

UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Efficient Computing in Hyperdimensional Space

Permalink

<https://escholarship.org/uc/item/64t2x17z>

Author

Kang, Jaeyoung

Publication Date

2023

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA SAN DIEGO

Efficient Computing in Hyperdimensional Space

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

in

Electrical Engineering (Communication Theory & Systems)

by

Jaeyoung Kang

Committee in charge:

Professor Tajana Šimunić Rosing, Chair
Professor Mingu Kang
Professor Duygu Kuzum
Professor Jishen Zhao

2023

Copyright
Jaeyoung Kang, 2023
All rights reserved.

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

University of California San Diego

2023

DEDICATION

To my parents and my sister,

For their endless love and support

EPIGRAPH

The best way to predict the future is to invent it.

— Alan Kay

TABLE OF CONTENTS

Dissertation Approval Page	iii
Dedication	iv
Epigraph	v
Table of Contents	vi
List of Figures	ix
List of Tables	xi
Acknowledgements	xii
Vita	xv
Abstract of the Dissertation	xviii
Chapter 1 Introduction	1
1.1 Accelerating Hyperdimensional Computing with GPUs	3
1.2 Graph-based Learning with HDC	4
1.3 HDC for Mass Spectrometry-based Proteomics	5
Chapter 2 A GPU-Powered Framework for Hyperdimensional Computing	7
2.1 Introduction	7
2.2 Background	10
2.2.1 Preliminaries of Hyperdimensional Computing	10
2.2.2 ML with HDC	12
2.3 OpenHD Framework	13
2.3.1 OpenHD Workflow	15
2.3.2 Data Type Definition	16
2.3.3 Access Fusion	16
2.3.4 Automated Data Type Mutation	17
2.3.5 Optimization of Built-in Functions	19
2.4 OpenHD Optimization Techniques	20
2.4.1 PARTRAIN: Parallel Training for HDC Classification	20
2.4.2 HDC-Based Memory Optimization	21
2.5 Evaluation	24
2.5.1 Experimental Setup	24
2.5.2 Comparison of HDC to the State-of-the-art ML	26
2.5.3 Accuracy of PARTRAIN	27
2.5.4 OpenHD Efficiency	28

	2.5.5	Effectiveness of ENC-STREAMER and L2-RECYCLE	32
	2.5.6	Code Generation Overhead	34
	2.5.7	Comparison with Other Hardware Implementation	34
	2.5.8	Evaluation on High-performance Hardware	36
	2.5.9	Case Study: OpenHD on Other HDC Tasks	38
	2.6	Related Work	39
	2.6.1	Automated GPU Acceleration	39
	2.6.2	Hardware Acceleration of HDC	40
	2.7	Conclusion	41
Chapter 3		A Graph-based Machine Learning in High-dimensional Space	43
	3.1	Introduction	43
	3.2	Background and Related Work	45
	3.2.1	HDC-based ML on Graph Domain	45
	3.2.2	In-memory Computing with FeFET	46
	3.3	RelHD Algorithm Overview	47
	3.3.1	Encoding Node Features	49
	3.3.2	Relation Embedding	50
	3.3.3	Training and Inference	51
	3.4	FeFET-based PIM Acceleration for RelHD	51
	3.4.1	Cluster-based PIM Architecture	52
	3.4.2	Processing Flow	55
	3.4.3	Graph-aware Optimization	57
	3.5	Evaluation	59
	3.5.1	Experimental Setup	59
	3.5.2	Comparison to Baselines running on GPU	61
	3.5.3	FeFET-based RelHD Accelerator Efficiency	62
	3.5.4	Comparison to the state-of-the-art GNN accelerator	63
	3.5.5	RelHD Scalability	64
	3.5.6	Overhead Analysis	64
	3.5.7	RelHD Accuracy & Dimensionality	65
	3.6	Conclusion	65
Chapter 4		Massively Parallel Open Modification Spectral Library Searching with Hyperdimensional Computing	67
	4.1	Introduction	67
	4.2	Background and Related Work	72
	4.2.1	Mass Spectrometry Background	72
	4.2.2	Accelerated Spectral Library Searching	73
	4.2.3	Memory-centric Computing and HDC	74
	4.3	HyperOMS Algorithm	74
	4.3.1	Data Preprocessing	75
	4.3.2	Encoding: Spectrum Vectors to Hypervectors	77

4.3.3	Hamming Similarity Search	80
4.4	HyperOMS Hardware Architecture	81
4.4.1	Motivation	81
4.4.2	HyperOMS Accelerator Design	82
4.4.3	In-memory Encoding	84
4.4.4	Near-memory Hamming Similarity Search	86
4.4.5	Data Organization Optimization	90
4.5	Evaluation	93
4.5.1	Methodology	93
4.5.2	Impact of Encoding Configuration	96
4.5.3	Search Quality	97
4.5.4	HyperOMS Performance Comparison on GPU	99
4.5.5	Performance Improvement of HyperOMS Accelerator	100
4.5.6	Hypervector Folding Parameter Search	101
4.5.7	Effectiveness of Optimization Scheme	102
4.5.8	Scalability and Overhead Analysis	103
4.6	Conclusion	104
Chapter 5	Summary and Future Work	106
5.1	Dissertation Summary	107
5.2	Future Directions	108
Bibliography		110

LIST OF FIGURES

Figure 2.1:	Overall execution workflow of OpenHD.	13
Figure 2.2:	Example implementation using OpenHD. <code>@hd.run</code> decorator passes the entire function component to the abstraction layer of OpenHD. <code>add_row</code> adds the hypervector <code>id_test</code> for each row of hypermatrix <code>id_base_hx</code>	14
Figure 2.3:	The example of data type mutation.	18
Figure 2.4:	Overview of the proposed parallel training method.	19
Figure 2.5:	OpenHD of ENC-STREAMER-applied HDC encoding.	21
Figure 2.6:	Design of L2-RECYCLE-applied training module in HDC-based classification.	23
Figure 2.7:	Comparison between OpenHD-based HDC and the state-of-the-art ML method. (a) Model size of DNN-based vs. HDC-based classification (b) Runtime of DNN-based vs. HDC-based classification (c) K-means vs. HDC-based clustering	26
Figure 2.8:	Accuracy comparison between PARTRAIN and the conventional HDC training method.	28
Figure 2.9:	Speed comparison of OpenHD vs. baseline HDC implementations.	29
Figure 2.10:	Speed comparison of OpenHD vs. TF2 GPU-based HDC clustering.	31
Figure 2.11:	Energy consumption comparison of GPU (OpenHD) vs. HDC running on the embedded CPU.	32
Figure 2.12:	Execution time reduction by proposed optimizations.	33
Figure 2.13:	OpenHD inference compared to FPGA and ASIC.	35
Figure 2.14:	OpenHD training compared to FPGA.	36
Figure 2.15:	Improvement of OpenHD compared to baselines running on high-end hardware.	37
Figure 2.16:	Stochastic nature of HDC for other cognitive tasks.	39
Figure 3.1:	Two types of FeFET PIM block.	46
Figure 3.2:	Overview of node classification problem	47
Figure 3.3:	Overview of encoding and relation embedding stage in RelHD. (a) Node feature encoding (b) Aggregation of 1-hop neighbor information (c) Aggregation of 2-hop neighbor information. Green-colored hypervectors are used for relation hypervector generation (Eq. 3.1).	48
Figure 3.4:	Architecture of RelHD PIM accelerator.	53
Figure 3.5:	Conflict issue and the proposed optimization scheme.	56
Figure 3.6:	Improvement of RelHD over baselines on GPU. For RD, GCN and GAT on GPU ran out of memory (OOM).	60
Figure 3.7:	Performance of RelHD PIM accelerator versus RelHD GPU implementation	62
Figure 3.8:	The scalability of RelHD.	64
Figure 3.9:	Accuracy comparison. (a) RelHD accuracy according to hypervector dimensionality (D) (b) RelHD versus baselines. OOM denotes out of memory.	65

Figure 4.1:	Overview of spectral library search. Standard search uses a narrow precursor m/z tolerance, while OMS uses a wide precursor m/z tolerance during the searching.	69
Figure 4.2:	Overview of OMS process using HyperOMS.	75
Figure 4.3:	Data preprocessing in HyperOMS.	76
Figure 4.4:	Encoding in HyperOMS.	77
Figure 4.5:	Position hypervector generation. (a) Strategy overview. (b) Pairwise similarity (Hamming similarity normalized by the hypervector dimension size) between position hypervectors.	78
Figure 4.6:	Hamming similarity search in HyperOMS.	80
Figure 4.7:	Overview of the HyperOMS accelerator architecture. Encoding and Hamming similarity search step is performed in PuM and NMP, respectively. . .	82
Figure 4.8:	Per-stage profiling on NVIDIA Geforce GTX 1080Ti	83
Figure 4.9:	In-memory encoding scheme for spectrum vectors.	85
Figure 4.10:	Statistics of Hamming similarity values. (a) Relevance between θ_D and $\theta_{D/2}$. (b) Similarity between query and matched reference spectrum (normalized to D).	88
Figure 4.11:	Near-memory HSE design.	89
Figure 4.12:	(a) Uniform mapping. (b) Proposed reference data mapping scheme. (c) Address mapping process and reference index retrieving process based on (b). . .	91
Figure 4.13:	Search quality comparison on the small scale dataset. (a) HyperOMS with different hypervector dimensionality. (b) HyperOMS with different number of bit flips. (c) HyperOMS with various encoding strategies. (d) Results of baseline tools.	96
Figure 4.14:	Search result analysis from HyperOMS and baseline tools on the large scale dataset. (a) Total number of identifications. (b) Venn diagram of b1927 query search result.	97
Figure 4.15:	Performance and energy efficiency comparison.	99
Figure 4.16:	HyperOMS runtime breakdown.	101
Figure 4.17:	Hypervector folding parameter search. (a) Hypervector folding success rate (FSR) according to K, T . (b) Parameter selection.	101
Figure 4.18:	Speed comparison of HyperOMS with different configurations. (a) The cumulative effects of optimization strategies (A: Data organization optimization, B: Hypervector folding). (b) Impact of the number of banks.	102

LIST OF TABLES

Table 2.1:	HDC framework on various hardware platforms	8
Table 2.2:	The OpenHD API examples	17
Table 2.3:	Statistics of the datasets	25
Table 3.1:	Architecture Parameters	59
Table 3.2:	Statistics of the datasets	60
Table 4.1:	System specifications of HyperOMS accelerator.	94
Table 4.2:	Spectrum preprocessing settings.	95

ACKNOWLEDGEMENTS

First and foremost, I would like to appreciate my advisor, Prof. Tajana Rosing, for her invaluable guidance, unwavering support, and continuous encouragement throughout the development of this thesis. Her insightful critiques and wholehearted encouragement have been instrumental in shaping the direction and outcome of my research. Without her mentorship, my Ph.D. experience would not have been as successful and memorable as it has been. I would also like to thank my doctoral committee members, Prof. Mingu Kang, Prof. Duygu Kuzum, and Prof. Jishen Zhao, for their valuable feedback that helped me to improve my dissertation.

I would like to extend my sincerest appreciation to my colleagues in System Energy Efficiency Lab (SEELab) and our collaborators for their constructive feedback, team effort, and friendly atmosphere. Their contributions and suggestions have greatly enhanced the quality of my research. Specifically, I would like to thank Prof. Yeseong Kim for his mentorship that helped me adjust to research and the lab during my early Ph.D. studies, and Dr. Behnam Khaleghi, Minxuan Zhou, and Weihong Xu for their collaboration and help in many of my projects. Also, I would like to thank Prof. Niema Moshiri and Prof. Wout Bittremieux for their support and encouragement in pursuing interdisciplinary research.

During my Ph.D. journey, I had the privilege to experience a distinctive blend of both academic and industry-based research. I would like to appreciate Dr. Euicheol Lim, Dr. Byung Il Koh, Dr. Joonseop Sim, and Jungmin Choi at SK Hynix; Dr. Adam Li and Hai Xiao at AMD; Cecile Foret, Dr. Zheng Li, Dr. Zhile Ren, and Matthew Kay Fei Lee at Apple; Dr. Ameen Akel, Sean Eilert, Justin Eno, and Ken Curewitz at Micron. The hands-on experience and professional insights gained through the collaboration have greatly motivated my research.

I thank my dearest friends in San Diego for their friendship and the memories we have shared. Specifically, I would like to appreciate Dr. Byung Hoon Ahn for being such a wonderful friend and mentor, who always gave me constructive advice both on the research side and in life. I appreciate all my friends whose support and vivacity sustained me on a daily basis.

Last but not least, I owe a debt of gratitude to my family: my father, Insung Kang; my mother, Youngman Choi; my sister, Yunjin Kang; brother-in-law, Donghyun Lee; my niece, Chaerin Lee, who has provided me with constant love and support throughout my academic pursuits. Their constant encouragement and belief in me have been a source of strength and motivation and made this achievement possible.

My research was made possible by funding from National Science Foundation (NSF) Grant 1730158, 1826967, 1911095, 2003279, 2052809, 2100237, 2112167, 2112665, and 2120019; CRISP, one of six centers in JUMP; PRISM and CoCoSys, centers in JUMP 2.0, an Semiconductor Research Corporation (SRC) program sponsored by Defense Advanced Research Projects Agency (DARPA); and SRC-Global Research Collaboration (GRC) grant.

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

Chapter 2 contains material from “XCellHD: An Efficient GPU-Powered Hyperdimensional Computing with Parallelized Training”, by Jaeyoung Kang, Behnam Khaleghi, Yeseong Kim, and Tajana Rosing, which appears in Asia and South Pacific Design Automation Conference (ASP-DAC), 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 2 contains material from “OpenHD: A GPU-Powered Framework for Hyperdimensional Computing”, by Jaeyoung Kang, Behnam Khaleghi, Tajana Rosing, and Yeseong Kim, which appears in IEEE Transactions on Computers (TC), 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 3 contains material from “RelHD: A Graph-based Learning on FeFET with Hyperdimensional Computing”, by Jaeyoung Kang, Minxuan Zhou, Abhinav Bhansali, Weihong Xu, Anthony Thomas, and Tajana Rosing, which appears in IEEE International Conference on Computer Design (ICCD), 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 4 contains material from “Massively Parallel Open Modification Spectral Library Searching with Hyperdimensional Computing”, by Jaeyoung Kang, Weihong Xu, Wout Bittrem-

ieux, and Tajana Rosing, which appears in International Conference on Parallel Architectures and Compilation Techniques (PACT), 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 4 contains material from “DRAM-based Acceleration of Open Modification Search for Mass Spectrometry-based Proteomics”, by Jaeyoung Kang, Weihong Xu, Wout Bittremieux, Niema Moshiri, and Tajana Rosing, which is under submission. The dissertation author was the primary investigator and author of this paper.

VITA

2019	Bachelor of Engineering (Great Honors), Korea University
2021	Master of Science, University of California San Diego
2023	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

Weihong Xu, Jaeyoung Kang, Wout Bittremieux, Niema Moshiri, and Tajana Rosing, “HyperSpec: Ultra-fast Mass Spectra Clustering in Hyperdimensional Space”, *Journal of Proteome Research*, 2023.

Weihong Xu, Jaeyoung Kang, and Tajana Rosing, “FSL-HD: Accelerating Few-Shot Learning on ReRAM using Hyperdimensional Computing”, *Design Automation and Test in Europe Conference (DATE)*, 2023.

Derek Jones, Jonathan E. Allen, Xiaohua Zhang, Behnam Khaleghi, Jaeyoung Kang, Weihong Xu, Niema Moshiri, and Tajana Rosing, “HD-Bind: Encoding of Molecular Structure with Low Precision, Hyperdimensional Binary Representations”, *arXiv preprint arXiv:2303.15604*, 2023.

Rebecca Fielding-Miller, Smruthi Karthikeyan, Tommi Gaines, Richard S Garfein, Rodolfo A Salido, Victor J Cantu, Laura Kohn, Natasha K Martin, Adriane Wynn, Carrissa Wijaya, Marlene Flores, Vinton Omaleki, Araz Majnoonian, Patricia Gonzalez-Zuniga, Megan Nguyen, Anh V Vo, Tina Le, Dawn Duong, Ashkan Hassani, Samantha Tweeten, Kristen Jepsen, Benjamin Henson, Abbas Hakim, Amanda Birmingham, Peter De Hoff, Adam M Mark, Chanond A Nasamran, Sara Brin Rosenthal, Niema Moshiri, Kathleen M Fisch, Greg Humphrey, Sawyer Farmer, Helena M Tubb, Tommy Valles, Justin Morris, Jaeyoung Kang, Behnam Khaleghi, Colin Young, Ameen D Akel, Sean Eilert, Justin Eno, Ken Curewitz, Louise C Laurent, Tajana Rosing, Rob Knight, Nathan A Baer, Tom Barber, Anelizze Castro-Martinez, Marisol Chacón, Willi Cheung, Evelyn S Crescini, Emily R Eisner, Lizbeth Franco Vargas, Charlotte Hobbs, Alma L Lastrella, Elijah S Lawrence, Nathaniel L Matteson, Karthik Gangavarapu, Toan T Ngo, Phoebe Seaver, Elizabeth W Smoot, Rebecca Tsai, Bing Xia, Stefan Aigner, Catelyn Anderson, Pedro Belda-Ferre, Shashank Sathe, Mark Zeller, Kristian G Andersen, Gene W Yeo, Ezra Kurzban, “Safer at school early alert: an observational study of wastewater and surface monitoring to detect COVID-19 in elementary schools”, *The Lancet Regional Health – Americas*, 2023.

Jaeyoung Kang, Weihong Xu, Wout Bittremieux, and Tajana Rosing, “Massively Parallel Open Modification Spectral Library Searching with Hyperdimensional Computing”, *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2022.

Jaeyoung Kang, Minxuan Zhou, Abhinav Bhansali, Weihong Xu, Anthony Thomas, and Tajana Rosing, “RelHD: A Graph-based Learning on FeFET with Hyperdimensional Computing”, *IEEE International Conference on Computer Design (ICCD)*, 2022.

Jaeyoung Kang, Behnam Khaleghi, Tajana Rosing, and Yeseong Kim, “OpenHD: A GPU-Powered Framework for Hyperdimensional Computing”, IEEE Transactions on Computers (TC), 2022.

Jaeyoung Kang, Behnam Khaleghi, Yeseong Kim, and Tajana Rosing, “XCellHD: An Efficient GPU-Powered Hyperdimensional Computing with Parallelized Training”, Asia and South Pacific Design Automation Conference (ASP-DAC), 2022.

Behnam Khaleghi, Jaeyoung Kang, Hanyang Xu, Justin Morris, and Tajana Rosing, “GENERIC: Highly Efficient Learning Engine on Edge using Hyperdimensional Computing”, Design Automation Conference (DAC), 2022.

Weihong Xu, Jaeyoung Kang, and Tajana Rosing, “A Near-Storage Framework for Boosted Data Preprocessing of Mass Spectrum Clustering”, Design Automation Conference (DAC), 2022.

Rishikanth Chandrasekaran, Kazim Ergun, Jihyun Lee, Dhanush Nanjunda, Jaeyoung Kang, and Tajana Rosing, “FHDnn: Communication Efficient and Robust Federated Learning for AIoT Networks”, Design Automation Conference (DAC), 2022.

Behnam Khaleghi, Uday Mallappa, Duygu Yaldiz, Haichao Yang, Monil Shah, Jaeyoung Kang, and Tajana Rosing, “PatterNet: Explore and Exploit Filter Patterns for Efficient Deep Neural Networks”, Design Automation Conference (DAC), 2022.

Minxuan Zhou, Weihong Xu, Jaeyoung Kang, and Tajana Rosing, “TransPIM: A Memory-based Acceleration via Software-Hardware Co-Design for Transformer”, IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2022.

George Armstrong, Cameron Martino, Justin Morris, Behnam Khaleghi, Jaeyoung Kang, Jeff DeReus, Qiyun Zhu, Daniel Roush, Daniel McDonald, Antonio Gonazlez, Justin P Shaffer, Carolina Carpenter, Mehrbod Estaki, Stephen Wandro, Sean Eilert, Ameen Akel, Justin Eno, Ken Curewitz, Austin D Swafford, Niema Moshiri, Tajana Rosing, and Rob Knight, “Swapping metagenomics preprocessing pipeline components offers speed and sensitivity increases”, American Society for Microbiology mSystems Journal, 2022.

Saransh Gupta, Behnam Khaleghi, Sahand Salamat, Justin Morris, Ranganathan Ramkumar, Jeffrey Yu, Aniket Tiwari, Jaeyoung Kang, Mohsen Imani, Baris Akansali, and Tajana Rosing, “Store-n-Learn: Classification and Clustering with Hyperdimensional Computing across Flash Hierarchy”, ACM Transactions on Embedded Computing Systems (TECS), 2022.

Saransh Gupta, Mohsen Imani, Joonseop Sim, Andrew Huang, Fan Wu, Jaeyoung Kang, Yeseong Kim, and Tajana Rosing, “COSMO: Computing with Stochastic Numbers in Memory”, ACM Journal on Emerging Technologies in Computing Systems (JETC), 2022.

Sahand Salamat, Jaeyoung Kang, Yeseong Kim, Mohsen Imani, Niema Moshiri, and Tajana Rosing, “FPGA Acceleration of Protein Back-Translation and Alignment”, Design Automation and Test in Europe Conference (DATE), 2021.

Yunhui Guo, Mohsen Imani, Jaeyoung Kang, Yeseong Kim, Sahand Salamat, Justin Morris, Baris Aksanli, and Tajana Rosing, “HyperRec: Efficient Recommender Systems with Hyperdimensional Computing”, Asia and South Pacific Design Automation Conference (ASP-DAC), 2021.

Jaeyoung Kang, Illsoo Sohn, and Sang Hyun Lee, “Enhanced Message-Passing Based LEACH Protocol for Wireless Sensor Networks”, Sensors, 2018.

ABSTRACT OF THE DISSERTATION

Efficient Computing in Hyperdimensional Space

by

Jaeyoung Kang

Doctor of Philosophy in Electrical Engineering (Communication Theory & Systems)

University of California San Diego, 2023

Professor Tajana Šimunić Rosing, Chair

The rapid proliferation of data has expedited the development of cutting-edge machine learning (ML) methods, including deep neural networks, to find underlying insights from data. However, the large amount of data movement has driven up energy requirements, posing a new threat of surpassing global energy production. In this dissertation, we explore a novel computing paradigm, hyperdimensional computing (HDC), which mimics attributes of the human brain’s neuronal circuits with lightweight arithmetic on low-precision high-dimensional vectors and significantly enhances speed and energy efficiency. We present OpenHD, a GPU-based software infrastructure for HDC, which automatically generates optimized GPU code of HDC applications with the Just-in-Time compilation to ease the development and optimization efforts

for the deployment. The proposed framework achieves $4.5\times$ and $146\times$ speedup on average for GPU-based HDC classification and clustering, respectively.

Using OpenHD, we first expand the variety of data structures that can be supported in HDC applications. We design an HDC-based ML solution that supports graph data, and implement it on GPU. Our solution, RelHD, enables graph-based ML by aggregating the relationship between data and features of data into a single high-dimensional vector. We further accelerate RelHD using processing in-memory (PIM) based on FeFET. The PIM accelerator offers $10\times$ speedup and $986\times$ energy efficiency improvement over the state-of-the-art crossbar memory-based graph neural network accelerator.

While most existing HDC-based applications have used small-scale datasets, we show that HDC can scale to tackle the large-scale problem. We present an HDC-based approach called HyperOMS for large-scale open modification spectral library searching (OMS) in mass spectrometry-based proteomics analysis. We develop a novel HDC-based OMS algorithm and accelerate it on GPU using the OpenHD framework. To run HyperOMS efficiently, we devise a DRAM-based PIM accelerator with optimization strategies to maximize parallelism. Evaluation results show that the accelerator yields up to $100\times$ speedup and has $1,337\times$ higher energy efficiency over the state-of-the-art OMS tool running on GPU while offering comparable search quality to competing solutions.

Chapter 1

Introduction

Over the last decade, there has been exponential growth in data generated from various sources, such as Internet-of-Things (IoT) devices. In order to draw meaningful and useful insights from the data deluge, advanced machine learning (ML) techniques, e.g., deep neural networks (DNNs), have been developed, which involve complex execution pipelines and a large number of computations. The proliferation of high-performance hardware [1–3] helped to handle massive parallelism and run these algorithms in a reasonable time. However, running them accompanies high energy consumption and excessive carbon emission [4–7]. An emerging computing paradigm based on hyperdimensional computing (HDC) is lightweight and capable of parallelization offering promising opportunities for significantly enhancing energy efficiency.

HDC is based on a long-term memory model, Sparse Distributed Memory (SDM) [8], which emulates characteristics of the human brain’s neuronal circuits with vector operations in a high-dimensional (HD) space. A key difference from the conventional computing method that uses real numbers is that the HDC uses *hypervectors*. A hypervector includes the pattern of information encoded into tens of thousands of bits.

Compared to the conventional DNN-based solutions, HDC has several advantages: (i) it is highly parallelizable [9] mainly due to the coordinate-wise independent nature of its operations,

(ii) it can perform training and inference with the energy-efficient computation, and (iii) the hypervector representation is robust to noise [10]. Many applications have been proposed based on HDC with high accuracy and efficiency, e.g., DNA sequencing [11,12], language recognition [13], robotics [14], activity identification and voice recognition [15], multimodal sensor fusion [16], recommender system [17], and bio-signal processing [18]. Several companies are also actively studying HDC to implement more generalizable learning, e.g., Google [19], IBM [20], and Numenta [21].

HDC reduces the complexity of the operations required to process data and uses ultra-low precision (as low as 1-bit) numbers. Since HDC deals with large hypervectors, the conventional CPU-centric architecture is not suitable for such computations. HDC is easily parallelizable and can benefit from hardware accelerators. Earlier works have focused on new hardware accelerator designs for HDC, e.g., application-specific integrated circuit (ASIC) [9, 13, 22–24] and processing in-memory (PIM) accelerator [25–29], showing promising speedup and energy efficiency over the CPU-based HDC. Nevertheless, HDC confronts the following challenges in a real-world deployment.

