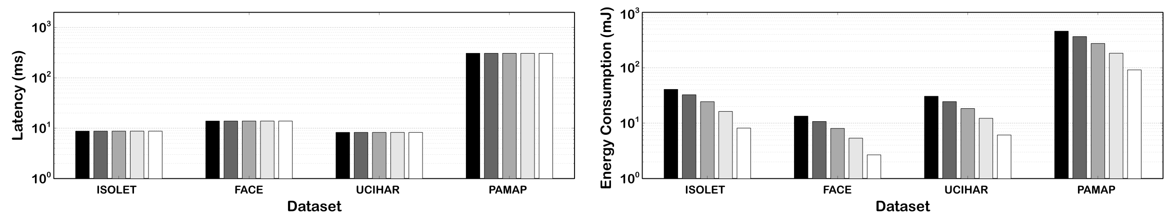
(a) Encoding



(b) Training



(c) Retraining



(d) Inference

Figure 2.14. Latency and energy consumption of Tri-HD for different dimensions.

utilizing different learning rates with varying dimensionality. The optimal learning rate may be different for each dataset. Here we experimentally observed that, as shown by this graph, a learning rate of 8.0 gave close to the best results on average. The learning rate determines how strongly each misprediction during retraining effects the model being trained.

Performance and Energy: Figure 2.13 shows the latency and energy consumption of a similarity check (dot product) for the two rounding approaches. As expected, similarity metric with rounding before multiplication is on average 3.4× faster and 2.3× more energy-efficient than similarity with rounding after multiplication because it avoids the comparatively slow and high energy consuming in-memory multiplication. For rounding before multiplication, in-memory search happens twice, once for the input and the other for the shifted output. While in the case of rounding after, in-memory search is performed only for the output. We also observe that the number of classes has negligible effect on the latency of similarity check. Whereas, the energy consumption increases almost linearly with the number of classes because Tri-HD needs to check the similarity of an input vector with all the class hypervectors.

2.7.6 Tri-HD Energy-Accuracy Trade-off

HD computing works based on the pattern of neural activity which are in high dimensional space. In theory, the dimensional of the hypervector should be large enough (e.g., $D = 10,000$) to ensure the randomly generated base hypervectors are nearly orthogonal. However, HD computing shows robustness to scaling the hypervector dimensions. Figure 2.11 shows the HD accuracy when the hypervector dimension scales from 2000 to 10,000. The result shows that for all applications the HD can provide the similar accuracy as 10,000 when the hypervector dimension scales to 8000. In addition, in reducing the hypervector dimensions from 10000 to 2000, HD loses on average only 1.6% in accuracy.

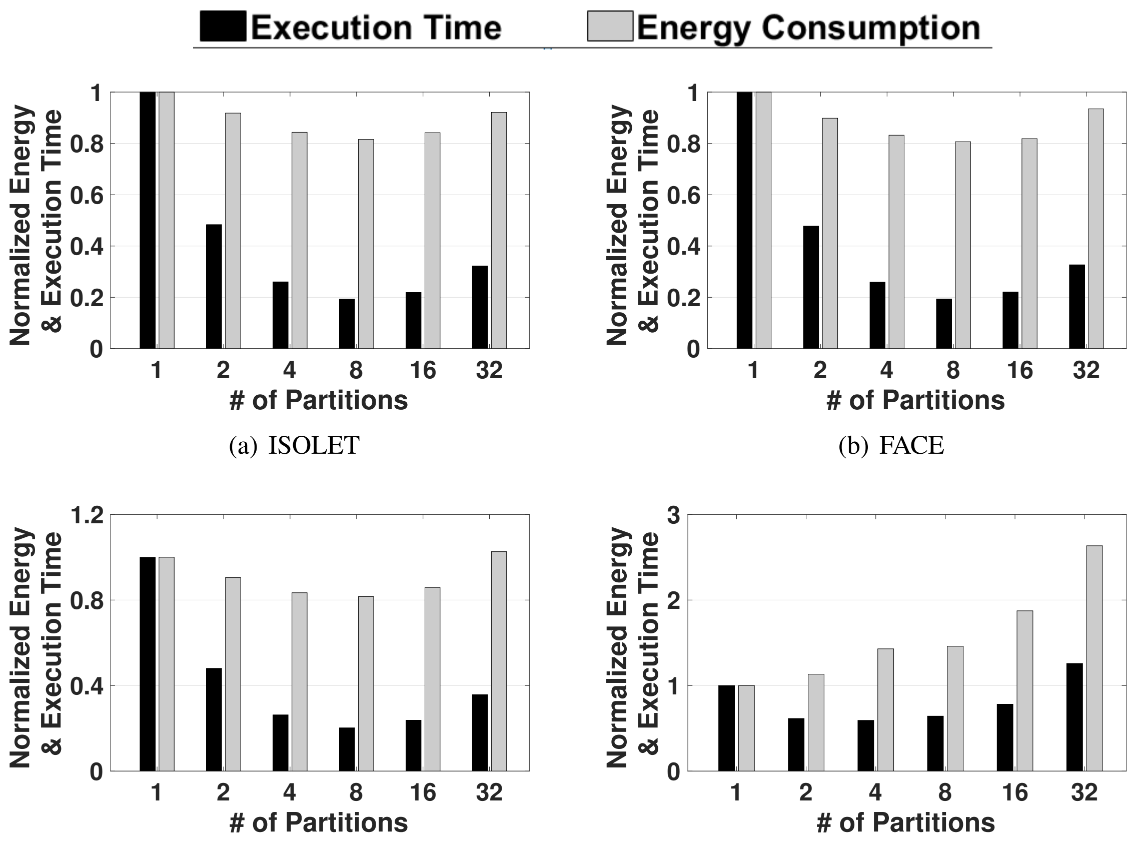
Tri-HD can exploit the robustness of HD to dimensionality in order to reduce the computation cost. Figure 2.14 shows the latency and energy consumption of running HD classification in memory. Our evaluation shows that reducing the hypervector dimension reduces Tri-HD energy consumption. This efficiency comes from the less number of class elements and operations that HD needs to store and process in lower dimension. Our result shows that Tri-HD memory requirement decreases linearly with the hypervector dimensions. For example, HD with D=2000 dimensions consumes 78% lower energy. Note that the latency of Tri-HD does not change with the hypervector dimensions. In fact, Tri-HD is designed to perform bit parallel operations where all computations can be parallelized across different dimensions.

Hence, there is a trade-off between the accuracy and efficiency when the hypervector dimension reduces. The results are relative to Tri-HD architecture running the baseline HD with $D = 10,000$ dimensions. Let the quality loss, ΔE, be defined as the difference between the HD classification accuracy in low dimension and $D = 10,000$. When our design ensures 0.5% quality loss ($\Delta E = 0.5\%$), the Tri-HD can provide 25% energy efficiency as compared to the baseline HD model. Similarly, ensuring quality loss of less than 1% (2%), Tri-HD energy and memory efficiency further improve by 65% and 78% receptively.

2.7.7 Tri-HD Parallelism

The HD efficiency depends on the amount of parallelism which we can apply to different modules. The most area efficient method for HD encoding is to store all ID and level hypervectors in a single memory partition and perform XOR operation between each ID and corresponding level in series. This method serially processes the features and its performance is directly related to the number of features. Our design parallelizes the encoding module by partitioning the memory block as discussed in Section 2.3. For instance, HD using two memory partitions can process two features at the same time. In the best case, the number of memory partitions can be equal to the number of features that exist in application. For example, for ISOLET with 617 features, the encoding operation can be fully parallelized by dividing the memory block into 617 partitions. Each memory partition computes the XOR operation of ID and one of the level hypervectors, which is selected depending on the feature value. There are two overheads of using multiple memory partitions (i) The effective memory requirement increases, since each partition needs to replicate the level hypervectors and assign rows to perform computation and store the XOR result. (ii) All XOR results need to be written in a single memory in order to add together and generate an encoded data. This write operation needs to perform sequentially and degrades the efficiency of using multiple memory block.

Figure 2.15 compares the impact of number of memory partitions on the energy efficiency, execution time and memory requirement of HD computing. Our evaluation shows that



(a) ISOLET

(b) FACE

(c) UCIHAR

(d) PAMAP

Figure 2.15. Impact of number of partitions on the energy consumption and execution time of Tri-HD encoding running different HD applications.

increasing the number of partitions at first improves the performance and energy efficiency of the computation, however, it results in less efficiency when the number of partitions surpasses 8. For example, encoding ISOLET with 16 partitions is 10% slower than encoding the same dataset with 8 partitions. For more than 8 partitions, the cost of combining the results from different partitions exceeds the benefits provided by parallelism due to partitions.

This chapter showed how hyperdimensional computing can be accelerated with processing in-memory. Chapter 3 shows how in-storage computing can enable hyperdimensional computing on larger datasets, while further reducing the amount of data transfers.

Chapter 2, in part, is a reprint of the material as it appears in S. Gupta, M. Imani, and T. Rosing, “FELIX: Fast and Energy-Efficient Logic in Memory,” *IEEE/ACM International*

Conference On Computer Aided Design (ICCAD), 2018. The dissertation author was the primary investigator and author of this material.

Chapter 2, in part, is currently being prepared for submission for publication of the material. S. Gupta, J. Morris, X. Shen, M. Imani, B. Aksanli, and T. Rosing, “Tri-HD: Train, Re-train, and Infer with Hyperdimensional Computing in Memory.” The dissertation author was the primary investigator and author of this material.

Chapter 3

Accelerating Hyperdimensional Computing in Storage

Chapter 2 presented a PIM architecture to accelerate HD computing. However, PIM may not be able to store very large datasets and may fetch data from disk. Recent work has introduced computing capabilities to solid-state disks (SSDs) to process data in storage [26, 27, 28, 1]. This not only reduces the computation load from the processing cores but also processes raw data where it is stored. However, the state-of-the-art in-storage computing (ISC) solutions either utilize a single big accelerator for a SSD or limit the gains by using complex power-hungry accelerators down the storage hierarchy [2]. Such architectures are not able to fully leverage its hierarchical design.

In this chapter, we propose an in-storage computing (ISC) based HD computing system that spans multiple levels of the storage hierarchy. We exploit the internal bandwidth and hierarchical structure of SSDs to perform HDC operations over multiple data samples in parallel. Our main contributions are as follows:

- We present a novel ISC architecture for HDC which performs HDC classification and clustering completely in storage. It enables computing at multiple levels of SSD hierarchy, allowing for highly-parallel ISC. Our hierarchical design provides parallelism and hides a significant part of the performance cost of ISC in the storage read/write operations.
- We introduce the concept of batching in HDC and utilize it to make our ISC implementation

more efficient. During training, we batch together multiple data samples encoded in the HDC domain in storage. This allows us to partially process data without accessing all encoded hypervectors. Batching enables us to have a minimal aggregation hardware requirement. Batching also reduces the amount of data sent out of storage.

- Store-n-Learn utilizes die-level accelerators to convert raw data into hypervectors locally in all the flash planes in parallel. Unlike previous work [2], our accelerator is simpler and hides its computation latency by the long read times of raw data from flash arrays. Our die-level accelerators can perform both batched and non-batched encoding efficiently in flash planes. For batched encoding, the accelerator processes multiple inputs in a page in parallel. Whereas, it generates multiple dimensions corresponding to an input in parallel during non-batched encoding. This flexibility is enabled by our innovative adder tree design.

- We present a top-level SSD accelerator, which aggregates the data from different flash dies. This accelerator is implemented on an FPGA-based device controller. We implement new and efficient FPGA designs for HDC training, retraining, inference, and clustering. While HDC training provides sufficiently accurate initial models, retraining significantly improves the accuracy of the models by iterating over training data and updating the models multiple times. Store-n-Learn inference allows the users to directly obtain the classification result from the storage drive without sending the entire model to the host. Moreover, Store-n-Learn clustering leverages the FPGA already present in storage and iteratively processes the datasets multiple times to generate high quality cluster centers.

- We also present host-side and drive-side primitives to enable the FPGA to work seamlessly with the die-level accelerators.

- We evaluate Store-n-Learn over ten popular classification and clustering datasets. Our experimental results show that Store-n-Learn is on average 222$\times$ (543$\times$) faster than

CPU and 10.6$\times$ (7.3$\times$) faster than the state-of-the-art ISC solution, INSIDER for HDC classification (clustering).

3.1 Related Work

Hyperdimensional Computing: Prior work applied the idea of hyperdimensional computing to a wide range of learning applications, including language recognition [45], speech recognition [42], gesture detection [65], human-brain interaction [66], and sensor fusion prediction [7]. For example, work in [9] proposed an HD encoder based on random indexing for recognizing a text's language by generating and comparing text hypervectors. Work in [65] proposed an encoding method to map and classify biosignal sensory data in high dimensional space. Work in [8] proposed a general encoding module that maps feature vectors into high-dimensional space while keeping most of the original data. Prior work also designed different training framework to enable sparsity and quantization in HD computing [67, 68]. Prior work also tried to design different hardware accelerators for HD computing. This include accelerating HD computing on existing FPGA, ASIC, and processing in-memory platforms [69, 48, 44]. However, these solutions do not scale well with the number of classes and dimensions, primarily due to the data movement issue. In addition, the existing processing in-memory architectures only accelerate the encoding, training, or associative search and they are not scale with number of classes of hypervector dimensions. Moreover, they work with binary hypervector which has been shown to provide very low classification accuracy in HD space [70]. In contrast, our proposed Store-n-Learn accelerates all the phases of HDC classification and clustering by fundamentally addressing data movement and memory requirement issues. In addition, Store-n-Learn scales with the size of data and the complexity of learning task.

In-Storage Computing: The major bottlenecks in the current storage systems include the slow flash array read latency and the SSD to host I/O latency [71]. To alleviate these issues prior work introduced ISC architectures [72, 28]. These work exploit the embedded cores present in the SSD controller to implement ISC. Another set of work in [27, 26, 2] used ASIC accelerators in SSD for specific workloads. The work in [1] proposed a full-stack storage system to reduce the host-side I/O stack latency. However, all these works propose single-level computing in storage. Store-n-Learn on the other hand, is the first work to push the computing all the way down to the flash die to extract maximum parallelism. It also uses a top level accelerator to provide addition layer of computing. The combination provides a faster implementation that overcomes the SSD to host transfer bottleneck for HDC.

3.2 Hyperdimensional Computing

Brain-inspired Hyperdimensional (HDC) computing has been proposed as the alternative computing method that processes the cognitive tasks in a more light-weight way [6, 45]. HDC offers an efficient learning strategy without over-complex computation steps such as back propagation in neural networks. HDC works by representing data in terms of extremely large vectors, called hypervectors, on the order of $10,000$ dimensions. HDC has been shown to incur minimal error rates, providing accuracy similar to the state-of-the-art learning algorithms like DNNs [59] and k-means [11]. However, the high-dimensional space of HDC makes it robust to external noise sources and hardware-induced errors like device failures [73], stuck-at-fault errors [48], errors from low-precision hardware [45], and noisy communication [74]. Hence, in noisy and error-prone systems HDC proves superior to algorithms like DNNs and k-means that incur large accuracy losses. HDC performs the learning task after mapping all training data into the high-dimensional space. The mapping procedure is often referred to as *encoding*. Ideally, the encoded data should preserve the distance of data points in the high-dimensional space. For example, if a data point is completely different from another one, the corresponding hypervectors should be orthogonal in the HDC space. There are multiple encoding methods proposed in literature [8, 9]. These methods have shown excellent classification accuracy for different data types. In the following, we explain the details of HDC classification steps.

3.2.1 Encoding

Let us consider an encoding function that maps a feature vector $\mathbf{F} = \{f_1,\ f_2, \ldots,\ f_n\}$, with n features ($f_i \in \mathbf{N}$) to a hypervector $\mathbf{H} = \{h_1,\ h_2, \ldots,\ h_D\}$ with D dimensions ($h_i \in \{0, 1\}$). We first generate a projection matrix **PM** with D rows and each row is a vector with n dimensions randomly sampled from $\{-1, 1\}$. This matrix is generated once offline and is then be used to encode all of the data samples. We generate the resulting hypervector by calculating the matrix vector multiplication product of the projection matrix with the feature vector:

$$\mathbf{H}' = \mathbf{PM} \times \mathbf{F} \tag{3.1}$$

After this step, each element h_i of a hypervector $\mathbf{H}'$ has a non-binary value. In HDC, binary (bipolar) hypervectors are often used for the computation efficiency. We thus obtain the final encoded hypervector by binarizing it with a sign function ($\mathbf{H} = sign(\mathbf{H}')$) where the sign function assigns all positive hypervector dimensions to '1' and zero/negative dimensions to '-1'. The encoded hypervector stores the information of each original data point with D bits.

3.2.2 Training for Classification

In the training step, we combine all the encoded hypervectors of each class using element-wise addition. For example, in an activity recognition application, the training procedure adds all hypervectors which have the "walking" and "sitting" tags into two different hypervectors. Where $H^i_j = \langle h_D, \cdots, h_1 \rangle$ is encoded for the j^{th} sample in i^{th} class, each class hypervector is trained as follows:

$$C^i = \sum_j H^i_j = \langle c^i_D, \cdots, c^i_1 \rangle \tag{3.2}$$

3.2.3 Classification Retraining

HD classification training requires only a single pass over training data and delivers reasonable accuracy. However, some critical applications and/or situations may demand higher accuracy. In such cases, HD classification retraining can significantly improve the accuracy of the base trained hypervectors by iterating over the training data multiple times. Consider a training input hypervector H_x which belongs to class j but is incorrectly assigned to class k, the retraining step proceeds as follows:

$$C_i = C_i - H_x \tag{3.3}$$

$$C_j = C_j + H_x \tag{3.4}$$

This step can be repeated several times for the whole dataset until the desired accuracy is achieved.

3.2.4 Classification Inference

The main computation of inference is the encoding and associative search. We perform the same encoding procedure to convert a test data point into a hypervector, called a *query hypervector*, $Q \in \{-1,1\}^D$. Then, HDC computes the similarity of the query hypervector with all k class hypervectors, $\{C_1, C_2, \cdots, C_k\}$. We measure the similarity between a query and a i^{th} class hypervector using: $\delta\langle Q,\ C_i\rangle$, where δ denotes the similarity metric. The similarity metric most commonly used is Cosine Similarity as it provides the highest accuracy. However, other similarities metrics like dot product and hamming distance for binary class hypervectors are also used. After computing all similarities, each query is assigned to a class with the highest similarity.

3.2.5 Clustering

The HD clustering algorithm is very similar to the popular K-means algorithm. HD clustering, like K-means, first starts off with random centers. Each cluster center is assigned a unique hypervector. Then, the algorithm iterates through all of the data points while comparing their corresponding hypervectors with those of the cluster centers using cosine similarity metric.

Each data point is assigned the center with maximum similarity. After all the points are labeled, the new centers are chosen by superimposing the corresponding points to form an updated set of cluster centers:

$$C_k^{t+1} = \sum_{H_X \in C_k^t} H_X \tag{3.5}$$

where $H_X \in C_k^t$ indicates the set of all data points assigned to the cluster represented by C_k after iteration t. The process is repeated until convergence or the maximum number of iterations is reached. Convergence occurs when no point is assigned to a different cluster compared to the previous iteration.

3.2.6 Challenges

HDC is light-weight enough to run at acceptable speed on a CPU [70]. Utilizing a parallel architecture can significantly speed up the execution time of HDC [44]. However, with the constantly increasing data sizes along with the explosion in data that occurs due to HDC encoding, running this algorithm on current systems is highly inefficient. All of these platforms need to fetch the extremely large hypervectors from memory/disk in order to process them. They also require huge memory space to store HDC hypervectors and train on them. With the available parallelism across thousands of dimensions and simple operations needed, in-storage computing (ISC) is a promising solution to accelerate HDC encoding and training.

General-purpose ISC solutions partially address the data transfer bottleneck but still are not able to fully exploit the huge internal SSD bandwidth [1]. The state-of-the-art application specific ISC [2] try to exploit the internal SSD bandwidth but provide only one-level of computing, which fails to accelerate applications which either (i) have a computing logic that is too complex to implement using the small accelerator or (ii) require post-processing computation steps. Store-n-Learn aims to overcome these issues by breaking complex HDC algorithms into simpler, both data-size and computation-wise, parallelizable tasks. Then, Store-n-Learn utilizes two levels of computation within the SSD, one at the chip-level and other at the SSD level, to efficiently implement those tasks.

3.3 Store-n-Learn Design

Store-n-Learn is an ISC design that performs HDC classification and clustering completely in storage. Figure 3.1 shows an overview of Store-n-Learn SSD architecture. A flash die consists of multiple flash planes, each of which generates a page during a read cycle. Store-n-Learn inserts a simple low-power accelerator, die-level accelerator (green on the right in Figure 3.1), in each plane to encode every read page into a hypervector. These hypervectors are then sent to a SSD-level FPGA, which accumulates these hypervectors in batches in the top-level accelerator (green on bottom left in Figure 3.1). The FPGA is also used for retraining, inference, and clustering on the encoded hypervectors received from the flash planes. Store-n-Learn uses a scratchpad (green on top left in Figure 3.1) in the controller to store the projection matrix, which it receives as an application parameter from the host. Batching ensures that data generated by each SSD-wide read operation is used in training as soon as it is available, without waiting for the remaining data. The top-level accelerator is a FPGA which uses INSIDER acceleration cluster [1] to implement all HDC operations other than encoding. We utilize the INSIDER's software stack to connect Store-n-Learn to the rest of the system. We modify the SSD drivers and INSIDER virtual files mechanism to enable computing in flash chips and make it visible to the FPGA.