- **Hardware availability:** The design of ASICs and PIM for HDC takes a long time to synthesize and fabricate and is very costly, limiting the development and deployment of new HDC algorithms. HDC applications require flexibility, e.g., hypervector lengths, the precision of operations, and encoding algorithms, which are infeasible in existing HDC accelerators. It is also important to be able to support high-precision numbers as the state-of-the-art HDC algorithms benefit from high-precision encoding to improve accuracy [28, 30–32].
- **Limited support on structured data:** Existing HDC solutions are designed as an alternative to classical ML, which assumes that the input data is represented with simple arrays. However, real-world data cannot always be expressed with an array or tabular. For example, a graph is needed to model complex relationships and dependencies between

entities in a broad spectrum of domains, e.g., social networks, and molecular structure.

- **Large-scale applications:** Current HDC applications have shown that existing algorithms can be redesigned to be highly parallelizable, lightweight, and use only a small number of computations, but they mostly focus on small-scale problems, such as training and inference on IoT device and pattern matching [11, 12, 33]. HDC-based solutions for large-scale applications have not been studied, which is essential as many real-world problems require processing and analyzing massive amounts of data.

In this dissertation, we present solutions for the above challenges. To address the hardware availability problem, we first build a graphics processing unit (GPU)-based software framework for HDC called OpenHD, which automatically generates optimized GPU code for HDC applications. In turn, we develop a learning solution for the graph domain called RelHD, which is useful for modeling complex relationships and dependencies between entities using HDC operations. As a proof of concept, we implement it on GPU using the OpenHD framework and devise a PIM accelerator tailored for the algorithm. Finally, we show that HDC can successfully scale to large-scale datasets. We implement an HDC-based solution for mass spectrometry-based proteomics on GPU using OpenHD and further accelerate the HDC-based open modification search (OMS) using a DRAM-based accelerator. In the rest of the introduction, we discuss the contributions of this dissertation in more detail.

1.1 Accelerating Hyperdimensional Computing with GPUs

HDC has emerged as an alternative lightweight learning solution to deep neural networks. A key characteristic of HDC is the great extent of parallelism that can facilitate hardware acceleration. Among various parallel hardware platforms, GPUs can offer programmability with high parallelism and superior efficiency on high-precision numbers. Even though GPU generally have high power consumption, the off-the-shelf low-power embedded GPU platforms

like NVIDIA Jetson [34–37] are readily available on the market. However, previous hardware implementations of HDC seldom focus on GPU designs [9], which were also inefficient partly due to the complexity of accelerating HDC on GPUs.

We present OpenHD in Chapter 2, a flexible and high-performance GPU-powered framework for automating the mapping of general HDC applications, including classification and clustering to GPUs. OpenHD takes advantage of memory optimization strategies specialized for HDC, minimizing the access time to different memory subsystems and removing redundant operations. We also propose a novel training method to enable data parallelism in HDC training.

Our evaluation result shows that the proposed training rapidly achieves the target accuracy, reducing the required training epochs by $4\times$. With OpenHD, users can deploy GPU-accelerated HDC applications without domain expert knowledge. Compared to the state-of-the-art GPU-powered HDC implementation [38], our evaluation on NVIDIA Jetson TX2 shows that OpenHD is up to $10.5\times$ and $314\times$ faster for HDC-based classification and clustering, respectively. Compared with DNN-based classification and K-means clustering on GPUs, OpenHD-based HDC is $11.7\times$ and $53\times$ faster at comparable accuracy.

1.2 Graph-based Learning with HDC

Advances in graph neural network (GNN)-based algorithms enable machine learning on graphs by analyzing latent relationships between entities. It has been used to analyze social networks [39, 40], protein interactions [41, 42], and disease classification [43]. However, GNNs are computationally demanding since they rely upon backpropagation over the graph data, that has sparse and irregular characteristics.

In Chapter 3, we propose a lightweight graph-based ML framework based on HDC called RelHD. It maps the features of each node into an HD space and embeds relationships between nodes. Using lightweight HDC operations, RelHD enables ML training and inference on graph

data without backpropagation, which leads to significant speedup and enhanced energy efficiency compared to existing GNN-based solutions. We implement the proposed algorithm based on our GPU-based HDC framework, OpenHD. We also design a scalable PIM architecture based on the emerging FeFET technology to implement RelHD and optimize data allocation and operation scheduling to maximize the accelerator performance by addressing the sparseness and irregularity of the graph. Our evaluation results show that RelHD offers comparable accuracy to the popular GNN-based algorithms [44, 45] while being up to $32\times$ faster on GPUs. Also, our FeFET-based accelerator is $33\times$ faster and $59,287\times$ more energy efficient on average over RelHD on GPU. It is $10\times$ faster and $986\times$ more energy efficient on average compared to the state-of-the-art in-memory processing-based GNN accelerator [46].

1.3 HDC for Mass Spectrometry-based Proteomics

HDC has also been used for pattern-matching tasks in addition to classical learning tasks, for example, DNA pattern matching [11] and sequence matching [12]. In Chapter 4, we show that HDC can run effectively on the large-scale application, open modification spectral library search for mass spectrometry-based proteomics. HDC maximizes parallelism, enables the utilization of lightweight operations, and simplifies the execution pipeline.

Mass spectrometry, commonly used for protein identification, generates a massive number of spectra that need to be matched against a large database. In reality, most of them remain unidentified or mismatched due to unexpected post-translational modifications. Open modification search (OMS) has been proposed as a strategy to improve the identification rate by considering changes in spectra, but it expands the search space exponentially [47–49].

We propose HyperOMS, an algorithm-hardware co-design for boosted OMS, to cope with the enlarged database and expanded search space. HyperOMS algorithm encodes spectral data into binary vectors and performs the efficient OMS in high-dimensional space. We accelerate

the HyperOMS algorithm using a DRAM-based PIM accelerator, which combines in-memory processing and near-memory processing technologies. In order to maximize the parallelization and efficiency of the accelerator, we optimize the data allocation and devise an approximation strategy for similarity computation. Experimental results show that HyperOMS accelerator yields up to $6.9\times$ speedup and $341\times$ higher energy efficiency compared to running HyperOMS on GPU, and up to $100\times$ speedup and $1337\times$ higher energy efficiency over the state-of-the-art OMS tool, ANN-SoLo [50], while providing comparable search quality to competing tools.

Chapter 2

A GPU-Powered Framework for Hyperdimensional Computing

2.1 Introduction

The growing demand for real-time data analysis has accelerated the trend of on-device ML systems. Deep learning (DL) is a popular method to extract higher-level features but running those workloads on conventional systems results in high energy consumption and slow processing speed. Furthermore, mobile devices need to perform online learning tasks in various environments, but data can be vulnerable to noise. The limitations lead researchers to seek alternative computing methodologies for lightweight learning like HDC. HDC enables various cognitive tasks with hypervector arithmetic, e.g., adding multiple hypervectors to memorize them and learning their dominant pattern with a single hypervector.

To run the HDC on the GPU, one could integrate HDC with the parallel computing models oriented to vector processing, e.g., TensorFlow. Recent work [9] used TensorFlow on embedded GPU, but it showed a comparable speed to CPU-based implementation. In our initial experiment, TensorFlow XLA-based HDC classification running on NVIDIA Jetson TX2 was only $1.74\times$

Table 2.1: HDC framework on various hardware platforms

	Platform	Adaptive Code Generation	Precision Support	Implementation Flexibility
F5-HD [51]	FPGA	O	Fixed-point	X
HD-Core [52]	FPGA	X	Fixed-point	X
BinHD [22]	ASIC	X	Binary	X
XCelHD [38]	GPU	X	Binary, Fixed-point, FP32	X
OpenHD (Proposed)	GPU	O	Binary, Fixed-point, FP32	O

faster than CPU, with a low GPU utilization of at most 40%. It creates a lot of HDC data transfer between CPU and GPU, which causes a memory bottleneck. Furthermore, TensorFlow-based HDC often fails to effectively leverage the memory hierarchy on the GPU since HDC data are large-sized and have specific memory access patterns.

The state-of-the-art GPU-based HDC solution, XCelHD [38], introduced a way to optimize HDC classification on GPU, achieving up to $35\times$ speedup over the TensorFlow-based HDC implementation. However, it fails to achieve optimal performance when data characteristics, device architecture, or HDC algorithm change. A hand-optimizing kernel requires a lot of effort and is not flexible to data characteristics, HDC algorithm, and hardware changes. As such, it is critical to find optimal configurations and parameters by adaptively reacting to data, hardware, and HDC design rather than relying on a hand-optimized kernel. Also, as we attribute the success of many ML techniques to the development of easy-to-use, flexible, and efficient libraries like TensorFlow, the HDC method also requires a readily-available environment to experiment with new ideas on the GPU. To this end, it is essential to develop a new library specifically designed for HDC, to leverage the parallel computation pattern in an efficient, flexible, and automated manner.

In this chapter, we present a GPU-powered HDC framework, dubbed OpenHD. Unlike existing HDC accelerators based on FPGA and ASIC, OpenHD supports various types of HDC-based algorithms, including classification and clustering (see Table 2.1). The proposed solution

offers a software interface capable of automatically generating optimized CUDA code of commonly used HDC applications with the Just-in-Time (JIT) compilation to ease the development and optimization efforts. OpenHD adopts various optimization strategies that effectively mitigate the low parallelization and memory access issues of the existing HDC-based applications. Furthermore, we propose a highly parallel HDC training, called PARTRAIN, that elevates the GPU parallelism by redesigning the HDC training algorithm. It converts the sequential HDC training process into multiple simultaneous subtasks, each of which trains submodels with different training samples while updating the final model based on the partial models. Our contributions are summarized as follows:

- We present OpenHD for GPUs, which supports various HDC-based learning tasks. In this chapter, we apply OpenHD to two representative HDC-based algorithms: classification and clustering. OpenHD efficiently parallelizes the HDC learning procedure in a GPU-friendly way while intelligently optimizing memory allocation/access patterns.
- Our OpenHD framework automatically (i) extracts the HDC-related parts from a given Python program, and (ii) maps them to highly-optimized CUDA code. It allows users to implement various GPU-accelerated applications with HDC philosophy even when they are unaware of the detailed acceleration mechanism on the underlying GPU subsystems.
- We propose a novel HDC training method for high parallelism. It enables full utilization of hardware resources and reduces the number of required training epochs by $4\times$ on top of the GPU-accelerated HDC.
- Our strategies specially designed for GPU maximize data reuse and enhance cache utilization, which accelerates the prediction (up to $12\times$) and the encoding step (up to $1.94\times$) in HDC with simple parallelization.
- We show that OpenHD achieves comparable accuracy to the state-of-the-art DNN-based

classification and clustering (K-means) algorithms running on the same GPUs but at a much higher speed.

We implemented and evaluated OpenHD with various datasets using NVIDIA Jetson TX2. Our evaluation results show that HDC-based classification has a $9\times$ smaller model and runs $11.7\times$ faster while offering comparable accuracy to the DNN-based method. HDC-based clustering is $53\times$ faster than the K-means algorithm with comparable quality. Moreover, our results indicate that OpenHD is up to $10.5\times$ and $314\times$ faster than the state-of-the-art classification and clustering HDC implementation running on the GPU, respectively.

2.2 Background

2.2.1 Preliminaries of Hyperdimensional Computing

Based on the understanding that brain events involve the simultaneous activity of a massive number of neurons, HDC models it with vectors in a HD space [53]. This HD vector is called *hypervector*, and HDC processes data in the unit of hypervectors.

Hypervector: The human memory associates different information and comprehends the relationship between them. HDC represents each datum with a *hypervector* and measures the correlation with the distance in the HD space. For example, let there exist two hypervectors with D -dimensionality, and each has random components between $+1$ and -1 , i.e., $\{+1, -1\}^D$. In the vector space, the two bipolar hypervectors are near-orthogonal. We can represent two distinct items using the two randomly generated hypervectors. Generally speaking, finding the relationship and distinguishing information is done by measuring the similarity between a pair of hypervectors, $\delta(\vec{H}_1, \vec{H}_2)$, where $\vec{H}_1$ and $\vec{H}_2$ are two hypervectors. Existing works primarily use three different metrics: hamming distance for binary hypervectors, dot product similarity, and cosine similarity for non-binary hypervectors [54]. Note that the hypervector is a holographic

representation of information in that there is no specific important dimension in a hypervector. This independence property of hypervector allows resiliency to corruption in components.

Addition (bundling) / Multiplication (binding): Using the element-wise addition and multiplication between hypervectors, we can *memorize* and *associate* different information. The element-wise addition operation, called bundling, produces a hypervector that preserves all similarities of the combined members, mimicking the memorization functionality. The element-wise multiplication, called binding, results in a hypervector that associates two different hypervectors. The resulting hypervector represents new information, which is near-orthogonal (dissimilar) to both of the associated hypervectors.

Permutation: To process sequential information, the HDC has an operation called permutation. The permutation operation, which is denoted by $\rho^n(\vec{H})$, shuffles components of $\vec{H}$ with n -bit(s) rotation. The results of the operation creates a near-orthogonal, i.e., $\delta(\rho^n(\vec{H}), \vec{H}) \simeq 0$ for non-zero n . Also, the results of permutation is reversible, meaning that $\rho^{-n}(\rho^n(\vec{H})) = \vec{H}$. Hence, the permutation can be utilized to ordered information. For instance, $\rho^1(\vec{H}_1)$, $\rho^2(\vec{H}_2)$ and $\rho^3(\vec{H}_3)$ may represent a sequence of three words.

Flip: To create hypervectors that maintain a certain degree of similarity, HDC uses the flip operation. Let $\vec{H}$ be a hypervector randomly sampled. If we flip $D/2$ elements, e.g., changing the sign bit of -1 and $+1$, the result hypervector is orthogonal to the original hypervector; flipping $D/4$ elements creates a hypervector that is 50% similar to the original $\vec{H}$ in the vector space. Using this property, we can represent different levels of information ranging from 0% to 100%. We first create random hypervectors for 0%, which is denoted by $\vec{H}_0$. Then, for the target level p in percentage, $\vec{H}_p$, we can represent this by flipping $(D/2) \times (p/100)$ elements of $\vec{H}$.

With these simple hypervector operations, prior research has shown that it is possible to implement diverse cognitive tasks. In the following section, we discuss how they solve the classical ML problems based on HDC.

2.2.2 ML with HDC

Classification

In [11, 15, 55–57], the authors employ HDC to perform classification. It consists of the encoding, training, and inference stage. **Encoding** maps (encodes) real-world data into the HD space. There are various encoding methods. One of the most popular ones is the ID-level method [15, 27, 51, 55]. For a given feature vector, this encoding method represents each feature value using the flip operation and each feature position using a randomly generated hypervector. It then combines the hypervector pair of the feature position and value using element-wise multiplication. In particular, we encode an input data to an *input hypervector* $\vec{T}$ as follows:

$$\vec{T} = \sum_f \vec{ID}_f \odot \vec{LV}_m \quad (2.1)$$

where $\odot$ indicates an element-wise multiplication. Here, $\vec{ID}_f$ indicates *ID hypervectors* which assigned to an individual f th index of feature. We generate *level hypervectors* $\vec{LV}_m$ to capture different values m in f th index of feature. The ID-level method is more complex but also encompasses all the steps of random projection (RP) encoding, which means that any implementation that can do ID-level can also implement the RP [15].

The **training** stage builds *HDC model* (class hypervectors). We initialize HDC model by combining all hypervectors $\vec{H}_i^k$ belonging each class k using the element-wise addition, i.e., $\vec{C}_k = \sum_i \vec{H}_i^k$ where $\vec{C}_k$ indicates class hypervector for class k . Iterative *training* is used to achieve higher accuracy. HDC calibrates class hypervectors based on the prediction using the current state of class hypervectors. It checks the ground truth label is equivalent to the predicted class. We can make predictions by selecting a class with the maximum similarity between the input hypervector and class hypervectors. If the model gives an incorrect prediction, we update the class hypervectors by subtracting the encoded hypervector from the wrongly predicted class and

adding the pattern again to the correct target class hypervector to amplify the pattern. We can apply the learning rate ($\eta \in (0, 1]$) for scaling the input hypervector during the update [58]. Here, we define *epoch* as an iteration over all data.

The **inference** stage maps test data into query hypervectors using the same method in the encoding stage. The class that shows the maximum similarity to query hypervectors is selected as the prediction result.

Clustering

Earlier work [59] proposed an HDC-based clustering algorithm. After mapping raw data to the HD space, their algorithm performs clustering by iterating two stages: similarity computation and cluster center update. The algorithm aims to learn the representative hypervector for each cluster head. We assign a cluster head with the highest similarity to each data. For the cluster center update stage, the element-wise addition among data that points to the same cluster head is performed. Then, each component of the hypervector is binarized, and that hypervector is assigned as a representative hypervector of the cluster head.

2.3 OpenHD Framework

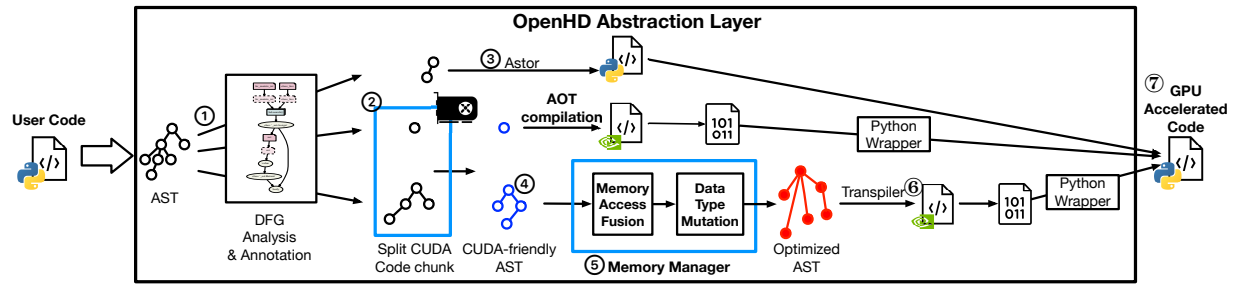


Figure 2.1: Overall execution workflow of OpenHD.

OpenHD provides a runtime environment to implement HDC applications seamlessly integrating with GPU for high efficiency. The OpenHD software interface reduces the user’s

```

1  hd.init(D=D, context=global())
2  @hd.run
3  def add_row(F):
4      id_base_hx = hd.hypermatrix(F)
5      id_test = hd.draw_random_hypervector()
6      for f in range(F):
7          id_base_hx[f] += id_test
8          id_base_hx[f] += id_test
9      return id_base_hx, id_test

```

Figure 2.2: Example implementation using OpenHD. `@hd.run` decorator passes the entire function component to the abstraction layer of OpenHD. `add_row` adds the hypervector `id_test` for each row of hypermatrix `id_base_hx`.

need for understanding the GPU details and minimize the need for hand-optimization. It also enables the implementation of various HDC-based algorithms. The abstraction layer of OpenHD automates the memory allocation of GPU memory and execution of the accelerated code. OpenHD uses Python as the front-end with NumPy-style syntax. At runtime, it automatically generates optimized CUDA code to accelerate the implemented HDC applications on the state-of-the-art GPU systems.

Programming interface: To ease the understanding of how users can implement HDC applications with OpenHD, Figure 2.2 shows an example code. As shown in the example code, users can create either hypermatrix, i.e., an array of hypervectors, or hypervectors, and then perform hypervector arithmetic operations. When the `@hd.run` decorator is applied to the code (Line 1), the abstraction layer of OpenHD performs JIT compilation. It generates highly optimized CUDA code based on HDC characteristics and compiles using the CUDA compiler installed locally. Also, it finds the optimized parameters (kernel configuration) required for execution and binds the execution file to the Python front-end through PyCUDA [60]. Through OpenHD library, users can run GPU-accelerated HDC applications with less effort in a highly optimized fashion.

2.3.1 OpenHD Workflow

Figure 2.1 illustrates how the abstraction layer of OpenHD handles the given Python code. Python code is transformed into abstract syntax trees (AST) by the built-in library. In the process of scanning the AST, OpenHD annotates each node of the AST to identify whether it is a JIT-compatible tree. Then, we build a data flow graph (DFG) using the annotated AST (❶). One of the goals of the abstraction layer is to decide whether the code is to be offloaded or not. In other words, we need to find code chunks that can be efficiently handled with the GPU (❷). Using the results obtained from the static analysis of the AST, we can distinguish those two. First, the code component that needs to be executed on the CPU is regenerated as python code from a node tree using Astor library [61] (❸). The remaining code is converted into the optimized CUDA code. OpenHD’s CUDA code generator takes into account the GPU memory hierarchy (e.g., global memory, shared memory, constant memory) to maximize performance and efficiency.

The AST of the corresponding Python code fragment cannot be used as-is for the CUDA code generation (❹). It needs to be converted into a CUDA-understandable AST. Also, we need to consider the hardware specifications and input data characteristics to generate optimized code. After this step, AST addresses in the annotation nodes are invalidated. Note that the code fragment that handles HDC-specific operations such as permutation and shuffle is substituted with a single HDC operation. Before generating code, the memory manager of OpenHD (❺) considers access fusion to minimize global memory access. Furthermore, we apply data type mutation to save memory allocated to variables and maximize memory coalescing by analyzing and managing the minimum data type required by each variable in OpenHD.

CUDA-compatible AST is transpiled into actual compilable code by the transpiler of OpenHD (❻). Since the generated code cannot be executed standalone, the transpiler adds the required headers and definitions automatically. Then, the final CUDA code is wrapped with PyCUDA-based Python code. Finally, the abstraction layer binds the CPU side code and the GPU code (❼). Instead of running the code generation for all code wrapped with `@hd.run` every

time, OpenHD caches the generated code for future reuses. If same code requests for the JIT compilation, OpenHD intelligently utilizes previously compiled code.

2.3.2 Data Type Definition

HDC’s data units are hypervectors. Thus, we defined two additional built-in data types in OpenHD: `hypervector` and `hypermatrix`. Hypervector is a HD vector with arbitrary D dimensionality. Hypermatrix corresponds to the multiple hypervectors; it can be used to express class hypervectors. HDC uses a fixed dimensionality during application execution. In OpenHD, users declare it in an initializing function that applies to the global scope, and it is used in the subsequent process. The result of an operation like the similarity computation may not follow the hypervector or hypermatrix. In OpenHD, users do not need to declare a variable to store the result explicitly. Instead, variables reside in GPU memory for later computation, and they can be copied to the host through the `to_numpy()` method.

Deploying the hypermatrix/hypervector as built-in data types creates a serious design challenge – hypervectors usually have a large data size, incurring both high communication and computation costs on the GPU subsystems. In the rest of this section, we describe the representative optimization schemes applied in OpenHD.

2.3.3 Access Fusion

To improve kernel performance, it is essential to reduce latency by minimizing access to GPU off-chip global memory, i.e., GDDR. The TensorFlow-based implementation assumes a hypervector as a tensor, so each line of code is represented by a node. Therefore, access to the global memory occurs every time the line is executed, leading to reduced performance. OpenHD uses the NumPy-style syntax for ease of programmability. If the AST is used as-is, the same problem may arise. Therefore, in OpenHD, coping with how the user wrote the code, the global

Table 2.2: The OpenHD API examples

Function	Semantics
<code>hd.init(D, context)</code>	Initialize HDC application with D dimensionality in designated context (e.g., <code>global()</code>)
<code>hd.hypervector()</code> / <code>hd.hypermatrix(N)</code>	Declare hypervector or hypermatrix with N hypervectors.
<code>hd.draw_random_hypervector()</code> / <code>hd.draw_gaussian_hypervector()</code>	Generate a random hypervector from uniform distribution or normal distribution
<code>hd.permute(T, N)</code>	Perform HDC permutation to hypervector T
<code>hd.shuffle(T, S)</code>	Randomly shuffles hypervector T with seed S
<code>hd.search(C, A)</code> / <code>hd.sim(C, A)</code>	Perform associative search between class hypervectors C and hypermatrix or hypervector A
<code>hd.cos(A)</code> / <code>hd.sign(A)</code>	Apply element-wise cosine / sign flip on hypermatrix or hypervector A

memory access corresponding to hypervector or hypermatrix fuses through *memory access fusion* to reduce the memory access. For example, Line 7–8 of the code in Figure 2.2 is fused to a single kernel so that the two consecutive memory accesses are replaced with global memory access and register access.

Given the CUDA-understandable AST, OpenHD generates a control flow graph and traverses it. The variable node that reads or writes occurs is recorded individually, and we can check a propagable path. For example, if multiple reads or writes occur in the same variable without dependency, those nodes can be fused. We reflect changes to the corresponding node in the AST.

2.3.4 Automated Data Type Mutation

For some GPU-powered embedded devices, the CPU and GPU may share the same system memory through unified virtual addressing (UVA), and it is essential to reduce the memory footprint. In HDC, the operation is performed on a HD vector as a unit. If the components of the hypervector use a data type with a large size, the amount of memory occupied by a single hypervector will increase accordingly. To reduce the memory footprint caused by hypervectors,

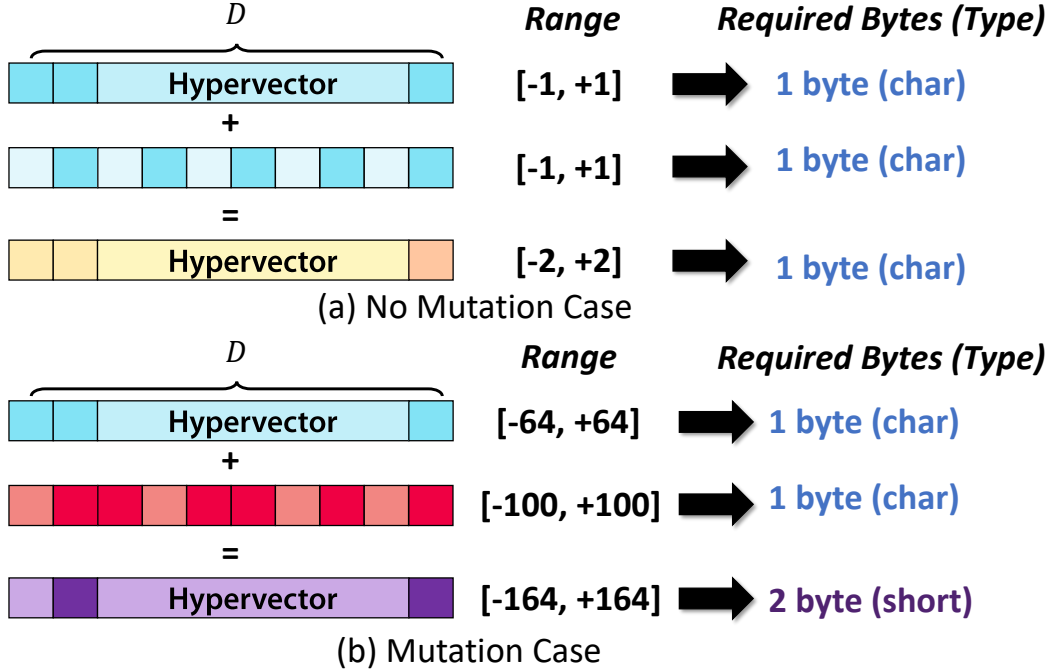


Figure 2.3: The example of data type mutation.

we applied *automated data type mutation* optimization technique that inferences for the data type of hypervector components.