3.3.1 Batched HDC Training in Store-n-Learn

The size of raw data (number of data points) combined with the size of each hypervector (size of each encoded data point) makes it unrealistic to store all the encoded hypervectors and then perform HDC training over them. Hence, we employ batching to perform partial training with the hypervectors available at any given moment. As mentioned in Section 3.2, the initial HDC training algorithm to create a class hypervector (3.2) is to add up all of the encoded samples

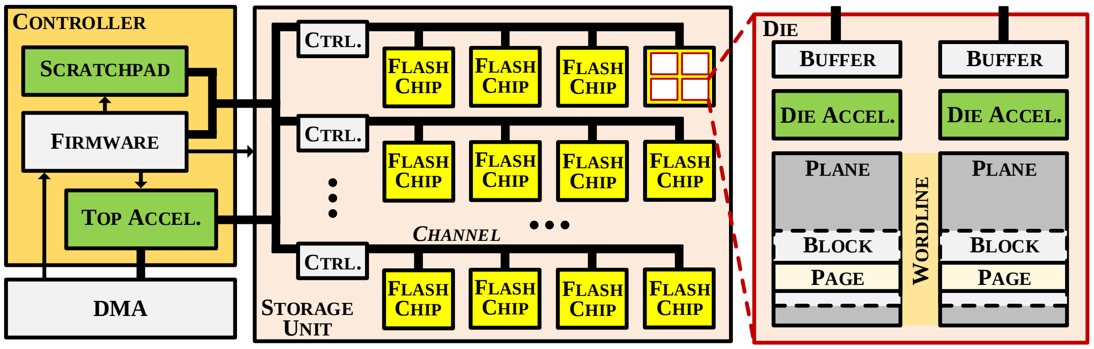


Figure 3.1. Store-n-Learn SSD Overview. The components added by Store-n-Learn are shown in green.

belonging to a given class. This summation can be spit up into batches of partial sums and maintain the same result. For example, say there are s samples for each class, the total sum can be split up into k partial sums or batches and the batch size defined as $b = s/k$, as shown in Equation 3.6.

$$C^i = \sum_{j=1}^{b} H_j^i + \sum_{j=b+1}^{2b} + \ldots + \sum_{j=((s-1)b)+1}^{s} H_j^i \tag{3.6}$$

Batching allows Store-n-Learn to process a subset of encoded hypervectors together. Store-n-Learn chip-level accelerators encode raw data into hypervectors and send them to the top-level SSD FPGA accelerator for further processing. All flash chips operate in parallel to encode some of their data, send the hypervectors to FPGA, and operate on the next set. Each of these hypervectors belongs to a specific *class*. For an application with C classes, we allocate enough memory in the top-level accelerator to store C *model hypervectors*, each assigned to a class. We batch all incoming hypervectors from flash that belong to the same class together and bundle the result with the corresponding model hypervector. This is continued until all required data has been encoded and used to train model hypervectors. In the end, the top-level model hypervectors represent a fully trained model of the data. Batching provides us with two benefits. First, it minimizes the memory requirement during training. Second, it reduces its effective latency by combining hypervectors as soon as they are generated. This hides a major part of training latency with the time taken to read data from flash.

What if the size of model hypervectors is too large to store at top-level FPGA accelerator? Some application may need too many dimensions or have too many classes to store all model hypervectors at the FPGA, which at best may have few MBs of blocked RAMs (BRAMs). In such a case, even with balanced data, it will not be possible to train the model completely in storage. However, Store-n-Learn can still perform training in batches and reduce the amount of data sent to the host for processing. Now, instead of allocating FPGA BRAMs for all model hypervectors, it is dynamically allocated according to the encoded input hypervectors available at a time. If an input hypervector does not belong to one of the present models, a model hypervector is sent out to the CPU host and an empty model hypervector corresponding to the class associated with incoming hypervector is allocated instead. The implementation details are presented in Section 3.3.3. The host is then responsible for combining various batched training hypervectors together.

In this operating mode, Store-n-Learn still reduces the amount of data movement compared to sending the raw low dimensional data. Here, we define n as the number of features or dimensionality of the original data, D as the dimensionality of the encoded hypervectors, and b as the batch size. When $nb > D$, the total data movement of the resulting batched hypervectors is less than the amount of original data sent in low-dimensional space when the batched hypervector uses the same bitwidth as the original data. However, we can utilize lower bitwidth representations as we encode the data into a hypervector whose elements are $\{-1, 1\}$ and then bundle the hypervectors with element-wise addition. Therefore, the range of data in any given dimension can be defined by the normal distribution with a mean of 0 and standard deviation of $\sqrt{b}$. We can represent each dimension of the batched hypervector with $(\log_2 4\sqrt{b}) + 1$ bits while maintaining an accurate representation. We multiply by 4 to capture 4 standard deviations away and add one to account for the sign bit. In this case, assuming the original data is represented with 32 bits, Store-n-Learn sends less data than the data movement required to send the original data in low-dimensional space when $32nb > D((\log_2 4\sqrt{b}) + 1)/32$

3.3.2 Encoding Near Data via Flash Hierarchy

The modern SSD architecture is hierarchical in nature. An SSD has multiple channels. Each channel is shared by 4-8 flash chips as shown in Figure 3.1. The flash chip may consist of several flash dies which are further divided into flash planes, each plane consisting of a group of blocks, each of which store multiple pages. Each plane has a page buffer to write the data to. Operations in SSD happen in page granularity where the size of pages usually ranges from 2KB-16KB [75]. To fully utilize the flash hierarchy, we introduce accelerators for each flash plane as shown in Figure 3.1. The aim of this added computing primitive is to process the data where it has no conflict or competition for resources.

Chip-level Accelerator Design:

Store-n-Learn plane-accelerator encodes an entire page with raw data to generate a D dimensional hypervector. Let us assume the SSD page size to be 4KB (p_s) with each data point being 4 bytes (d_s). This translates to 1K data points (p_s/d_s). Let the feature vector contain 1K features. Assuming that the feature vectors are page-aligned, each page stores one feature vector. HDC encoding multiplies n-size feature vector with a projection matrix containing $D \times n$ 1-bit elements. Our accelerator calculates the dot product between two page-long vectors, one read from the flash array and another being a row-vector of the projection matrix. This involves element-wise multiplication of the two vectors and adding together all the elements in the product. Since the weights in the projection matrix $\in \{1, -1\}$, we reduce the bits required to store the weights by mapping them such that $1 \rightarrow 1$ and $(-1) \rightarrow 0$. We use 2's complement to break the multiplication into an inversion using XNOR gates and then adding the total number of inverted inputs to the accumulated sum of XNOR outputs. The accelerator is shown in Figure 3.2. It consists of an array of 32K XNOR gates followed by a 1K input tree adder (labeled CSA in Figure 3.2). The tree adder is a pruned version of the Wallace carry-save tree, where the operand size throughout the tree is fixed to 4B. It reduces 1024 inputs to 2, which is followed by a carry look ahead addition (labeled CLA in Figure 3.2). This gives us the dot product of the two vectors.

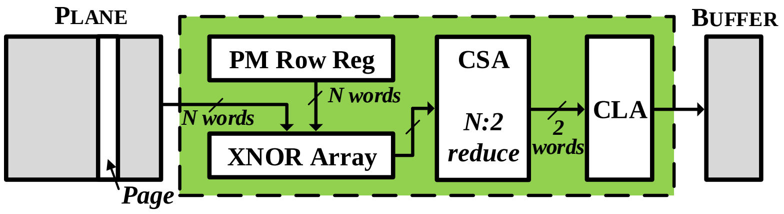


Figure 3.2. Store-n-Learn die accelerator

It is the value of one dimension of the encoded hypervector. The accelerator is iteratively run D times to generate D dimensions. Depending upon the power budget, Store-n-Learn may employ multiple parallel instances of this accelerator to reduce the total number of iterations. Since D is generally large, the generated D-dimensional vector is multi-page output. Store-n-Learn writes the output of the accelerator to the page buffer of the plane, which serves as the response to the original SSD read request.

Storing Input Data:

The accelerator above assumed the size of the feature vectors to be exactly the same as that of a page. However, this is rarely the case. State-of-the-art ISC designs use page-aligned feature vectors, which may lead to poor storage utilization if the feature vector size is too small or just larger than the page size. For example, in a page-aligned feature vector setting, a 4KB page may fit only one 512B feature instead of eight. Also, a 5KB feature vector may occupy two complete pages. To alleviate the issue, we propose a cross-plane storing scheme, which considers all the planes in a chip when storing data, with the goal of increasing the traditional ISC storage utilization while being accelerator-friendly. We first describe the case when the size of the feature vector is smaller than the page size. The scheme, shown in Figure 3.3 on the left, divides an n-sized feature vector into n_p equal segments such that the most efficient storage is given when:

$$\underset{c}{\operatorname{argmax}} \; (c \times n \, + \, d.n/n_p \leq p_s)$$

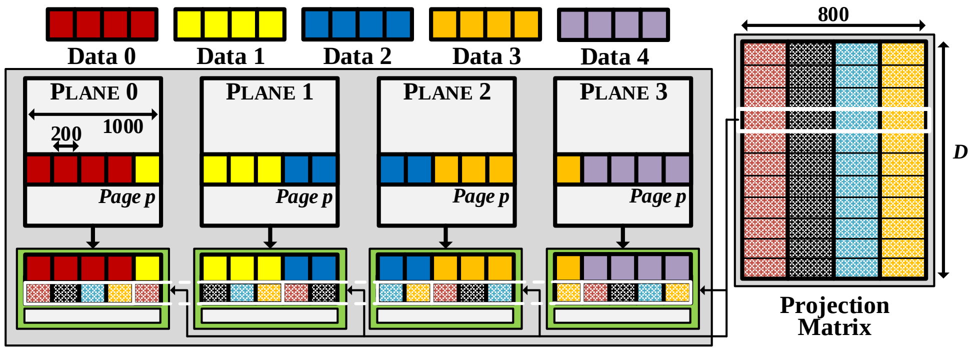


Figure 3.3. Data storage scheme in Store-n-Learn and the corresponding segmentation of the projection matrix. Data represents a feature vector.

where c is the number of complete n-sized feature vectors in a p_s-sized page, n_p is the number of planes per chip, and $d \in \{0, 1, ...n_p\}$. Hence, a page would contain $c \times n_p + d$ segments in total. Having n_p equal segments instead of any variable segmentation allows the accelerator to have a simple segment-wise weight allocation. Each row-vector in the projection matrix of a plane is divided into the same sized segments as the feature vector as shown in Figure 3.3 on the right. This allows Store-n-Learn to increase storage efficiency while minimizing the control overhead of the accelerator.

If the size of the feature vector is less than the page size, Store-n-Learn uses the same segmentation size. However, the number of segments in a page are given by:

$$\underset{d}{\operatorname{argmax}} \ (d.n/n_p \leq p_s)$$

A drawback of this scheme is that individual reads for small feature vectors may require reading two pages instead of one. However, our main purpose is to obtain trained vectors and not raw feature vector values. Moreover, since a feature split across two planes shares the same block and page number, they are both read at the same time.

Encoding on Store-n-Learn Flash Chips:

While the new data storing scheme improves the page utilization, it does not suit well the chip-level accelerator. As proposed before, our accelerator is a dot product engine. It processes an entire page from the flash array to generate values of different dimensions of the corresponding hypervector. In the new data storage scheme, this would result in an encoded hypervector consisting of multiple and also partial feature vectors. An easy fix would be to just process one feature vector at a time by setting the remaining inputs of the accelerator to 0. However, this would increase the total latency of the accelerator. The situation is worse if the size of feature vectors is very small. We address this problem by extending the concept of batching in Store-n-Learn.

As detailed in Section 3.3.1, a set of encoded hypervectors can be added dimension-wise without interfering with HDC training process as long as they belong to the same final trained hypervector, for example the same class model. An encoded dimension (d_i) of a feature vector (FV) is obtained by a dot product between the feature values (FV_i) and the corresponding row of the projection matrix (PM), i.e.,

$$d_i = FV_0 \times PM_{i,0} + FV_1 \times PM_{i,1} + ...FV_{n-1} \times PM_{i,(n-1)}$$

Now, to add multiple FVs together, we just need to make sure that an element in a FV is being multiplied with the corresponding weight of the PM. In that case, we would achieve the same effect as batching, only at a lower level of abstraction. This also works when we have partial features. In this case, the encoded hypervector for the current page would just have partial information and may not correctly represent the data. Some part of this information is contained in the encoded hypervector of another page. However, all these hypervectors will be added together during training. Hence, the final hypervector will contain all the information.

To support this strategy in Store-n-Learn accelerator, the flash controller segments the projection matrix in the same way as the feature vectors in the planes and sends the corresponding segments to the accelerator in each plane. It is important to note that only the features belonging to the same class are added together in batches. So, a chip-level accelerator performs a bitwise comparison between the labels of feature vectors in a page and only processes those belonging to the same final model together.

Encoding without Batching:

The encoding acceleration discussed above works well while training class hypervectors for classification because training inputs samples can be batched together. However, other tasks like clustering, retraining, and inference operate on individual data samples. Hence, they cannot utilize batched hypervectors and require access to individual ones.

As discussed before, encoding individual data points is slow and doesn't fully utilize the adder tree present in the encoding accelerator. Hence, unlike batched encoding where we could get away with generating just one dimension per iteration of the accelerator, here we need to generate multiple dimensions in parallel. Since each dimension is independent, one way to improve the latency of encoding individual vectors would be to introduce multiple adder trees, each computing one dimension. However, this would linearly increase the power and area of the accelerator. Moreover, the optimal size and number of trees would differ for each application.

Instead, we preserve the current single adder tree and introduce carry look ahead adders (CLAs) at intermediate stages as shown in Figure 3.4. The figure shows only a part of our complete 20-stage adder tree. Our 32-bit CLA implementation has a latency similar to four sequential carry save additions, i.e. four stages of the carry save adder tree (CSA). Hence, we generate our tree using four-stage CSAs. We also add CLAs after every four stages, as shown with blue and purple boxes in Figure 3.4. For example, a 16 (20) stage CSA consists of 113 (455) smaller and independent 4-stage, 24 (97) 8-stage, 4 (19) 12-stage, and 1 (3) 16-stage CSAs. We insert a 32-bit CLA for each of these independent CSA. This results in a total of 141 (574) intermediate 32-bit CLAs. Each of these CLA-enabled independent trees can generate one output (dimension) each. Hence, in the case when the size of feature vector is significantly

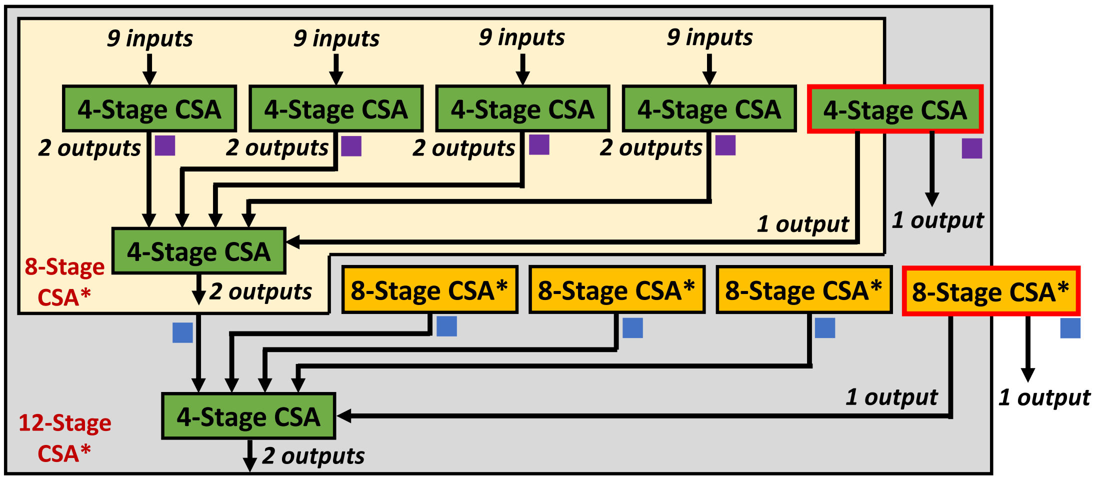


Figure 3.4. Modified CSA in die-accelerator to encode individual feature vectors in Store-n-Learn. The blue and purple squares represent intermediate CLAs.

smaller than the page-size and less than the input size of any of the CLA-enable smaller trees, these trees can generate a dimension each. So, if a feature vector has a size of, say 32 (8), we utilize the stage-8 (stage-4) CLAs, i.e. the blue (purple) boxes in Figure 3.4. For a 1024-input adder, we can generate 24 dimensions of the hypervector corresponding to this feature vector in parallel. To enable this, the feature vector is input to each smaller CSA. Moreover, the projection submatrix corresponding to the 24 dimensions is flattened and supplied to the accelerator. These above modification allows us to generate multiple dimensions in parallel, significantly boosting the performance of single feature vector encoding.

3.3.3 Training at Top-Level

The encoded hypervectors from flash chips are used for training in the top-level accelerator, which is implemented on an FPGA present in the SSD. We use FPGA because it is flexible with the application parameters and can be configured using the primitives provided by INSIDER [1]. The encoded hypervectors come with class labels. During training, they are accumulated into the corresponding class (or model) hypervectors. At the end of training we obtain an output hypervector for each class that in turn represents all the input samples belonging to that class.

In the FPGA, we first allocate memory for the final class hypervectors. For each class, the FPGA has an input queue, where the input hypervectors belonging to that class are indexed, and an accumulator, which serially accumulates the vectors in the input queue to generate the final class hypervector. The class label of an incoming hypervector is used to index it to the corresponding class input queue. The size of the queue is determined based on the frequency of the inputs, the number of classes, and dimensionality D. The introduction of class-wise input queues removes the input data dependency of the accumulator by pre-processing class labels. An accumulator simply needs to read the input index from its queue and operate on the corresponding data. It makes the computation for different classes independent and parallelizable. The accumulators for each class then operate in parallel to add an input hypervector from the queue to the corresponding class hypervector. While the computation can also be fully parallelized over all dimension, the large size of hypervectors and the limited read ports of the memory make it impractical. Hence, we divide the hypervectors into partitions to allow partial parallelism. The final class hypervectors are sent to the host.

If an application has too many classes or requires extremely large number of dimensions, then the FPGA may not have enough space to store all the class hypervectors. In such a case, we allocate the memory for the maximum number of class hypervectors, C_{max}. We assign labels to these classes with respect to the incoming hypervectors. Hence, the first set of incoming hypervectors belonging to C_{max} different classes are processed as before. We introduce an addition queue that indexes, along with their labels, the incoming hypervectors not associated with any of the active C_{max} class. Whenever the queue is full, one of the C_{max} class hypervectors is sent to the CPU host. The corresponding memory is allocated for the class to which the first hypervector in the queue belongs. The class hypervector sent to the host is the one that has accumulated the most incoming hypervectors.

3.3.4 Retraining and Inference at Top-Level

The initial training iteration builds the HD model, a class hypervector for each class. However, to fine-tune the HD model and increase accuracy, multiple retraining iterations may be needed. As explained in subsection 3.2.3, the retraining step consists of reading the encoded hypervectors from the storage, performing the inference, comparing the classification output with the data label, and adjusting the HD model in case of misprediction. To adjust the model, the encoded hypervector is added to the class it belongs to and is subtracted from the mispredicted class hypervector. Store-n-Learn supports inference and retraining on FPGA by leveraging the flash chips that encode data into hypervectors. Previous works perform data encoding on the FPGA to avoid storing the encoded data which requires more storage space. Supporting HD encoding on FPGA consumes a lot of FPGA resources, thus limits the performance of the accelerator. Store-n-Learn uses Flash chips to encoded the data in real-time that saturates the internal SSD bandwidth. Thus, Store-n-Learn dedicates all the FPGA resources for HD training, inference, and retraining, thereby providing higher performance as conventional FPGA-based accelerators.

Figure 3.5(a) shows the architecture of the Store-n-Learn FPGA-based accelerator for HD training, retraining and inference. The encoded hypervector is read from the storage device and stored into the “encoded hypervector” buffer. During HD inference, Store-n-Learn in every clock cycle reads d dimensions of the encoded hypervector and since HD operations can be parallelized in the dimension level, it calculates the partial similarity metric between the dimensions of the encoded hypervector and corresponding dimensions of the class hypervectors. In HD inference, in every clock cycle, Store-n-Learn calculates the similarity metric between d dimensions of the encoded hypervector and d dimensions of C class hypervectors. For each class, Store-n-Learn performs d multiplications, and accumulates the multiplication results in a tree adder with d inputs. At the end, the class with the maximum similarity is the inference result. In each cycle, Store-n-Learn calculates a part of similarity metric and the entire inference is executed in $\frac{D}{d}$ cycles. d directly affects the required resources for training, retraining, and inference of HD. On the other hand, Store-n-Learn reads the encoded hypervector from the storage device. Hence, the value of d depends on the available resources on the FPGA and the SSD-to-FPGA bandwidth.