Figure 2.3 shows how OpenHD tracks the range of the hypervectors to verify when to mutate the data type, saying the no mutation case (a) and mutation case (b). The first hypervector created in the HDC application is $\{+1, -1\}^D$, where D is the dimensionality of the vector. Thus, it starts from a vector of `char` type (1 byte), and the required range changes as HDC addition or multiplication are performed in the subsequent process. For example, for a variable of `char` type, the minimum required data type may increase in the order of `short` (2 byte), `int` (4 byte), and `long` (8 byte) during iteration.¹ Therefore, this optimization module tracks the range of variables along with the dependency graph in kernel units. We hereby track three types: variable nodes, conditional statements, and the dominator. Finally, the data type in nodes (the operation inside the kernel, the return type of the kernel, and the input type of the subsequent kernel) in the AST is overridden with the inferred one.

¹It may vary depending on the system environment.

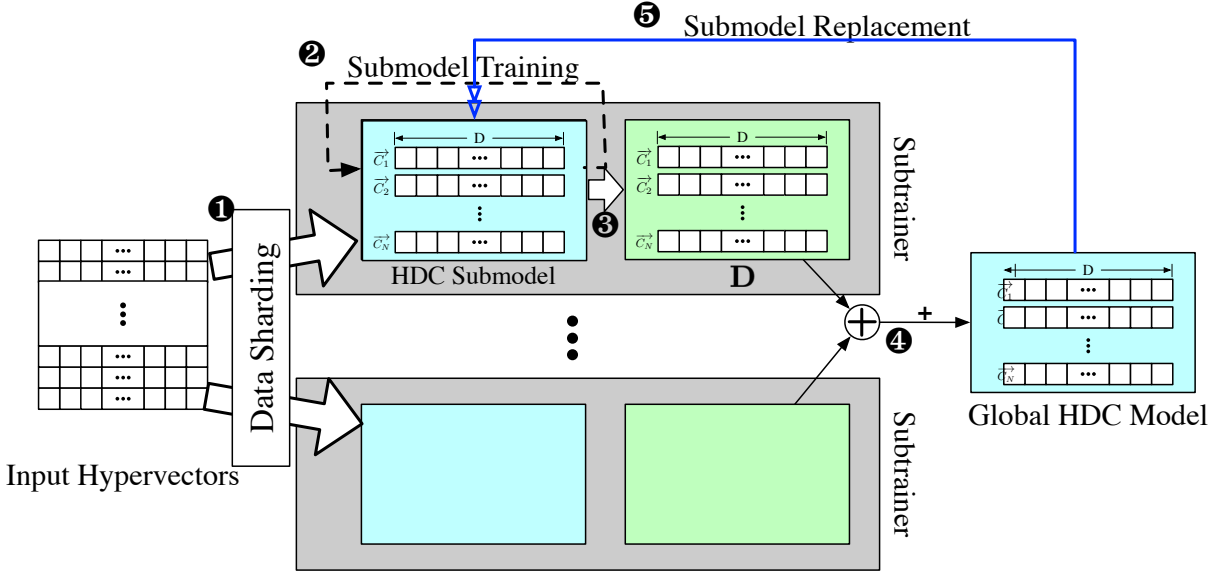


Figure 2.4: Overview of the proposed parallel training method.

2.3.5 Optimization of Built-in Functions

OpenHD supports HDC’s standardized operations as an API (see Table 2.2). It reduces syntax analysis and code generation time. The corresponding operations are unnested from AST and compiled in the ahead-of-time (AOT) as runtime library APIs.

Permutation/shuffle operation accompanies non-coalesced memory access if source data reside on global memory. For shuffle operations where many reading transactions can occur, we place the source hypervector in the constant memory. Even if a user uses multiple permutations, the memory access fusion module combines operations into a single kernel. Thus, it avoids performance degradation by storing intermediate results in registers.

2.4 OpenHD Optimization Techniques

2.4.1 PARTRAIN: Parallel Training for HDC Classification

The iterative training (retraining) enhances the HDC model accuracy. In contrast to the HDC encoding, the existing iterative training process has limited parallelism in nature since it feeds data sequentially. In other words, the hardware can be underutilized and become a bottleneck of the HDC application. We propose PARTRAIN, which enables a parallel iterative training process and expedites model refinement by maximizing hardware utilization.

Figure 2.4 illustrates the design of PARTRAIN. We spawn *subtrainers*. Subtrainers can be created until it fills up available hardware resources. Each subtrainer contains two sets of C hypervectors, where C is the number of classes. The first set contains an HDC submodel, while the other stores the changes of the submodel. We initialize an HDC submodel using single-pass training and the other set with zeros. PARTRAIN aims to maximize the data parallelism during the training. We split the data equally according to the number of subtrainers (❶). Each subtrainer refines its HDC submodels(❷) and accumulates changes of it to $\vec{D}$ (❸). After training with assigned datapoints, we update the global HDC model by calculating $\vec{G} = \sum_i \vec{D}_i$ (❹). Note that i indicates i th subtrainer. The global model $\vec{G}$ replaces all HDC submodels (❺). The whole process is defined as *one epoch*.

Note that PARTRAIN is different from data-parallel training used in DNNs wherein gradients are aggregated from multiple GPUs, which is equivalent to having a larger batch on a larger GPU. In contrast, PARTRAIN performs submodel updates before the global model aggregation. Hence, the training result (model) of PARTRAIN is different from the sequential HDC training as all updates are postponed to the end of the epoch (even the submodels are not being sequentially updated as opposed to the conventional HDC algorithm).

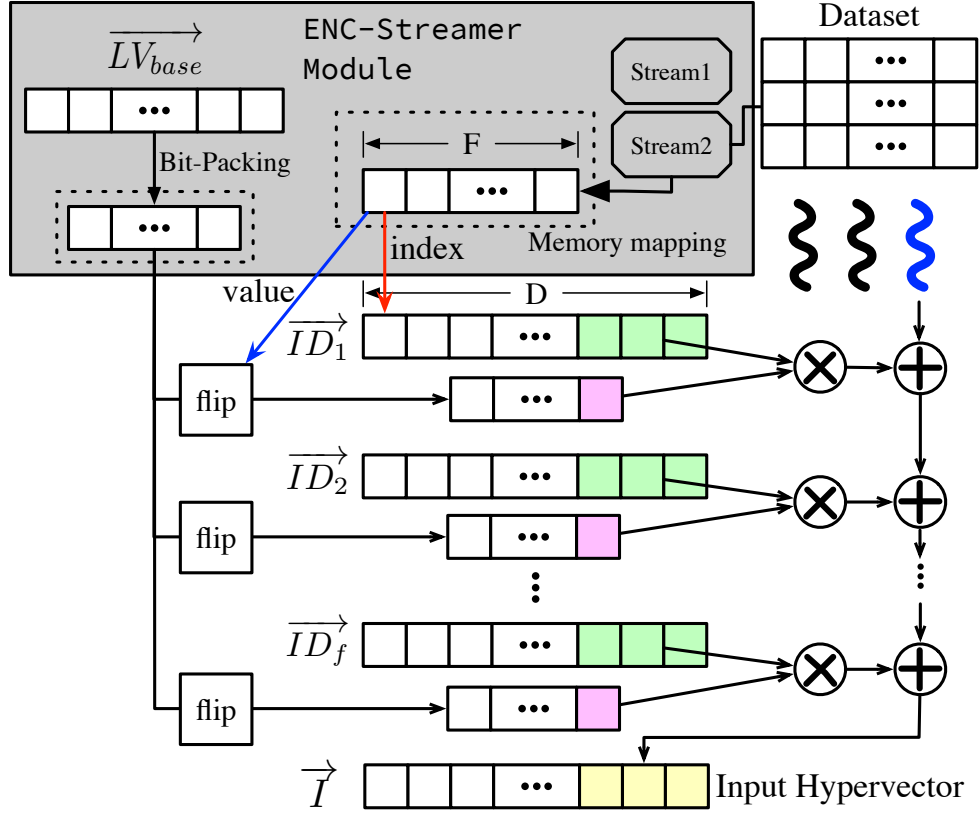


Figure 2.5: OpenHD of ENC-STREAMER-applied HDC encoding.

2.4.2 HDC-Based Memory Optimization

OpenHD uses two techniques to optimize memory transactions which is the main challenge in HDC that uses large-sized hypervectors. ENC-STREAMER optimizes the encoding stage of HDC and L2-RECYCLE optimizes the training (subtrainer in case of PARTRAIN).

HDC Encoding Optimization

ENC-STREAMER allocates data to the proper GPU memory hierarchy to achieve high efficiency during encoding and inference stages. The encoding is the bottleneck of the HDC application, as it takes up to 70% of the overall runtime of HDC on CPU [62]. Here we focus on one of the popular HDC encoding techniques, the ID-level encoding, but the same strategy can be applied to a simpler RP method. A naive way to implement the ID-level encoding on GPU is to

parallelize over datapoints and each dimension of input hypervectors.

On top of the parallelization, ENC-STREAMER enhances memory transactions by leveraging GPU memory hierarchy. The GPU memory hierarchy includes several memories, e.g., global memory, shared memory, and constant memory, which have different characteristics in terms of size and latency. The execution time varies depending on a data allocation strategy.

Figure 2.5 shows the HDC encoding module with ENC-STREAMER. We use two streams to overlap the computation and the memory transfer. Two streams use the same optimization techniques synchronously and run until all datapoints are covered in an interleaved fashion. Furthermore, ENC-STREAMER applies an additional caching strategy on raw features and level hypervectors, which is a frequently used term as shown in Eq. 2.1. We cache raw feature values on the shared memory to enhance memory coalescing. ENC-STREAMER optimizes access level hypervectors. The access to the level hypervector is irregular as intensity values in raw data are random. Since the level hypervector is generated using flips in a regular way, we can generate $\overrightarrow{LV}$ in-place. Hence, ENC-STREAMER stores 0-th level hypervector, $\overrightarrow{LV_{base}}$, on the constant memory as it is used frequently and unchanged during the entire encoding process. We bit-pack the base-level hypervector and store it in the constant memory. Since hypervectors require a large size array, packing is essential to satisfy the constant memory constraint. Empirically, packing eight bits (`char`) showed the best performance. Each thread takes a packed level hypervector (pink-colored boxes) and encodes the eight components of the input hypervector (yellow-colored boxes). We use encoded data for the rest of the HDC application runtime, thus *completely eliminating the data movement costs*. For the GPUs with UVA, we use a single stream and use managed memory since raw data requires one read transaction and does not have to be cached. This mechanism helps to alleviate the memory capacity limitation that is critical in embedded GPUs.

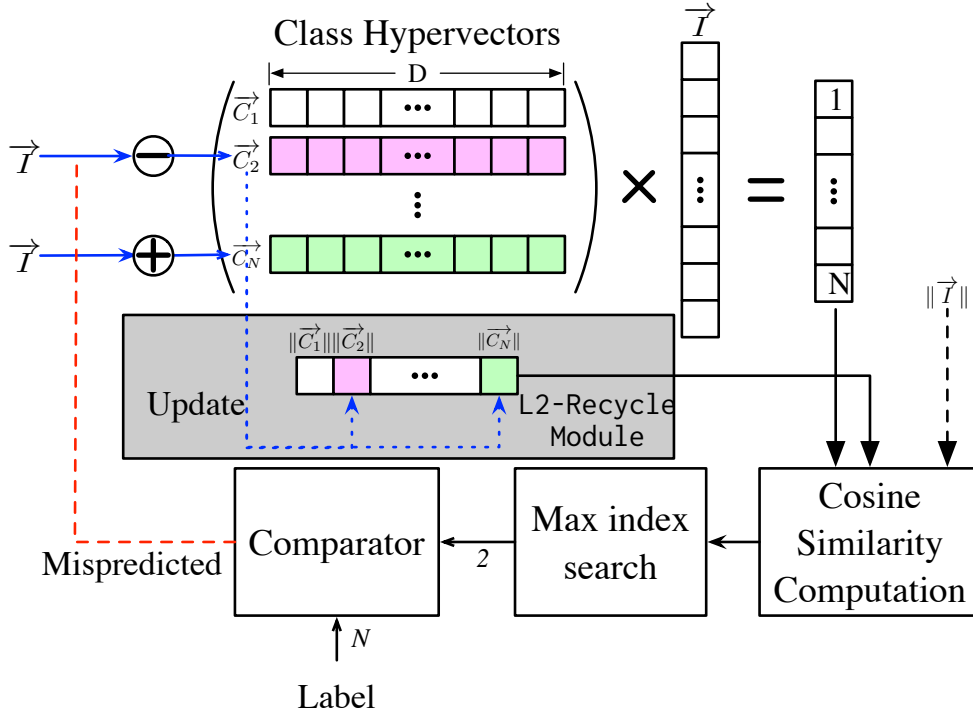


Figure 2.6: Design of L2-RECYCLE-applied training module in HDC-based classification.

HDC Training Optimization

Pairwise similarity computation is the next slowest component of HDC in both classification and clustering. Since OpenHD is specialized for non-binary hypervectors, we use cosine similarity metric which defined as $\delta(\vec{x}, \vec{y}) = (\vec{x} \cdot \vec{y}) / (\|\vec{x}\| \cdot \|\vec{y}\|)$. To expose higher parallelism, we separately compute the numerator and the denominator, which can be done using the parallel reduction technique [63] on large dimensionality. However, it showed marginal speedup over the CPU-based implementation.

In addition to parallelization, L2-RECYCLE minimizes the memory access by identifying *invariant calculation results* during the similarity computation. It computes the L2 norm components only when the corresponding hypervector is updated. Figure 2.6 illustrates the training in HDC classification with L2-RECYCLE. Since the training procedure computes the similarity ($s[k]$) between the k^{th} class hypervector and input hypervectors, we *pre-compute* their L2 norms

when updating them in the training loop. It separately stores the L2 norm of class hypervectors. Our strategy removes unnecessary computation and minimizes memory access to the large-sized hypervectors.

2.5 Evaluation

2.5.1 Experimental Setup

To verify the practical value of OpenHD, we implemented HDC-based classification [15] and clustering algorithms [59]. We evaluated the implementation of the CPU and GPU in the NVIDIA Jetson TX2 device with Jetpack 4.4.1 SDK. We set the dynamic voltage and frequency scaling governor.

Comparison to the state-of-the-art ML

We first compare the HDC-based classification and clustering with the optimized DL model generated by AutoKeras [64] and K-means algorithm [65] in terms of quality, model size, and runtime.

HDC-based Classification

We compare the execution time of our OpenHD-based implementation with the TensorFlow 2 (TF2)-based HDC and the state-of-the-art GPU-powered HDC classification, XCelHD [38]. Note that we used our TF2-based baseline since the existing TensorFlow-based HDC [9] has higher per-inference energy than ours. For a fair comparison, we measure the speed of the second run since OpenHD caches and reuses function chunks once compiled. We discuss the details about the compilation overhead in Section 2.5.6. Next, to show the energy efficiency of OpenHD, we compare the energy consumption with the CPU-based HDC classification and clustering implementation, which uses Python with a C++ backend to take full advantage of the parallel

Table 2.3: Statistics of the datasets

Dataset	Features	Classes	Training set	Test set
CARDIO	21	2	1913	213
UCIHAR	561	12	6213	1554
ISOLET	617	26	6238	1559
FACE	608	2	21441	2494
PAMAP2 ²	75	5	22500	22500

capabilities of SIMD. We included the communication time, e.g., memory copy time, between the GPU and the host. For the power consumption measurement, *tegrastats* utility is used. We set fixed quantization level Q to 100 and epochs to 20. The hypervector dimensionality, D , is set to 10,000 and 4,000. Also, PARTRAIN-powered OpenHD is configured with 10 subtrainers.

To observe the performance focusing on the actual use-case of HDC-based classification, we evaluated the implementation on a wide range of following benchmark datasets [66, 67]. **CARDIO**: a medical diagnosis based on patients’ information, **UCIHAR**: detecting human activity based on 3-axial linear acceleration and angular velocity data, **ISOLET**: recognizing audio of the English alphabet from different people, **FACE**: classifying images with faces/non-faces, each sampled from Caltech’s web faces dataset, CIFAR-100, and Pascal VOS 2012 datasets, and **PAMAP2**: classifying five human activities based on a heart rate and inertial measurements. Attribute information for these datasets is listed in Table 2.3.

HDC-based Clustering

We compare the execution time of our OpenHD-based implementation and the TF2-based HDC clustering running on the GPU. We used ten epochs in our HDC-based clustering algorithm. The clustering phase can be early-terminated as the HDC model generally converges before ten epochs. However, we fixed the number of iterations for a fair comparison. We evaluate the HDC-based clustering using the fundamental clustering problem suite (FCPS) [68], which

²We randomly sampled from the original dataset as it cannot be fitted to the Jetson TX2 memory.

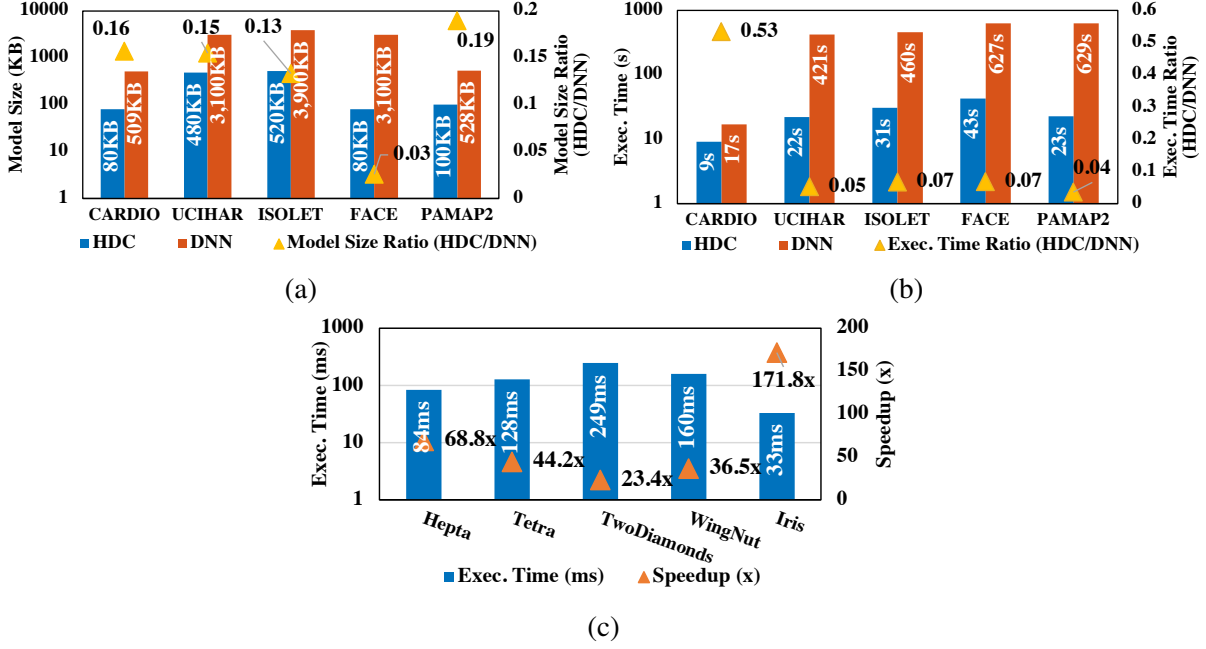


Figure 2.7: Comparison between OpenHD-based HDC and the state-of-the-art ML method. (a) Model size of DNN-based vs. HDC-based classification (b) Runtime of DNN-based vs. HDC-based classification (c) K-means vs. HDC-based clustering

addresses general challenges for clustering. We experiment with a subset of FCPS, including **Hepta** (consists of six classes with different inner class variances), **Tetra** (dataset with small inter class distances), **TwoDiamonds** (two classes with touching classes) and **WingNut** two classes with inter class density variation). We also experimented with the **Iris** [66] pattern recognition dataset.

2.5.2 Comparison of HDC to the State-of-the-art ML

To compare HDC-based and DNN-based classification, we run the neural architecture search (NAS) for the DNN model with the maximum accuracy using AutoKeras [64]. The comparison between HDC and the most accurate DNN model found is done on the NVIDIA Jetson TX2. The HDC model is trained for ten epochs, and the execution time for the NAS is excluded. For the comparison of HDC-based clustering, we compare it with the K-means algorithm running on GPU.

Classification accuracy: HDC-based classification has a comparable accuracy to DNN-based classification. The former offers 95.5% of accuracy on average, while the latter shows 94.7% on average. HDC-based classification accuracy remains stable once sufficient dimension size is reached [69]. Additional dimensionality can be used for better noise resilience [53].

Classification model size: Figure 2.7(a) illustrates the HDC-based and the DNN-based model size. HDC is on average $9\times$ smaller than the DNN-based models. By decreasing D , The HDC model can reduce the model size by 20% with an accuracy drop of less than 1%.

Classification training and inference time: As depicted in Figure 2.7(b), HDC-based classification runs $11.7\times$ faster than the DNN-based method on average. While existing DNN-based applications in embedded devices focus only on optimization of inference through TensorRT [70] and TVM [71] libraries, HDC-based solutions enable both lightweight training and inference in embedded devices.

Clustering accuracy, speed, and efficiency: To evaluate HDC-based Clustering, we set D to 1024, as it gives a comparable normalized mutual information score to K-means, which is a significantly smaller hypervector dimensionality compared to classification, resulting in much smaller model sizes. The score difference was within 3.1% on average. Figure 2.7(b) illustrates the execution time ratio between K-means and HDC-based clustering running on GPU. HDC-based clustering is $53\times$ faster on average compared to K-means, at nearly the same power consumption.

2.5.3 Accuracy of PARTRAIN

PARTRAIN enhances data parallelism of HDC-based training using multiple trainers. Thus, it can look up more data than the traditional HDC training technique in the same period. We compared PARTRAIN-enabled HDC classification to the existing training method. In particular, two baselines are used: single-pass training (Single Shot) and iterative training with 20 epochs (Retraining-20). We also observed PARTRAIN with different epoch settings, with single

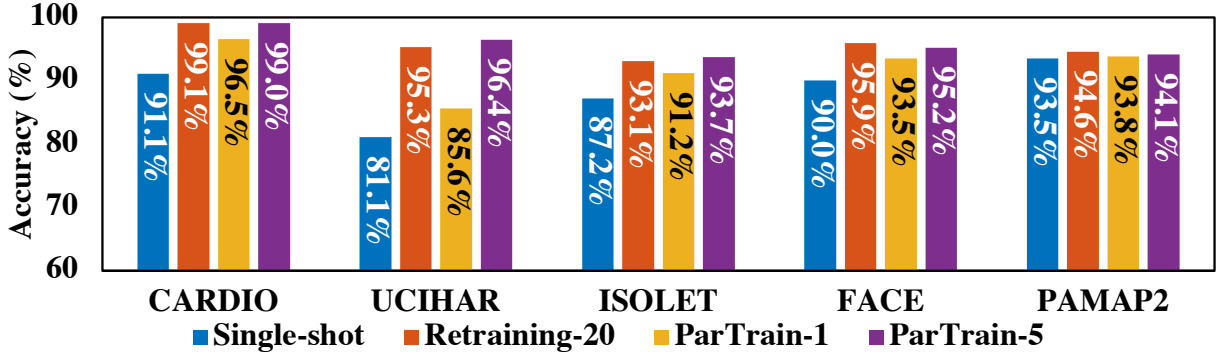


Figure 2.8: Accuracy comparison between PARTRAIN and the conventional HDC training method.

(PARTRAIN-1) and five (PARTRAIN-5) iterations. Figure 2.8 shows the achieved peak accuracy with different training settings. PARTRAIN-5 case achieves a similar accuracy compared to the original HDC with 20 epochs. We can reduce required epochs by $4\times$. It implies that PARTRAIN can achieve the target accuracy with fewer epochs. Furthermore, the overhead of gathering information from subtrainers was negligible, which is less than 5% of the runtime.

2.5.4 OpenHD Efficiency

Speed Improvements in Classification

Figure 2.9 compares the execution time of the HDC classification implemented with OpenHD, XCelHD and TensorFlow 2. Compared to the TF2-based HDC, the encoding speed of XCelHD and OpenHD is improved by ENC-STREAMER which flexibly maps the data on the proper CUDA memory hierarchy. OpenHD adds optimization using data type mutation on top of XCelHD. Assigning data type reduces the memory footprint of HDC applications and allows storing more data in the local memory. Moreover, OpenHD enables efficient use of memory bandwidth which is significant for large-sized hypervector. Note that OpenHD efficiently leverages GPGPU memory hierarchy as encoding API of OpenHD adopts the optimization in XCelHD. As shown in Figure 2.9(a), OpenHD-powered HDC encoding is on average $1.37\times$ and

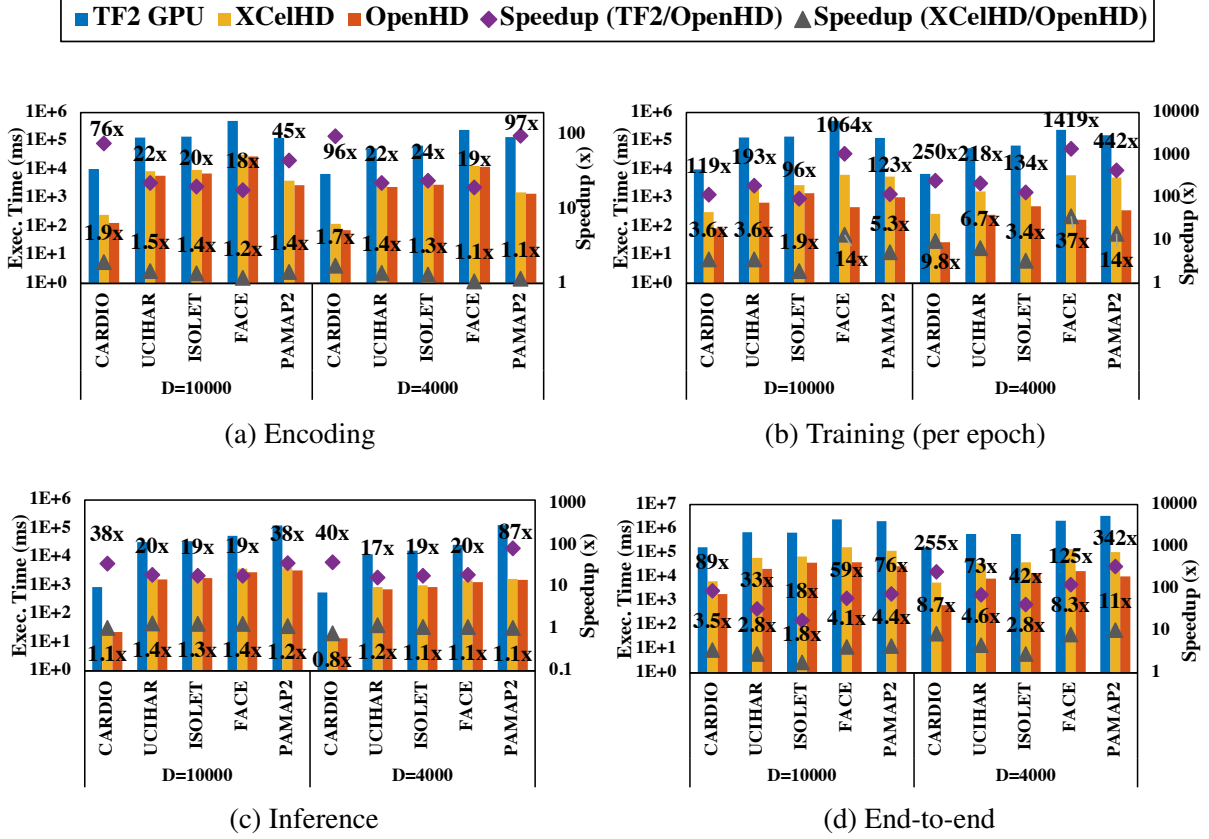


Figure 2.9: Speed comparison of OpenHD vs. baseline HDC implementations.

35 $\times$ faster than the XCelHD and the TensorFlow 2-based HDC, respectively.

The iterative training stage mainly benefits from the memory access fusion technique and L2-RECYCLE. In addition to XCelHD with L2-RECYCLE, OpenHD further optimizes the TF2-based HDC training with memory access fusion. The prediction is based on `hd.search` being AOT compilation, which is primarily benefitted by L2-RECYCLE. Also, during the update of class hypervectors, operations need to be written line-by-line in code due to the Python syntax. However, memory access fusion reduces global memory access overhead. The kernel is generated automatically according to the maximum number of epochs. The encoding stage includes the writing operation of the hypervectors only once. In contrast, we apply operations to the large-sized hypervector repeatedly. Memory access fusion greatly impacts the speedup compared to XCelHD, leading to a larger speedup. Note that more subtrainers can be added on the OpenHD-based HDC

since we assign the proper data type to the hypervector and reduce the memory footprint. As shown in Figure 2.9(b), OpenHD provides on average $6.8\times$ speedup over XCelHD and $258\times$ speedup over TF2-based HDC in the HDC training module.

The inference stage consists of encoding the test set and similarity computation. Since OpenHD adaptively generates code, the memory mapping policy is changed according to the test set configuration. Then the similarity search operation for hypermatrix is applied by the `hd.sim` function. Since the encoding takes up a large portion of the inference stage, more than 99%, the trend was similar to the encoding stage (see Figure 2.9(a) and (c)). In the case of the CARDIO dataset, the speed of OpenHD was slower than the XCelHD. It is due to the communication overhead between the Python front-end and CUDA backend, while both implementations have similar optimizations. For other cases, OpenHD consistently shows speedup over baselines. OpenHD-powered inference stage yields on average $1.17\times$ and $27\times$ of speedup compared to XCelHD-based and TF2-based HDC, respectively.

Overall, on the HDC classification, OpenHD runs faster than baselines. In particular, OpenHD-based implementation shows the best improvement on the training stage. PARTRAIN helps to overcome limited parallelism and underutilization during the training process in GPU-based HDC. By adaptively generating CUDA code with regard to the characteristics of the dataset and the encoding strategy, the training and inference stage benefitted. For the end-to-end execution of the HDC process, OpenHD-powered implementation gains on average $4.5\times$ and $78\times$ over the state-of-the-art GPU-based HDC and TF2-based HDC classification, respectively, as shown in Figure 2.9(d).

Speed Improvements in Clustering

Figure 2.10 shows the clustering speed improvements of OpenHD as compared to TF2-based implementation for the encoding and the clustering stages. For these two stages, we get $252\times$ and $94\times$ speedup on average. During the encoding stage of OpenHD, a thread iterates over

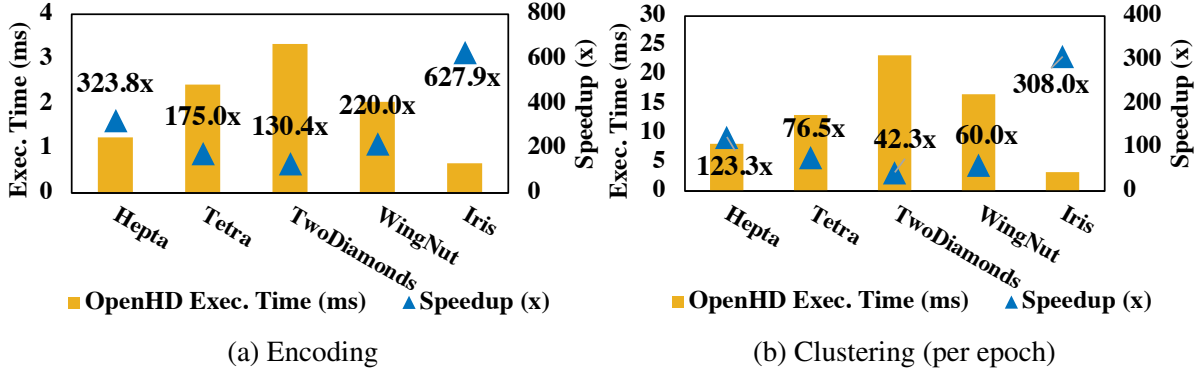


Figure 2.10: Speed comparison of OpenHD vs. TF2 GPU-based HDC clustering.

the features. The clustering dataset has a smaller number of features than the dataset used in the classification evaluation. Hence, while it shows a similar trend to Figure 2.9(a), the execution time reduction is substantial. In the clustering stage, we iterate over datapoints. Comparing the absolute value of the dataset size to the feature size, the former is significantly larger than the latter, leading to a longer execution time during the clustering. Therefore, it shows a more limited speedup than the encoding stage. For the end-to-end execution of HDC-based clustering, OpenHD-based implementation is on average $96\times$ (up to $146\times$) faster than TF2-based HDC.

Energy Efficiency Improvements

We compared the energy consumption of OpenHD to the state-of-the-art HDC classification running on the low-powered CPU. CPU-based HDC uses SIMD, which supports a smaller degree of parallelism than the hypervector dimensionality. Since it has limited parallel computing resources, we disabled PARTRAIN. Figure 2.11 shows the energy efficiency improvement of OpenHD over the baseline. The encoding stage enables high parallelism and consumes more power, but reduced execution time improves energy efficiency. The training module uses low power in nature due to limited parallelism. However, compared to the CPU-based HDC, L2-RECYCLE significantly reduces the execution time by removing redundant operations; $125\times$ energy consumption reduction on average. We disabled PARTRAIN in this experiment, but

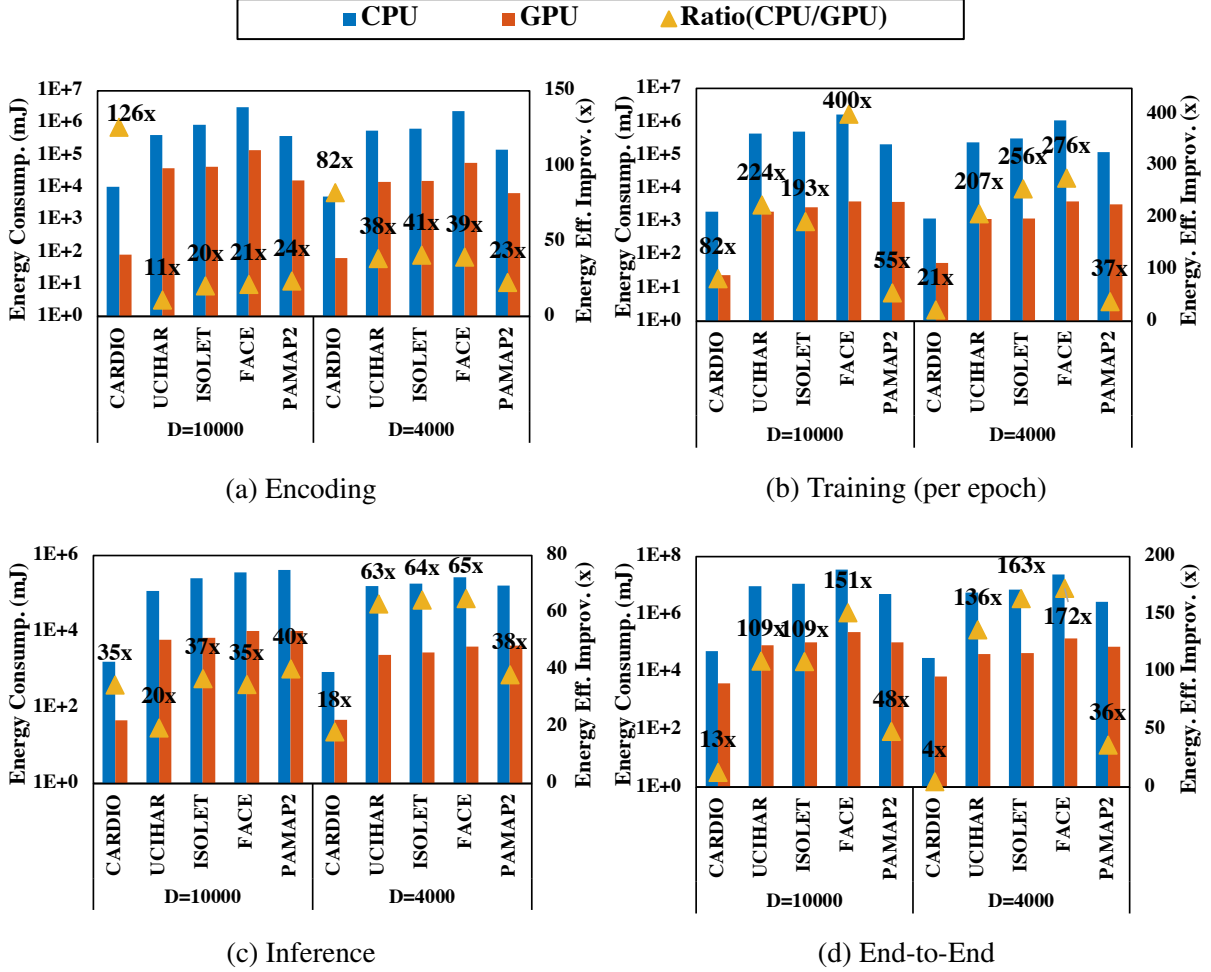


Figure 2.11: Energy consumption comparison of GPU (OpenHD) vs. HDC running on the embedded CPU.

enabling it can enhance energy efficiency further. Even if it has more parallelism and consumes more power, the reduced number of iterations compensates for the increased power consumption. Ultimately, OpenHD achieves up to $172\times$ energy efficiency improvement over the CPU-powered HDC classification.

2.5.5 Effectiveness of ENC-STREAMER and L2-RECYCLE

In this section, we observe the effectiveness of ENC-STREAMER and L2-RECYCLE by comparing the execution time with and without applying strategies. We set several scenarios to

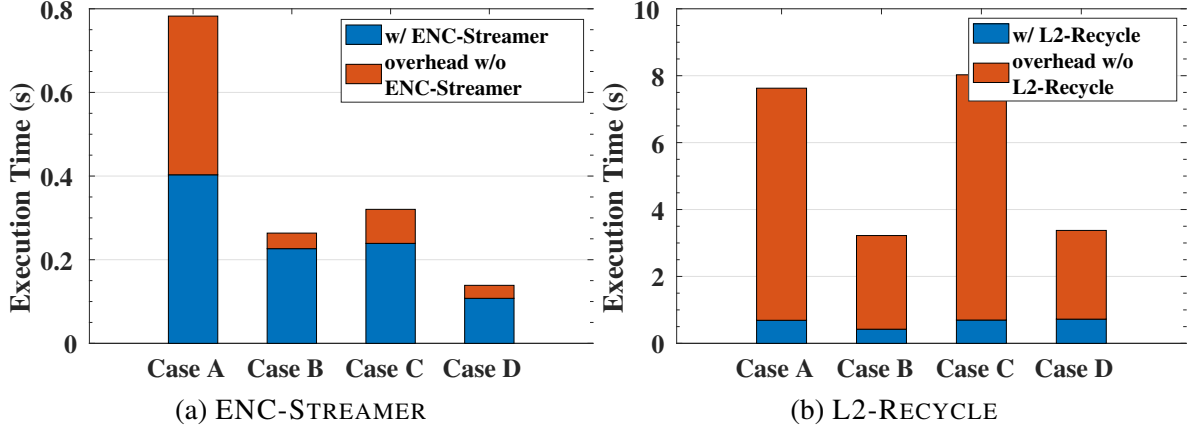


Figure 2.12: Execution time reduction by proposed optimizations.

measure general effectiveness and fixed the dataset size to 10,000. ENC-STREAMER mainly optimizes the encoding stage, which is affected by dimension size D and the number of features F . Our experiment conducted on four scenarios, (F, D) : Case A) (500, 10000), case B) (500, 4000), case C) (250, 10000) and case D) (250, 4000). As shown in Figure 2.12(a), ENC-STREAMER is most effective on large hypervector dimensionality, achieving $1.94\times$ in Case A. Moreover, ENC-STREAMER improves energy efficiency since the power consumption of the two variants is similar.

L2-RECYCLE is effective on the iterative training of HDC classification. We measured execution time by varying the number of classes C and the dimensionality D . We evaluated on four scenarios: Case A) (10, 10000), case B) (10, 4000), case C) (20, 10000) and case D) (20, 4000). On the large D , L2-RECYCLE shows significant speedup up to $12\times$ (see Figure 2.12(b)). Without L2-RECYCLE, the execution time increases dramatically on large dimensionality. Our optimization strategy offers similar training time regardless of D . The overhead is minimal; in the worst case, case C requires a small additional memory cost of 80 Byte.

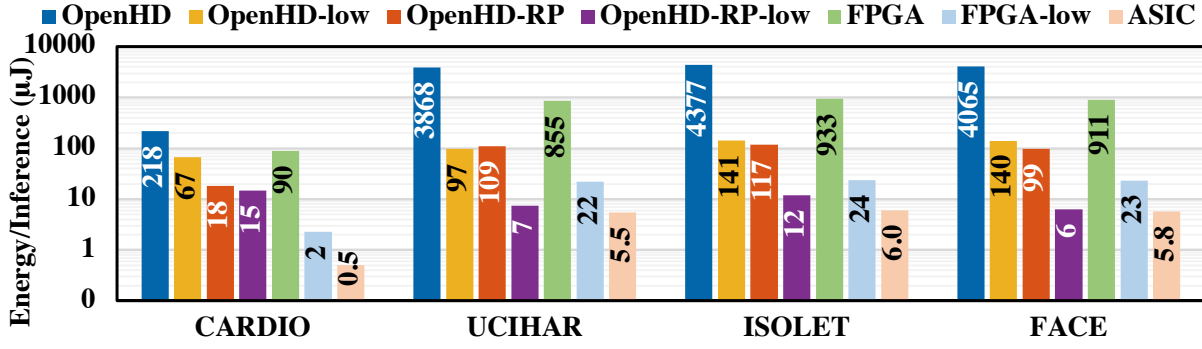
2.5.6 Code Generation Overhead

As OpenHD performs JIT compilation, there is an overhead of code analysis. Nevertheless, since OpenHD caches the compiled code chunk once, there is no overhead when rerun. We compared the execution time for the first run and the second run to measure the overhead of JIT compilation. The overhead of code generation and compilation was at most 88%. Besides, the profiling results show that the compute utilization and throughput of GPU are improved by $1.72\times$ and $1.82\times$, respectively.

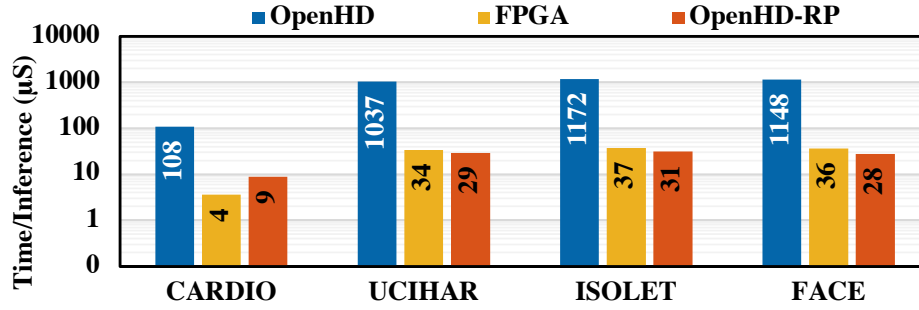
2.5.7 Comparison with Other Hardware Implementation

We compare OpenHD with an efficient FPGA design implemented and verified on the Xilinx Alveo U200 accelerator card and with the ASIC design of [9]. The FPGA implementation leverages RP encoding, which relies on matrix-vector multiplication. It is inherently more efficient but less accurate in certain applications such as time-series. Therefore, in Figure 2.13, we also present a variant of OpenHD that uses RP, OpenHD-RP. Since the ASIC implementation achieves lower accuracy, for a fair comparison, we also compare OpenHD-low and FPGA-low, reduced-dimension GPU and FPGA that yield the same accuracy of ASIC.

Using 10,000 dimensions, the average per-inference energy consumption of OpenHD, which uses ID-level encoding, is $3.9\times$ of FPGA. However, OpenHD-RP, which uses the same encoding as FPGA, consumes $7.3\times$ less energy than FPGA. OpenHD-RP-low (reduced dimension with a similar accuracy of ASIC) improves the energy by $1.35\times$ with respect to FPGA-low and only consumes $3.05\times$ more energy than the full-custom ASIC. In higher dimensions, OpenHD-RP shows better improvement over FPGA ($7.3\times$ vs $1.35\times$). FPGA uses on-chip SRAM memory with single-cycle access regardless of the dimensionality, so its execution and energy improve linearly in lower dimensions. For GPU, the data movement overhead manifests more significantly in lower dimensions (wherein computation cost is smaller).



(a) Inference energy



(b) Inference time

Figure 2.13: OpenHD inference compared to FPGA and ASIC.

Figure 2.13(b) compares the per-inference execution time of OpenHD and FPGA. Using the more-complicated ID-level encoding, OpenHD inference time is $30.8\times$ higher than the FPGA, while OpenHD-RP is only $1.08\times$ slower than the FPGA (with $7.3\times$ better energy as mentioned above). Note that the ASIC [9] has not provided execution time numbers, so we omitted it from comparisons.

Figure 2.14 compares the training energy and execution time of OpenHD and FPGA for 20 epochs. OpenHD improves the end-to-end and iterative training (retrain)-only energy by $1.63\times$ and $2.78\times$. The retrain-only energy does not include the encoding for which OpenHD consumes more energy due to using ID-level encoding. Therefore, the energy improvement without considering encoding is higher (with RP encoding, the end-to-end energy improvement would be even higher). Finally, the end-to-end and iterative retrain of OpenHD is, respectively,

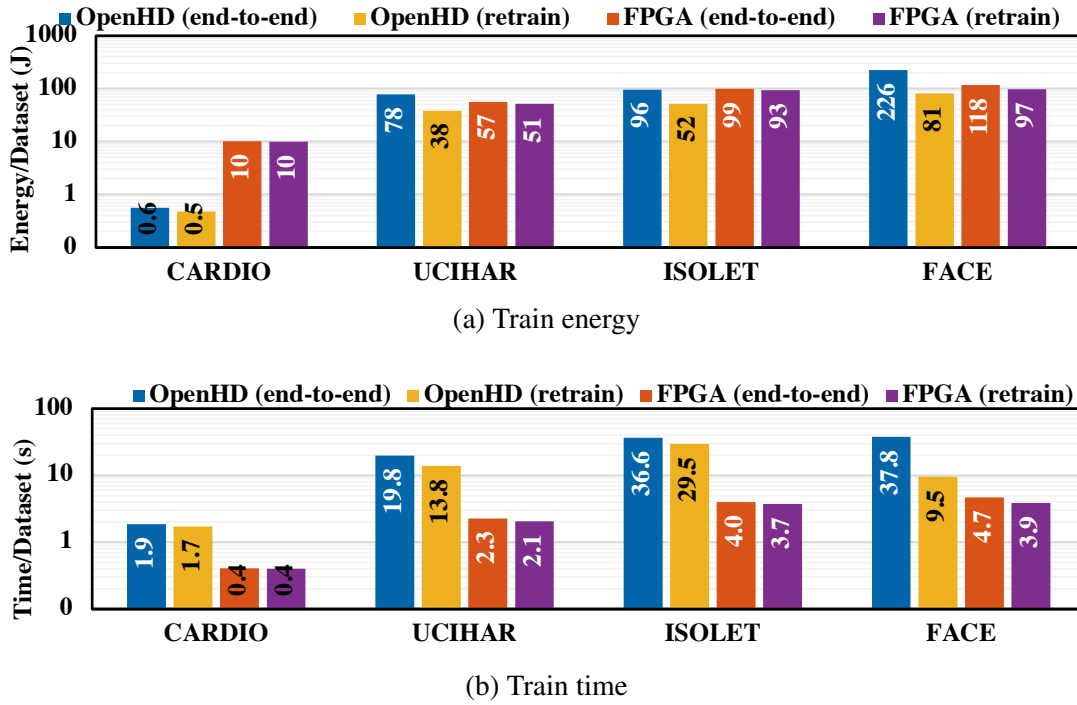
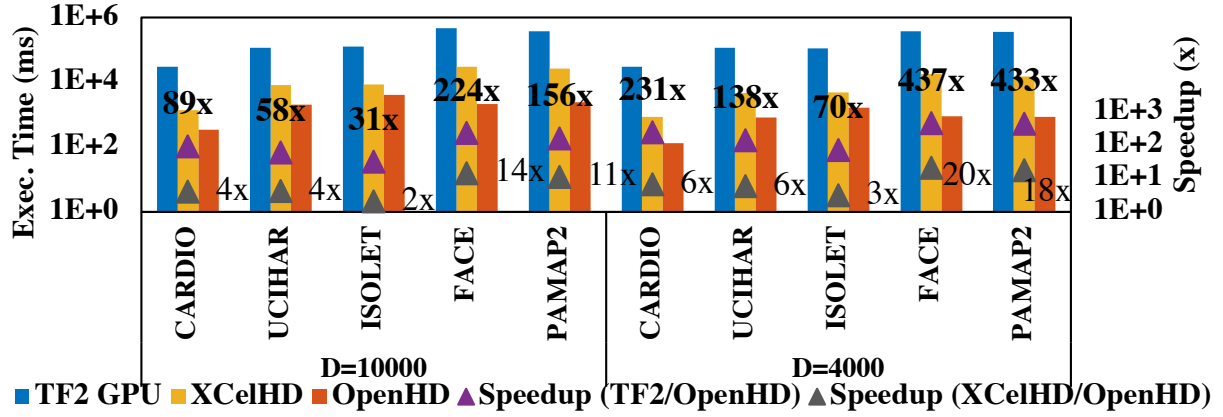


Figure 2.14: OpenHD training compared to FPGA.

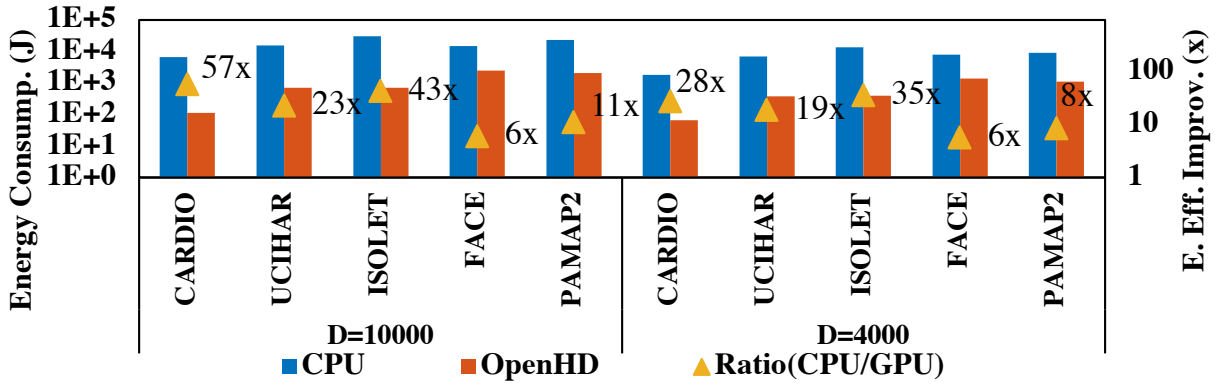
$7.4\times$ and $4.9\times$ slower than FPGA. Using RP encoding can improve the encoding and the end-to-end speed, but retrain time will improve intact as it deals with already encoded hypervectors. Note that Xilinx Alveo U200 is a high-performance FPGA and offers massive parallelism. Compared with low-end or mid-range FPGA, we expect OpenHD could excel in performance, as well.

2.5.8 Evaluation on High-performance Hardware

High-performance hardware can also be utilized to process HDC in the real world. A workload can be offloaded to the cloud, or the user may develop a new HDC algorithm on the high-end GPUs. OpenHD is runnable on the desktop-class GPU since it uses CUDA as a backend. As such, we measured the execution time and energy consumption of OpenHD and baselines for HDC-based classification task on the high-performance system, Intel i7-8700K with 16GB RAM and NVIDIA Geforce GTX 1080Ti. We measured the power consumption using Intel PowerLog and `nvidia-smi` command.



(a) Speedup



(b) Energy Efficiency

Figure 2.15: Improvement of OpenHD compared to baselines running on high-end hardware.