To perform HD retraining, Store-n-Learn first performs HD inference, and compares the

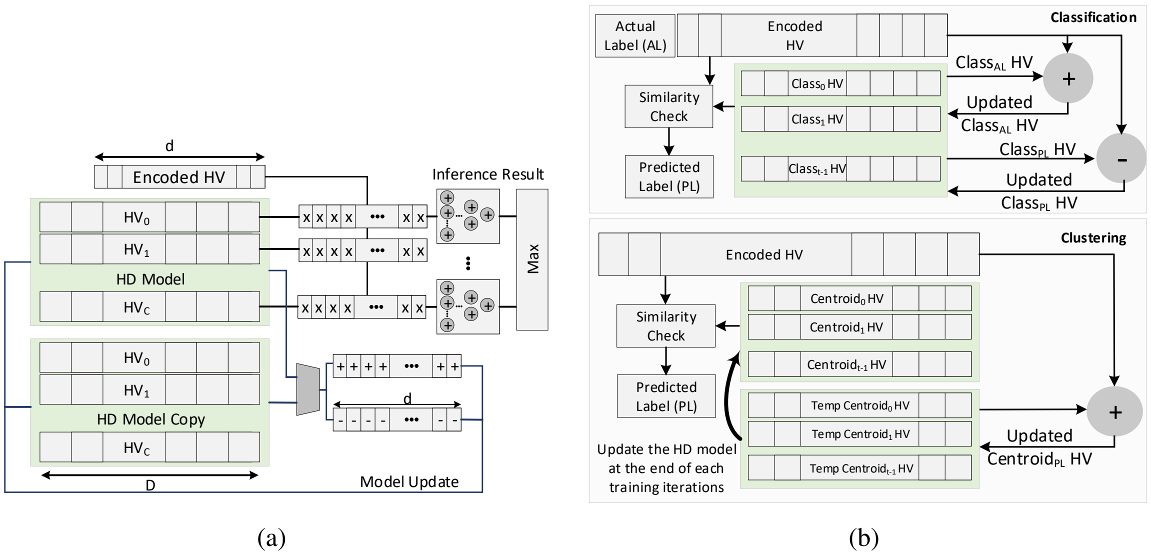


Figure 3.5. Store-n-Learn top-level FPGA design. (a) Retraining and inference for HD classification and (b) HD clustering as compared to the retraining step in HD classification.

prediction label with the original data label. In case of misprediction, it adjust the HD model by subtracting the encoded hypervector from the mispredicted class and adding it to the actual class. As illustrated in Figure 3.5(a), for each misprediction, one addition and one subtraction is needed. Since during the retraining stage, entire encoded hypervector is needed, Store-n-Learn locally stores it on FPGA BRAMs. If Store-n-Learn predicts the label correctly, it reads the next encoded input; otherwise, it performs the model adjustment in $\frac{D}{d}$ cycles. In each cycle d dimensions of the encoded hypervector is added to the class hypervector with actual label and subtracted from the predicted class.

3.3.5 Clustering at Top-Level

Store-n-Learn supports HD clustering, consisting of multiple clustering iterations on FPGA. As explained in subsection 3.2.5, in each clustering iteration, HD uses the existing centroids to clusters the input data, and uses the clustered data, at the end of iteration, to update the cluster centroids. Multiple clustering iterations are required for HD clustering model to converge. Store-n-Learn proposes a novel FPGA-based accelerator for HD clustering. During HD clustering, similar to HD classification, Store-n-Learn reads the encoded hypervector from the storage device. It initializes the clustering HD model with randomly selecting an encoded input hypervector for each cluster hypervector. Then it reads d dimensions of the encoded hypervector in every clock cycle and calculates the similarity metric between the encoded hypervector and cluster hypervectors. Store-n-Learn assigns the cluster with the highest similarity to the input. Store-n-Learn uses the average of the encoded hypervectors assigned to a cluster as the cluster centroid, giving us the initial clustering model. This is followed by multiple clustering iterations. For each iteration, Store-n-Learn uses a copy of the latest HD clustering model to update the centroids. In an iteration, Store-n-Learn uses the clustering model to perform similarity check for each encoded input and assigns a cluster to it. Then, the corresponding input hypervector is added to the predicted cluster centroid of the iteration's copy of the HD clustering model. At the end of each clustering iteration, i.e. after processing all the input data, the HD clustering model is replaced by the updated copy of the HD clustering model.

Figure 3.5(b) highlights the differences between Store-n-Learn HD retraining and HD clustering. To support HD clustering, along with HD classification, Store-n-Learn reuses the similarity check module of HD classification to find the predicted cluster; it additionally, needs a copy of the HD model to update the centroids. As illustrated in the figure, Store-n-Learn requires a duplicate of the HD model memory to support HD clustering and it reuses the adder array to update the cluster centroids. Therefore, Store-n-Learn supports HD clustering with double BRAM utilization and with minimal logic overhead, only for generating related control signals. In each cycle, similar to classification inference, the similarities between the encoded hypervector and the clustering centroid hypervectors are calculated. Finding the closest cluster takes $\frac{D}{d}$ cycles. Then Store-n-Learn uses the predicted clusters, to update the centroids. Updating the centroids, takes another $\frac{D}{d}$ cycles. Hence, clustering each input, including the centroid updating, takes a total of $\frac{2\times D}{d}$ cycles.

3.3.6 Software Support

Store-n-Learn derives its base system-architecture from INSIDER [1]. The INSIDER framework is an API which, while being compatible with POSIX, allows us to implement an ISC accelerator cluster. The INSIDER API takes a C++ or RTL code as an input and programs the acceleration cluster (running on drive FPGA) accordingly. The drive program interface has three FIFOs. The data input (output) FIFO takes in the input (output) data that is needed (generated) by the accelerator. The parameter FIFO contains the runtime parameters for the FPGA which are sent by the host. INSIDER keeps control and data planes of ISC separated. The drive control and standard operations are handled by the SSD firmware while all compute data from flash chips are intercepted by the top level FPGA accelerator for computing. The FPGA doesn't care about the source and/or destination of the data.

Store-n-Learn Host-Side Support:

INSIDER API uses POSIX-like I/O functionaly to communicate with the driver. INSIDER has a standard block device driver with changes made to the virtual read and write functionalities to accommodate for the programmable accelerator clusters in the drive. However, the current abstraction allow us to pass directive/parameters only to the ISC FPGA and not the drive. We define a new API, `send_mode`, which defines the mode for read and write operations, further discussed in Section 3.3.6. It passes a single integer, *mode*, to the drive firmware while opening a virtual file. For a non-ISC read/write from the drive, *mode* is set to '0.' During an ISC read, *mode* represents the $expansion\ factor\ (EF)$. EF defines the increase in the size of raw data after encoding. For example, $EF = 5$ means that each page of raw data generates five pages of encoded data (due to large D). In this case, *mode* is set to 5. This parameter is necessary to enable the drive to read the required number of pages from the flash chips. Since EF is dependent on the number of features of the data and dimensionality requirement of the application, it remains constant for an entire run. Similarly, a non-zero *mode* signifies ISC write. In this case, the data being sent to the drive contains the elements of the HDC projection matrix and is written to the controller scratchpad. No data is written to the flash chips. During write we only care about whether *mode* is zero or non-zero.

Store-n-Learn Drive-Side Architecture:

Store-n-Learn implements its top-level accelerator described in Section 3.3.3 as an INSIDER acceleration cluster, which enables the final training step. However, INSIDER system doesn't support Store-n-Learn's die-level acceleration because the standard read/write drive operations can't readily accommodate on-the-fly change in data size while reading encoded pages and writing projection matrix elements to the on-die accelerator.

Store-n-Learn introduces the processing capability between flash planes and page buffers but sometimes only raw data may be required. Hence, Store-n-Learn employs two read modes, normal and compute. It uses the die-level accelerator in multiplexed mode where a read page is sent to the accelerator for processing only in compute mode, shown in Figure 3.6. In normal mode, the plane directly writes the original page to the page buffer. Moreover, response type in the two modes also differs. A normal read results in just one page while a compute read responds with multiple but fixed number of pages. Store-n-Learn uses application specifications such as feature vector size and dimensionality requirement to generate an expansion factor, which is supplied to the SSD firmware by the host, as explained in Section 3.3.6. The firmware uses this factor to calculate the response size for page read commands in compute mode.

Store-n-Learn also employs two write modes, normal and compute. The compute mode is used to supply projection matrix data to the on-die accelerators. In normal mode, data is written in the data buffer and then programmed in the flash array. In compute mode, the data in data buffer is sent to the accelerators as shown in Figure 3.6. The writes in compute mode are fast since the data is just latched in CMOS registers instead of flash arrays. Unlike a compute mode read, where the same command can be issued to all the chips, compute mode write requires individual commands for each plane to configure their respective on-die accelerators. This follows from Figure 3.3. Each plane gets the same segments but their positions may differ for

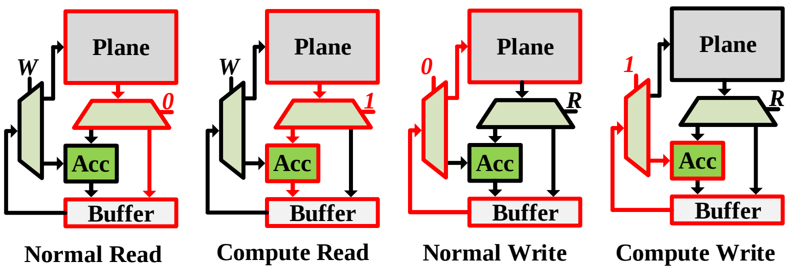


Figure 3.6. Different read and write modes in Store-n-Learn. The components in red are active during the operation.

different planes. A write configuration command is separately issued for each plane. For each plane, it configures the size of segment (seg_S), number of input segments (seg_{in}), actual number of segments in the plane (seg_{act}), and the ID of the first segment (seg_{one}). The format of the command is $[seg_S, seg_{in}, seg_{act}, seg_{one}]$. For example, the command for plane 0 and plane 2 in Figure 3.3 would be $[200,4,5,0]$ and $[200,4,5,2]$ respectively. While sequential, this step has negligible latency overhead because it can be performed in parallel for all the flash chips.

As discussed briefly in Section 3.3.2, the flash controller sends the projection matrix elements to the respective accelerators. SSD receives the projection matrix from host. We introduce a dedicated scratchpad in the flash controller to store the matrix. The controller sends the elements in page-sized frames to the die accelerators. The frames consist of multiple segments and are used by the die-accelerators according to the configuration command, as shown in Figure 3.3.

3.4 Results

3.4.1 Experimental Setup

We developed a simulator for Store-n-Learn which supports parallel read and write accesses to the flash chips. We utilized Verilog and Synopsys *Design Compiler* to implement and synthesize our die-level accelerator at 45nm and scale it down to 22nm. The top-level FPGA accelerator has been synthesized and simulated in Xilinx Vivado. For Store-n-Learn drive simulation, we assume the characteristics similar to 1TB Intel DC P4500 PCIe-3.1 SSD connected to an Intel(R) Xeon(R) CPU E5-2640 v3 host. The parameters for Store-n-Learn are shown in Table 3.1.

We compare Store-n-Learn with 7th Gen 2.4GHz Kaby Lake Intel Core i5 CPU with 8MB RAM and 256 GB SSD. We also compare it with a 3.5GHz Intel(R) Xeon(R) CPU E5-2640 v3 CPU server with 256GB RAM and 2TB local disk. We also compare Store-n-Learn with INSIDER [1] and DeepStore [2], the state-of-the art ISC solutions. INSIDER is a full-stack storage system and uses a top-level FPGA accelerator in the drive for ISC. DeepStore is an ISC implementation for query-based workloads which employs specialized accelerators in SSD. For all our experiments, including those for other ISC solutions, the data is assumed to be channel-striped and stored using Store-n-Learn's proposed scheme.

3.4.2 Classification Workloads

We evaluate the efficiency of Store-n-Learn on five popular classification applications, as listed below:

Speech Recognition (`ISOLET`): The goal is to recognize voice audio of the 26 letters of the English alphabet [55].

Face Recognition (`FACE`): We exploit Caltech dataset of 10,000 web faces [56]. Negative training images, i.e., non-face images, are selected from CIFAR-100 and Pascal VOS 2012 datasets [64].

Activity Recognition (`UCIHAR`): The dataset includes signals collected from motion sensors for 8 subjects performing 19 different activities [57].

Medical Diagnosis (`CARDIO`): This dataset provides medical diagnosis based on cardiotocography information about each patient [76].

Gesture Recognition (`EMG`): The dataset contains EMG readings for five different hand gestures [77].

Table 3.1. Store-n-Learn Parameters

Capacity	$1TB$	**Channels**	32
Page Size	$16KB$	**Chips/Channel**	4
External BW	$3.2GBps$	**Planes/Chip**	8
BW/Channel	$800MBps$	**Blocks/Plane**	512
Flash Latency	$53us$	**Pages/Block**	128
FPGA	$XCKU025$	**Scratchpad Size**	$4MB$
Avg Power/DA	$8mW$	**DA Latency**	$1.02ns$

*DA: Die-accelerator

3.4.3 Clustering Workloads

We evaluate Store-n-Learn on FCPS, the fundamental clustering problem suite [78], which has been widely used in the literature. We also evaluate HD clustering on the pattern recognition dataset [79]. The specific datasets used are:

FCPS Hepta [78]: 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 [78]: The Tetra data set consists of 400 data points in four clusters that have large intra-cluster distances. The clusters are nearly touching each other, resulting in low inter-cluster distances.

FCPS TwoDiamonds [78]: 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.

FCPS WingNut [78]: The Wing Nut dataset 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.

Pattern Recognition (Iris) [79]: The data set consists of samples from each of three species of Iris flower with four features are present from each sample. One class is linearly separable from the other 2; the latter are not linearly separable from each other.

3.4.4 Comparison with CPU and CPU Server

We first compare Store-n-Learn with CPU and CPU-based server running state-of-the-art implementations of HDC classification and clustering over the five datasets with $D = 10k$. In addition, we generate a synthetic dataset with 10 classes and each data sample having 512 features. We vary the size DS (number of data points) of the synthetic dataset from 10^3 to 10^7.

HDC Classification with Single-pass Training: The runtime of single-pass classification for different platforms is shown in Figure 3.7. We observe that Store-n-Learn is on average $3405\times$ and $1612\times$ faster than CPU and CPU-server, respectively. Our evaluations show that the improvements from Store-n-Learn increases linearly with an increase in the dataset size. This happens because more data samples result in more huge hypervectors to generate and process. In conventional systems, this translates to a huge amount of data transfers between the core and memory. It should be noted that the CPU system runs out of memory while encoding for 10^6 samples and kills the process. The CPU server faces a similar situation for 10^7 samples. In contrast, since Store-n-Learn generates hypervectors (encoding) while reading data out of the slow flash arrays and processes (training) them on the disk itself, there is minimal data movement involved.

Figure 3.7 also shows the size of raw input data in each case normalized to the size of the corresponding trained class hypervectors. While Store-n-Learn only sends class hypervectors from drive to the host, CPU-based systems fetch all data samples from the disk. We observe that the ratio increases linearly with an increase in the data size. In fact, the size of class hypervectors does not change with an increase in data size as long as the number of classes and required dimensions remain the same.

HDC Classification with Retraining: Figure 3.8 shows the runtime of HD classification with 50 epochs of retraining for different platforms. We observe that Store-n-Learn is on average $222\times$ and $81\times$ faster than CPU and CPU-server, respectively. The improvements are lower than those in case of single-pass classification because now FPGA-based retraining, specifically the search component of retraining, is the major latency bottleneck. Our evaluations also show that the performance of Store-n-Learn classification with retraining increases with an increase in either the dataset size or the number of classes. In addition to processing more hypervectors for a larger dataset, more classes increase total number of the latency critical search operations. Store-n-Learn is able to process much larger datasets than CPU and CPU-server, both of which run out of memory while working with 10^6 data samples. The trend for total SSD to host data transfers remains similar to that of single-pass training, where the amount of data transfers saved increases linearly with an increase in the data size.

HDC Clustering: Figure 3.9 shows the runtime of HDC with 50 epochs of clustering for different platforms. We observe that Store-n-Learn is on average 543× and 187× faster than CPU and CPU-server, respectively. Moreover, the latency of Store-n-Learn clustering increases with both dataset size and the number of classes. However, the relative improvements from Store-n-Learn also increase with an increase in dataset size. Store-n-Learn is able to process much larger datasets than CPU and CPU server, both of which run out of memory while clustering 10^6 data samples.

The amount of data transfers saved increases linearly with an increase in the data size. For very small clustering datasets [78, 79], transferring hypervectors of cluster centers instead of raw data increases the data transfers between SSD and host. However, data transfers become a system bottleneck for large datasets, in which case Store-n-Learn significantly reduces the total transfers. For example, Store-n-Learn transfers ˜5000× less data as compared to CPU-based systems for the synthetic dataset with one million samples.

3.4.5 Store-n-Learn Efficiency

Figure 3.10 shows the breakdown of Store-n-Learn single-pass classification latency normalized to the total latency. Here I/O shows the time spent in sending the generated class hypervectors to the host. For small datasets, `CARDIO` and `EMG`, the latency is dominated by the encoding. However, as the data size increases, the internal SSD channel bandwidth becomes a

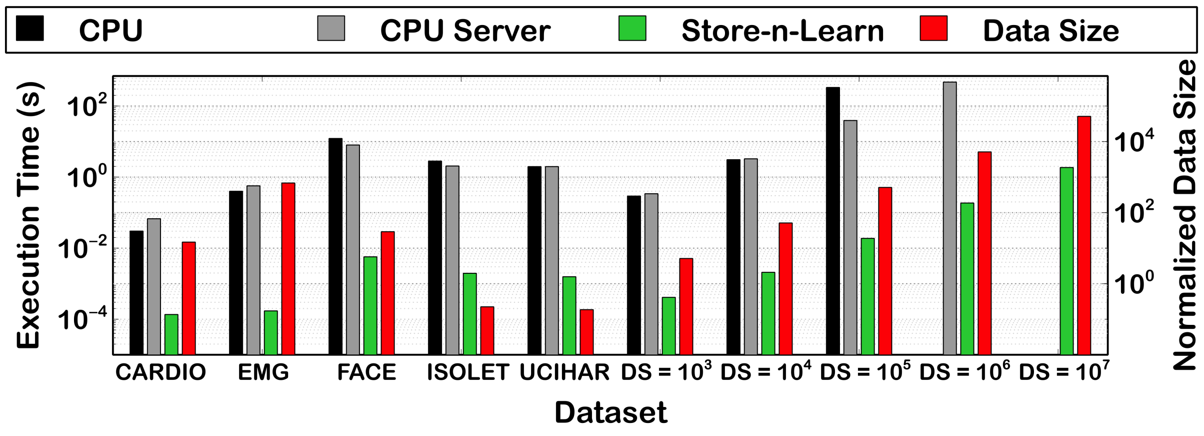


Figure 3.7. Runtime comparison of HDC encoding and single-pass classification training in Store-n-Learn with other platforms. The bars in red shows the size of raw data normalized to the total size of corresponding class hypervectors in Store-n-Learn.

bottleneck. This indicates that Store-n-Learn is able to completely utilize and saturate the huge internal SSD bandwidth. In addition, a significant amount of time spent in training and some part of the encoding is hidden by the SSD channel latency. As a result, the combined latency is less than sum of the latency for individual stages. For the example of `FACE` dataset, even though the training takes more than half of the total latency, a negligible portion of it actually contributes to the overall latency. It shows that Store-n-Learn stages are able to hide some of their latency. However, in the case of HD retraining and clustering, top-level FPGA accelerator becomes the latency bottleneck. This happens due to the iterative nature of these algorithms.

To demonstrate the scalability provided by Store-n-Learn, we evaluate it over a synthetic dataset with 10^4 samples each with 512 features. We vary the dimensions D from 10^3 to 10^5. Figure 3.11a shows that the latency of Store-n-Learn increases linearly with an increase in the number of dimensions, showing that Store-n-Learn is able to scale with D. Additionally, an increase in D results in longer class hypervectors for the same input data. Hence, the ratio of raw data to hypervector size decreases with an increase in dimensions, falling from from 512 for $D = 1k$ to 2.5 for $D = 10^5$.

We also scale the dataset with the number of class, while keeping its size fixed to 10^4

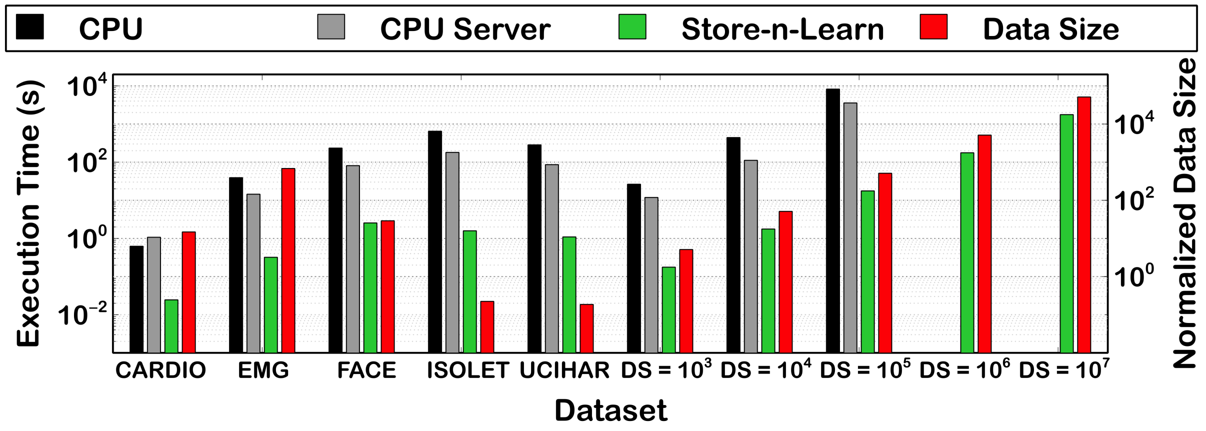


Figure 3.8. Runtime comparison of HDC classification with retraining in Store-n-Learn with other platforms. The bars in red shows the size of raw data normalized to the total size of corresponding class hypervectors in Store-n-Learn.

samples and D as 10^3. Figure 3.11b shows that the Store-n-Learn latency has minor changes with the number of classes when we have less than 50 classes. This is because our FPGA has enough resources to train up to 54 classes with $D = 10k$ dimensions. The latency almost doubles for 100 classes. However, when number of classes increases further, the size of model hypervectors is too large to store in the FPGA. Hence, partially trained hypervectors are then sent to the host for further processing. This can be seen by a jump in the latency for 500 classes in Figure 3.11b. In addition to the time spent in training, transferring the class hypervectors to host creates a major bottleneck. This is also evident from the data size ratio which declines for large number of classes. A ratio of less than 1 signifies that the size of generated hypervectors is larger than the raw data.