Figure 2.15 shows the speedup and energy efficiency improvement results for datasets tested on high-performance hardware. On high-end hardware, OpenHD shows a similar speedup trend to the embedded environment. For large tasks, OpenHD also shows a high degree of acceleration. Even if the high-end GPU consumes more power, the energy efficiency improvement is larger than that of the embedded device due to the short execution time. Overall, our design achieves an average speedup of $6.7\times$ over XCelHD and energy efficiency improvement of $18\times$ on average over CPU.

2.5.9 Case Study: OpenHD on Other HDC Tasks

Since OpenHD is a framework to implement various applications using the HDC paradigm, we can also implement non-ML applications. Here, we introduce two additional HDC application examples implemented with OpenHD to discuss the potential of HDC.

Key-value Storage

With the element-wise addition and multiplication, we can effectively implement a hash table-like data structure (also accelerated on GPU for high efficiency): $\vec{H} = \sum_i \vec{K}_i \times \vec{V}_i$ where $\vec{K}_i$ and $\vec{V}_i$ are the encoded hypervectors for the key and value. The single hypervector includes multiple key-value relations. We can check if a target value $\vec{V}$ is associated with a key $\vec{K}$: $\delta(\vec{H} \times \vec{K}, \vec{V})/D \simeq 1$ when using the dot product for the similarity metric. Unlike the traditional hash table, the key-value structure can *categorize* information effectively. For example, let us give an analogous example – “which fruits are red?” To memorize the relationship between the fruit and color, we can encode each fruit and color to $\vec{K}_i$ and $\vec{V}_i$. If $\vec{Y}$ is the hypervector that represents the yellow color, $\vec{H} \times \vec{Y}$ is similar to all hypervectors associated with $\vec{R}$; near-orthogonal for other fruits. An interesting property of HDC is that the dimension size determines the amount of information that can be stored in a single hypervector.

Figure 2.16 (a) shows the recovery rate according to the number of pairs in HDC-based key-value storage. Because the number of relations that can be distinguished by a single hypervector is limited, the recovery rate decreases as the number of key-value pairs increases. However, as the dimension size of the hypervector increases, more relations can be contained.

Structure Comparison

The human memory easily recognizes differences in structured objects, e.g., chronological changes in train station connections due to constructions. Let us consider a problem that compares two lists where a list has different items. We may encode the structure of the list using adjacent

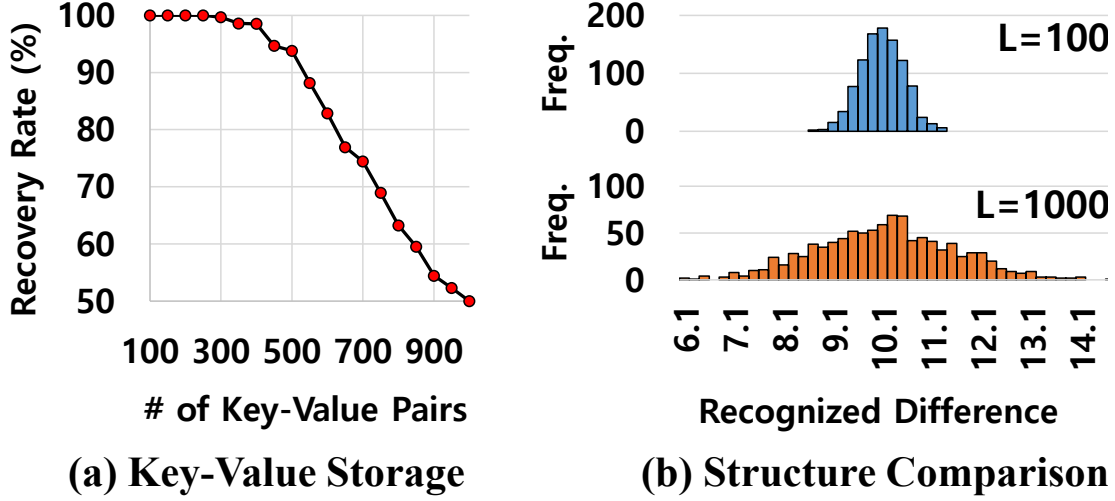


Figure 2.16: Stochastic nature of HDC for other cognitive tasks.

items, say $\vec{L}_i$ and $\vec{R}_i$: $\vec{M} = \sum_i \vec{L}_i \times \vec{R}_i$. When we have two hypervectors encoded for the different lists, $\vec{M}_1$ and $\vec{M}_2$, we can identify the number of indels using a single similarity computation: $\delta(\vec{M}_2 - \vec{M}_1, \vec{M}_2)/D$.

Figure 2.16 (b) shows the distribution of recognized differences according to the list length L when data is encoded and stored in a single hypervector. When a single hypervector contains more information, i.e., when L is large, it shows a lower resolution for a relation between data. In contrast, for small L , the HDC-based solution captures the difference well. However, as in the example of key-value storage, using a single hypervector is efficient regarding the data size. Also, the resolution issue can be mitigated by increasing the dimensionality of the hypervector.

2.6 Related Work

2.6.1 Automated GPU Acceleration

There are previous studies that accelerate algebraic operations on the GPU without users' awareness of the GPU. CuPy [72] offers GPU-accelerated application with NumPy-compatible functions. Their work leverages existing CUDA-based libraries such as cuDNN, cuSPARSE,

and cuBLAS. In [73], the authors present the system that generates CUDA code from the C syntax. Their work mainly enhances data access patterns using the existing polyhedral compiler optimization technique. Numba [74] accelerates a Python code using the LLVM compiler library, generating machine code at runtime. Their work enables the parallelization of the application using multithreading, SIMD vectorization, and GPU. Libraries such as Kokkos [75] and RAJA [76] have been released to increase the portability of CUDA code with the abstraction for parallel execution and memory management. The library uses modern C++-like syntax and provides an abstraction for operations not only on loop execution, but loop partitioning, reorder, and tile. Furthermore, for simple operations, execution configurations are automatically set. Our work is different from these general-purpose libraries in that OpenHD intelligently utilizes the characteristics of HDC, adapting various optimization schemes specialized to HDC for higher efficiency.

2.6.2 Hardware Acceleration of HDC

HDC consists of many bit-level arithmetic operations, including addition and multiplication. It can effectively be parallelized and handled on various parallel hardware platforms. The ASIC designs are proposed, e.g., similarity computation circuits [10] and text classification ASIC design [26] based on advanced memory technology. The work in [27] also shows how to accelerate the HDC encoding scheme using PIM techniques. In [77], the authors present a hybrid ASIC architecture that runs both DL and HDC. [51] proposes the HDC-based classification running on the FPGA.

Although FPGA, ASIC, and PIM implementations offer superior efficiency, design and synthesis time impede rapid development. Prior works explored the GPU-powered HDC implementation. The work in [9] shows TensorFlow for HDC classification acceleration on GPU. Hypervectors can be treated as a tensor data type. Hence, HDC operations can be implemented and accelerated with the GPU with tensor operations in TensorFlow. The results show that TensorFlow-based HDC running on the NVIDIA Jetson Nano consumes more energy and runs

just as slow as HDC on the CPU. Thus, we need a tool such as OpenHD, which automatically creates an efficient mapping of HDC applications to GPUs. OpenHD maximizes parallelism specific to HDC applications, while effectively leveraging the memory hierarchy and minimizing the memory accesses.

2.7 Conclusion

In this chapter, we presented a GPU-based HDC acceleration framework called OpenHD. OpenHD allows users to implement various HDC applications in Python while automatically accelerating them on GPU with highly optimized techniques. We address the memory access challenges in the current HDC application running on the GPU with efficient cache utilization and data reuse. The proposed optimization module benefits the encoding and the prediction of the HDC, which affects the end-to-end pipeline of HDC applications. Furthermore, we propose a parallelized training method, called PARTRAIN, to enhance hardware utilization and data-parallel benefits. Users can use proposed techniques with NumPy-style Python syntax and automatically get fully optimized CUDA code with the best possible performance for commonly used HDC applications such as classification and clustering. For the standardized HDC operations, our framework offers API to compile in AOT as a runtime library. In our experiments, OpenHD-based classification offers comparable accuracy to the DNN-based solution with $9\times$ smaller model and $11.7\times$ faster execution time. OpenHD-based clustering is $53\times$ faster compared to the K-means algorithm at comparable accuracy. Moreover, the proposed PARTRAIN reduces the required epochs to obtain comparable accuracy by $4\times$. Evaluation results with the low-powered embedded GPU show that OpenHD is $4.5\times$ and $146\times$ faster execution time on average for classification and clustering applications, respectively, compared to the state-of-the-art GPU-based HDC implementation.

This chapter contains material from “XCellHD: An Efficient GPU-Powered Hyperdimen-

sional Computing with Parallelized Training”, by Jaeyoung Kang, Behnam Khaleghi, Yeseong Kim, and Tajana Rosing, which appears in Asia and South Pacific Design Automation Conference (ASP-DAC), 2022. The dissertation author was the primary investigator and author of this paper.

This chapter contains material from “OpenHD: A GPU-Powered Framework for Hyperdimensional Computing”, by Jaeyoung Kang, Behnam Khaleghi, Tajana Rosing, and Yeseong Kim, which appears in IEEE Transactions on Computers (TC), 2022. The dissertation author was the primary investigator and author of this paper.

In the next chapter, we introduce an HDC solution capable of processing graph data. We propose an HDC-based graph ML framework, which allows for both graph-based ML training and inference only with lightweight HDC operations. We demonstrate that the proposed algorithm can be efficiently run on GPUs and a FeFET-based PIM accelerator, outperforming the speed and energy efficiency of existing GNN-based approaches.

Chapter 3

A Graph-based Machine Learning in High-dimensional Space

3.1 Introduction

In the previous chapter, we proposed a GPU-powered framework for HDC which automatically generates optimized CUDA code. It enables users to easily run HDC applications with GPU acceleration in a significantly optimized manner. This chapter focuses on demonstrating the application of this library to a novel HDC algorithm. We first develop the HDC-based graph ML solution and apply the framework to accelerate on GPU. Furthermore, we implement the proposed algorithm on PIM hardware and compare the performance and efficiency of the GPU and PIM implementations.

Beyond DNN-based algorithms that have shown high efficacy in ML tasks, GNNs have emerged as they yield state-of-the-art performance in graph-based ML. By analyzing the underlying relationships between data, GNN has demonstrated its efficacy on tasks, e.g., node classification, graph classification, and link prediction. Various areas rely on GNNs, such as analysis of social networks, protein interactions, disease classification, and many others. However,

training GNNs rely on backpropagation over the irregular and sparse graph data structure, which have high computational complexity [46, 78].

HDC, in contrast, is very efficient and lightweight. Previous works have tackled graph-related workloads using the HDC principle. In [79], the authors represent the edge in HD space by binding the hypervectors assigned to the source and destination nodes. [80] exploited a method based on holographic reduced representation to map nodes into HD space but relied on a neural network during the inference. Recent work [81] exploits the memorization ability of HDC and encodes the graph structure into hypervector. It enables information retrieval and graph reconstruction based on memorized information. The work in [82] encodes a graph into an hypervector and performs graph classification. Unfortunately, no existing HDC-based solutions consider the node’s structural role in a graph when addressing graph-based ML tasks. Also, they lack high-efficiency hardware for HDC-based ML on the graph domain.

In this chapter, we propose RelHD, which is capable of efficiently learning on graph data by leveraging the strengths of HDC. RelHD encodes the information of the neighboring nodes and trains a model with a lightweight HDC operations. Hence, the proposed algorithm is easily parallelizable, making it much faster compared to existing graph-based ML algorithms, e.g., Node2Vec [83], GCN [44], and GAT [45], on GPU. Furthermore, we devise a ferroelectric field-effect transistor (FeFET)-based hardware accelerator tailored to the proposed algorithm because running HDC on PIM has been shown to be much faster and more energy-efficient than on GPUs [25–27]. In contrast to previous FeFET-based HDC accelerators which limit HDC operation on a single FeFET memory block, RelHD accelerator is scalable with customized hardware components for efficient inter-block computation and communication. Our solution enables graph-based ML with scalable FeFET memory blocks and optimizes data allocation and processing flow to improve the efficiency of irregular graph-based algorithms. Experimental results on GPU show that RelHD offers comparable accuracy to existing solutions but is at most $32\times$ faster than popular GCN-based algorithms. The proposed accelerator further improves speed

and energy efficiency by $33\times$ and $59,287\times$, respectively, over RelHD on GPU. Also, RelHD PIM accelerator yields $2.5\times - 14.7\times$ speedup and $127\times - 10,282\times$ improved energy efficiency over the state-of-the-art PIM-based GNN accelerator, PIM-GCN [46].

3.2 Background and Related Work

3.2.1 HDC-based ML on Graph Domain

Several recent studies attempted to represent graph data in HD space and solve graph-based ML tasks. GrapHD [81] gathers node and neighbor information into a single hypervector. The encoded result can be used to reconstruct the original graph. It helps several cognitive tasks, such as object detection, by pipelining with a convolutional neural network (CNN)-based algorithm. Furthermore, they proposed a PIM-based hardware accelerator to expedite the processing. However, it is incapable of performing both on-device training and inference on data represented by a graph. Also, it only supports HDC operations on binary hypervectors, which can lead to data loss.

The work in [82] encodes a graph into an hypervector by bundling edge hypervectors, where edge hypervectors are the result of the binding of node hypervectors. It natively supports graph classification using the HDC principle, but it neglects the node’s structural role and is only implemented on GPU, even though HDC provides superior speed and energy efficiency on PIM-based hardware platforms [25–29]. In contrast, RelHD embeds the relationship between nodes and pipelines to an HDC-based training algorithm, thereby considering the structural role of each node and enabling on-device graph-based ML. Furthermore, we show a PIM-based solution optimized for RelHD algorithm, which offers high energy efficiency and speed over existing PIM solutions for GNN-based algorithm.

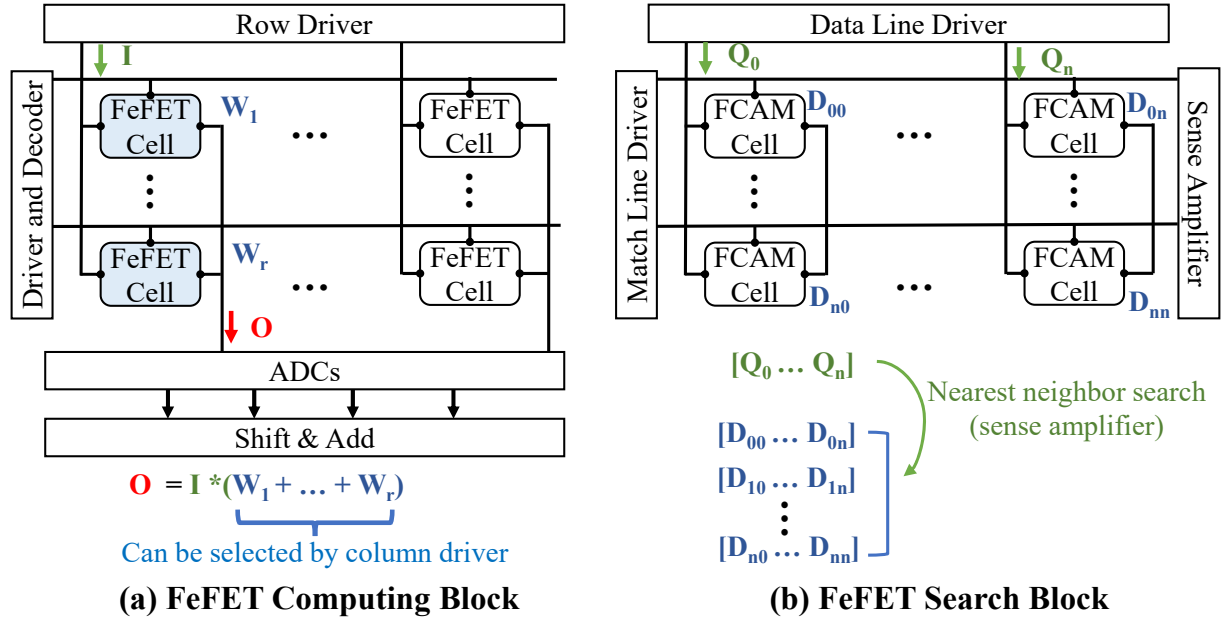


Figure 3.1: Two types of FeFET PIM block.

3.2.2 In-memory Computing with FeFET

A FeFET has emerged as a strong solution for in-memory computations, as it is compatible with CMOS, has low write energy, high I_{on}/I_{off} ratio, and low latency [84,85]. In FeFET memory, the polarization of the Fe layer determines the threshold voltage of the FeFET device. Under the positive gate pulse, the polarization direction is towards the channel, and this results in a low- V_{th} state corresponding to a logic 0 in the device. Similarly, a negative pulse results in a high- V_{th} state corresponding to a logic 1. FeFETs can store multiple V_{th} levels through partial polarization of the Fe layer. The V_{th} of the device can reach intermediate states by applying voltage pulses to the gate of the FeFET.

Previous works have demonstrated the use of FeFET memory array to perform in-memory logic operations [86, 87] as a ternary CAM [85] or an analog CAM [88]. HDC encoding and MCAM-based nearest-neighbor search are accelerated in FeFET by [89], and both operations require supplementary analog circuitry to perform multi-bit operations. Figure 3.1 shows the organization of two types of FeFET blocks and the supported operations.

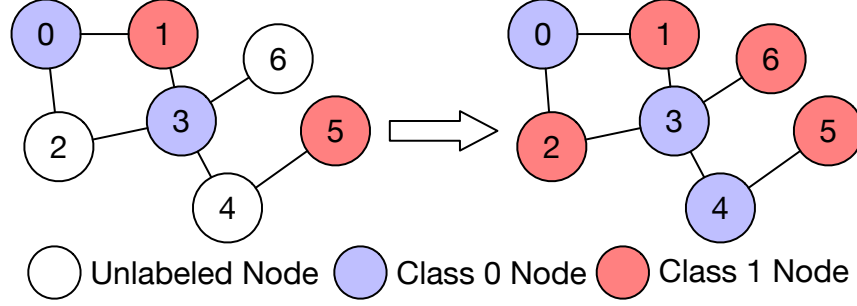


Figure 3.2: Overview of node classification problem

For in-memory computation (see Figure 3.1(a)), we can obtain the analog weighted sums between the input signal I and the weights stored in FeFET cells (W) by sensing the output current of the bit-line (O) as demonstrated in [87]. The results are converted into the digital domain via ADCs, which are connected to a shift-and-add circuit to support high bit-precision (i.e., larger than 3 in this work). These operations have demonstrated that FeFET memory arrays can be used to overcome the data transfer bottleneck, which significantly affects the runtime and total power. Figure 3.1(b) shows the structure of FeFET block for nearest-neighbor search using multi-bit CAM FeFET cells (FCAM cells). Specifically, we can encode the query as the input to FCAM bit-lines and search the most similar rows based on FeFET CAM distance metric [90]. These operations have demonstrated that FeFET memory arrays can be used to overcome the data transfer bottleneck, which has been shown to take a significant portion of the execution time and total power. We can exploit these operations to support all HDC operations.

3.3 RelHD Algorithm Overview

RelHD is a lightweight HDC-based learning framework for graphs. Beyond previous HDC-based algorithms for a graph that focus on memorization of graph structure and limited reasoning capability that requires neural networks, our solution empowers ML on graphs with HDC operations. In this chapter, we focus on the node classification problem, which is a key task of graph-based ML. Figure 3.2 describes the node classification problem in graph. Node 0 and 3

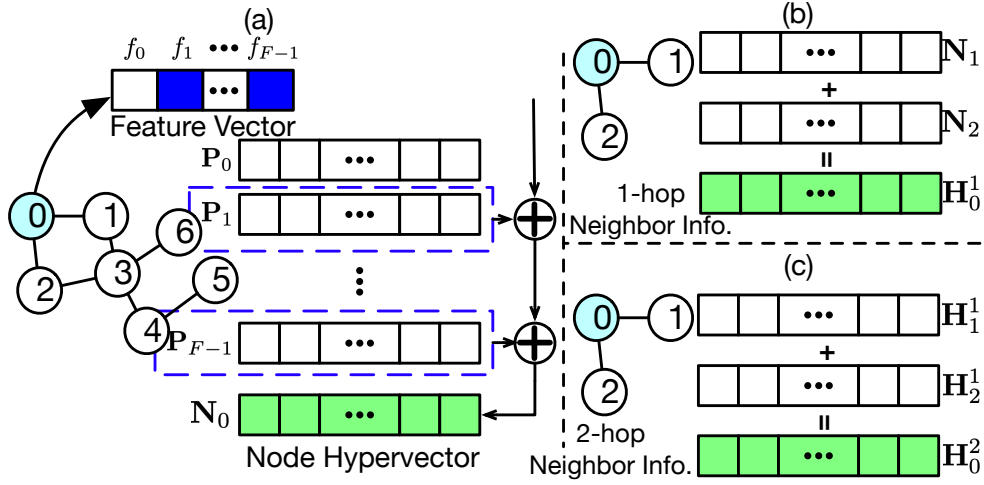


Figure 3.3: Overview of encoding and relation embedding stage in RelHD. (a) Node feature encoding (b) Aggregation of 1-hop neighbor information (c) Aggregation of 2-hop neighbor information. Green-colored hypervectors are used for relation hypervector generation (Eq. 3.1).

are known as Class 0, Node 1 and 5 are Class 1, and other nodes are unlabeled. The goal of node classification is to predict the class of unlabeled nodes given the graph topology, labeled nodes, and node features. It is essential to realize the structural role of the node in a graph.

To this end, RelHD encodes graph nodes by bundling not only the node features but also the multi-hop relations between different nodes. Using binding and bundling HDC operations on graph nodes and their neighbors, the proposed algorithm provides good reasoning capability on graph data. First, RelHD represents the feature of each node in the encoding phase as an HD vector called node hypervector (Figure 3.3(a)). In turn, in the relation embedding phase, information of 1-hop and 2-hop neighbors (Figure 3.3(b), (c)) are collected, and these values and node hypervector are bound to relation hypervector. The training phase generates hypervector representing each class by bundling relation hypervectors with the same label. Finally, the inference finds the most similar class hypervector with the relation hypervector of the unlabeled node.

3.3.1 Encoding Node Features

This section discusses the **encoding** scheme that maps graph node information to the HD space. It encodes features of each node to the *node hypervector*, $\mathbf{N}$. For example, in Cora dataset [91], which predicts the category of a paper given the citation network, a paper is represented as a node, and an edge between nodes indicates the citation link. Each node has features that are described by a bag-of-words (BoW) model—a binary vector indicating the presence of words. The feature vector of node k , i.e., $\mathbf{F}_k$, is $\{f_0, f_1, \dots, f_{F-1}\} = \{0, 1\}^F$ where F is the number of features and $k \in V$.

We propose a new encoding strategy for a dataset with a BoW model, which is inspired by the random indexing methodology [92, 93]. Figure 3.3(a) shows our node feature encoding method for dataset with a BoW model. For each feature, we generate orthogonal hypervectors $\mathbf{P} = \{+1, -1\}^D$ and assign each of them to distinguish the feature positions. Rather than randomly generating HD vectors to achieve orthogonality, we use HDC permutation (cyclic shift). We randomly generate one base hypervector $\mathbf{P}_{base} = \{+1, -1\}^D$. For feature index i , $\rho^i(\mathbf{P})$ is assigned. This method can reduce memory consumption by creating $\mathbf{P}$ in place and be implemented in a hardware-friendly manner. We build a node hypervector based on feature vector and $\mathbf{P}$ by accumulating $\mathbf{P}$ corresponding to feature index only when feature value f exists (blue-colored box in Figure 3.3(a)). In other words, the node hypervector of node k is computed by $\mathbf{N}_k = \sum_{j \in A} \mathbf{P}_j$ where $A = \{x \mid f_x \neq 0, f_x \in \mathbf{F}_k\}$.

In the case of graph dataset with dense features, e.g., Reddit [94], we can create node hypervector from node features using existing HDC encoding methods such as non-linear encoding [28] and ID-Level encoding [15, 27]. In RelHD, we use ID-Level encoding since the proposed FeFET accelerator (Section 3.4) works on fixed-point numbers.

3.3.2 Relation Embedding

Given the node representation in HD space, RelHD learns the graph structure by gathering information of nearby nodes in the **relation embedding** stage. In particular, we create *relation hypervector* for each node, which embeds the (1) node hypervector, (2) 1-hop neighbor information, and (3) 2-hop neighbor information. As illustrated in Figure 3.3(b), relation embedding stage collects information regarding 1-hop and 2-hop neighbor nodes for each node. We achieve this with an hypervector bundling operation. Let the hypervector representation of 1-hop neighbors of node k is $\mathbf{H}_k^1$. We aggregate node hypervectors of neighbors, i.e., $\mathbf{H}_k^1 = \sum_{j \in B} \mathbf{N}_j$. B is a set of the index of node k 's 1-hop neighbors. It can be easily obtained from the edge set E , which has the origin and the destination of edges.

In a similar manner, 2-hop neighbor node information can be aggregated as shown in Figure 3.3(c). Backtracking to search for a 2-hop neighbor can be computationally demanding. Instead, we reuse $\mathbf{H}^1$ and set B to build hypervector representation of 2-hop neighbors, $\mathbf{H}^2$. Specifically, for node k , $\mathbf{H}_k^2$ can be computed by $\sum_{j \in B} \mathbf{H}_j^1$. Here we reuse B from the 1-hop neighbor hypervector ($\mathbf{H}^1$) computation step.

The encoding and the relation embedding stages build three properties for each node: $\mathbf{N}$, $\mathbf{H}^1$, and $\mathbf{H}^2$. RelHD treats each of them as characteristics of a node. Using the basic operations of HDC, we can bind these to relation hypervector, $\mathbf{I}$. For node k , $\mathbf{I}_k$ is computed by

$$\mathbf{I} = \mathbf{N} \odot \phi_0 + \mathbf{H}^1 \odot \phi_1 + \mathbf{H}^2 \odot \phi_2 \quad (3.1)$$

where $\odot$ indicates the element-wise multiplication (binding operation). Note that ϕ_0 , ϕ_1 , and ϕ_2 are bipolar, orthogonal hypervector with D dimensionality, $\{+1, -1\}^D$.

3.3.3 Training and Inference

During the **training** stage, we generate an HDC model that consists of *class hypervectors* $\mathbf{C}$, which represents each class using hypervectors. We bundle all relation hypervectors ($\mathbf{I}$) of the same class, i.e., i th class hypervector $\mathbf{C}_i$ can be obtained with $\mathbf{C}_i = \sum_{j \in L} \mathbf{I}_j$ where L is a set of the index of node that belongs to class i . Here, the required bit for class hypervector increases dramatically since the number of nodes is much greater than the number of classes. We quantize the HDC model to reduce the memory requirements and increase inference efficiency. As such, we quantize the model by only taking sign bits, which have been used in previous HDC-based algorithms [15, 28]. Our experimental results showed that quantizing the model to a single bit hardly degrades the accuracy (less than 0.01%).

In some cases, the accuracy of the HDC model can be improved by **retraining**. Since the label of each node in the training set is known, RelHD compares the predicted class to the ground truth based on the current HDC model. If the prediction is incorrect, the class hypervector is updated by subtracting it from the incorrect class and adding it to the correct class. In other words, let the relation hypervector of current training node as $\mathbf{I}_{curr}$, ground truth label as i and the predicted label as j . If $i \neq j$, we subtract $\mathbf{I}_{curr}$ from $\mathbf{C}_j$. Subsequently, we add $\mathbf{I}_{curr}$ to $\mathbf{C}_i$. Note that retraining can cause overfitting, especially if the training set is smaller than the test set.

The **inference** stage predicts the label of an unlabeled node; it finds the most similar class hypervector to the test data. As such, we first build relation hypervector $\mathbf{I}$ of test nodes (queries) by reusing $\mathbf{N}$, $\mathbf{H}^1$, and $\mathbf{H}^2$. For each query ($\mathbf{I}$), the inference module calculates the similarity to the class hypervectors and selects a class with the highest similarity, i.e., $\arg \max_i \delta(\mathbf{C}_i, \mathbf{I})$.

3.4 FeFET-based PIM Acceleration for RelHD

We exploit the emerging FeFET technology to design an ultra-efficient PIM accelerator for RelHD. The proposed accelerator consists of multi-bit FeFET memory blocks that support

efficient analog-based arithmetic computations and nearest neighbor search. Previous FeFET accelerators can efficiently process HDC operations [89], but they can only encode limited feature vectors fitting in a single FeFET block. However, in the RelHD algorithm, we need to generate hypervectors for each node during the encoding and relation embedding stage. The number of hypervectors may easily exceed the number of rows available in a FeFET block for most real-world graphs. Furthermore, the randomness of a graph may lead to unbalanced utilization of hardware resources and poor parallelism because RelHD introduces data dependency between different graph nodes in the relation embedding phases.

We propose a hardware-software co-design optimized for RelHD. We first propose a new FeFET-based PIM architecture. The key novelty of RelHD hardware is the cluster-based architecture, which groups FeFET blocks into clusters where each cluster adopts near-memory logic to handle inter-block operations. With the near-memory logic, the PIM accelerator can exploit the high throughput of PIM for inter-block operations (i.e., hypervector summation). The cluster-based architecture is flexible to support various-sized graph data by carefully allocating memory resources in each step of RelHD. However, the memory allocation problem has a design space where a straightforward and naive strategy may result in inefficient performance. To efficiently address the memory allocation problem, we propose a graph-aware memory allocation algorithm that minimizes conflicts between different graph nodes and edges, thereby leading to high parallelism.

3.4.1 Cluster-based PIM Architecture

Figure 3.4 shows the proposed FeFET-based PIM architecture for RelHD. The FeFET memory consists of two types of FeFET blocks – compute memory blocks and content addressable memory (CAM) blocks, to support HDC operations. Here, we assume both types of blocks store up to 3-bit data in each memory cell. The compute blocks handle all arithmetic operations in the relation embedding and encoding, while the CAM blocks support a similarity check during the

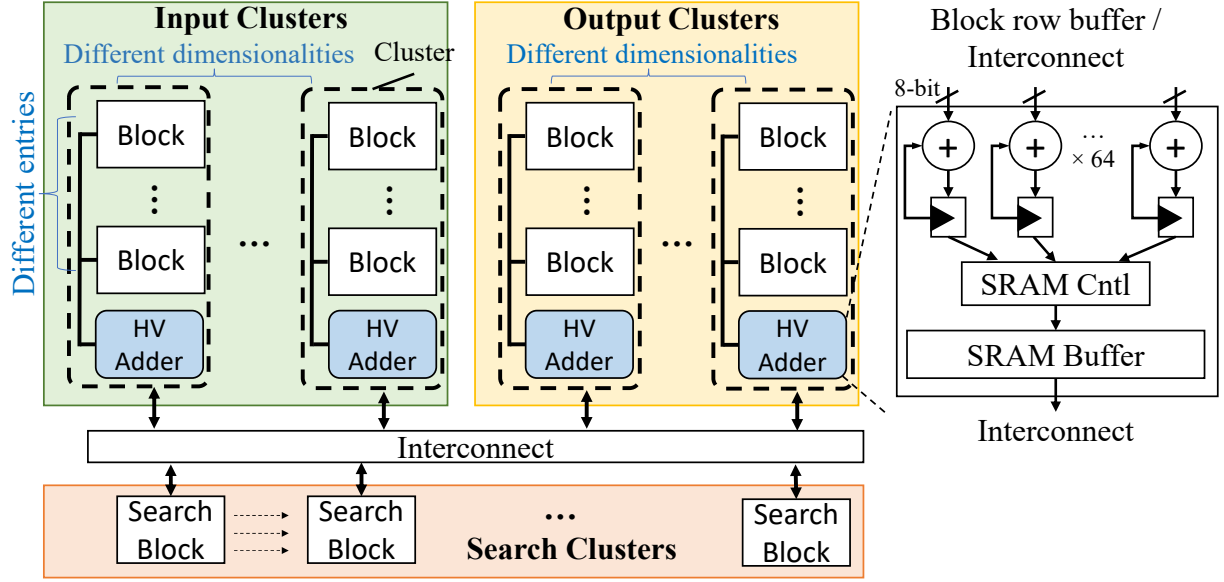


Figure 3.4: Architecture of RelHD PIM accelerator.

inference. On the top of the block-level, we add a cluster-level where we group a fixed amount of blocks into a cluster. We denote the cluster of the compute block as compute cluster and the cluster of the CAM block as the search cluster.

Compute Cluster

As introduced in Section 3.2.2, each FeFET compute block supports vector-matrix multiplications between the input signal and the cell values based on Kirchhoff’s law. We can exploit vector-matrix multiplications in each block to process the summation of multiple row vectors in a block, which can be utilized to support the basic operation in encoding/relation embedding steps of RelHD (i.e., sum up relevant hypervectors). However, in RelHD, the summation of relevant hypervectors depends on the graph structure, which is often random. If a memory block cannot store all vectors, vectors for each summation may be located in different memory blocks. In this case, we need to write partial sums of different blocks back to the memory to generate the global sum. Therefore, previous FeFET HDC accelerators [89] cannot scale to support RelHD because both data movements and memory writes are costly.

In the proposed PIM architecture, we exploit the cluster-level processing to handle inter-block hypervector summations. The key components of per-cluster logic include a hypervector adder (HVA) and an SRAM-based scratchpad. The HVA has a vector adder that sums hypervectors from different blocks. The HVA is connected with the data link from blocks in the cluster as well as the inter-cluster interconnect. Considering the vector adder supporting the full length of the hypervector (e.g., $D = 4096$) can be too costly, we adopt a relatively short vector adder and sums up all hypervectors in multiple iterations. Each HVA has 64 8-bit adders which can accumulate 64 elements at a time. If the input of HVA comes from a block in the cluster, HVA keeps reading 64 8-bit values from the latch of shift-add results. The partial sum of each vector block is stored in the SRAM-based buffer. With the cluster-level summation, compute blocks can only handle efficient vector-matrix multiplications, while the HVAs can process inter-block hypervector summation and data movement independently. Such architecture adds cluster-level parallelism, which improves the throughput and internal bandwidth utilization during the inter-block vector summation.

Search Cluster

CAM blocks in our accelerator can efficiently find the most similar hypervector to a query hypervector, similar to [90]. The search blocks store trained class hypervectors in rows. The query hypervector, provided as input to the CAM cells, is compared to all the stored class hypervectors, and the most similar class against the input is reported. When the dimension of hypervectors exceeds the number of columns in a CAM block, we concatenate several blocks and aggregate currents from all blocks during the search operation. We also adopt a cluster-based organization for CAM blocks. Specifically, we group several concatenated blocks as a cluster, where the concatenated length is sufficient for the most commonly used hypervector dimensions (e.g., $D = 10000$). Since the number of classes is usually small, each cluster can store a copy of trained hypervectors, and we can improve the throughput of the search by using several search

clusters.

Interconnect Network

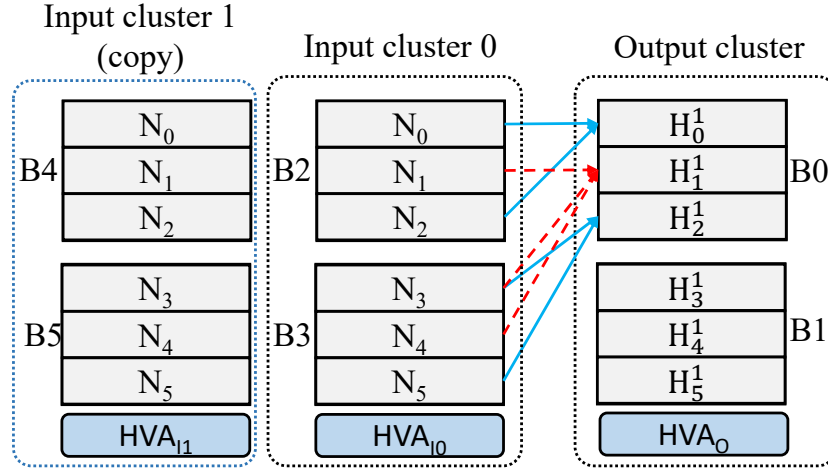
To handle the data movements required for each phase of RelHD, we connect all clusters through a shared-bus-based interconnect where per-cluster controllers can read/write data from/to other clusters. Each cluster controller has links to all blocks in the cluster, which can handle intra-cluster data movements.

3.4.2 Processing Flow

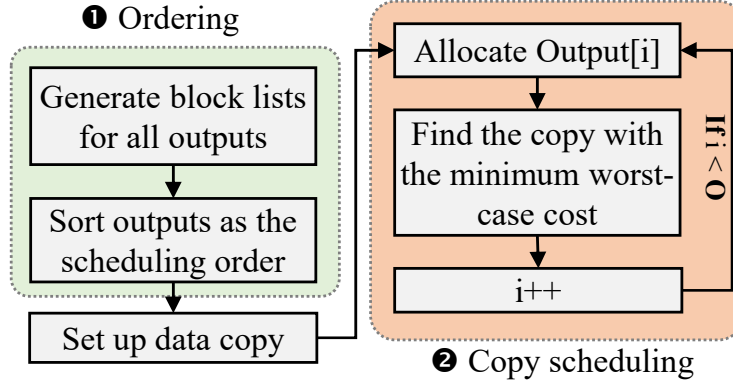
We propose a flexible processing flow using compute and CAM clusters for RelHD. During each encoding/relation phase, we allocate compute clusters for different data and operations. We first divide all clusters into input and output clusters, which store input data and output data, respectively. For each input vector or output vector, we allocate one memory row in a block or assign multiple memory blocks if the number of vectors is larger than the number of rows in a block. After allocation, we can accordingly schedule operations for allocated data in each cluster.

For example, to calculate an output 1-hop/2-hop hypervector by summing up all neighboring node hypervectors, each memory block in the input clusters first calculates the sum of selected node hypervectors by (1) activating corresponding rows and (2) sensing the summation with ADCs. Intra-block calculations are processed simultaneously in all blocks and finished at the same time because the latency of PIM vector-matrix multiplication is independent of the number of activated rows. Per-block partial sums are read by cluster HVAs to generate per-cluster partial sums. The output cluster collects partial sums from input clusters using the interconnect and uses its HVA to generate the 1-hop/2-hop hypervectors. They are stored back in the allocated memory row. During the training phase, the output clusters first calculate the class hypervectors and then send the class hypervectors to CAM clusters for inference.

In addition to the parallel computation on the partial sums for one output hypervector, we



(a) Conflicts of naïve scheduling



(b) Data and hardware-aware heuristic scheduler.

Figure 3.5: Conflict issue and the proposed optimization scheme.

parallelize computations for different output hypervectors if all computations do not have conflict when using hardware resources. We can also increase the compute throughput by duplicating data to resolve conflicts when the memory is sufficient. Therefore, the performance of PIM acceleration depends on the graph structure, data allocation as well as operation scheduling, resulting in a design space that needs exploration. In the next section, we propose a graph-aware optimization for data allocation and operation scheduling to maximize the efficiency of PIM acceleration.

3.4.3 Graph-aware Optimization

Operation scheduling and data allocation are critical to RelHD performance since the graph-based workload has complex data and computation patterns. The key constraint is that the PIM architecture cannot simultaneously sum up input vectors for two different outputs if they share a common set of inputs. Therefore, the performance of RelHD depends on the order of operations as well as the locations of data. Naive scheduling and allocation may cause a lot of conflicts for consecutive operations, significantly slowing down the performance. As such, we propose several data-aware optimization techniques to efficiently exploit the PIM architecture, especially effective for the relation embedding phase of RelHD that uses edge information.

Maximizing parallelism in clusters

We divide the clusters into input and output for the relation embedding phase. Input/output vectors are allocated to different rows of memory blocks in the corresponding clusters. Figure 3.5 shows an example of calculating 1-hop hypervectors ($\mathbf{H}_{0-5}^1$) with node hypervectors (N_{0-5}). In the naive allocation, we allocate $\mathbf{H}_{0-5}^1$ to the output cluster (B0 and B1), and N_{0-5} to the input cluster 0 (B2 and B3). Calculation of each 1-hop hypervector requires corresponding blocks to sum up node hypervectors. For example, to calculate $\mathbf{H}_0^1$, B2 first sums up N_0 and N_2 . The HVA of input cluster 0 (HVA_{I0}) can directly read the sum of N_0 and N_2 from B2 and send it to the output cluster. For $\mathbf{H}_1^1$, HVA_{I0} needs to sum up partial sums from both B2 ($\mathbf{N}_1$) and B3 ($\mathbf{N}_3 + \mathbf{N}_4$). We can exploit different forms of parallelism during the relation embedding to improve the throughput because the calculation of an output vector is independent of other output vectors. As shown in the example of Figure 3.5(a), we can calculate $\mathbf{H}_0^1 = \mathbf{N}_0 + \mathbf{N}_2$ and $\mathbf{H}_2^1 = \mathbf{N}_3 + \mathbf{N}_5$ on B0 and B1 in parallel. However, we cannot simultaneously schedule $\mathbf{H}_0^1 = \mathbf{N}_0 + \mathbf{N}_2$ and $\mathbf{H}_1^1 = \mathbf{N}_1 + \mathbf{N}_3 + \mathbf{N}_4$ because of conflict on B0. As in real-world graphs, where we need to allocate thousands of nodes in memory, the naive allocation may cause a lot of conflicts that significantly limit the system's performance.

One way to reduce the conflict is to schedule independent output vectors together. For example, we can schedule $\mathbf{H}_1^1$ calculation after $\mathbf{H}_0^1$ and $\mathbf{H}_2^1$ so that the memory can fully utilize both B2 and B3 for all calculations. However, finding the best ordering is an NP-hard problem that can only be solved by exhaustively evaluating all permutations. Another way to reduce conflict and improve parallelism is data duplication: different copies can calculate different output vectors independently without conflict. As shown in Figure 3.5, we duplicate N_{0-5} to the input cluster 1 (B4 and B5) which can handle the computation of $\mathbf{H}_1^1 = \mathbf{N}_1 + \mathbf{N}_3 + \mathbf{N}_4$. In this case, we can calculate $\mathbf{H}_0^1$, $\mathbf{H}_1^1$, and $\mathbf{H}_2^1$ in parallel without conflict. However, duplication will significantly increase the complexity of the original scheduling problem. We designed a scheduler that minimizes conflicts while considering data patterns.

Allocation Optimization

Figure 3.5(b) depicts the proposed algorithm for data allocation. It helps to avoid block conflicts by allocating the $\mathbf{O}$ output vectors with a large number of block conflicts to different input copies. The allocator runs in two steps: (1) ordering and (2) copy scheduling. In the ordering step, we sort the output vectors based on the hardware utilization. The allocator starts by generating the input block list for each output vector based on the input-output relationship as well as the hardware configuration. For example, in Figure 3.5(a), $\mathbf{H}_0^1$ only has B0 in its block list, while $\mathbf{H}_1^1$ has both B0 and B1. In turn, we sort the outputs based on the number of blocks, starting from the output vector with the highest number of blocks. With the sorted outputs, we can balance the block usage more easily by allocating outputs requiring large numbers of blocks to different copies.

Before the copy scheduling step, the allocator needs to determine the number of copies for input data based on the workload and hardware configurations. Since different RelHD phases run sequentially, we maximize the number of input copies under a certain capacity to fully utilize the memory. After setting up data copies, it allocates an input copy for each output in the sorted order

Table 3.1: Architecture Parameters

Block	64* 64 3-bit cells, 8 columns/ADC
Cluster	2048 blocks, 64 8-bit adders, 8KB SRAM buffer
Tile (1Gb)	128 clusters, interconnect bandwidth = 8GB/s
FeFET compute	$t_{Add}/t_{RD} = 2.65ns$, $e_{Add} = 48.06pJ$, $t_{WR} = 11ns$, $e_{WR} = 104.18pJ$
FeFET CAM	$t_{Search} = 1.5ns$, $e_{Search} = 0.069fJ$

using an architectural simulation, which updates the timing information of memory blocks based on scheduled computations. When allocating the computation of each output, the simulation evaluates the cost of using each input copy based on the current system status. The allocator chooses the allocation that results in the minimum worst-case cost (the maximum latency among all blocks). The simulator updates the status of memory blocks by simulating the computations on allocated blocks. The allocation algorithm completes after allocating the input copy for all output elements.

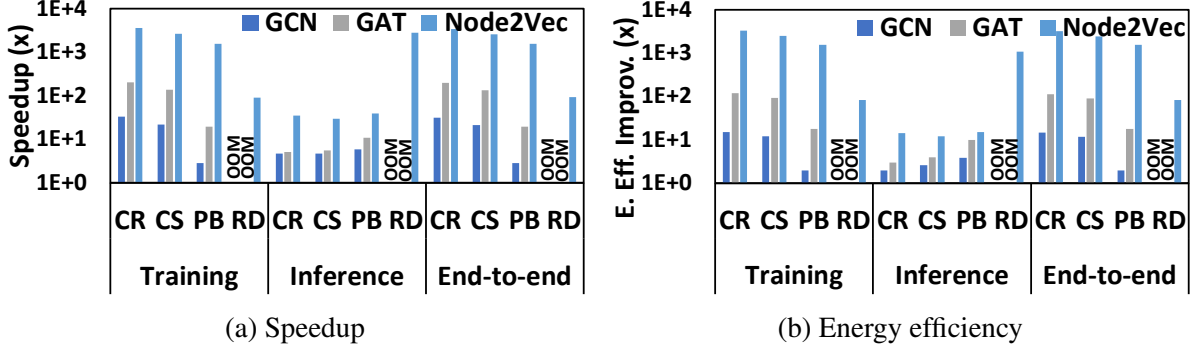
3.5 Evaluation

3.5.1 Experimental Setup

We implement RelHD on NVIDIA Geforce GTX 1080Ti GPU with CUDA v11.3 and the proposed FeFET accelerator. We measured the GPU power consumption using `nvidia-smi`. We implemented RelHD on GPU using optimized GPU-based HDC implementation [38,95]. Baseline algorithms are implemented with PyTorch Geometric [96]. Table 3.1 shows the architectural parameters for the proposed FeFET-based accelerator. We use the published FeFET device data [89, 90] (45nm technology). NeuroSim [97] generated latency and energy estimates for different operations in FeFET blocks as well as for peripherals (e.g., ADCs, shift-and-add, etc.). We implement the customized logic (i.e., HVA) using Verilog HDL and synthesize the design on Synopsys Design Compiler using 65nm library (scaled to 45nm). The synthesized design is placed and routed using Synopsys IC Compiler. We integrate these values with a PyTorch-based

Table 3.2: Statistics of the datasets

Dataset	Nodes	Edges	Features	Classes
Cora (CR)	2708	5429	1433	7
CiteSeer (CS)	3327	4732	3703	6
Pubmed (PB)	19717	44338	500	3
Reddit (RD)	232965	114615892	602	41

**Figure 3.6:** Improvement of RelHD over baselines on GPU. For RD, GCN and GAT on GPU ran out of memory (OOM).

simulation infrastructure to evaluate the end-to-end performance of RelHD accelerator.

Datasets and Baselines We used representative network dataset, Cora (CR) [91], CiteSeer (CS) [98], Pubmed (PB) [99], and Reddit (RD) [94]. Attribute information for these datasets is listed in Table 3.2. We compare RelHD to state of the art algorithms - Node2Vec [83], GCN [44] and GAT [45]. We used the same hyperparameters as in the original publications (e.g., number of hidden layers) and sparse matrix compression in Node2Vec. RelHD uses hypervector dimensionality (D) of 4096. Our PIM architecture is scalable; we can increase the number of tiles to accommodate more data duplication for better parallelism. We use enough memory tiles to enable 4-duplication for each dataset. Specifically, we use 1/1/2/72 tiles for CR/CS/PB/RD respectively.

3.5.2 Comparison to Baselines running on GPU

RelHD trains the model with vector addition, while baseline algorithms require compute-intensive operations such as a random walk or matrix multiplication. Furthermore, the proposed algorithm does not use backpropagation and achieves high accuracy with a single training epoch. RelHD reaches the maximum accuracy without retraining steps on CR, CS, and RD. In the case of PB, RelHD only two epochs are required for retraining to achieve peak accuracy. Conversely, baseline algorithms typically need at least 70 epochs to get the best accuracy. Note that GCN and GAT showed out of memory on GPU, which is consistent with the result in existing works [100, 101].

First, we compare the training time. For a fair comparison, we include the execution time of encoding, relation embedding, training, and retraining phase in RelHD since baseline algorithms include feature extraction during the training step. Figure 3.6(a) compares the training time of RelHD and baselines. RelHD yields an average speedup of $1087\times$, $13\times$, and $82\times$ compared to Node2Vec [83], GCN [44], and GAT [45], respectively. In the case of RD, the speedup of RelHD compared to baseline is minimal compared to other datasets; it is not directly proportional to the graph size. Datasets other than RD have a sparse feature vector, while RD has a dense feature vector. The RD case accompanies more arithmetic operations for the encoding stage than other datasets, i.e., RD achieves less speedup. Besides, the training time of Node2Vec increases exponentially as the size of the graph increase (e.g., PB, RD dataset). Therefore, RelHD achieves significant speedup compared to other baseline algorithms.

During the inference stage in RelHD, we measure pairwise similarity between relation hypervectors of query nodes and class hypervectors, as the graph structure is pre-encoded to relation hypervector. As shown in Figure 3.6(a), RelHD is $104\times$, $5\times$ and $6.7\times$ faster on average compared to Node2Vec [83], GCN [44], and GAT [45], respectively. RelHD achieves significant speedup for end-to-end execution compared to the baselines. As illustrated in Figure 3.6(a), RelHD is on average $12\times$, $80\times$, and $1071\times$ faster compared to baseline algorithms:

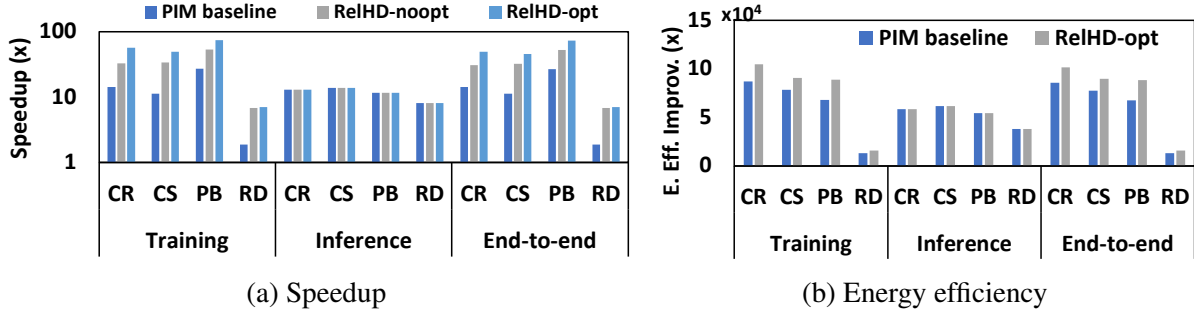


Figure 3.7: Performance of RelHD PIM accelerator versus RelHD GPU implementation

Node2Vec [83], GCN [44], and GAT [45], respectively.

We compared the energy consumption of RelHD and baseline algorithms on GPU. RelHD improves total energy efficiency by $1007\times$, $7\times$, and $57\times$ on average over Node2Vec [83], GCN [44], and GAT [45], respectively, as shown in Figure 3.6(b). Here, RelHD inference showed speedup and energy efficiency improvement, $19\times$ and $10\times$ on average, respectively. Improvement in inference was less than training since RelHD requires similarity computation on large hypervectors.

3.5.3 FeFET-based RelHD Accelerator Efficiency

We compared the performance and efficiency of RelHD accelerator to GPU and two FeFET-based accelerators. The “PIM baseline” is similar to MIMHD [89], a previous FeFET-HDC accelerator which processes all computations by using FeFET analog operations. We fit MIMHD blocks in the same architecture as RelHD accelerator, except that the “PIM baseline” does not contain HVAs. “PIM baseline” exploits the FeFET blocks of the output vectors for summing up inter-block input vectors, resulting in many writes. It also does not use our scheduling optimizations. The “RelHD-noopt” uses the same hardware as the proposed RelHD accelerator, but does not use the scheduling optimizations. The “RelHD-opt” is our final PIM design, which applies our optimization schemes on the “RelHD-noopt”.

Our evaluation shows that “RelHD-opt” achieves $33\times$ of speedup and $59,287\times$ energy

efficiency improvement on average over the GPU (see Figure 3.7). RD shows limited speedup and energy improvement compared to other datasets as the graph in RD is dense. Compared to “PIM baseline”, RelHD is $3.5\times$ faster and $1.2\times$ more energy efficient. As compared with “RelHD-noopt” systems, our optimized allocation and scheduling strategy improves the relation embedding phase speed by leveraging the data relationship information in graphs. It maximizes parallelism, resulting in $1.3\times$ speedup on average. Despite the reduced runtime, the difference of energy consumption between “RelHD-opt” and “RelHD-noopt” is small because our optimizations only change the order and the memory locations of the computations.