3.4.6 Store-n-Learn vs Other Algorithms

We compare Store-n-Learn with the best existing algorithms for classification and clustering. For classification, we compare our work with the state-of-the-art DNN network for ISOLET [80]. In our evaluation, Store-n-Learn runs HDC classification with 50 epochs of retraining while DNN is trained on the CPU. We observe that Store-n-Learn is 9.4× faster than DNN while incurring less than 1% accuracy loss. We also compare our design with DNN running on FPGA.

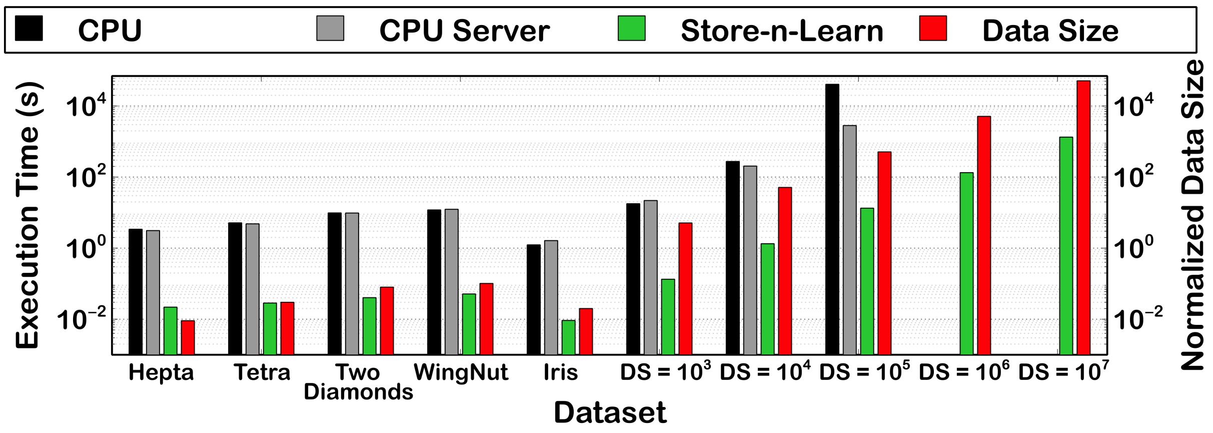


Figure 3.9. Runtime comparison of HDC clustering in Store-n-Learn with other platforms. The bars in red shows the size of raw data normalized to the total size of corresponding cluster center hypervectors in Store-n-Learn.

No FPGA implementation completely trains DNNs due to the complexity of operations and lack of sufficient on-board resources. Hence, we compare the inference performance of our design with that of DNN running on FPGA [81]. Store-n-Learn is 17.7× faster than FPGA for ISOLET dataset, with less than 1% accuracy loss.

For clustering, we compare Store-n-Learn with the k-means algorithm [82] for the five clustering datasets on CPU. Store-n-Learn runs HDC clustering with 50 epochs of clustering. The quality of clustering is measured in terms of mutual information score, which is one when the predicted labels are perfectly correlated with the ground truth and zero when they are totally uncorrelated. Store-n-Learn is on average 1.3× faster than k-means on CPU while providing the same mutual information score. We also compare Store-n-Learn clustering with k-means running on FPGA and observe that Store-n-Learn is 47× faster. Store-n-Learn is faster than the state-of-the-art algorithms for both classification and clustering due to the latency overhead of data transfers in traditional systems. Moreover, the higher complexity of operations in traditional algorithms further makes them slower on FPGA.

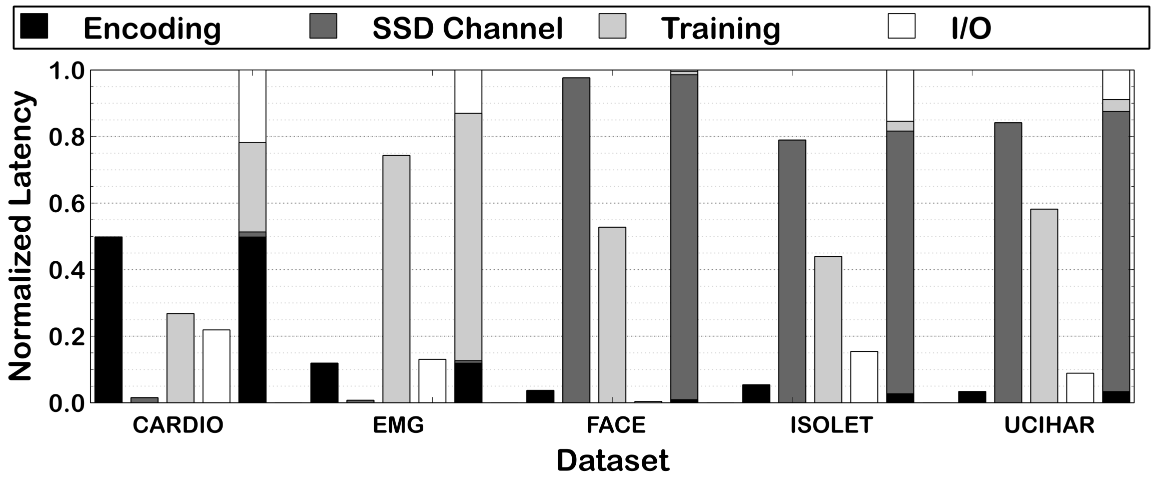


Figure 3.10. Breakdown of latency of different stages of HDC single-pass classification normalized to the total latency.

3.4.7 Comparison with Existing ISC Solutions

We compare the performance and data transfer efficiency of Store-n-Learn with state-of-the-art ISC designs INSIDER [1] and DeepStore [2]. In our experiments, INSIDER performs both encoding and training/clustering using the FPGA accelerator in SSD and sends the class hypervectors to the host. Since DeepStore was intended for a completely different application, we replace its accelerator with Store-n-Learn die-level accelerator. During ISC, Deepstore encodes the raw data into hypervectors and sends those hypervectors to the host for training/clustering.

HDC Single-Pass Classification: Figure 3.12 shows the change in latency and data transfer size of single-pass classification for the three ISC solutions. We observe that Store-n-Learn is on an average 14.4× and 446.8× faster than INSIDER and DeepStore, respectively. While encoding in DeepStore takes the same time as Store-n-Learn, transferring hypervector from SSD to host and further training on them on CPU increases the execution time of DeepStore significantly. On the other hand, the SSD channel bottleneck faced by Store-n-Learn is relaxed in case of INSIDER since it only transfers raw data. However, the FPGA-based HDC encoding+training are on an average 21× slower as compared to FPGA-based training. Also, since INSIDER performs training in SSD, it transfers the same amount of data to the host as

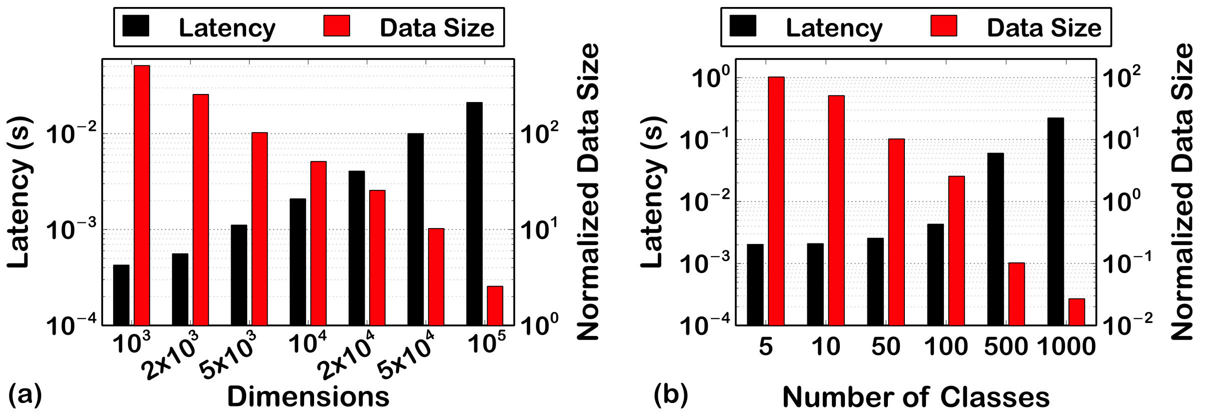


Figure 3.11. Change in HDC runtime and raw data size to hypervector ratio with (a) dimensions and (b) number of classes.

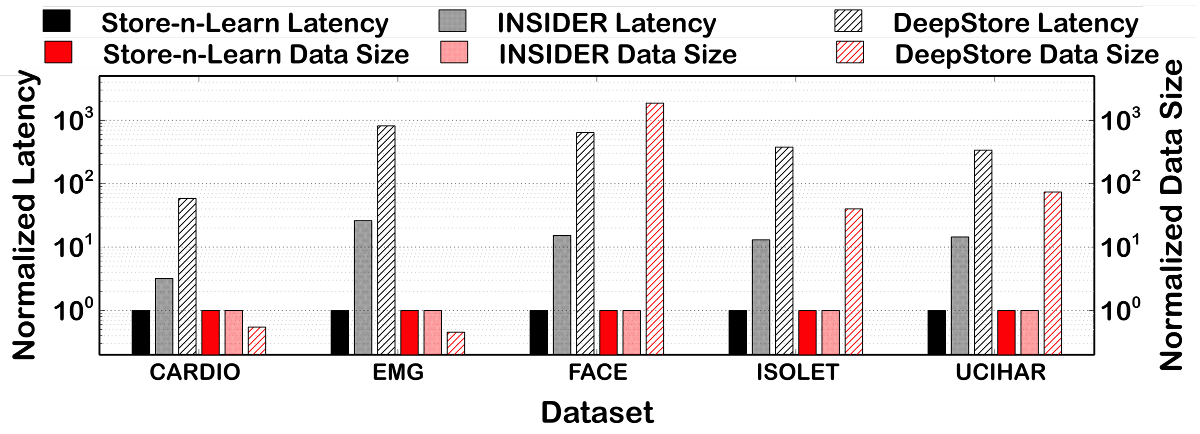


Figure 3.12. Runtime and data transfer size comparison of Store-n-Learn classification without retraining with INSIDER [1] and DeepStore [2]

Store-n-Learn. However, by transferring untrained hypervectors, DeepStore increases the amount of data transferred on an average by 397× as compared to Store-n-Learn.

HDC Classification with Retraining: Figure 3.13 shows the change in latency and data transfer size for complete HDC classification. We observe that Store-n-Learn is on an average 10.6× and 179× faster than INSIDER and DeepStore, respectively. For DeepStore, transferring hypervector from SSD to host and further retraining on them for 50 epochs on CPU increases the execution time of DeepStore significantly. INSIDER's FPGA-based HDC encoding and retraining are on an average 10.7× slower as compared to Store-n-Learn's FPGA-based

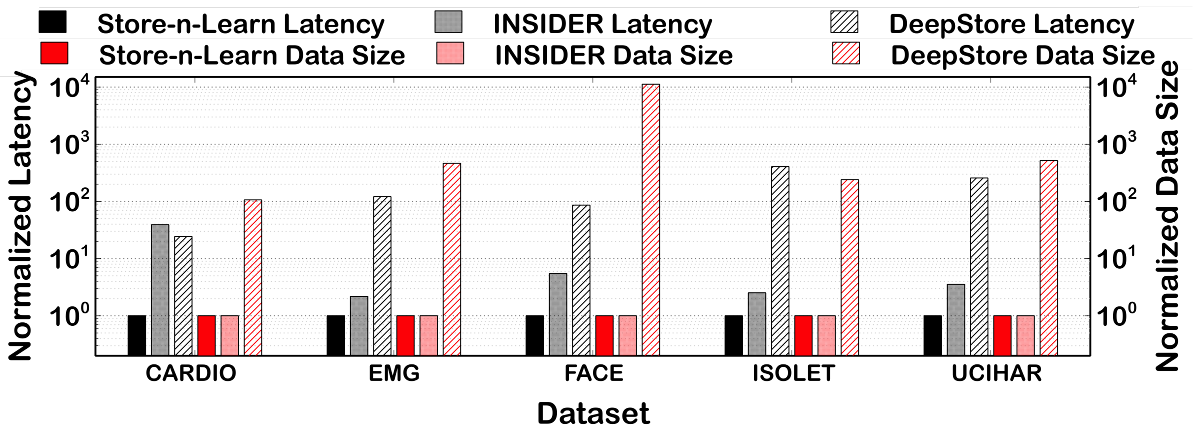


Figure 3.13. Runtime and data transfer size comparison of Store-n-Learn classification with retraining with INSIDER [1] and DeepStore [2]

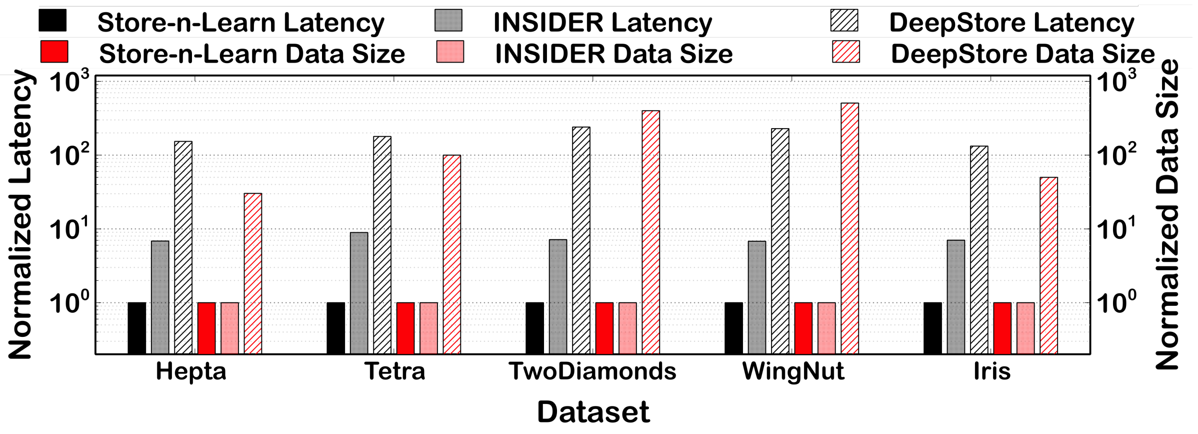


Figure 3.14. Runtime and data transfer size comparison of Store-n-Learn clustering with INSIDER [1] and DeepStore [2]

retraining because encoding consumes a significant amount of FPGA resources, leaving less resources to accelerate latency critical retraining. While INSIDER transfers the same amount of data to the host as Store-n-Learn, DeepStore increases the amount of data transferred on an average by 2510× as compared to Store-n-Learn. This is 6.3× worse than the data transferred in single-pass classification because in retraining hypervectors are sent for individual data points, eliminating the gains from batched encoding.

HDC Clustering: Figure 3.14 shows the change in latency and data transfer size for HDC clustering. We observe that Store-n-Learn is on an average 7.3× and 187× faster than

INSIDER and DeepStore, respectively. The latency results follow the same trends and reasoning as those for HDC classification with retraining. For data transfers, DeepStore increases the amount of data transferred on an average by 217$\times$ as compared to Store-n-Learn. The data transfer overhead of DeepStore worsens with an increase in the dataset size.

This and the previous chapters accelerated computations on raw data in memory and storage. The next chapter shows how similar computations can be performed in a server-client setup where the data is always encrypted.

Chapter 3, in part, is a reprint of the material as it appears in S. Gupta, J. Morris, M. Imani, R. Ramkumar, J. Yu, A. Tiwari, B. Aksanli, and T. Rosing, "THRIFTY: Training with Hyperdimensional Computing across Flash Hierarchy," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2020. The dissertation author was the primary investigator and author of this material.

Chapter 3, in part, is currently being prepared for submission for publication of the material. S. Gupta, B. Khaleghi, S. Salamat, J. Morris, R. Ramkumar, J. Yu, A. Tiwari, M. Imani, B. Aksanli, and T. Rosing, "Store-n-Learn: Classification and Clustering with Hyperdimensional Computing across Flash Hierarchy." The dissertation author was the primary investigator and author of this material.

Chapter 4

Secure and Privacy-Preserving Computing with Fully Homomorphic Encryption in Memory

Chapters 2 and 3 presented in-memory and in-storage HD computing accelerators. However, some critical healthcare, finance, or insurance applications may rely on cloud computing due to the complexity of algorithms, very large and dynamic computation/learning models, proprietary algorithms, or the need to simultaneously learn from multiple users [29, 30, 31, 32]. Since they deal with extremely sensitive information, most of the users' data is encrypted. The solutions proposed earlier cannot process such data without decrypting it first.

Fully homomorphic encryption (FHE) allows us to apply functions of arbitrary complexity on encrypted data (ciphertext) without the need to decrypt it. This eliminates the need for private key exchanges and decrypting data at the server, raising the bar on security and privacy. However, computing on encrypted data comes at a huge data and computation cost, resulting in large performance and memory overheads. For example, encrypting an integer in homomorphic domain may explode its size from meagre 4B to more than 20KB. Moreover, homomorphically multiplying two FHE encrypted integers may require 10s of millions of operations. Further, computing with encrypted data may limit the complexity of the function that can be evaluated for a set of encryption parameters. The work in Gentry [83] proposes a procedure, called bootstrapping, to reduce the growth of noise during function evaluation in FHE domain, allowing

FHE to perform more complex operations. However, it is extremely expensive and increases the latency of evaluating a homomorphic function by 100-1000$\times$. Recent proposals in [84, 85, 86] make bootstrapping faster and computationally less expensive. Unfortunately, bootstrapping still remains expensive and is the major limiting factor while using FHE to evaluate real workloads. The encryption keys used in such schemes may reach up to GBs in size, adding to the huge capacity and data transfer bottleneck of FHE.

The works in [87, 88, 89, 90, 91, 92] proposed CPU and GPU implementations of RGSW-based FHE schemes [84, 37, 93]. However, they cannot scale enough to provide the speedup needed to make FHE feasible. Most operations in these schemes are based on polynomials and vectors, which are difficult to accelerate due to the limited parallelism and data access provided by current systems. Other hardware-acceleration work in [94, 95, 96, 97] accelerate previous generation schemes which are not truly FHE and support limited functionality.

Processing in-memory is an excellent match for the FHE since it provides extensive parallelism, bit-level granularity, and an extensive library of compatible operations which dramatically improving both performance and energy efficiency [33, 34, 35, 36]. It addresses the issue of large data movement by processing data in memory where it is stored. We use Resistive RAM (RRAM) which has low energy requirements, high switching speed, is scalable, and compatible with the CMOS fabrication process.

In this chapter, we present the first latest generation end-to-end acceleration of FHE cryptosystem based on [37]. Unlike previous HE proposals, which supported a library of functions, the latest RGSW-based cryptosystem allows computing arbitrary functions on encrypted data. Our proposed MemFHE has two main components, the client and the server PIM accelerators. The client PIM accelerator runs ultra-efficient in-memory operations to not only encode and decode data but also enables ring learning with errors (RLWE) to encrypt and decrypt data. The encrypted data (ciphertext), along with an encrypted version of secret key, are sent to the server PIM accelerator for processing. Server PIM receives the ciphertext from multiple clients and performs operations on ciphertext to generate output. To enable this, server PIM uses

PIM-enabled bootstrapping which keeps the accumulated noise low so that the output ciphertext can be decrypted by the intended client. This ciphertext is sent back to the client. In MemFHE, only the client has the means to decrypt the output ciphertext and access the unencrypted data.

To summarize, our specific contributions are:

- We present the first end-to-end acceleration of fully homomorphic encryption in memory. Our design accelerates both the encryption/decryption and the full FHE computation pipelines. While individual PIM operations are slower than in CPU, MemFHE employs ciphertext-level and operation level parallelism combined with operation-level pipelining to achieve orders of magnitude of performance improvement over the traditional systems.

- Our server PIM design includes fast bootstrapping, key switching, and modulus switching in memory. It distributes the key memory units to reduce the instances of data contention. It sequentially processes different inputs in different pipeline stages for the best processing throughput.

- We accelerate the bottleneck process of bootstrapping by using a highly pipelined architecture. Our bootstrapping introduces parallel accumulation units, which supports two different types of bootstrapping techniques. We propose a novel implementation for the core bootstrapping operation, NTT. Unlike existing works, our NTT doesn't require any special interconnect structure. Moreover, it is flexible and can process many NTT stages without needing extra hardware.

- Our client PIM design includes encryption and decryption. MemFHE enables encryption efficiently in memory by exploiting bit-level access and accelerates dot product with a new in-memory implementation.