3.5.4 Comparison to the state-of-the-art GNN accelerator

We compared our accelerator to a state-of-the-art GNN inference accelerator, PIM-GCN [46], which is the fastest GCN accelerator based on emerging crossbar memory. For the inference, RelHD accelerator uses pre-encoded relation hypervectors, which was done during the model training. Our evaluation shows that RelHD accelerator takes 3.8×10^{-6} seconds and consumes 2.2×10^{-6} J for CR, CS, and PB workloads because a 64-row block can store all classes for all workloads, leading to a constant search time for each node. In the case of RD, our accelerator runs in 2.1×10^{-4} seconds and uses 1.2×10^{-4} J of energy. As a comparison, PIM-GCN takes 5.6×10^{-5} , 9.5×10^{-6} , 3.3×10^{-5} , and 7.6×10^{-3} seconds, while consuming 2.8×10^{-4} , 9.2×10^{-4} , 3.8×10^{-3} , and 1.26 J on CR, CS, PB, and RD dataset, respectively. The results show that RelHD accelerator is $2.5\times$ to $14.7\times$ faster than PIM-GCN, while improving the energy efficiency by $127\times$ to $10,282\times$. RelHD PIM accelerator achieves much better energy efficiency because of the MCAM search operations. Also, note that RelHD accelerator showed the best speedup and energy efficiency improvement in large-sized graph, RD.

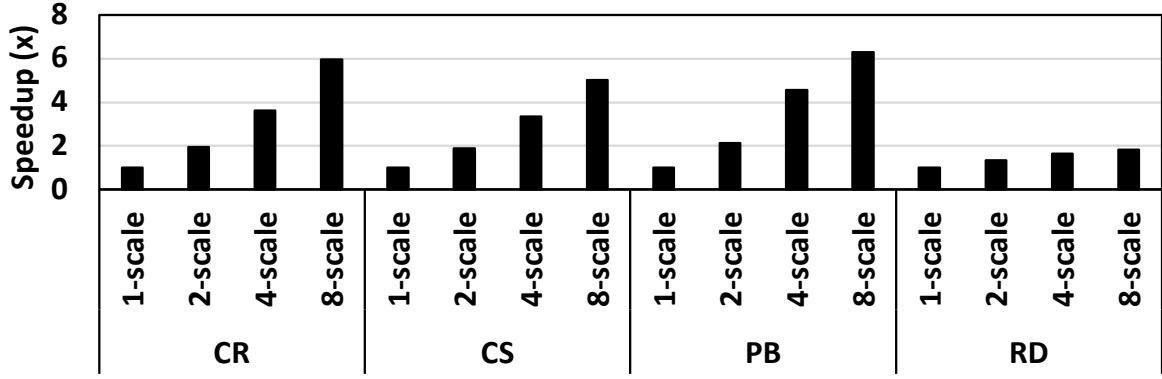


Figure 3.8: The scalability of RelHD.

3.5.5 RelHD Scalability

Similar to the GCN and GAT, large-size graphs can be challenging to run. RelHD accelerator can offer a high degree of parallelism and scalable memory bandwidth by increasing the number of compute and CAM clusters. Also, unlike existing FeFET-based HDC accelerators [89], the proposed hardware supports massive HDC operations in a scalable manner. Figure 3.8 shows the speedup according to the number of compute and CAM clusters. Because different workloads have different memory requirements (from hundreds of Mbs to tens of Gbs), we scale the memory capacity from the minimum requirement (i.e., 1-scale) to $8\times$ of the minimum requirement for each workload. Our results show that increasing the memory by $8\times$ can provide $6\times$, $5\times$, $6\times$, and $2\times$ speedup for CR, CS, PB, and RD respectively. It implies that the RelHD accelerator scales well for different datasets. RD provides less speedup than other datasets because data is large and dense, introducing more data movement overhead when using more memory.

3.5.6 Overhead Analysis

HVA and 8KB SRAM-based scratchpad takes $0.057mm^2$ chip area and consume $27.34mW$ power in each cluster. The area and power overhead of the proposed hardware are only 6.14% and 2.13% respectively. The most area and power consuming components are FeFET blocks ($0.42mm^2 / 153.6mW$), ADC ($0.31mm^2 / 512.0mW$), DAC ($0.022mm^2 / 512.0mW$), and shift-and-

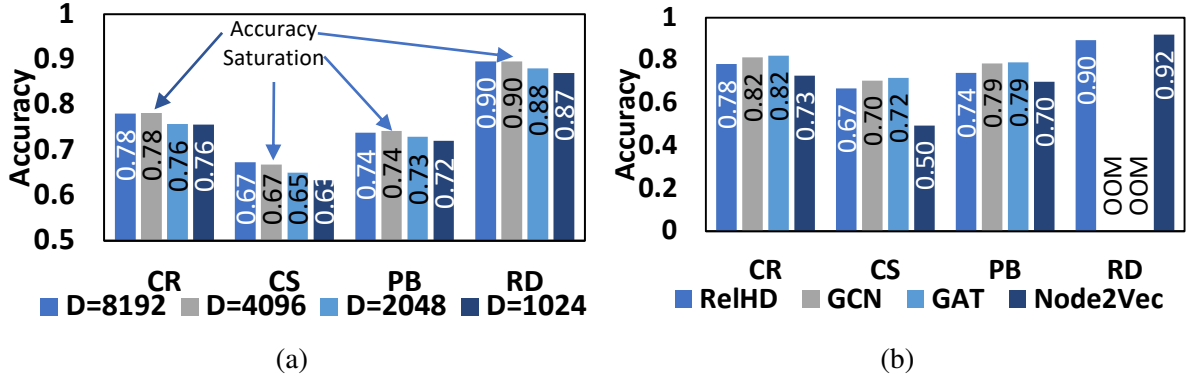


Figure 3.9: Accuracy comparison. (a) RelHD accuracy according to hypervector dimensionality (D) (b) RelHD versus baselines. OOM denotes out of memory.

add $(0.12mm^2 / 102.4mW)$.

3.5.7 RelHD Accuracy & Dimensionality

Hypervector dimensionality D is critical since it affects accuracy but at the cost of computational complexity and memory capacity. Figure 3.9(a) shows accuracy changes of RelHD as a function of D . When $D \geq 4096$, the accuracy of RelHD converges. We conclude that at least 4096 dimensionality is needed to encode the node features and the relation between the nodes into hypervectors. Figure 3.9(b) depicts the achieved accuracy of RelHD and the baselines. RelHD yields better accuracy than Node2Vec while showing a slight accuracy drop (0.04 on average) compared to GNN-based solutions. For RD, the node labels are highly dependent on graph topology information. Node2Vec utilizes this information well, thereby showing high accuracy [102].

3.6 Conclusion

In this chapter, we presented RelHD, a lightweight graph-based ML framework based on HDC. It embeds neighbor information and node features in HD space and predicts the label of an unlabeled node with simple HDC operations. We designed a scalable FeFET-based PIM

accelerator for RelHD, with optimized data allocation and scheduling to maximize the parallelism on an irregular graph. Evaluation results show that RelHD is up to $32\times$ faster compared to GNN-based algorithm on GPU while offering comparable accuracy. Furthermore, the proposed PIM accelerator significantly enhances speed and energy efficiency by $33\times$ and $59,287\times$ on average, respectively, over GPU. RelHD on PIM yields on average $10\times$ speedup and $986\times$ energy efficiency improvement compared to the state-of-the-art crossbar memory-based GNN accelerator.

This chapter contains material from “RelHD: A Graph-based Learning on FeFET with Hyperdimensional Computing”, by Jaeyoung Kang, Minxuan Zhou, Abhinav Bhansali, Weihong Xu, Anthony Thomas, and Tajana Rosing, which appears in IEEE International Conference on Computer Design (ICCD), 2022. The dissertation author was the primary investigator and author of this paper.

In the next chapter, we demonstrate that HDC can effectively handle large-scale problems. We introduce a new solution for mass spectrometry-based proteomics that significantly improves speed and energy efficiency. The proposed solution consists of a highly parallel HDC-based open modification spectral library search algorithm with hardware acceleration using GPUs and a DRAM-based accelerator.

Chapter 4

Massively Parallel Open Modification

Spectral Library Searching with

Hyperdimensional Computing

4.1 Introduction

In the previous chapter, we discussed the HDC-based graph ML algorithm and its implementation on GPU and PIM. Current HDC-based applications, including RelHD, showed remarkable speed and energy efficiency improvement compared to existing ML approaches. However, HDC-based solutions seldom focus on a large-scale problem, which takes a significant portion of the ever-rising energy demand for computation. In this chapter, we scale out the target task for HDC. We focus on a large-scale pattern-matching problem in bioinformatics, which plays a crucial role in personalized healthcare and drug discovery.

Proteomics plays an essential role in understanding the molecular mechanisms of proteins, which are responsible for various tasks in a life of a cell. Protein biomarkers are used to predict disease progression and severity. It can provide early diagnosis and aid the therapeutic strategy

design. Mass spectrometry (MS) is one of the most popular and reliable approaches to identifying and quantifying proteins and peptides in biological samples. During shotgun proteomics experiments, first proteins are digested into peptides, i.e., short sequences of amino acids. Next, the peptides are ionized to obtain a charge, and they are measured using tandem mass spectrometry (MS/MS) to obtain fragmentation spectra. In a typical MS experiment, a method called tandem mass spectrometry (MS/MS) often generates millions of spectra data [103]. Researchers determine peptide annotations of the MS/MS spectra via spectral library searching [104]. Peptide sequences are assigned to experimental MS/MS spectra by matching them against a spectral library of known peptides (see Figure 4.1).

The challenge of spectral library searching is that a significant portion of MS/MS spectra acquired during an experiment cannot be directly identified [105] using searching approaches with conventional similarities, such as cosine similarity. In reality, proteins undergo one or more post-translational modifications (PTMs), which change their mass and MS/MS fragmentation pattern. PTMs can be introduced during sample preparation as an artifact of MS measurement [106], or biologically relevant PTMs arise *in vivo* [107]. However, spectral libraries mainly contain reference spectra for unmodified peptides, so PTMs make experimental spectra difficult to identify as they no longer exactly match the reference spectra.

Open modification searching (OMS) is a promising approach to circumvent this limitation and identify modified spectra [108]. Standard spectral library searching only compares experimental spectra to reference spectra with a similar precursor mass, i.e., the mass of the unfragmented peptide, as matching peptides should have an identical mass. In contrast, OMS performs spectra matching on a wider range of reference spectra. It compares modified query spectra to their unmodified reference variants, even when their precursor mass differs due to PTMs. OMS' wider searching range empowers higher identification capability, thus enabling the study of more complex protein interaction in virus-host [109] and proteomics analysis of non-model organisms [110].

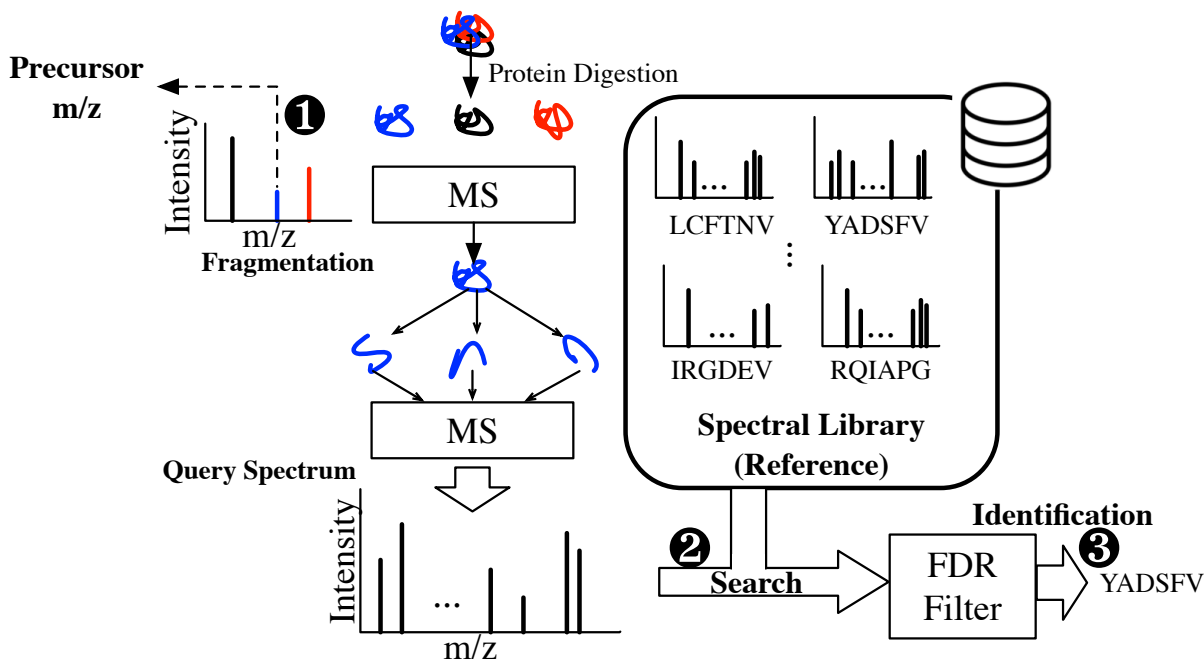


Figure 4.1: Overview of spectral library search. Standard search uses a narrow precursor m/z tolerance, while OMS uses a wide precursor m/z tolerance during the searching.

Compared to standard searching, the OMS suffers from low speed due to the drastically increased search space [47]. This problem is further exacerbated by the increasing spectral data due to the cost reduction in the MS experiment ($2\times$ in recent two years) [111]. Also, large spectral libraries created by repository-scale mining of open MS data become available [112]. For example, the size of human HCD (higher energy collisional dissociation) spectral libraries hold 2.15 million data points, which is $4\times$ larger than the previous NIST-HCD [112]. MassIVE repository [113] contains 6.4 billion spectra, which corresponds to over 500TB in size (as of June 2023).

Several tools have been introduced to efficiently perform OMS [47, 50, 114–116]. These tools use various techniques to refine the search space, such as fragment ion indexing [114], nearest neighbor searching [47, 50], or tag-based filtering [115]. For example, the state-of-the-art OMS tool ANN-SoLo performs nearest neighbor searching using GPU and computes shifted cosine similarities on candidates [50]. The current solutions involve a complex execution pipeline,

have a limited data parallelism, and necessitate high-precision floating-point (FP32) arithmetic for optimal search quality, such as shifted cosine similarity [47]. However, as novel spectral libraries keep growing in size, OMS tools face a “memory wall”, with the runtime dominated by IO operations and limited GPU memory capacity. To cope with the trend, we need an OMS solution that (1) reduces computational complexity and (2) minimizes the cost of data movement.

Memory-centric computing systems, like PIM, are being developed as potential solutions to the “memory wall” problem. They offer extensive parallelism with scalable memory bandwidth, and reduces overhead caused by data movement between processor and memory. Several accelerators [117–119] use near-memory processing (NMP) technique, which integrates computing logic and buffer in an advanced IC package and exposes large internal memory bandwidth. Meanwhile, other PIM-based accelerators, like those described in [12, 28], process data inside memory cells using analog technology, specifically categorized as processing-using-memory (PuM). Several studies [120–122] demonstrated that the integration of PuM and NMP technology can effectively manage each stage of the algorithm. However, implementing existing OMS solutions that use high-precision floating-point arithmetic [47, 50, 115, 116], on current memory-centric architectures is challenging since they require costly peripheral circuits, such as analog-to-digital/digital-to-analog converters [123] and floating-point units [124]. In contrast, the proposed algorithm uses lightweight Boolean arithmetic with high parallelism and a simplified execution pipeline, which can minimize the necessity of peripheral circuits. This approach is ideal for PIM-based hardware acceleration due to its memory-centric and highly parallel nature.

In this work, we propose HyperOMS, an algorithm-hardware co-design for OMS acceleration. Our approach involves a redesign of the OMS algorithm, utilizing the HDC that has shown high efficiency for pattern-matching tasks, particularly on parallel computing platforms. By mapping data into HD space, HDC enhances data separability and robustness [125, 126]. Leveraging HDC’s robustness, we aim to minimize the impact of PTMs. To further increase the efficiency and speed, we develop an accelerator with PuM-NMP hybrid processing on DRAM

that provides significantly higher internal memory bandwidth, lower data movement cost, and extensive data parallelism. To summarize, our contributions are as follows:

- We propose a novel HDC-inspired hardware-friendly OMS algorithm that encodes spectra to a binary vector. Our method reflects the spatial and value locality of peaks in the spectrum, making the encoded data resilient to peak shifts and intensity changes.
- We implement the HyperOMS algorithm on the high-performance HDC library [38, 95] on GPU, obtaining up to $17\times$ speedup and $6.4\times$ energy efficiency over the state-of-the-art OMS solution, ANN-SoLo [50], while offering comparable search quality.
- Based on the GPU profiling result, we identify that the HyperOMS algorithm is a memory-intensive workload. Thus, we design a DRAM-based HyperOMS accelerator that combines PuM and NMP technologies for extended memory bandwidth. To the best of our knowledge, this is the first work that exploits DRAM technology to accelerate an HDC-based algorithm.
- To address hardware utilization challenges that come from the filtering step in OMS, we introduce a scheme for optimizing data organization. The proposed data organization scheme effectively tackles the issue of bank under-utilization that arises during reference filtering, resulting in $3.7\times$ speedup.
- For the first time, HyperOMS tackles a large-scale pattern-matching problem with HD vectors. To alleviate the pressure from large dimensionality in our algorithm, we introduce a strategy that computes a sub-vector similarity only unless the result is larger than the given threshold to accommodate more parallelism on the NMP hardware. We show that it can reduce the execution time by $1.8\times$ with only 0.4% identification loss.
- Our evaluation result demonstrates that the HyperOMS accelerator provides up to $6.9\times$ ($100\times$) speedup and $341\times$ ($1,337\times$) enhanced energy efficiency over HyperOMS running on GPU (ANN-SoLo [50] on GPU).

4.2 Background and Related Work

4.2.1 Mass Spectrometry Background

MS is used to study the biological process in proteomics via the analysis of protein expression or state in cells or tissue. Proteins are ubiquitous building blocks of life, and they are composed of peptides, which are chains of amino acids, which can be described as a string of letters.

During MS data acquisition, peptides are ionized to receive a charge, and their mass-over-charge (m/z) is measured. First intact ions are measured in an MS scan using data-dependent acquisition, and the resulting MS spectrum contains the corresponding m/z values. The most intense peaks in the MS spectrum are selected. It is further analyzed in MS/MS scans, i.e., the second mass spectrometer. Ions with matching m/z are isolated and fragmented to generate MS/MS spectra. Fragmentation occurs along the peptide backbone in between its constituent amino acids. Peptides are split into their possible amino acid subsequences. We record the m/z and intensity values of all fragments, and the measured spectrum forms a *unique fingerprint of the measured peptide*. Thus, each MS/MS spectrum consists of fragment m/z , *spectrum charge*, and its intact m/z from the preceding MS scan, called the *precursor m/z* (Figure 4.1-❶).

Spectral library searching determines which peptide corresponds to the measured spectra [104]. (Figure 4.1-❷). A spectral library contains reference spectra, each with known peptide labels. We first select the reference candidates with a similar precursor m/z to a query spectrum. Next, similarities between the query and all candidates are computed. Finally, the query spectrum is assigned the same peptide label as its highest-scoring reference match. Here, we apply a false discovery rate (FDR) filter on search results [127] (Figure 4.1-❸), which is called target-decoy strategy [127] in MS/MS analysis to reduce false positives. Decoy spectra that cannot exist are added to the spectral library besides the real (target) spectra. Decoy spectra selected by the search tool are filtered out. The number of target SSMs and decoy SSMs at a specific score can be

used to compute the FDR. Typically, an FDR threshold of 1% is used to minimize the number of incorrect identifications. The quality of different search tools can be compared by *the number of identified spectra* at a fixed FDR threshold.

A **standard searching** strategy can identify directly matching spectra. It assumes that precursor m/z of query and matched reference spectra are similar (narrow precursor m/z tolerance). However, as spectral libraries mainly contain unmodified reference spectra, they cannot be used to identify modified ones. Modified ones have a different intact mass, as the modifications induce mass shifts. **Open modification searching** addresses these issues by (i) using a wide precursor m/z tolerance that exceeds mass shifts induced by modifications to select reference candidates [108], and (ii) using alternative spectrum similarity measures that take peak shifts due to modifications into account [47]. Using a wide precursor m/z tolerance enables finding (partial) matches between unmodified reference spectra and their modified variants. However, a large number of candidates need to be evaluated for each query spectrum, which can be computationally demanding.

4.2.2 Accelerated Spectral Library Searching

OMS has recently become an increasingly popular search strategy, and there have been several studies to accelerate searches on parallel hardware platforms other than CPU. Several studies have focused on accelerating spectral library searching using GPUs for efficient spectrum–spectrum similarity computation [128]. [129] used a CPU-FPGA architecture in which multiple FPGAs are used for scalability and parallelism. However, none of these studies have tackled the OMS. ANN-SoLo [50] is a state-of-the-art OMS tool that uses GPU-powered nearest neighbor searching to prune the search space. However, GPU-based ANN-SoLo [50] suffers from limited memory capacity and high data movement cost when handling large databases, i.e., repository-scale spectral library searching. Unlike previous approaches, the proposed accelerator tackle these challenges with PIM technologies and offer scalable memory bandwidth, memory capacity,

and promising efficiency.

4.2.3 Memory-centric Computing and HDC

Previous works have proposed a memory-centric architecture to alleviate the “memory wall” challenge by moving data operations closer to the memory module. There are various choices to realize the computation: (1) integrating computing logic near the bank IO or near-subarray circuits [118, 130], or (2) computing within the memory array using memory commands [131–133].

Besides, existing works have shown that running HDC on PIM hardware is much faster and more energy-efficient than on other parallel hardware platforms like GPUs [25–28, 32]. For instance, [25–28] used ReRAM and recently, [32] used FeFET to enable HDC-based machine-learning. However, as we tackle the large-scale search problem, these technologies are not suitable; the accelerator needs to support a large memory capacity as well as high in-memory computation parallelism. Compared to ReRAM [121, 134] and FeFET [135] devices, DRAM [119] has more mature manufacturing techniques, which make it practical to implement DRAM-based PIM. Additionally, DRAM provides faster writing speed and high-density memory at a low cost and is easier to scale up. Furthermore, it offers higher internal data parallelism, as it allows for the activation and access of more than thousands of bit lines simultaneously [119]. As such, we exploit DRAM to accelerate the HyperOMS algorithm in memory-centric architectures.

4.3 HyperOMS Algorithm

In this section, we introduce the HyperOMS algorithm. HyperOMS encodes raw spectral data into binary HD vectors called *hypervectors* and performs spectral library searching using them. During the encoding process, HyperOMS captures the positions and intensities of peaks while also considering their spatial and value locality. Although peaks may be shifted or have

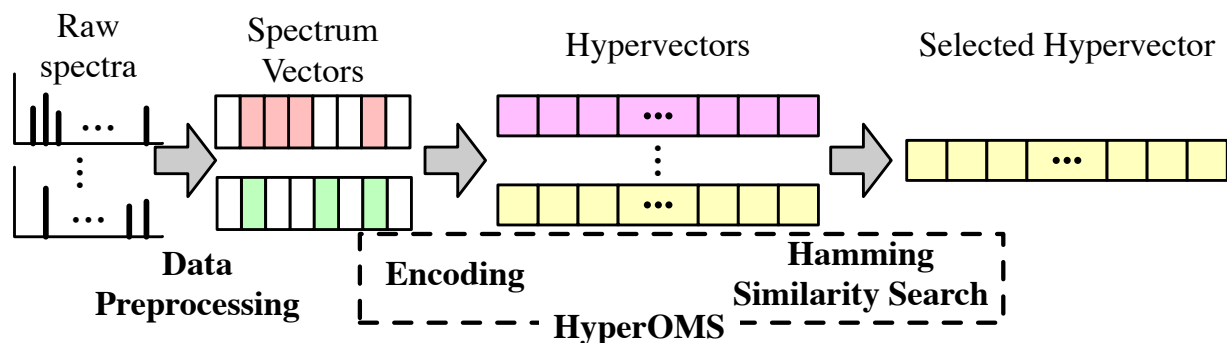


Figure 4.2: Overview of OMS process using HyperOMS.

varying intensities due to PTMs, the similarity between a query spectrum and a matching reference spectrum remains stable. Additionally, since a binary vector representation is used, HyperOMS enables searching using a simple Hamming similarity computation.

Figure 4.2 shows the flow of HyperOMS. It starts with a data preprocessing step, a common step for OMS. It refines and vectorizes raw spectra and compresses them, resulting in spectrum vectors. HyperOMS encodes spectrum vectors into hypervector during the encoding step. Next, the Hamming similarity search step finds the most similar spectra by filtering reference spectra according to the query’s precursor m/z and spectrum charge and computing Hamming similarity between query hypervectors and candidate reference hypervectors.

4.3.1 Data Preprocessing

The preprocessing step (1) refines the raw spectra by removing redundant peaks and (2) vectorizes refined spectra (see Figure 4.3). First, raw spectra are refined to gather meaningful peaks (❶). We remove peaks whose intensity is below 1% of the most intense peaks. Low-intensity peaks are considered noise. In turn, we retain 50 to 150 most intensive peaks of the spectra. Existing studies [47, 50, 116] have shown that we can effectively refine spectra in this manner.