- We evaluate MemFHE for various security-levels and compare it with state-of-the-art CPU implementations for Ring-GSW based FHE. MemFHE is up to $20k\times$ faster than CPU for

FHE arithmetic operations and provides on average 2007$\times$ higher throughput than [3] while implementing neural networks with FHE.

4.1 Background and Motivation

4.1.1 FHE Schemes

Many fully homomorphic encryption schemes have been developed during the past decade. The first generation include the original design from [83] and its subsequent optimizations. However, they have limited homomorphic capacity due to rapid noise growth during evaluation, restricting the evaluation to few gates at a time. Second generation schemes reduce the noise growth from linear to logarithmic and are based on more standard hardness assumptions. However, they are slow, requiring minutes to perform simple gate operations (HElib-IBM [98]).

The third generation schemes use weaker hardness assumptions to minimize the bootstrapping time and provide slower noise growth [99, 84, 93]. The work in [37] presented a framework to enable fast bootstrapping for such schemes under different security assumptions. While being the most general, supporting arbitrary functions, allowing many bootstrapping iterations without the need to decrypt, and providing providing control over security-levels, these schemes bootstrap individual boolean gates. They may be slower overall when implementing multi-bit operations. Recent works [100, 101, 3] have shown efficient extension of these schemes for multi-bit operations. The work in this direction promise to deliver faster bootstrapping and better overall application latencies, while providing the ability to perform functions of arbitrary complexity in encrypted domain.

4.1.2 FHEW Cryptosystem

FHEW cryptosystem [37] is based on the latest generation of FHE schemes, namely FHEW [84] and TFHE [85], and evaluates logic functions on encrypted data, i.e. *ciphertexts*, by evaluating look-up tables (LUTs). This is a foundational work toward realizing the full potential of FHE with more efficient encryption (less data size explosion), and faster bootstrapping for the same level of security as the previous generation schemes. It operates at bit-level, where each data bit is encrypted into pair consisting of a polynomial and an integer using a secret key, s, with learning-with-error (LWE) scheme. The encryption is performed for given application parameters, q and n, where n is the degree of the polynomial. All operations and data are taken modulus q. The typical values of n and q, presented in Section 4.7, results in a bit of data being encrypted into a 0.5-1kb ciphertext. In some cases, FHEW further breaks the ciphertext integers (including each polynomial coefficient) into d_r numbers, each with base B_r, to control the growth rate of noise. This further increases the ciphertext size. FHEW operates on LWE-encrypted ciphertexts, utilizing two different encrypted versions of s, EK_B and EK_S. The encrypted keys may have memory footprint in GBs.

FHEW employs cyclotomic ring-based encryption technique, namely RGSW [99], to operate on the ciphertexts. For each function, like NOR or XOR, that should be applied on the input ciphertexts, FHEW stores a corresponding FHE function in the LUTs. For example, an AND operation between two bits in *plaintext*, translates to simple addition of their corresponding ciphertexts, followed by AND-specific coefficient mapping. This is followed by bootstrapping, which reduces the noise accumulated in the output ciphertext due to function implementation. If not bootstrapped, the output ciphertext may become undecryptable. Most operations in bootstrapping happen over the polynomial part of output ciphertext, using the encrypted version EK_B of s. The ciphertext undergoes several *accumulation* iterations during bootstrapping. Bootstrapping works on parameters with similar functionality as that of LWE encryption but have different values, namely N, Q, B_g, and d_g. Here, all operations in accumulation happen on integers that have each been decomposed into d_g digits with base B_g. The final accumulation output is a pair of polynomials of degree N and modulus Q. The final output ciphertext, with reduced noise, is *extracted* out of the accumulation output. It is further treated with EK_S encrypted version of s to convert it back to the original LWE-encrypted domain. This process is called key-switching. The key-switched ciphertext can then be decrypted to obtain the output bit.

Apart from the large memory requirements of different FHEW components, the iterative nature and high polynomial degrees of FHEW operations makes it a slow and a memory-intensive process. Most data operations in FHEW are applied over polynomials which have a large compute and memory transfer bottleneck [89]. Efficient polynomial multiplication converts the polynomial into the frequency domain with number theoretic transform (NTT). The digit-decomposed computations of FHEW (i.e. breaking integers into d_r or d_g digits), required back-and-forth polynomial conversions between normal (coefficient) and NTT domain. Cumulatively, these operations make the implementation of FHEW on CPUs/GPUs very slow. Moreover, the huge memory requirement of the third generation FHEW cryptosystem, restricts the development of an effective FPGA/ASIC implementations. In contrast, MemFHE presents the first memory-centric architecture for FHEW cryptosystem. While MemFHE benefits from the large memory density due to its memory-centric approach, processing in memory further enables efficient computations, extreme parallelism, and significantly reduced data movement.

4.1.3 Processing in Memory

Many PIM techniques using RRAM have been proposed recently which implement bitwise operations, arithmetic, and search operations in memory [33, 102, 59, 103, 104], with support for varying bit-widths and data types including binary, integer, fixed point, and floating point. They use the switching-based RRAM processing in memory logic, where operations are governed by the voltage applied at the memory bitlines [33, 59]. The work in [59, 40] implement addition and multiplication using the bitwise operations. A b-bit addition is implemented with b serial 1-bit additions, which are further implemented with operations like AND, OR, and XOR. Where, a multiplication operation is implemented by first generating partial products using bitwise AND and then adding them using 1-bit additions.

4.2 MemFHE System Overview

MemFHE employs an end-to-end privacy-preserving computing system consisting of both client and server implementations. Our architecture is based on the FHEW cryptosystem

[37] which provides the slowest noise growth and hence is the most generally applicable class of FHE. MemFHE is implemented completely in memory, using homogeneous crossbar memory arrays and exploits processing in memory to implement all FHE operations.

All computations in the MemFHE-server happen in encrypted domain. It inputs the encrypted *ciphertexts* and performs the desired operations on the ciphertexts in the basic function unit, U_{FUNC}, without decrypting them. Computing in FHE domain leads to the accumulation of noise in the resultant ciphertext. To reduce this noise and keep it below the threshold, server utilizes the MemFHE-bootstrapping. Bootstrapping is the most important but also the slowest process in the MemFHE-server pipeline due to its iterative nature. Hence, we heavily pipeline bootstrapping architecture, so that the slowest operations in bootstrapping happens on different pipeline stages. We introduce novel architectures for various sub-components of bootstrapping and perform operation level optimizations in the bootstrapping core. As a result, MemFHE-server can achieve a high throughput of 170 inputs/ms even for high security parameters, which is 20k$\times$ higher than the latest CPU implementation [92].

In addition to the server, we also present MemFHE-client, which provides the input ciphertexts and receives the output of the server. The client is responsible for converting raw data into FHE domain, using a client-specific secret key. The client in FHEW cryptosystem encrypts a bit of data into an LWE ciphertext. MemFHE-client accelerates LWE utilizing efficient in-memory multiply-accumulation and shift operations. The encrypted ciphertext is sent to server along with an encrypted version of the client's secret key. Client also decrypts the output from the server to obtain the result of FHE computation in the plaintext form.

4.3 MemFHE-Server Architecture

Figure 4.1 shows an overview of the server's architecture. The goal of MemFHE's server is to provide a high throughput for operations on encrypted data. To achieve this, we create a deep pipeline. As discussed later and evaluated in experiments, bootstrapping is the major bottleneck

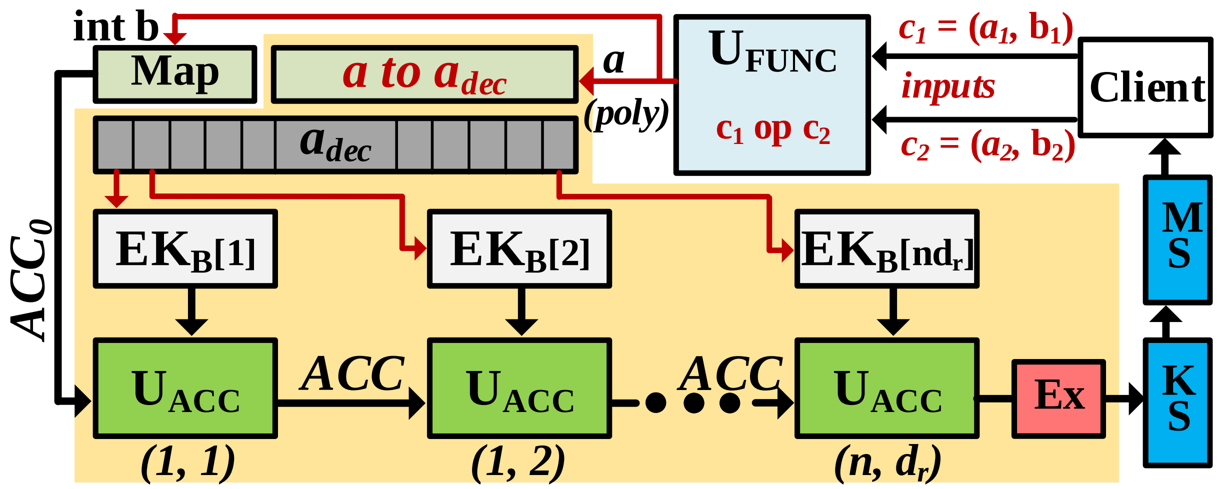


Figure 4.1. MemFHE Server Architecture

of the server-side computations. Hence, we use the latency of the slowest bootstrapping stage to set the maximum latency of any pipeline-stage in the server. We next present in-memory implementations of all the server components.

4.3.1 FHEW Function Implementation

The main strength of FHEW lies in its ability to implement arbitrary functions. FHEW achieves this by translating each boolean function into one or more homomorphic computation steps and then mapping the integer output to a bootstrapping-compatible polynomial, m_b. Each element of m_b is set to either $Q/8$ and $-Q/8$, the FHE equivalents of binary '1' and '0'. MemFHE allocates a memory block which stores these translations for all functions. Function implementation is the only process in MemFHE server that follows the client's parameters, n and q. FHEW uses polynomial addition, subtraction, and scaling by a constant as computing steps. For example, an AND between two bits is implemented by first homomorphically adding the corresponding ciphertexts (both the polynomial and the integer parts), followed by mapping the integer part of the output ciphertext to N-degree polynomial, m_b. Then, each coefficient of m_b in $[3q/8, 7q/8)$ is set to $Q/8$ and the others are set to $-Q/8$. A complete list of boolean gates and their corresponding FHEW translations are presented in [37]. MemFHE implements computation steps in a memory block, U_{FUNC}, executing polynomial additions and subtractions as described in Section 4.6. Scaling is performed using a series of shift-add operations. Since mapping happens within server's parameters, MemFHE performs it during the initialization stage of bootstrapping discussed in Section 4.4.1.

4.3.2 Bootstrapping

Implementing functions homomorphically in encrypted domain introduces noise in the ciphertext, which may make it impossible to decrypt the ciphertext. Bootstrapping reduces this accumulated noise. A majority of MemFHE's resources are dedicated to the bootstrapping core. MemFHE transfers the output of U_{FUNC} to bootstrapping. The initialization phase of bootstrapping coverts the output of U_{FUNC} into a server-compatible encryption and initializes a cryptographic accumulator, ACC. Then, bootstrapping utilizes a series of accumulation units, U_{ACC}, to modify the contents of ACC. The accumulation uses EK_B to "decrypt away" the accumulated noise from the output of U_{FUNC}. MemFHE supports two types of accumulation schemes, AP [105] and GINX [106]. While GINX is more efficient for binary- and ternary-distributed secret keys, AP is more efficient in other cases [37]. MemFHE chooses the accumulation scheme based on the client's encryption procedure. The output ciphertext with reduced-noise is then extracted from the ACC. Section 4.4 details the implementation of different bootstrapping steps in MemFHE.

4.3.3 Key Switching

Bootstrapping encrypts the output with a different key, EK_B instead of the original key s. Key switching is performed to obtain an output encrypted with s, so that it can be decrypted by the client. It utilizes the switching key, EK_S, which is sent by the client to the server along with the refreshing key, EK_B. As shown in [37], key switching uses a base B_s that breaks the integers into d_s digits. The N domain output of ACC gets converted to a client-compatible n. Key switching initializes a ciphertext, c_s, with an empty polynomial and the integer value of the extracted ACC. The ciphertext c_s has the parameters n and Q. Each coefficient of the ACC polynomial part, selects elements (n, Q ciphertext) from EK_S and then subtracts them from the existing value of c_s. This is repeated for d_s iterations. At the end of each iteration, the ACC polynomial coefficients are divided by the switching base B_s.

All operations in key switching are performed modulo Q. MemFHE first implements $(d_s - 1)$ divisions as shown in Figure 4.1. Since B_s is known, MemFHE pre-computes and stores the value of $1/B_s$. Division is now a multiplication with $1/B_s$. To prevent losing data due to rounding errors, the multiplication with $1/B_s$ is performed in full precision, generating twice the number of bits than needed. This happens in parallel for all the coefficients in a row-parallel way. This is followed by a modulo operation with B_s. Here we utilize in-memory Montgomery reduction (Section 4.6) to obtain the modulus of the divided coefficients. Now, we have $N \times (d_s - 1)$ coefficients, that select as many ciphertexts from EK_S, and perform sequential ciphertext subtractions. MemFHE employs a tree structure to subtract the ciphertexts. Each computing element of this tree is a memory block. Each blocks perform x sequential subtractions so that the total latency of these subtractions is less than the throughput of the design. Hence, we pipeline the tree stage-by-stage. It takes $\lceil log_2(N.(d_s - 1)/x) \rceil$ tree stages to implement all the subtractions. Each subtraction is followed by Barrett reduction (Section 4.6 with modulo Q. The final output of the tree, c_s, represents the key-switched output.

4.3.4 Modulus Switching

Lastly, the output of key switching is converted from a modulo Q ciphertext to a modulo q ciphertext. To achieve that, each element is multiplied with q and divided by Q and then rounded off to the nearest integer. MemFHE implements modulus switching in a single memory block. The key-switched ciphertext c_s, including its integer part, and is stored vertically in the memory block so that each coefficient is in a separate row. Similar to key switching, MemFHE prestores the value q/Q. All the ciphertext coefficients are hence multiplied with q/Q in a row parallel way. Then, a value of 0.5 is added to all the products in parallel using row-parallel addition as

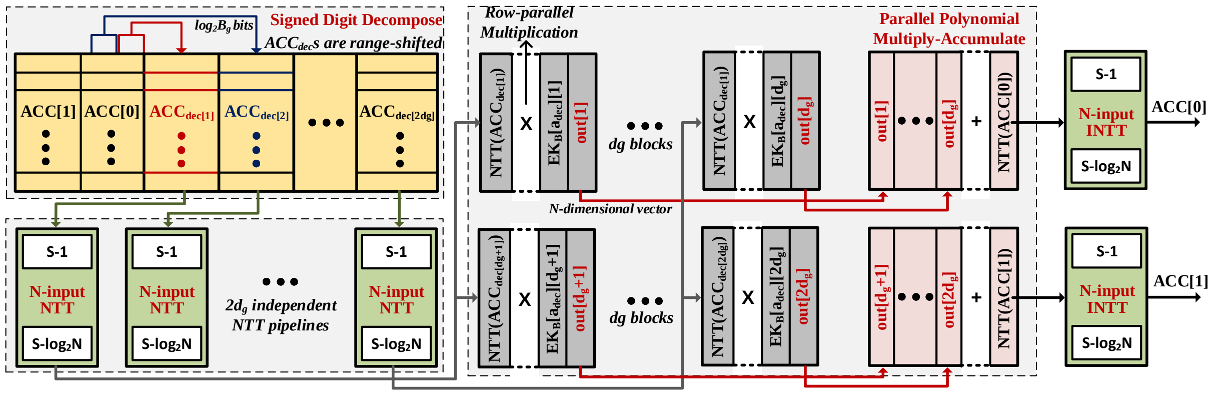


Figure 4.2. Accumulation Unit U_{ACC} of MemFHE

detailed in Section 4.6. Now, for each memory row, the integer part represents the integer nearest to the corresponding coefficient of $c_s.(q/Q)$. We finally take modulus of the output with q. Since q is a power of 2 for all security parameters that MemFHE considers, modulo is equivalent to reading $log_2 q$ LSBs of the output. If q is not a power of 2, we use Barrett reduction instead. The output of modulus switching, also the output of server, is a ciphertext with parameter n and q, encrypted with secret key, s of the client.

4.4 MemFHE Bootstrapping

Bootstrapping inputs an encrypted version of the private key, EK_B, also called the refreshing key, along with a ciphertext. The output is a ciphertext corresponding to the input ciphertext but with reduced noise. Bootstrapping performs iterative computations on a cryptographic accumulator, *ACC*. The process involves first *initializing ACC* with the input ciphertext, then implementing an iterative *accumulation* over *ACC*. Each accumulation involves a series of multiplication and addition operations over polynomials. Finally, an element of the final *ACC* is *extracted* to obtain the output ciphertext. In this section, we discuss the implementation of each of these steps in MemFHE.

4.4.1 Initialization

The initialization phase of bootstrapping performs two tasks (i) setting the initial value of *ACC* and (ii) ensures that the input ciphertext's polynomial is compatible with the decomposed refreshing key.

Initializing ACC: MemFHE performs the mapping discussed in Section 4.3.1 in this phase. The coefficients of the bootstrapping-compatible polynomial, m_b are each mapped to $Q/8$ and $-Q/8$ based on whether they lie inside or outside an operation-dependent range (lb, ub), $[3q/8,\ 7q/8)$ in the case of AND. To implement this mapping operation in parallel for all the coefficients of m_b, we utilize search-based PIM operations. Using exact bitwise-search operations, MemFHE implements in-memory compare operation, which can search a set of memory columns for all the numbers greater, equal, or less than the query. The details of the operation are presented in Section 4.6. First MemFHE inputs lb as a query and searches for all the numbers greater than lb. Then, MemFHE performs another query of the filtered numbers with ub as an input, searching for the numbers less than ub. The final filtered-out rows are then initialized to $Q/8$, while the remaining rows are initialized to $-Q/8$. The resultant m_b is assigned as the initial value of ACC.

Polynomial's Compatibility with $\mathbf{EK_B}$: The input ciphertext's polynomial a, needs to be made compatible with the decomposed refreshing key, EK_B. The polynomial a undergoes the same set of operations as those discussed in key switching, except for subtractions, with parameters n, B_r, and d_r instead of N, B_s, and d_s. It results in $n \times d_r$ coefficients for each input. We call them a_{dec}. For the bootstrapping pipeline to work, all of the $n \times d_r$ U_{ACC} units should receive elements from a_{dec}s belonging to different inputs. Hence, we introduce an $n \times d_r$-sized register, in which word$_i$ is fed directly to U_{ACC-i} as shown in Figure 4.1.

4.4.2 Accumulation

The inputs to the accumulation function include the decomposed representation of a (a_{dec} from the initialization step, an RGSW encrypted refreshing key, EK_B, and the output of initialization step, a pair of polynomials of degree N. Accumulation preforms iterative multiplication of this key with ACC and then addition back to ACC. It is the slowest part of bootstrapping due to high data dependency between the iterations. It adds the result of multiplication in each iteration to the accumulator. The dependency of the input of one ciphertext element on the output of the previous one further prohibits the functions from being parallelized across the ciphertext elements. However, each ciphertext element is a high-degree polynomial, providing an opportunity to parallelize over the polynomial length.

AP Bootstrapping:

Traditionally, refreshing key is an n-dimensional vector where each element of the vector is either an N-degree polynomial or a pair of those. However, in AP bootstrapping instead of each element of EK_B being an N-degree polynomial, it is a pair of $2d_g$ polynomials of degree N. Each dimension of the vector is further represented using the pair (B_r, d_r). Hence, the AP refreshing key is a three dimension matrix where each element of the matrix is a pair of $2d_g$ N-degree polynomials. MemFHE stores the refreshing key in $n \times d_r$ memory blocks such that each block stores $2B_r.d_g$ polynomials. Each EK_B memory block is assigned to the corresponding accumulation unit. The main computation of the AP bootstrapping is to perform accumulation function on ACC $n \times d_r$ times. Each step involves a multiplication of the current ACC value with an element of EK_B as $ACC \leftarrow ACC \diamond EK_B$.

Accumulation Unit (U_{ACC}): We design a bootstrapping pipeline such that the accumulation logic consists of $n \times d_r$ accumulation units, U_{ACC}. The unit address (i, j), where $0 \leq i < n$ and $0 \leq j < d_r$, corresponds to the $(i \times d_r + j)$th accumulation iteration. While the units cannot operate on multiple iterations of a single ciphertext in parallel, they can process different ciphertexts in a pipelined fashion. Each unit receives the corresponding value from a_{dec} memory and uses it to select an element from EK_B for multiplication. Since all units input EK_B in each iteration, it introduces a fetch bottleneck at the EK_B. To reduce this problem, EK_B is split over multiple memory blocks, with each U_{ACC} having a local EK_B memory. EK_B is independent of the inputs and populated once.

Since FHEW is based on RGSW encryption scheme, the multiplication in the accumulation stage happens on digit-decomposed operands to reduce the growth of noise. As explained later, the SDD tile in U_{ACC} performs digit decomposition on the two N-degree polynomials of ACC, splitting each coefficient of ACC into d_g numbers with $log_2 B_g$ bits each. EK_B is already digit-decomposed. The output of SDD tile, digit-decomposed ACC_{dec}, contains $2d_g$ polynomials of degree N, similar to each part of EK_B pair polynomials. Now U_{ACC} performs $4d_g$ polynomial-wise multiplications in parallel, $2d_g$ between ACC_{dec} and each part of the EK_B pair as shown in Figure 4.2. To make the multiplication efficient, all the polynomials are converted in NTT domain before multiplying. U_{ACC} employs $2d_g$ NTT pipelines and converts ACC_{dec} into NTT domain. The details of our NTT pipeline are presented in Section 4.4.2. EK_B is already in NTT domain. Polynomials in NTT domain are stored in a row-parallel way, such that each coefficient is stored in a separate row as shown in Figure 4.2. Then, we perform row-parallel multiplication between the polynomials. After multiplication, all products are accumulated to generate a pair of polynomials that serve as the output ACC. Before sending the output to the next unit, U_{ACC} converts it back to the coefficient (non-NTT) domain using the INTT pipeline.

Signed Digit Decompose (SDD): Signed digit decompose (SDD) decomposes a pair of polynomials into multiple polynomials. The core operation is to break each polynomial coefficient (originally $log_2 Q$ bits) into smaller $log_2 B_g$ bit signed numbers. As shown in Table 4.1, B_g is always a power of 2, making the process simpler. SDD consists of one or more memory blocks which perform iterative modulus-division operations, as shown in Figure 4.2. In each iteration, MemFHE selects $log_2 B_g$ LSBs (remainder of the division by B_g) from the coefficients, preserving the remaining bits (quotient of the division). The selected LSBs represent the first $log_2 B_g$-bit number. This process is repeated d_g times, decomposing all coefficients into into d_g

$log_2 B_g$-bit numbers. Hence, in the beginning of each iteration, we first change the range of the coefficients from $[0, Q)$ to $[-Q/2, Q/2]$ by subtracting Q from all inputs in $[Q/2, Q)$, mapping them to $[-Q/2, 0)$. MemFHE implements this operation in parallel for all the coefficients of the input polynomial. Coefficients are stored in different rows, occupying the same set of memory columns. We search for all numbers greater than $Q/2$ using MemFHE's in-memory parallel compare operation discussed in Section 4.6. MemFHE then subtracts Q from all the filtered coefficients. Similarly, the selected LSBs (remainders) are sign-extended, where MemFHE copies the $(log_2 B_g - 1)$th bit for all the coefficients in parallel. Then, all negative remainders are made positive. MemFHE achieves this by searching the MSB bits of all the remainders in parallel (one remainder per coefficient per iteration) and subtracting Q from the filtered remainders.

GINX Bootstrapping:

The decision to run either AP or GINX bootstrapping is based on the type of secret key used by the client. As shown in [37], GINX works better in case of binary and ternary secret keys, while AP works better for other. GINX bootstrapping differs from AP in two major ways. First, it utilizes binary secret keys, resulting in a smaller refreshing key EK_B. EK_B in GINX has a dimension of $n \times 2$, instead of AP's $n \times B_r \times d_r$. Each element consists of $2d_g$ polynomials of degree N, the same as AP. Second, the bootstrapping function in GINX involves extra multiplicative and additive terms to generate the effect of input-dependent polynomial rotation. Specifically, the bootstrapping follows:

$$ACC \leftarrow ACC + (X^m - 1)(ACC \diamond EK_B),$$

where $m = \lfloor a(i) \times (2N/q) \rfloor$ for ith coefficient of the input ciphertext polynomial a. $(X^m - 1)$ is a monomial representing GINX's "blind rotation" by m. This encodes the input in the form of the powers of polynomial. The state-of-the-art implementation PALISADE [92] pre-computes $(X^m - 1)$ for all possible values of $0 \leq m < 2N$ and maintains a library of their NTT counterparts.

Based on the m corresponding to a U_{ACC}, PALISADE selects a value from the library and then multiply it with U_{ACC}'s output. This creates a data transfer bottleneck in a pipelined architecture like MemFHE's, where many units need to access the library simultaneously. On the contrary, MemFHE exploits the bit-level access provided by PIM to implement this ”rotation” efficiently.

MemFHE uses the same architecture to implement GINX as that for AP. GINX requires $n \times 2$ U_{ACC} units. Here, unlike AP, EK_B input to U_{ACC} is independent of the polynomial part a of the ciphertext. Like in the case of AP, the SDD tile of U_{ACC} first decomposes input ACC, U_{ACC} then performs the same polynomial-wise multiplication and subsequent addition, and finally converts them to coefficient domain using INTT. Now, the output of addition represents $prod = (ACC \diamond EK_B)$ in coefficient domain. We now perform in-memory row-parallel rotation on $prod$ as discussed in Section 4.6. MemFHE finally adds the rotated $prod$, $prod_r$, to pre-decomposed ACC and finally subtracts $prod$. The output is the GINX accumulated ACC in coefficient domain.

NTT and INTT Pipeline

Number theoretic transform (NTT) is a generalization of fast Fourier transform (FFT) that performs transformation over a ring instead of complex numbers. In FHE, it is mainly used in polynomial multiplication where it converts a polynomial (by default in coefficient domain) into its frequency (NTT) domain equivalent. A polynomial multiplication in coefficient domain translates to an element-wise multiplication in NTT domain, enabling extensive parallelism for high-degree polynomials. However, the process of converting to and from NTT domain is complex. The state-of-the-art implementations of NTT [96, 25] utilize algorithms where the coefficient access pattern for an n-degree polynomial changes for each of the $log_2 n$ stages of NTT pipeline. Instead, we utilize Singleton's FFT algorithm proposed in [107] and later accelerated in [108, 109, 110] to implement MemFHE's NTT pipeline. Figure 4.3a shows the signal flow graph for Singleton's FFT algorithm. We observe that the coefficient access pattern for the algorithm remains the same for every stage. MemFHE exploits this property to avoid using NTT-specific interconnects.

Data Mapping: Figure 4.3b shows the data layout of one NTT stage in MemFHE. We write an n-degree input polynomial, a, in $n/2$ rows such that a pair of coefficients with indices $2i$ and $(2i+1)$ share the ith row of the memory block. All such pairs are hence written in separate rows, utilizing the same columns. A twiddle factor is associated with each pair, which is pre-computed and stored in the corresponding row. Each pair generates the ith and $(i+n/2)$th coefficients of the output polynomial in ith row of the block.

Computation: Each NTT stage of MemFHE performs three compute operations. First, we perform row-parallel multiplication between the coefficients with odd indices $(2i+1)$ and the corresponding twiddle factor W. Second, we add the generated products to the coefficients with even indices $(2i)$ in a row-parallel way to generate the first $n/2$ coefficients of the output polynomial. Lastly, we subtract the products from the even-indexed coefficients in a row-parallel way to obtain the remaining output coefficients. The details of the row-parallel operation execution are presented in Section 4.6.

Stage-to-Stage Data Transfer: Figure 4.3c shows the data transferred in each transfer phase. We perform column-wise data transfer, where each column consists of one bit from all (or a subset of) rows of the memory block. In one data transfer phase, q column transfers can transfer as many q-bit numbers as the rows in the memory. As discussed in data mapping, the output polynomial is present in $n/2$ rows such that indices $[0, n/2-1]$ are stored in one set of columns and the remaining indices in the another set of columns. Hence, we need four data transfer phases. The first data transfer reads the even-indexed coefficients from $[0, n/2-1]$ and write them to the next stage according to the data mapping scheme, while the second data transfer does the same for the even-indexed coefficients from $[n/2, n-1]$. Similarly, third and fourth data transfer phases deal with odd-indexed coefficients. These data transfers read selected rows from one memory block, send it over a conventional local interconnect, and write them at a contiguous location of the destination memory.

Operation Pipeline: We pipeline our NTT implementation at the granularity of an NTT stage. Each stage works in parallel over different inputs. As discussed in Section 4.7, each MemFHE memory block contains 1024 rows. Hence, one memory block can implement an NTT stage for up to 2048-degree polynomial, requiring a total of $11(log_2 2048)$ memory block for whole NTT. For $n < 2048$, we perform NTT over $m = 2048/n$ inputs at the same time in parallel, while requiring only $log_2 n$ stages in the pipeline. In order to maintain the computation and data transfer characteristics, we interleave the inputs as shown in Figure 4.3e. For example, if $m = 4$, then the first four rows of the memory block store the coefficients corresponding to indices 0 and 1 for the four inputs. The next four rows store coefficients for indices 2 and 3 for those inputs and so on. This ensures that the generated output can be transferred in four data transfer phases as before, without incurring any latency or hardware overhead. Here, the output throughput of the pipeline becomes $m\times$ the original throughput. For $n > 2048$, MemFHE allocates multiple memory blocks per stage and implements a deeper pipeline. For example, if $n = 8192$, MemFHE allocates four memory blocks per stage and thirteen stages per NTT pipeline, requiring a total of 52 memory blocks. Since MemFHE's NTT is stage-wise pipelined, the throughput of the larger NTT is the same as that for $n = 2048$.

Inverse NTT (INTT): NTT and INTT utilize the same hardware and have identical data-mapping, computation, transfer, and pipelining schemes. The two operations differ only in the twiddle factors they use. During pre-compute step, INTT pipeline generates the twiddle factors, w^{-k}, which are inverse of those used in NTT. The rest of the process remains the same.

4.4.3 Extraction

After accumulation, *ACC* consists of a pair of polynomials. Extraction is a simple mapping process that converts *ACC* to a ciphertext. The first polynomial of *ACC* represents the polynomial part of the bootstrapped output ciphertext. Whereas, the constant term (corresponding to degree-0) of the second polynomial represents the integer part. To reverse the mapping operation that occurred during initialization phase, $Q/8$ is added (modulo Q) to the integer part.

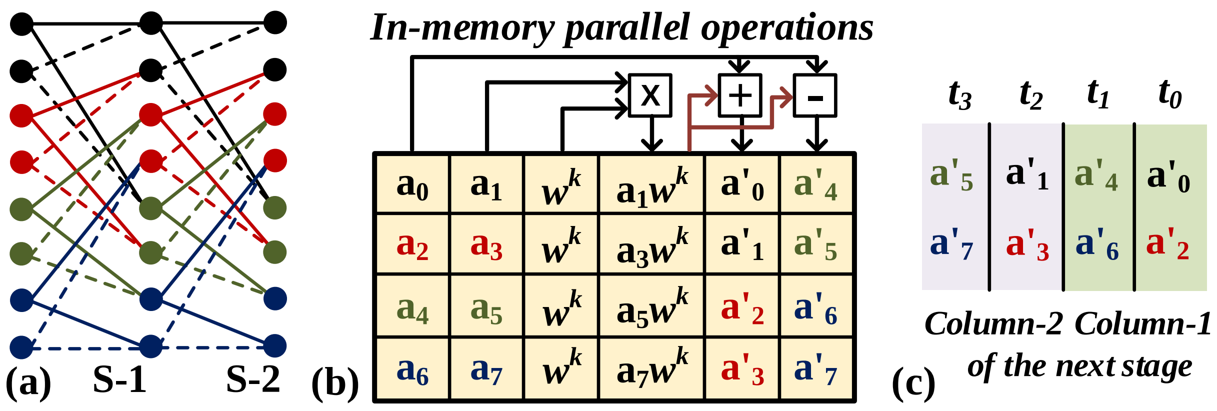


Figure 4.3. Singleton's NTT in MemFHE

4.5 MemFHE Client Architecture

MemFHE client has two functions, encryption and decryption. As discussed below, both require similar operations and can be implemented using the same memory block.

4.5.1 Encryption

Client encryption converts a message bit, m, into a ciphertext of the type (a, b), where a is an integer polynomial of length n, while b is an integer. This encryption utilizes learning with errors (LWE) encryption technique [111, 112, 37] and is defined as $LWE_s(m) = (a,b) = (a,(a.s+e+m')\ mod\ q)$, where m' is an encoded version of m, s is the secret key with the same data type as a, and e is an integer error added to the message.

Evaluating m' involves dividing the message, m, with a message modulus t and then multiplying the output with the application parameter, $q/2$. According to the state-of-the-art implementation in [92] and the security parameters presented in [37] and Section 4.7, t and q are always powers of 2. Hence, MemFHE scales m to m' using in-memory shift and add operations. We first extract the $log_2 t$ LSBs of m. Then, in-memory multiplication with $q/2$ is simply a left shift operation on $m\%t$ by $log_2(q/2)$. Since all the operations in encryption are done modulo q, we extract the $log_2 q$ LSBs of the output. In the case when q is not a power of 2, we perform modulo operations as described in Section 4.6.

Generating integer b requires a dot product between vectors a and s, followed by adding e and m'. To generate this dot product, we utilize the secret key memory, SK_mem. It stores the vector corresponding to secret key s in a row-parallel way such that all the elements of s occupy the same set of memory bitlines and each element is stored in a different row. The incoming vector a is written in the same way as s such that the corresponding elements of a and s are present in the same row.

We implement row-parallel integer multiplication between the elements of the two vectors. Our row-parallel execution performs vector-wide multiplication with the same latency as that of a single multiplication, discussed in Section 4.6. This is followed by an addition of all the products. To add, we perform column parallel in-memory addition operations on the output products such as those proposed in [34] but using the in-memory switching techniques instead of sense amplifier based operations of [34]. In the following discussion, we denote the bitwidth of each product (i.e. log_2q) with the letter p. Here, we accumulate each bit position independently, so that k p-bit numbers are reduced to p log_2k-bit numbers after $(k-2)$ column parallel 1-bit additions for each of the p bit position. To further reduce the output to a single number, we transpose the output of column-parallel addition so that the outputs for all p columns are stored in the same row. It takes p data transfers, log_2k bits per transfer, to read the outputs column-wise and store them in a row. We then perform bit-serial addition to obtain the final integer output, which takes $p \times log_2k$ 1-bit additions. This output represents the dot product $a.s$, to which we add integers e and m'.

4.5.2 Decryption

Client decryption converts the server's output ciphertext, (a,b), back to a bit message, m, as $Round(4/q*(b-a.s))$, where s is the client's private key. MemFHE first uses the dot product implementation of MemFHE's encryption to obtain $a.s$, followed by a subtraction operation with b. The subtraction is followed by a modulo q operation, where MemFHE simply reads the log_2q LSBs of the output. Scaling is done with $4/q$ by discarding the $log_2(q/4)$ LSBs. $Round(.)$ is implemented similar to the rounding function discussed during modulus switching in Section 4.3.4.

4.6 MemFHE Computations

In this section, we detail PIM implementation of the basic MemFHE operations.

Vectorized Data Organization: MemFHE implements vectorized-versions of its operations. An input vector, with n b-bit elements, is stored such that n elements occupy n different rows with but share the same b memory columns.

Row-parallel Addition and Multiplication: A b-bit addition in MemFHE is implemented using bitwise AND, OR, and XOR and requires $(6b+1)$ memory cycles [59]. Similarly, multiplication is performed by generating partial products and serially adding them. MemFHE optimizes the multiplication in [40] by sharing the memory cells among intermediate outputs of addition and utilizing faster operations proposed in [59]. This significantly reduces the time to perform full precision b-bit multiplication from $(13b^2-14b-6)$ to $(7b^2+4b)$ memory cycles, while the total memory required reduces from $(20b-5)$ to $13b$. This increase the maximum possible multiplication bitwidth from 51 bits in [40] to 78 bits in MemFHE.

Modulus/Modulo: Modulus operation gives the remainder of a division. In the context of FHE, modulus is used to avoid overflow during computation. Hence, most operations in MemFHE are followed by modulus. In most cases in MemFHE-server, modulus is taken with respect to a prime number. We perform PIM variants of Barrett [113] (for addition) and Montgomery [114] (for multiplication) reductions using shift and add operations, as done in [25]. This requires prior knowledge of the modulus base, which is governed by the security parameters (and hence known) in MemFHE. If taken with respect to a power of 2, then modulus just selects the corresponding LSBs of the input.

Comparison: Comparison operation in MemFHE can compare an input query with the data stored in MemFHE's memory blocks. We exploit the associative operations proposed in

[103] to search for a bit of data in a memory column. To compare data stored in *b* columns and *r* rows of a memory block with a *b*-bit query, we perform bit-by-bit search. Starting from MSB, associative search is applied for each memory column and all memory rows. All rows where there is a mismatch between the stored bit and the query bit, are selected by associative search circuit [103].

Rotation: Rotation in MemFHE is equivalent to reading out a memory row (column), bit-wise rotating them at the input register of the block and writing it back.

Shift: MemFHE implements shift operation by simply selecting or deselecting bitlines for the corresponding LSB/MSBs. If sign-extension is required, then MemFHE copies the data stored at the original MSB bitline.

4.7 Evaluation

4.7.1 Simulation Setup

We simulate MemFHE using a cycle-accurate simulator. The simulator considers the memory block size (1024×1024 bits in our experiments), the precision for each operation, the degree of polynomials, the locations and the organization of the data. We use HSPICE for circuit-level simulations and calculate energy consumption and performance of all the MemFHE operations with 28nm process node. We adopt an RRAM device with VTEAM model [63] and switching delay of 1.1ns [23]. The parameters of the model have been set to mimic the behavior of practical RRAM memory chips [115]. RRAM components of the design have a SET and RESET voltage of 2V and 1V respectively, with a high-to-low resistance ratio of $10M\Omega/10k\Omega$. A detailed list of parameters is presented in [33, 36]. However, the proposed architecture works with most processing in memory implementations based on digital data. The robustness of all circuits is verified using 5000 Monte Carlo simulations with 10% process variations on the size and threshold voltage of transistors.

Table 4.1. MemFHE Security Parameters

Set	Security	n	q	N	$\log_2 Q$	B_s	B_g	B_r
Classical								
STD128	128-bit	512	512	1024	27	25	2^7	23
STD192	192-bit	512	512	2048	37	25	2^{13}	23
STD256	256-bit	1024	1024	2048	29	25	2^{10}	32
Quantum – Safe								
STD128Q	128-bit	512	512	2048	50	25	2^{25}	23
STD192Q	192-bit	1024	1024	2048	35	25	2^{12}	32
STD256Q	256-bit	1024	1024	2048	27	25	2^7	32

4.7.2 Parameters and Security Guarantees

MemFHE is based on the FHEW cryptosystem of PALISADE library [92]. We perform our evaluation over multiple security parameter sets as described in [37] and summarized in Table 4.1. These parameters guarantee a wide range of security levels for both classical and quantum-safe FHE. MemFHE can be configured to work with any of these security settings.

4.7.3 MemFHE-Server Pipeline Analysis

Figure 4.4 shows the throughput, latency, energy consumed, and memory required for one MemFHE-server pipeline with different parameter settings. The throughput represents the number of input operations that MemFHE can process per millisecond (ms). The latency shows the end-to-end server-side execution time for one input. This also represents the time MemFHE-server takes to fill the pipeline. The energy consumption shows the total end-to-end energy consumed by an input. We compare the throughput-optimized and area-optimized implementations of the pipeline. The two implementations differ in the way they pipeline NTT/INTT. While the area-optimized version follows the stage-wise pipelining mechanism discussed in Section 4.4.2, the throughput-optimized design implements a finer-grained pipeline. It further break an NTT stage into three pipeline stages, first for multiplication with twiddle, second for reduction of the product and addition/subtraction, and the third for final reduction and data transfer to the next stage.

Throughput-Optimized MemFHE: We observe that the four design metrics change significantly with the security levels. MemFHE-server provides a throughput of 174 (51) inputs/ms while ensuring 128-bit classical (quantum-safe) security in the throughput-optimized configuration. Throughput is highly dependent on Q, the bitwidth of server-side computations. More precisely, throughput varies approximately with $(log_2 Q)^2$. This happens because the slowest operation of the pipeline, i.e. the coefficient-wise multiplication, has an implementation latency of $O(Q^2)$ in MemFHE. Similarly, MemFHE-server takes 29 ms (55 ms) to process an input and generate the output ciphertext for the client in 128-bit classical (quantum-safe) FHE setting.MemFHE's latency is dependent on Q^2 as well as the polynomial degree of input ciphertext, n, and parameter d_r and varies approximately with $n.d_r.(log_2 Q)^2$. MemFHE-server consumes a total energy of 34 mJ (164 mJ) for processing an input in 128-bit classical (quantum-safe) FHE setting. While the quantum-safe implementations consume higher energy than their classical counterparts, the difference reduces as the security-level increases. For example, MemFHE consumes 378% more energy in 128-bit quantum-safe mode as compared to the corresponding classical implementation. This reduces to 94% and 11% for 192-bit and 256-bit security levels respectively. The total memory consumed by MemFHE's server changes with different parameter settings as well. It varies approximately with $n.N.d_g$, consuming 37 GB (47 GB) for a complete server pipeline running 128-bit classical (quantum-safe) FHE. We further observe that the accumulation of cryptographic accumulator, ACC, consumes on average 96.5% of the total memory requirement of the server pipeline, while contributing 99.7% to the total latency.

Area-Optimized MemFHE: While MemFHE provides extensive throughput benefits, it takes considerable amount of area. Moreover, since memory is the main resource in MemFHE, we optimized our implementation for area. We observe that an area-optimized MemFHE-server pipeline consumes 2.5× less memory resources on average as compared to the throughput-

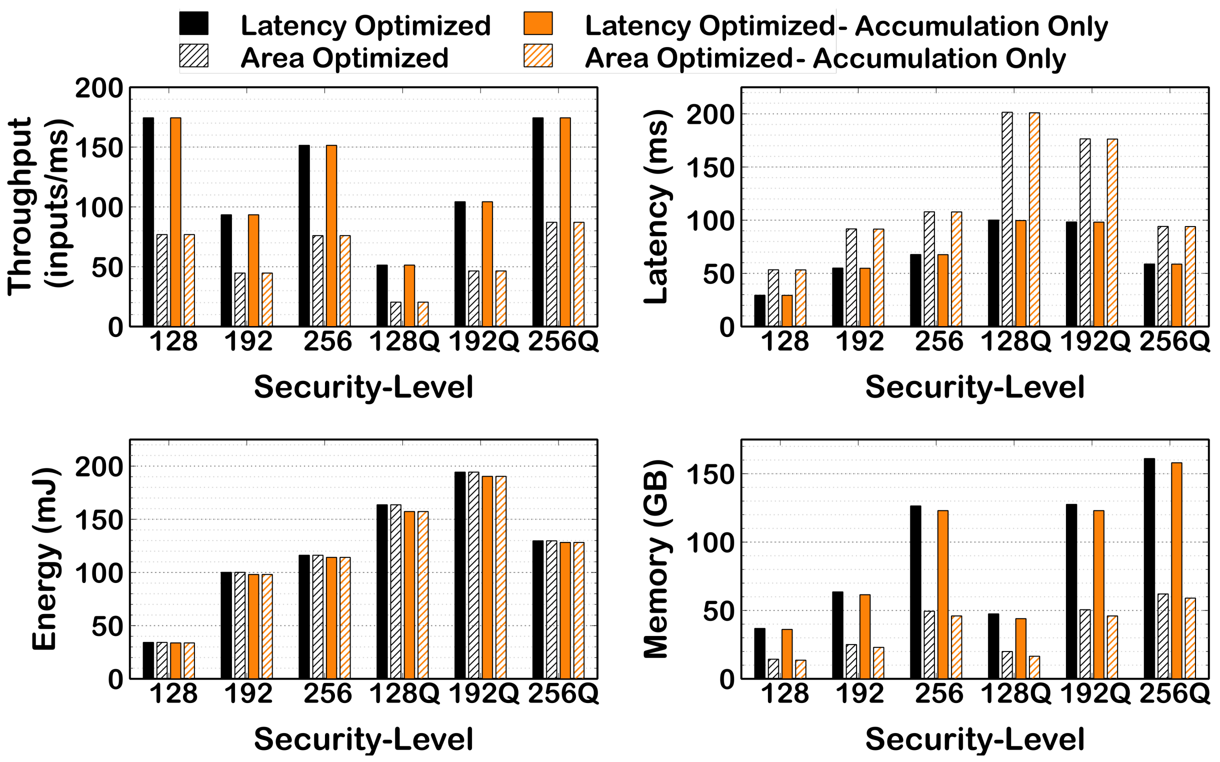


Figure 4.4. MemFHE-server pipeline results for a bitwise operation. The suffix Q represents quantum-safe security guarantee.

optimized design, while reducing the throughput by approximately 2.2×. In contrast, the latency increases by 75%. This happens because we reduce the number of pipeline stages by 3× in the area-optimized design but at the same time increase the latency of each pipeline stage by 2.2×. Since the operations remain the remain in both the designs, their total energy consumption is similar. This highlights one of the advantages of PIM as pipelining doesn't have operational and storage overhead since outputs of most operations are generated in the memory block and hence stored inherently. Similar to the throughput-optimized design, accumulation of *ACC* consumes on average 91.4% of the total memory requirement of the server pipeline, while contributing 99.8% to the total latency.

4.7.4 MemFHE-Server Scalability

To evaluate the scalability of MemFHE, we take the area-optimized version MemFHE for different security-levels and scale it to the given memory size. MemFHE has a minimum memory

Table 4.2. MemFHE Key Sizes (in MB)

	STD128	**STD192**	**STD256**	**STD128Q**	**STD192Q**	**STD256Q**
$\mathbf{EK_S}$	253	925	1269	1719	1750	1013
$\mathbf{EK_B}$ (**AP**)	322	897	1920	1150	2304	1792
$\mathbf{EK_B}$ (**GINX**)	14	39	60	50	72	56
Total (**AP**)	575	1822	3189	2869	4054	2805
Total (**GINX**)	267	964	1329	1769	1822	1069

requirement, which is storage needed for the refreshing and switching keys. The different key sizes in MemFHE are presented in Table 4.2. To scale down from a pipeline's ideal memory size described in Section 4.7.3 and Figure 4.4, we reduce the number of NTT cores in the memory. To scale up from the ideal memory size, we increase the number of pipelines.

Figure 4.5 shows the throughput of the server for different security levels under different memory constraints. Missing bars in the figure show the cases when the available memory is not sufficient to implement MemFHE. We observe that MemFHE's throughput changes almost linearly with the total memory availability. It increases from the ideal 77 inputs/ms with 14 GB memory consumption to 307 inputs/ms with 64 GB for 128-bit security level, while decrease to 7 inputs/ms with 2 GB memory size. However, in some cases the changes isn't linear. For example, for the quantum-safe 128-bit security configuration, MemFHE's throughput of 20 inputs/ms doesn't change when going from the ideal 20 GB to 32 GB. This happens because the increase in memory is not sufficient to support two pipelines. At the same time, increasing the memory availability further to 64 GB increases the throughput by $3\times$ to 61 inputs/ms because 64 GB memory has enough resources to fit three STD128Q pipelines.

4.7.5 MemFHE Client Analysis

MemFHE-client encrypts bits to ciphertexts and decrypts processed ciphertexts back to bits. Figure 4.6a shows the encryption latency and energy consumption for MemFHE-client at different security levels for a bit. Decryption involves the same operations and has roughly the same latency as that of encryption. The latency of encryption depends on the ciphertext modulus,

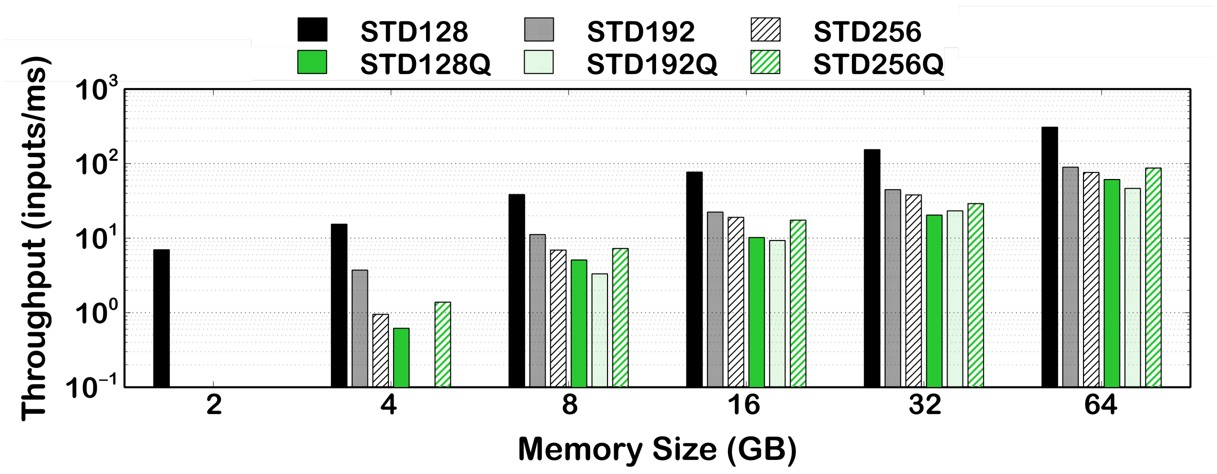


Figure 4.5. MemFHE-server throughput for different memory sizes. The missing bars represents memory lower than the minimum required size.

q, and the polynomial degree, n. As expected, the dot product $a.s$ is the slowest operation in encryption, taking 98% of the total latency. Encrypting a bit to a 128-bit (256-bit) quantum-safe ciphertext takes 3 us (5.5 us), while it consumes 4 nJ (9.8 nJ) of energy.

MemFHE requires a total of 128 KB (256 KB) memory (one memory block) for generating a 128-bit (256-bit) quantum-safe ciphertext. However, similar to MemFHE-server, the client is also scalable and employs multiple encrypting-decrypting memory blocks for processing multiple inputs in parallel. Figure 4.6b shows how the throughput of the MemFHE-client changes with the available memory sizes. The figure shows the combined encrypt-decrypt throughput. Each memory block in MemFHE can be dynamically configured to run either encryption or decryption. We observe that the client's throughput increases linearly with the increase in the total memory size, going from 0.2 inputs/us for 256 KB memory to nearly 47 inputs/us for 64 MB for quantum-safe 256-bit encryption.

4.7.6 Arithmetic Operations in MemFHE

In this subsection, we show the end-to-end performance of MemFHE while implementing addition and multiplication. We utilize Kogge-Stone adders for addition operation as well as accumulation of partial products during multiplication. This reduces the critical path of the circuits. It is essential because even though MemFHE provides large throughput, the end-to-end

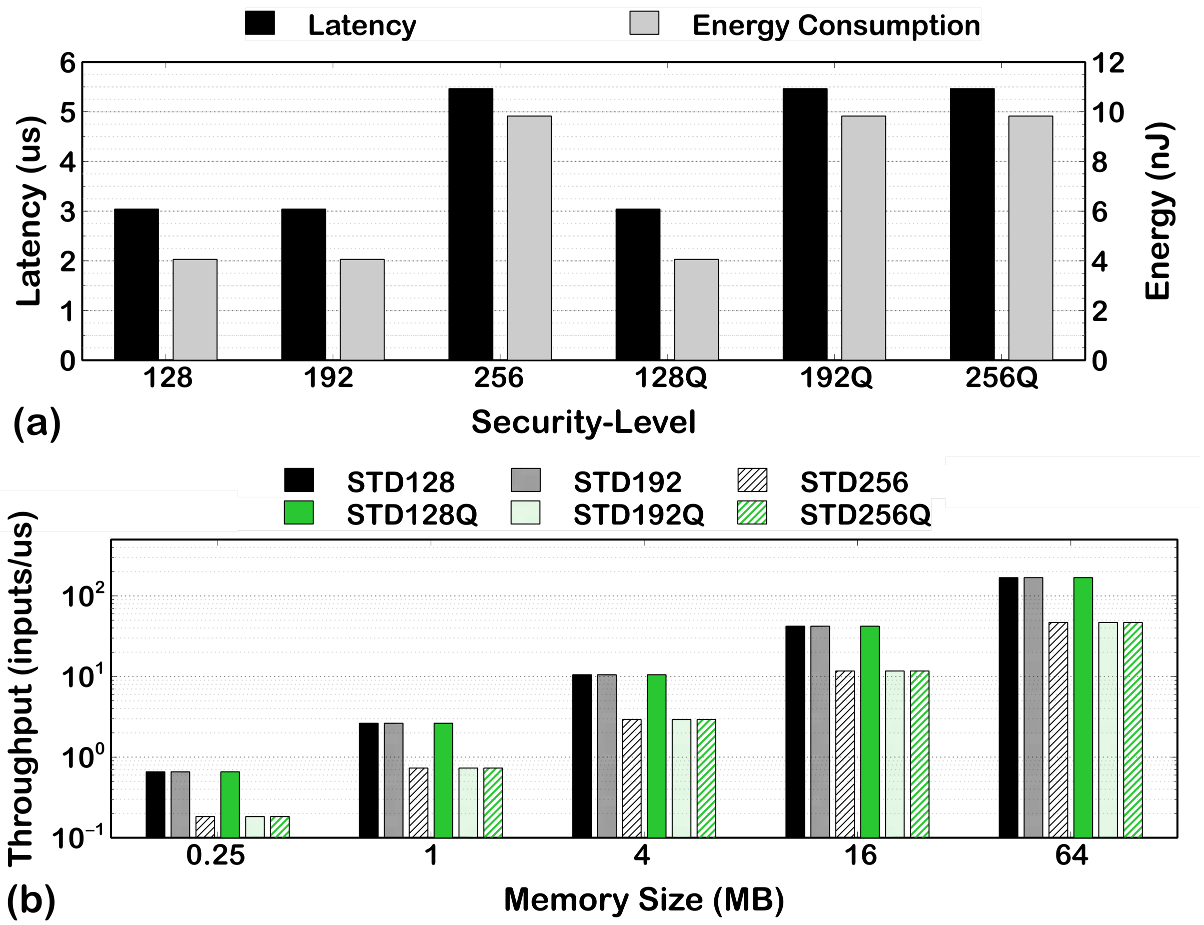


Figure 4.6. Encryption in MemFHE-client. (a) Latency and energy consumption and (b) throughput for different memory sizes.

latency for an input is comparatively high due to MemFHE's large pipeline depth. Hence, here we focus on the latency of executing each operation, rather than the associated throughput. Provided sufficient independent inputs, MemFHE can implement all these operations with the same throughput as shown in Section 4.7.3, processing up to 174 inputs/ms at 256-bit quantum-safe security.

Figure 4.7 shows the latency of running different types of additions and multiplications in MemFHE pipeline for various security settings. We observe that for individual operations, the latency is limited by their critical path. The latencies for individual addition vary with $O(log_2 b)$, where b is the bitwidth of operation, taking 353 ms (705 ms) for an 8-bit (64-bit) addition while providing 256-bit quantum-safe security. For a multiplication, the latency varies with $O(b.log_2 b)$,

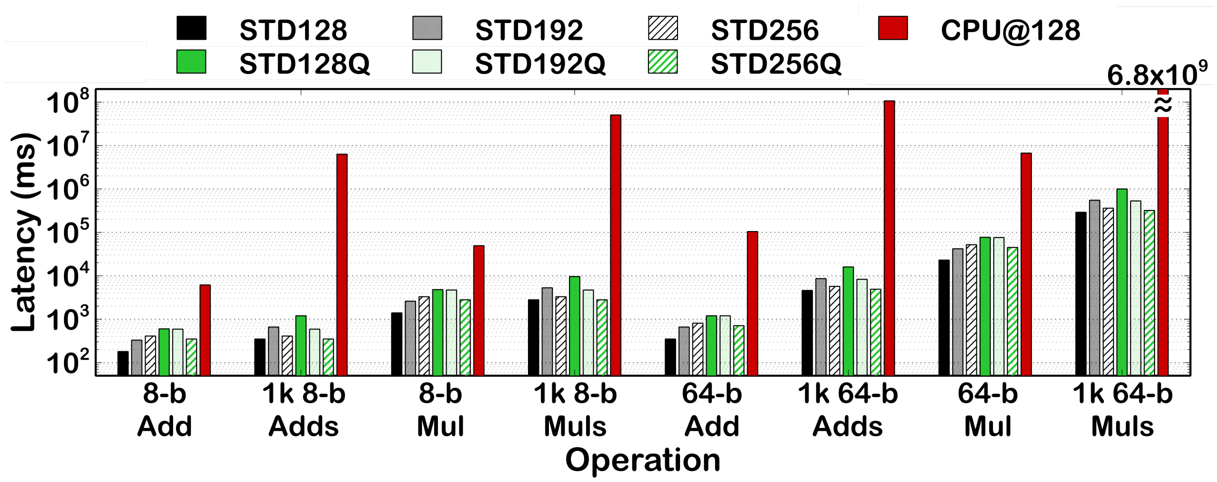


Figure 4.7. End-to-end latency for implementing additions and multiplications in MemFHE.

taking 2.8 s (45 s) for an 8-bit (64-bit) multiplication at the same security level.

Implementing 1024 independent additions and multiplications does not increase the latency significantly. Instead, these independent inputs fill up MemFHE's pipeline, which was otherwise severely underutilized. For example, performing 1024 8-bit additions/multiplication take only twice the total time as that for single addition/multiplication in 128-bit quantum-safe setting. For 256-bit quantum-safe FHE, the latency for 1024 8-bit additions/multiplications is actually similar to that for a single addition/multiplication. This happens because MemFHE pipeline for STD256Q is much deeper than that of STD128Q, allowing more operations to fill up the pipeline. Even for 1024 64-bit multiplications, MemFHE is at most 13× slower than one 64-bit multiplication. This shows that MemFHE truly shines when there are enough independent operations to fill the pipeline.

Lastly, Figure 4.7 also shows the latency of different addition and multiplication operations, normalized to MemFHE, for an Intel i7-9700 CPU with 64 GB of RAM in 128-bit classical security setting in log scale. The results were obtained using single-threaded implementation of the state-of-the-art PALISADE library [92] as detailed in [37]. We observe that CPU is on average 35× (295×) slower than MemFHE for individual 8-bit (64-bit) arithmetic operations. For 1024 arithmetic operations, MemFHE is on average 20573× faster than CPU. While a multi-threaded CPU implementation would theoretically reduce the latency by 8× in this case, it

Table 4.3. Workloads for Learning in MemFHE [3]

Dataset	**Network Topology**	**Accuracy**	**#GateOps**
MNIST	C-B-A-P-C-P-F-B-A-F[116]	99.54%	856K
CIFAR-10	[C-B-A-C-B-A-P]×3-F-F[117]	92.54%	211M
ImageNet	ShuffleNet [118]	69.4%	1.1G
Penn Treebank [119]	LSTM: t-step 25, 300-unit layer; ReLU [119]	89.8 PPW	24.4M

C: convolution layer; A: activation layer; B: batch normalization layer;
P: pooling layer; F: fully-connected layer; PPW: perplexity per word.

would still be much slower than MemFHE. This is due to the highly pipelined architecture of MemFHE that can deliver higher throughput for large data.

4.7.7 Learning in MemFHE

We show how MemFHE performs for complicated learning tasks. Our evaluation is inspired from the CPU implementation of TFHE-based deep neural networks (DNN) in [3], which we refer to as TDNN for simplicity. TDNN converts DNN operations into TFHE compatible functions. We use the same functions to evaluate MemFHE as it also supports TFHE. Table 4.3 details the datasets and the corresponding network topologies used for evaluation. TDNN works in both fully homomorphic (TDNN-FHE) mode as well as leveled mode (TDNN-Lvl). While TDNN-FHE bootstraps each gate operation, TDNN-Lvl bootstraps only higher-level operations like polynomial multiplications and additions [3].

Figure 4.8a shows the inference throughput of MemFHE and TDNN over various datasets. MemFHE is scaled to have a total of 64GB memory size. We then configure MemFHE based on the target security setting. While MemFHE provide a range of classical and quantum-safe security guarantees, TDNN provides 163-bit (152-bit) security guarantee in FHE (leveled) mode. We observe that as compared to TDNN-FHE, MemFHE provides on average 2007× higher throughput (inference/s) for classical FHE. Moreover, MemFHE has 827× higher throughput while ensuring quantum-safe FHE while TDNN-FHE just provides classical security. We also observe that MemFHE in quantum-safe provides similar throughput as TDNN-Lvl. This is a huge improvement because leveled HE accelerates computations on encrypted data by enabling

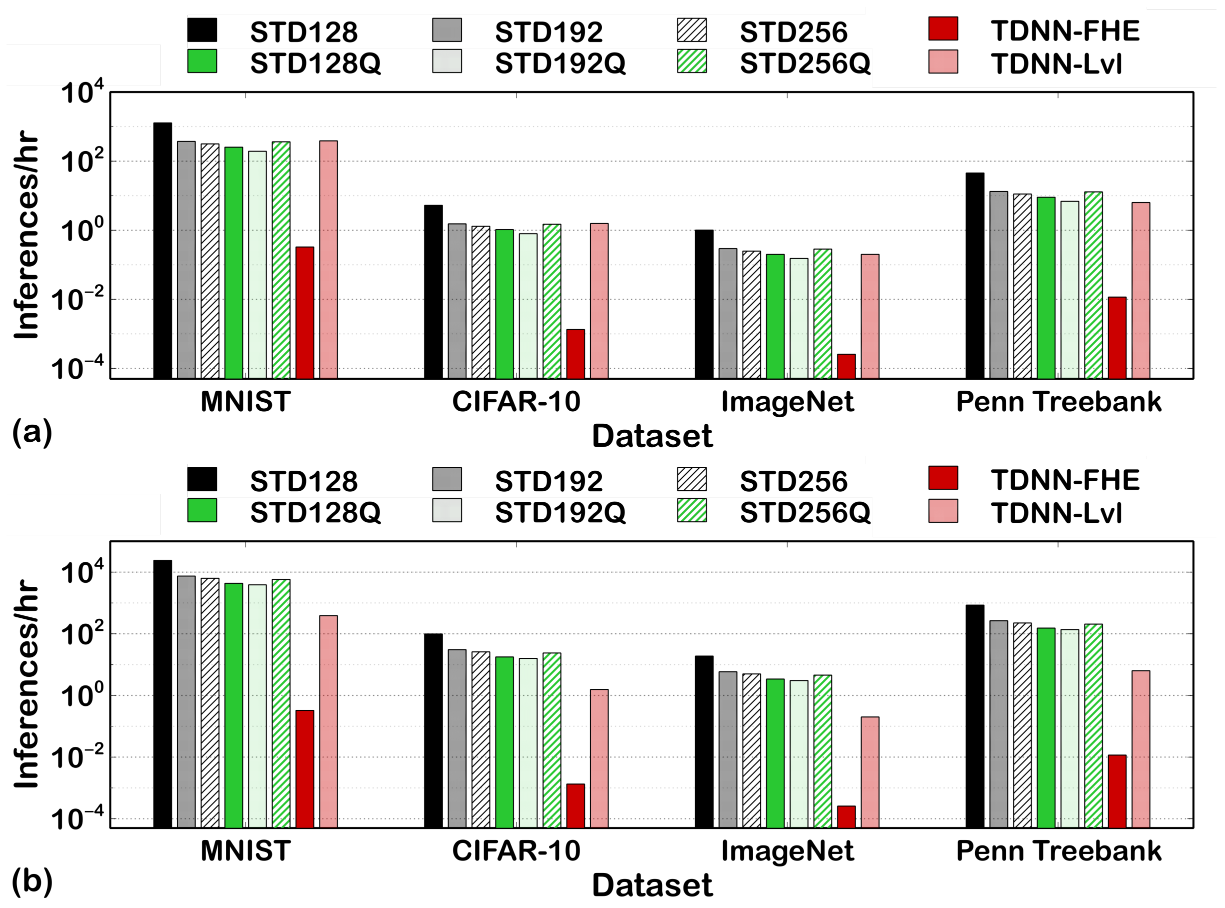


Figure 4.8. Inference throughput of MemFHE and TDNN [3] for different datasets. MemFHE utilizes (a) 64GB memory and (b) 1TB memory. TDNN-FHE and TDNN-Lvl provide 163-bit and 152-bit security guarantees.

operations without bootstrapping. However, it restricts the ciphertext sizes, further limiting the achievable security levels. Moreover, encrypting in leveled mode is dependent on the complexity of target operation, which introduces dependency between different inputs of an application. On the other hand, MemFHE achieves the throughput of a leveled implementation while running FHE.

We observe that TDNN presented in [3] runs on an Intel Xeon E7-4850 CPU with 1TB DRAM. To perform a similar memory evaluation, we scale MemFHE up to consume 1TB memory. Figure 4.8 summarizes the results. We observe that MemFHE's throughput further increases on average by 19$\times$ (17$\times$) for classical (quantum-safe) FHE. This translates to four orders of magnitude higher throughput than TDNN-FHE. This huge improvement in

MemFHE comes from (i) significant reduction in total data-transfers and (ii) the significantly higher number of processing in memory cores. In MemFHE, off-chip data-transfers consists only of the communication between client and server. On the other hand, traditional systems require a large number of back and forth core to memory transfers even during server-side computation. The high density of memory allows us to have a large number of PIM-enabled processing cores in the system, allowing for higher parallelism and deeper pipelining.

Chapter 4, in part, has been submitted for publication of the material as it may appear in S. Gupta and T. Rosing, "Accelerating Fully Homomorphic Encryption with Processing in Memory," *Design Automation Conference (DAC)*, 2021. The dissertation author was the primary investigator and author of this material.

Chapter 4, in part, is currently being prepared for submission for publication of the material. S. Gupta, R. Cammarota, and T. Rosing, "MemFHE: End-to-End Computing with Fully Homomorphic Encryption in Memory." The dissertation author was the primary investigator and author of this material.

Chapter 5

Summary and Future Work

The massive growth of data and the desire to process it with algorithms like machine learning have been pushing traditional computing systems to their limits. The aim of the dissertation is to make learning significantly more efficient and secure. In doing so, we utilize the brain-inspired hyperdimensional computing to make learning less complex and more hardware-friendly. We alleviate the pressure on today's computing platforms by enabling computations across the memory hierarchy, making it *intelligent*. We make end-to-end data privacy feasible in cloud computing based systems by designing accelerators that enable fully homomorphic encryption.

In this dissertation, we propose solutions for learning problems in two different scenarios. For efficient local learning, we accelerate HD computing using processing in-memory (Chapter 2) and propose an in-storage computing solution for learning with HD computing on large datasets (Chapter 3). To enable secure and privacy-preserved computing, we make fully homomorphic encryption feasible with processing in-memory (Chapter 4).

5.1 Dissertation Summary

Hyperdimensional Computing Across Memory Hierarchy: Chapter 2 proposed Tri-HD, the first ReRAM PIM architecture to implement the complete HD computing-based classification pipeline for non-binary data. Our design utilizes a novel distance metric that is

PIM-friendly and provides similar application accuracy as the more complex baseline metric. Our proposed architecture is enabled in PIM by fast and energy-efficient in-memory logic operations. We further increase the amount of in-memory parallelism by using in-block switches, which segment the bitlines to make parallel operations independent of each other. Our evaluation shows that for all applications tested using HD, Tri-HD provides on average 434x (2170x) speedup and consumes 4114x (26019x) less energy as compared to the CPU while running end-to-end HD training (inference). Tri-HD also achieves at least 2.2% higher classification accuracy than all existing PIM-based HD designs.

Chapter 3 proposed Store-n-Learn, an in-storage HD computing system that spans multiple levels of the storage hierarchy. We exploited the internal bandwidth and hierarchical structure of SSDs to perform HD classification and clustering in-storage. We proposed batched HD computing training to enable partial processing of HD hypervectors. We further proposed die-level accelerator for HD encoding and top-level FPGA accelerators for HD training, retraining, inference, and clustering. Our evaluation shows that Store-n-Learn is on average 222x (543x) faster than CPU and 10.6x (7.3x) faster than the state-of-the-art ISC solution, INSIDER [1] for HD computing-based classification (clustering).

Privacy-Preserving Computing with Fully Homomorphic Encryption: Chapter 4 presented MemFHE, the first end-to-end acceleration of fully homomorphic encryption in PIM. We designed accelerators for both client as well as server for the latest Ring-GSW based homomorphic encryption schemes. MemFHE alleviates the effect of FHE's data and compute explosion by reducing the data transfer bottlenecks and enabling extensive parallelism. MemFHE raises the bar of the security of today's systems, providing both classical as well as quantum-safe security guarantees. Our evaluation shows that MemFHE provides an average 2007x higher throughput for FHE-enabled neural networks than the state-of-the-art implementation.

5.2 Future Work

Our future plan is to design a heterogeneous system that can compute in processing core, cache, memory, and storage. To realize this effort, we aim to create a system infrastructure that can enable processing at different parts of the system. Such an infrastructure would include a code analyzer that can identify the parts of the program that could potentially be offloaded to memory and storage, followed by an operation library that maps those parts of the code to the PIM/ISC operations. This would be supported by PIM and ISC-specific compilers and data allocator. An initial proof-of-concept can be enabled with memory-centric interconnects like OpenCAPI and CXL as the underlying technologies.

On the application-side, we want to bridge the gap between hyperdimensional computing and state-of-the-art machine learning algorithms. We plan to add feature extraction capabilities to the current HD computing algorithms. This would allow HD to process data like images with higher accuracy. We also plan to extend our implementation of fully homomorphic encryption to natively support more data types in addition to bit operations. This would significantly reduce the latency of applications, bringing FHE closer to a real-world deployment.

Bibliography

[1] Z. Ruan, T. He, and J. Cong, "Insider: designing in-storage computing system for emerging high-performance drive," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, pp. 379–394, 2019.

[2] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. De Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W.-m. Hwu, "Deepstore: In-storage acceleration for intelligent queries," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 224–238, 2019.

[3] Q. Lou and L. Jiang, "She: A fast and accurate deep neural network for encrypted data," *Advances in neural information processing systems*, 2019.

[4] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the internet of things: A survey," *IEEE communications surveys & tutorials*, vol. 16, no. 1, pp. 414–454, 2013.

[5] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[6] 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.

[7] 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.

[8] M. Imani, C. Huang, D. Kong, and T. Rosing, "Hierarchical hyperdimensional computing for energy efficient classification," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] M. Imani, Y. Kim, S. Riazi, J. Messerly, P. Liu, F. Koushanfar, and T. Rosing, "A framework for collaborative learning in secure high-dimensional space," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 435–446, IEEE, 2019.

[14] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a micro-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[15] G. H. Loh, N. Jayasena, M. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski, "A processing in memory taxonomy and a case for studying fixed-function pim," in *Workshop on Near-Data Processing (WoNDP)*, pp. 1–4, 2013.

[16] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman, "Ac-dimm: associative computing with stt-mram," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 189–200, 2013.

[17] Q. Guo, X. Guo, Y. Bai, and E. Ipek, "A resistive tcam accelerator for data-intensive computing," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 339–350, IEEE, 2011.

[18] M. Imani, Y. Kim, A. Rahimi, and T. Rosing, "Acam: Approximate computing based on adaptive associative memory with online learning," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 162–167, 2016.

[19] L. Yavits, S. Kvatinsky, A. Morad, and R. Ginosar, "Resistive associative processor," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 148–151, 2014.

[20] S. Gupta, M. Imani, B. Khaleghi, V. Kumar, and T. Rosing, "Rapid: A reram processing in-memory architecture for dna sequence alignment," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2019.

[21] A. Siemon, S. Menzel, R. Waser, and E. Linn, "A complementary resistive switch-based crossbar array adder," *IEEE journal on emerging and selected topics in circuits and systems*, vol. 5, no. 1, pp. 64–74, 2015.

[22] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*, pp. 1–6, 2016.

[23] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.

[24] M. Imani, Y. Kim, and T. Rosing, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 757–763, IEEE, 2017.

[25] H. Nejatollahi, S. Gupta, M. Imani, T. S. Rosing, R. Cammarota, and N. Dutt, "Cryptopim: in-memory acceleration for lattice-based cryptographic hardware," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[26] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong, "Yoursql: a high-performance database system leveraging in-storage computing," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 924–935, 2016.

[27] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A framework for near-data processing of big data workloads," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 153–165, 2016.

[28] G. Koo, K. K. Matam, I. Te, H. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: trading communication with computing near storage," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 219–231, IEEE, 2017.

[29] M. Kim, A. Harmanci, J.-P. Bossuat, S. Carpov, J. H. Cheon, I. Chillotti, W. Cho, D. Froelicher, N. Gama, M. Georgieva, S. Hong, J.-P. Hubaux, D. Kim, K. Lauter, Y. Ma, L. Ohno-Machado, H. Sofia, Y. Son, Y. Song, J. Troncoso-Pastoriza, and X. Jiang, "Ultra-fast homomorphic encryption models enable secure outsourcing of genotype imputation," *bioRxiv*, 2020.

[30] H. Chen, Z. Huang, K. Laine, and P. Rindal, "Labeled psi from fully homomorphic encryption with malicious security," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1223–1237, 2018.

[31] M. Kim, Y. Song, B. Li, and D. Micciancio, "Semi-parallel logistic regression for gwas on encrypted data," *BMC Medical Genomics*, vol. 13, no. 7, pp. 1–13, 2020.

[32] E. J. Chou, A. Gururajan, K. Laine, N. K. Goel, A. Bertiger, and J. W. Stokes, "Privacy-preserving phishing web page classification via fully homomorphic encryption," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2792–2796, IEEE, 2020.

[33] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC – memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[34] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 383–396, IEEE, 2018.

[35] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 397–410, 2019.

[36] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 802–815, IEEE, 2019.

[37] D. Micciancio and Y. Polyakov, "Bootstrapping in fhew-like cryptosystems.," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 86, 2020.

[38] S. Gupta, M. Imani, J. Sim, A. Huang, F. Wu, M. H. Najafi, and T. Rosing, "Scrimp: A general stochastic computing architecture using reram in-memory processing," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1598–1601, IEEE, 2020.

[39] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (imply) logic: Design principles and methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2054–2066, 2013.

[40] A. Haj-Ali, R. Ben-Hur, N. Wald, and S. Kvatinsky, "Efficient algorithms for in-memory fixed point multiplication using magic," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, 2018.

[41] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "'memristive'switches enable 'stateful'logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

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

[43] 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.

[44] 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.

[45] 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.

[46] M. Zhou, M. Imani, S. Gupta, and T. Rosing, “Thermal-aware design and management for search-based in-memory acceleration,” in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[47] 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.

[48] 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 *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 492–494, IEEE, 2018.

[49] T. F. Wu, H. Li, P.-C. Huang, A. Rahimi, G. Hills, B. Hodson, W. Hwang, J. M. Rabaey, H.-S. P. Wong, M. M. Shulaker, and S. Mitra, “Hyperdimensional computing exploiting carbon nanotube fets, resistive ram, and their monolithic 3d integration,” *IEEE Journal of Solid-State Circuits*, vol. 53, no. 11, pp. 3183–3196, 2018.

[50] G. Karunaratne, M. L. Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian, “In-memory hyperdimensional computing,” *arXiv preprint arXiv:1906.01548*, 2019.

[51] 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 H.-S. P. Wong, “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.

[52] J. Liu, M. Ma, Z. Zhu, Y. Wang, and H. Yang, “Hdc-im: Hyperdimensional computing in-memory architecture based on rram,” in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 450–453, IEEE, 2019.

[53] S. Hamdioui, H. A. Du Nguyen, M. Taouil, A. Sebastian, M. L. Gallo, S. Pande, S. Schaafsma, F. Catthoor, S. Das, F. G. Redondo, G. Karunaratne, A. Rahimi, and L. Benini, “Applications of computation-in-memory architectures based on memristive devices,” in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 486–491, IEEE, 2019.

[54] M. Imani, X. Yin, J. Messerly, S. Gupta, M. Niemier, X. S. Hu, and T. Rosing, "Searchd: A memory-centric hyperdimensional computing with stochastic training," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[55] "Uci machine learning repository: Isolet dataset." http://archive.ics.uci.edu/ml/datasets/ISOLET, 1994.

[56] G. Griffin, A. Holub, and P. Perona, "Caltech-256 object category dataset," 2007.

[57] "Uci machine learning repository: Har dataset." https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities, 2012.

[58] A. Reiss and D. Stricker, "Creating and benchmarking a new dataset for physical activity monitoring," in *Proceedings of the 5th International Conference on PErvasive Technologies Related to Assistive Environments*, p. 40, ACM, 2012.

[59] 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.

[60] M. Imani, S. Gupta, S. Sharma, and T. S. Rosing, "Nvquery: Efficient query processing in nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 628–639, 2018.

[61] M. Imani, S. Gupta, A. Arredondo, and T. Rosing, "Efficient query processing in crossbar memory," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2017.

[62] M. Imani, S. Gupta, and T. Rosing, "Ultra-efficient processing in-memory for data intensive applications," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2017.

[63] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, 2015.

[64] M. Everingham, S. A. Eslami, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes challenge: A retrospective," *International journal of computer vision*, vol. 111, no. 1, pp. 98–136, 2015.

[65] 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 *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8, IEEE, 2016.

[66] A. Rahimi, P. Kanerva, J. d. R. Millán, and J. M. Rabaey, "Hyperdimensional computing for noninvasive brain-computer interfaces: Blind and one-shot classification of eeg error-related potentials," in *10th EAI Int. Conf. on Bio-inspired Information and Communications Technologies*, 2017.

[67] M. Imani, S. Bosch, S. Datta, S. Ramakrishna, S. Salamat, J. M. Rabaey, and T. Rosing, "Quanthd: A quantization framework for hyperdimensional computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[68] M. Imani, S. Salamat, B. Khaleghi, M. Samragh, F. Koushanfar, and T. Rosing, "Sparsehd: Algorithm-hardware co-optimization for efficient high-dimensional computing," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 190–198, IEEE, 2019.

[69] M. Schmuck, L. Benini, and A. Rahimi, "Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 4, pp. 1–25, 2019.

[70] 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.

[71] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 137–152, 2017.

[72] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: a user-programmable ssd," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pp. 67–80, 2014.

[73] 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.

[74] 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.

[75] W. Cheong, C. Yoon, S. Woo, K. Han, D. Kim, C. Lee, Y. Choi, S. Kim, D. Kang, G. Yu, J. Kim, J. Park, K.-W. Song, K.-T. Park, S. Cho, H. Oh, D. D. G. Lee, J.-H. Choi, and J. Jeong, "A flash memory controller for 15μs ultra-low-latency ssd using high-speed 3d nand flash with 3μs read time," in *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 338–340, IEEE, 2018.

[76] "Uci machine learning repository: Cardiotocography dataset," 2010.

[77] S. Benatti, E. Farella, E. Gruppioni, and L. Benini, "Analysis of robust implementation of an emg pattern recognition based control.," in *BIOSIGNALS*, pp. 45–54, 2014.

[78] M. C. Thrun and A. Ultsch, "Clustering benchmark datasets exploiting the fundamental clustering problems," *Data in brief*, vol. 30, p. 105501, 2020.

[79] "Uci machine learning repository: Iris dataset," 1988.

[80] M. Imani, M. S. Razlighi, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing, "Deep learning acceleration with neuron-to-memory transformation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–14, IEEE, 2020.

[81] 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.

[82] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[83] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 169–178, 2009.

[84] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 617–640, Springer, 2015.

[85] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[86] H. Chen and K. Han, "Homomorphic lower digits removal and improved fhe bootstrapping," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 315–337, Springer, 2018.

[87] M. S. Lee, Y. Lee, J. H. Cheon, and Y. Paek, "Accelerating bootstrapping in fhew using gpus," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 128–135, IEEE, 2015.

[88] X. Lei, R. Guo, F. Zhang, L. Wang, R. Xu, and G. Qu, "Optimizing fhew with heterogeneous high-performance computing," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 8, pp. 5335–5344, 2019.

[89] T. Morshed, M. M. Al Aziz, and N. Mohammed, "Cpu and gpu accelerated fully homomorphic encryption," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 142–153, IEEE, 2020.

[90] W. Dai and B. Sunar, "Cuda-accelerated fully homomorphic encryption library, august 2019."

[91] "Nufhe, a gpu-powered torus fhe implementation," *Available at https://github.com/nucypher/nufhe*, 2019.

[92] K. Rohloff and Y. Polyakov, "The palisade lattice cryptography library, 1.2017," *Library available at https://git. njit. edu/palisade/PALISADE*, 2017.

[93] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *international conference on the theory and application of cryptology and information security*, pp. 3–33, Springer, 2016.

[94] D. B. Cousins, K. Rohloff, and D. Sumorok, "Designing an fpga-accelerated homomorphic encryption co-processor," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 2, pp. 193–206, 2016.

[95] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International symposium on high performance computer architecture (HPCA)*, pp. 387–398, IEEE, 2019.

[96] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1295–1309, 2020.

[97] D. Reis, J. Takeshita, T. Jung, M. Niemier, and X. S. Hu, "Computing-in-memory for performance and energy-efficient homomorphic encryption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 11, pp. 2300–2313, 2020.

[98] S. Halevi and V. Shoup, "Algorithms in helib," in *Annual Cryptology Conference*, pp. 554–571, Springer, 2014.

[99] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Annual Cryptology Conference*, pp. 75–92, Springer, 2013.

[100] A. Guimarães, E. Borin, and D. F. Aranha, "Revisiting the functional bootstrap in tfhe," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 229–253, 2021.

[101] J. Zhou, J. Li, E. Panaousis, and K. Liang, "Deep binarized convolutional neural network inferences over encrypted data," in *2020 7th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2020 6th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, pp. 160–167, IEEE, 2020.

[102] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky, "Imaging: In-memory algorithms for image processing," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4258–4271, 2018.

[103] A. Ghofrani, A. Rahimi, M. A. Lastras-Montaño, L. Benini, R. K. Gupta, and K.-T. Cheng, "Associative memristive memory for approximate computing in gpus," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 2, pp. 222–234, 2016.

[104] M. Imani, S. Pampana, S. Gupta, M. Zhou, , Y. Kim, and T. Rosing, “Dual: Acceleration of clustering algorithms using digital-based processing in-memory,” in *Proceedings of the International Symposium on Microarchitecture*, IEE/ACM, 2020.

[105] J. Alperin-Sheriff and C. Peikert, “Faster bootstrapping with polynomial error,” in *Annual Cryptology Conference*, pp. 297–314, Springer, 2014.

[106] N. Gama, M. Izabachene, P. Q. Nguyen, and X. Xie, “Structural lattice reduction: generalized worst-case to average-case reductions and homomorphic cryptosystems,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 528–558, Springer, 2016.

[107] R. Singleton, “A method for computing the fast fourier transform with auxiliary memory and limited high-speed storage,” *IEEE Transactions on Audio and Electroacoustics*, vol. 15, no. 2, pp. 91–98, 1967.

[108] Z. Liu, Y. Song, T. Ikenaga, and S. Goto, “A vlsi array processing oriented fast fourier transform algorithm and hardware implementation,” *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 88, no. 12, pp. 3523–3530, 2005.

[109] H. E. Yantir, W. Guo, A. M. Eltawil, F. J. Kurdahi, and K. N. Salama, “An ultra-area-efficient 1024-point in-memory fft processor,” *Micromachines*, vol. 10, no. 8, p. 509, 2019.

[110] H. Cılasun, S. Resch, Z. I. Chowdhury, E. Olson, M. Zabihi, Z. Zhao, T. Peterson, J.-P. Wang, S. S. Sapatnekar, and U. Karpuzcu, “Crafft: High resolution fft accelerator in spintronic computational ram,” in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[111] V. Lyubashevsky, C. Peikert, and O. Regev, “On ideal lattices and learning with errors over rings,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 1–23, Springer, 2010.

[112] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography,” *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.

[113] P. Barrett, “Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor,” in *CRYPTO*, 1986.

[114] P. L. Montgomery, “Modular multiplication without trial division,” *Mathematics of computation*, 1985.

[115] J. J. Yang, D. B. Strukov, and D. R. Stewart, “Memristive devices for computing,” *Nature nanotechnology*, vol. 8, no. 1, pp. 13–24, 2013.

[116] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *International Conference on Machine Learning*, pp. 201–210, PMLR, 2016.

[117] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, and E. Prouff, "Privacy-preserving classification on deep neural network.," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 35, 2017.

[118] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 6848–6856, 2018.

[119] Q. V. Le, N. Jaitly, and G. E. Hinton, "A simple way to initialize recurrent networks of rectified linear units," *arXiv preprint arXiv:1504.00941*, 2015.