Next, we vectorize the filtered spectra (❷). The peaks are discretized by binning the

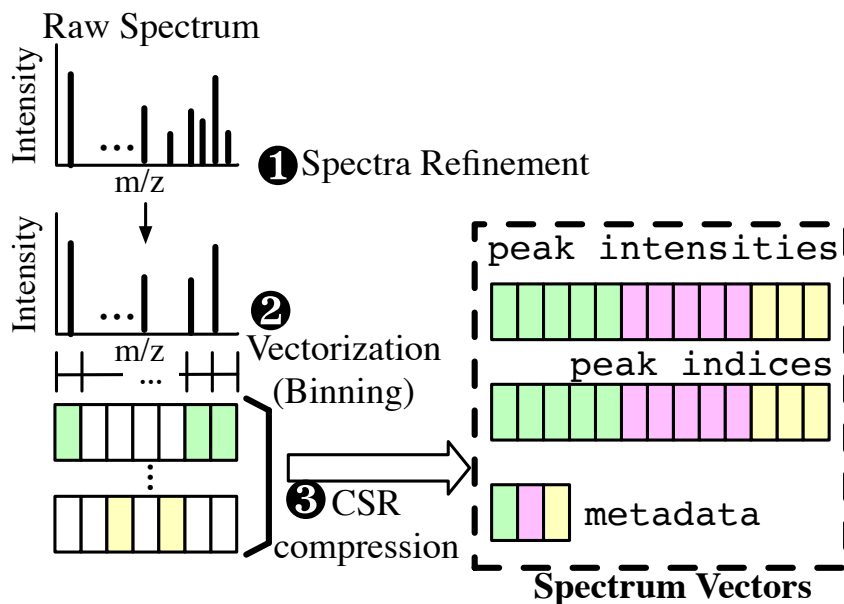


Figure 4.3: Data preprocessing in HyperOMS.

m/z range to represent a spectrum into a sparse vector of floating-point intensity values, called a *spectrum vector*. If multiple peaks are assigned to the same m/z bin, we sum their intensity values. A large bin width can lead to a loss of information when peaks are grouped into a single bin. For example, the mass range between 0 m/z and 2000 m/z and bin width 0.04 (based on the resolution of the mass spectrometer) results in a dimensionality of 50,000. The resulting spectrum vectors have sparsity less than 1%; there are 50 to 150 peaks for each spectrum vector and its dimensionality is 20,000 to 50,000. We compress spectrum vectors in a compressed sparse row (CSR) format (③). The preprocessing step is normally run offline, and the output is stored as a binary file for future use. In the following, we discuss the HyperOMS algorithm, which first encodes spectrum vector to hypervector and performs Hamming similarity search on hypervectors.

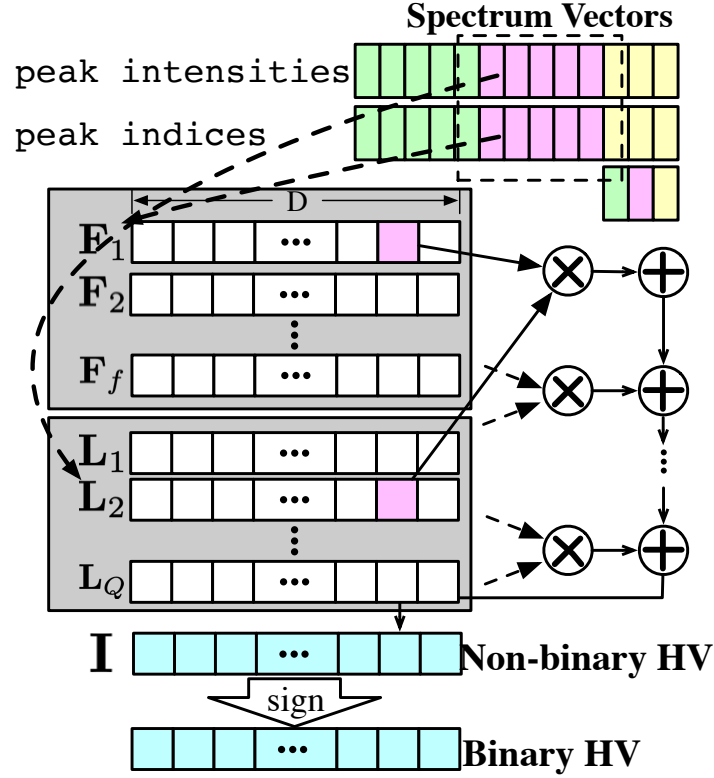


Figure 4.4: Encoding in HyperOMS.

4.3.2 Encoding: Spectrum Vectors to Hypervectors

HyperOMS encodes the data into a *binary vector representation*, which can enhance the computation efficiency. There have been several efforts to represent raw data in an HD binary vector, using Locality Sensitive Hashing (LSH) [28, 136] or HDC [15, 28, 51]. However, these strategies do not reflect the characteristics of OMS, including peak shifts and intensity changes. For example, they treat each feature position (corresponding to peak indices in the spectrum vector) as orthogonal. Peak shifts can lead to significant changes in similarity. Conversely, the proposed encoding takes both *spatial locality* (for peak shift) and *value locality* (for peak intensity change from instrument error or noise) of each feature into account. As a result, we can preserve similarity despite peak changes.

Figure 4.4 shows the encoding process of HyperOMS. Unique *position hypervectors*

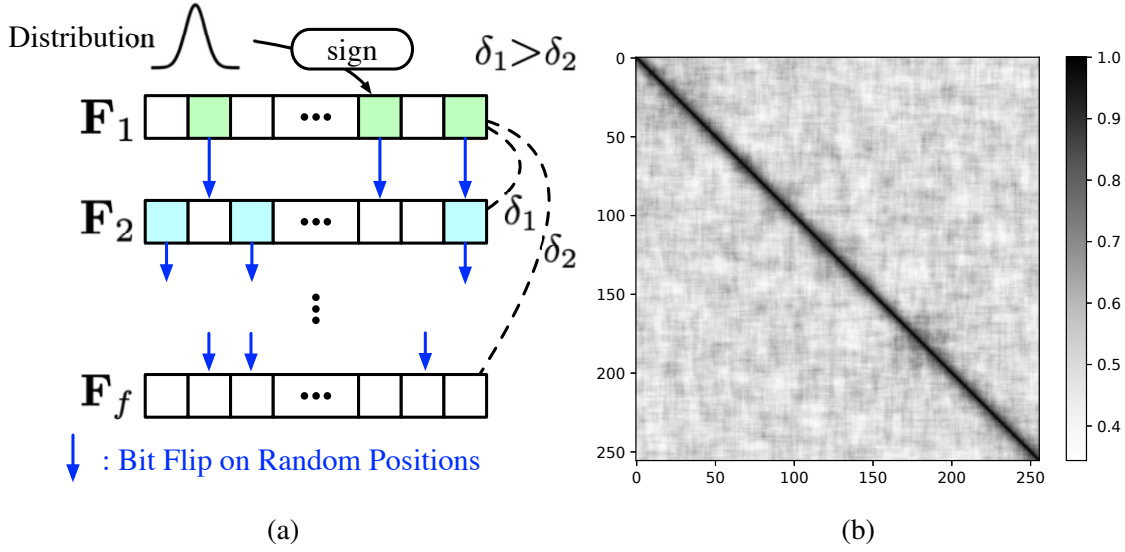


Figure 4.5: Position hypervector generation. (a) Strategy overview. (b) Pairwise similarity (Hamming similarity normalized by the hypervector dimension size) between position hypervectors.

$\mathbf{F}$ are assigned for each index in a spectrum vector, i.e., $\mathbf{F}_i$ corresponds to index i , and $\mathbf{F} \in \{\mathbf{F}_1, \mathbf{F}_2, \dots, \mathbf{F}_f\}$ where f is the dimensionality of spectrum vector. Similarly, we use *level hypervectors* $\mathbf{L}$ to capture different intensity values in each index. We quantize intensity range into Q levels, and $\mathbf{L}_i$ is assigned to each quantization level i where $i \in [0, Q]$. Given two sets of hypervectors, $\mathbf{F}$ and $\mathbf{L}$, a spectrum vector is encoded into an hypervector $\mathbf{I}$ as follows. Let $\mathbb{P}$ be the set of peaks in the spectrum vector, consisting of tuples (i, j) , with i the peak index and j the step value of its intensity. $\mathbf{I}$ is computed as follows:

$$\mathbf{I} = \sum_{(i,j) \in \mathbb{P}} \mathbf{F}_i \odot \mathbf{L}_j, \quad (4.1)$$

where $\odot$ indicates element-wise multiplication. In turn, we binarize the $\mathbf{I}$ for the computational efficiency on hardware; all positive elements are mapped to $+1$ and -1 otherwise. The final representation of the hypervector is a binary vector. Spectrum vectors corresponding to the query and reference spectra are encoded to query hypervectors and reference hypervectors, respectively. Encoding of reference spectra is done only once. The reference hypervectors are reused for

subsequent runs since they are already identified and unlikely to change.

Reflecting Spatial Locality

We introduce a novel position hypervector generation method to reflect the spatial locality. Previous studies [51] have used a permutation-based or random generation method, which makes $\mathbf{F}_i$ and $\mathbf{F}_j$ ($i \neq j$) nearly orthogonal. However, they are vulnerable to peak shifts that accompany changes in i ; the change of position hypervector results in significant similarity value changes of matched pairs.

Figure 4.5(a) shows the proposed position hypervector generation strategy. We randomly generate $\mathbf{F}_1 = \{+1, -1\}^D$. In turn, we flip α components in random positions. As more flips occur, the similarity between the original vector and the flipped vector decreases. For example, the similarity (δ_1) between $\mathbf{F}_1$ and $\mathbf{F}_2$ is larger than the similarity (δ_2) between $\mathbf{F}_1$ and $\mathbf{F}_f$. The proposed encoding method reflects the characteristics of peak shifts in OMS well: (1) neighboring positions should have spatial locality to deal with peak shifts, while (2) distant positions need to have adequate orthogonality since a dramatic peak shift rarely occurs in nature. The peak shift changes the index value corresponding to the intensity in the spectrum vector. With the proposed method, position hypervectors do not change significantly even if peak shifts occur; thus, the resulting representation can be tolerable to them. As depicted in Figure 4.5(b), for $\mathbf{F}_i$ and $\mathbf{F}_j$, the pairwise similarity has a high value when $i \approx j$ and is maximum when $i = j$ (diagonal elements). Note that we scaled down the f to 128 and D to 256 for better visibility.

Reflecting Value Locality

The intensity information of the spectrum vectors is captured. We use the level hypervectors generation method used in [15, 51]. We allocate a single bit to each of the hypervector components, i.e., $\mathbf{L}_i \in \{-1, 1\}^D$. $\mathbf{L}$ that is assigned to each quantization level needs to reflect the closeness of the intensity. The similarity between $\mathbf{L}_i$ and $\mathbf{L}_{i+1}$ should be higher than the similarity

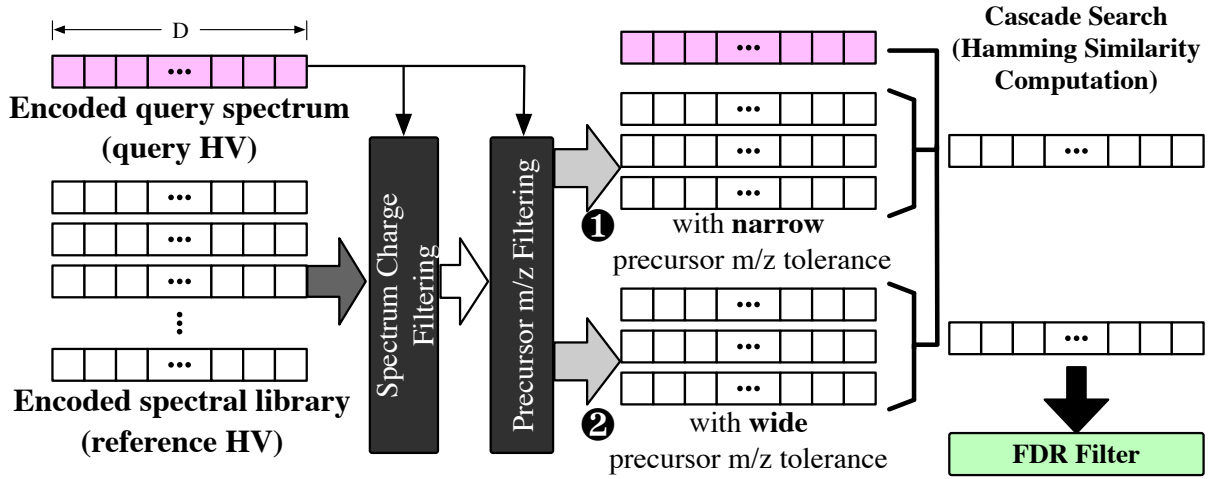


Figure 4.6: Hamming similarity search in HyperOMS.

between $\mathbf{L}_i$ and $\mathbf{L}_{i+100}$. For instance, for the target level p in percentage, $\mathbf{L}_p$, we can represent this by flipping $(D/2) \times (p/100)$ elements of $\mathbf{L}_0$.

4.3.3 Hamming Similarity Search

After the encoding step, HyperOMS finds the matched reference hypervector that is most similar to the query hypervector. It uses *Hamming similarity* (defined by the number of equal components in vector pairs) as a similarity metric. Here, reference spectra that need to be compared primarily need to satisfy spectrum charge and precursor m/z condition per query as discussed in Section 4.2.1. We gather reference spectra that (1) have the same spectrum charge as the query spectra and (2) satisfy the precursor m/z tolerance (precursor m/z difference between query and reference) condition.

OMS assumes that precursor m/z of selected reference spectra and query spectra can have a large difference. A wide precursor m/z tolerance is used to match modified spectra to their unmodified variants. However, we may miss the case of a reference spectrum with a similar precursor m/z that can pass through the FDR filter with high similarity. To avoid such misidentifications, we adopt *cascade search* [137]. A narrow precursor m/z tolerance is used for

standard search and FDR filtration is applied (Figure 4.6-①). In turn, remaining unidentified spectra are processed with a wide precursor m/z tolerance (Figure 4.6-②).

4.4 HyperOMS Hardware Architecture

4.4.1 Motivation

Insights from GPU Implementation

HyperOMS algorithm is a good match for highly parallel computing platforms. We implemented HyperOMS on GPU using GPU-based HDC framework [95] and optimization strategies in [38]. To maximize the computation efficiency, we represent binary hypervector as a 32bit integer array using bit packing, and similarity score computation is done by CUDA intrinsic. Our profiling results from `nvprof` shown in Figure 4.8 suggest that every stage of HyperOMS has low ALU (FP32, INT32) utilization (41.5% on average) compared to memory utilization (80% on average). The kernel of each stage is memory bandwidth bound. The encoding and the Hamming similarity search kernels are bound by the unified L1/texture cache and shared memory bandwidth, respectively. Each kernel implementation leverages the CUDA memory hierarchy to maximize data reuse and minimize access to the global memory for optimized performance.

Why we need DRAM-based PIM accelerator?

Two stages of the HyperOMS algorithm have low computational density: the encoding step requiring 8 operations per byte and the Hamming similarity calculation requiring 16 operations per byte. This implies that these steps are limited by the memory bandwidth. To this end, we leverage DRAM-based PIM hardware to accelerate the HyperOMS algorithm. Among various memory technologies for PIM, DRAM benefits from its mature manufacturing techniques, offering high-density memory cost-effectively and excellent scalability. It offers not only scalable memory

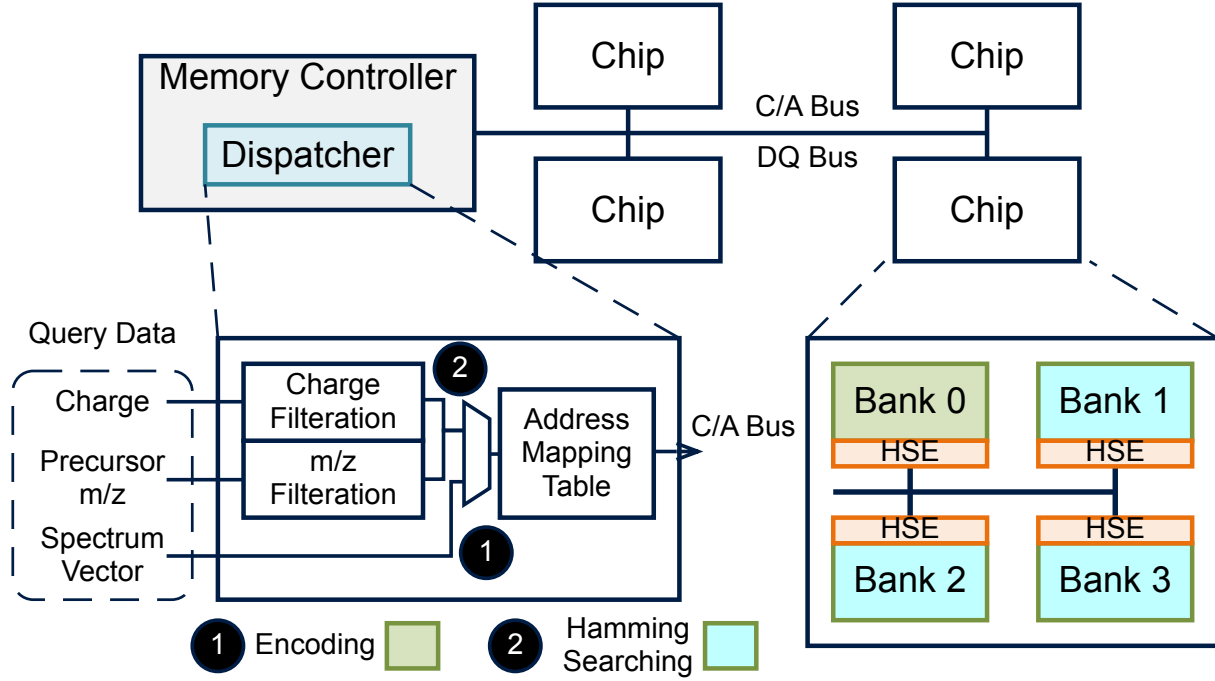


Figure 4.7: Overview of the HyperOMS accelerator architecture. Encoding and Hamming similarity search step is performed in PuM and NMP, respectively.

bandwidth but also provides a large memory capacity to accommodate the massive spectral data.

4.4.2 HyperOMS Accelerator Design

The HyperOMS accelerator uses the PuM technique for encoding and NMP for Hamming similarity search. Existing works [120–122] have shown that combining different PIM techniques, such as PuM and NMP, can optimize the efficiency of the PIM accelerator.

During the encoding stage, it is necessary to store intermediate results before binarization. The PuM technique reduces the necessity of peripheral circuits, including buffers. While the PuM technique requires more time to process the encoding stage compared to NMP, it takes a smaller portion of the total runtime. On the other hand, the search phase takes the majority of the time, especially in large datasets. As such, we accelerate the Hamming similarity search using the NMP technique.

Figure 4.7 illustrates the system diagram of the proposed HyperOMS accelerator that is

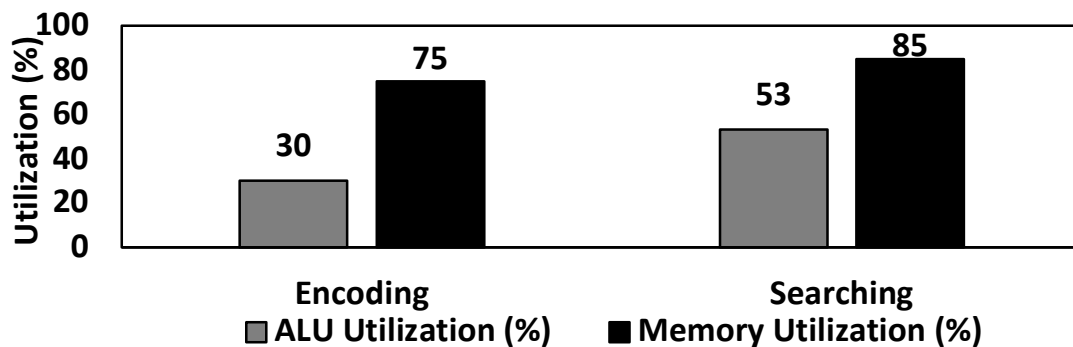


Figure 4.8: Per-stage profiling on NVIDIA GeForce GTX 1080Ti

composed of three main parts: (1) HyperOMS dispatcher, (2) encoding banks (Section 4.4.3), and (3) Hamming search banks (Section 4.4.4). We design the accelerator to be compatible with the DDR4 standard [138]. Each DRAM chip has one encoding bank, and the rest of the banks are working under the Hamming search mode. The encoding bank handles the encoding stage of the HyperOMS algorithm using PuM primitives. Each DRAM bank includes a near-bank Hamming similarity search engine (HSE). HSE can compute similarity values for multiple pairs simultaneously by adopting the hypervector folding strategy (refer to Section 4.4.4), which approximates the Hamming similarity value. The dispatcher is implemented in the memory controller (MC), working as the high-level scheduler and controller for query data fetching from the host, control/address (C/A) command generation, and query allocation to the DRAM banks. The query data (spectrum charge, precursor m/z , and spectrum vector) are fed into the HyperOMS accelerator via the dispatcher.

Dataflow: HyperOMS follows a map-reduce manner to fully utilize the internal data parallelism of DRAM. The encoding and Hamming similarity search phases are performed locally within **the encoding banks** and **Hamming search banks**, respectively. First, during the encoding phase (Figure 4.7-❶), the dispatcher converts the peak intensities and indices into corresponding memory addresses to perform in-memory encoding in the encoding banks. In turn, The encoded query hypervectors in the encoding banks are collected by the dispatcher and then broadcast to Hamming search banks. The dispatcher first performs the spectrum charge and precursor m/z .

filtering (Figure 4.7-②). It generates the destination memory address and distributes the encoded query from the encoding bank to Hamming similarity search banks based on the spectrum charge and precursor m/z information. The address mapping table stores two types of mapping data: (1) the memory address of position and level hypervectors in the encoding bank and (2) the mapping between the reference’s spectrum charges/precursor m/z and their memory addresses. Each Hamming search bank searches and finds the most similar reference hypervector among the stored reference data. The local search within each bank is performed asynchronously. After all Hamming search banks finish their searching process, the dispatcher gathers the results from each bank and further selects the best-matched reference hypervector in the reduction step. Here, spectrum charge and precursor m/z filtering can cause a bank under-utilization, which leads to speed and efficiency degradation. In order to balance the assigned workloads in banks and raise the bank utilization, we optimize the mapping of the reference data and the query dispatching, as described in Section 4.4.5.

Memory Modifications: To enable fast Hamming similarity search and extended bandwidth, HyperOMS accelerator involves three major memory modifications. Firstly, we add isolation transistors between subarrays to support fast inter-subarray row copy, which results in an additional area overhead of only about 0.8% [139]. Secondly, we implement the HSE design, which is connected to the local buffer of the bottom subarray in each bank. Finally, we add a new memory controller command (similar to [140]) to support data broadcast from the encoding banks to Hamming search banks. These modifications effectively balance the additional overhead and computation efficiency.

4.4.3 In-memory Encoding

The encoding process of query data is computed in the assigned encoding banks (see the green-colored bank in Figure 4.7) in conjunction with the PuM primitives [131–133]. We exploit the PuM technique since the encoding stage only needs simple XOR and addition operations that

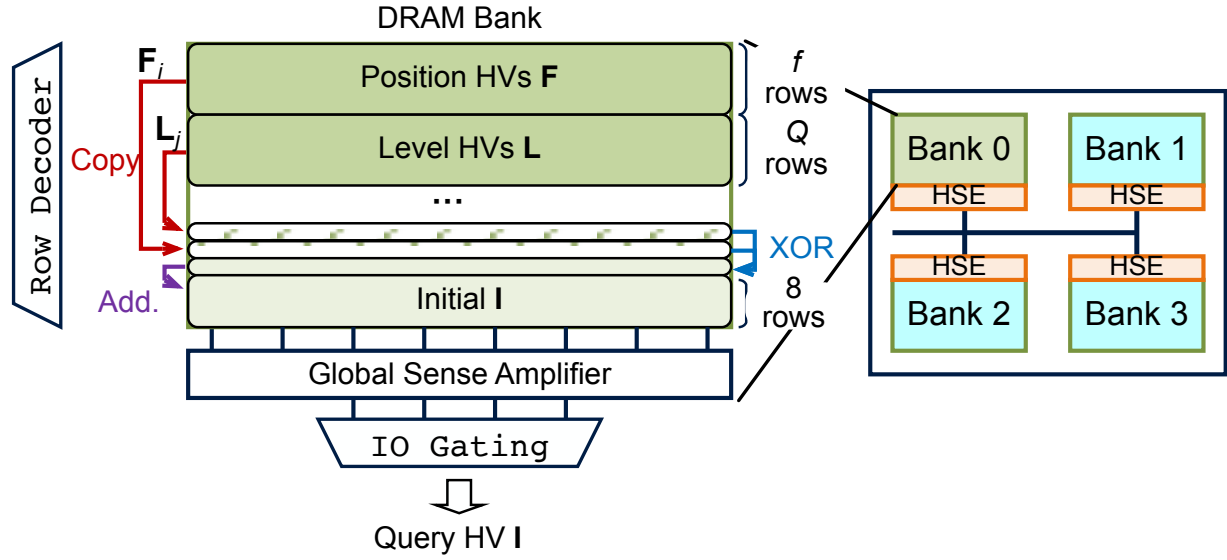


Figure 4.9: In-memory encoding scheme for spectrum vectors.

are well supported by existing PuM primitives [131–133]. Also, it needs to store intermediate results. By utilizing the PuM technique, the accelerator eliminates the need for a buffer to hold these results, which is beneficial for minimizing the density impact on DRAM. The encoding is a memory-intensive process characterized by massive data movement that arose from fetching the D -bit position or level hypervectors over $|\mathbb{P}|$ times each. The in-situ property of PuM avoids data fetching, thereby saving memory bandwidth as well as energy. Although PuM increases the processing time compared to NMP, the encoding process only occupies a small fraction of the overall runtime (see Figure 4.16).

The encoding bank execution includes three steps: (1) receiving the query data, (2) in-memory encoding, and (3) broadcasting encoded data to the query hypervector register within the HSE of other banks. As shown in Figure 4.9, the encoding bank pre-stores the position and level hypervectors in a total of $f + Q$ rows. In the first step, the DRAM bank receives the row addresses of $(i, j) \in \mathbb{P}$ from HyperOMS dispatcher. Then the corresponding position and level hypervectors, $\mathbf{F}_i$ and $\mathbf{L}_j$, are copied to new rows. The copy process can be achieved by the fast row copy in LISA [139]. Second, we compute the XOR for each pair of copied position and level hypervectors

using the majority-based arithmetic in [131, 133]. The third step is to aggregate the XOR results. The basic memory commands to compute in-memory bit-wise XOR and bit-serial addition are two back-to-back activation commands followed by a precharge command (AAP). Specifically, n -bit bit-serial addition needs $4n + 1$ AAP commands (XOR is regarded as 1-bit addition). We initialize the result rows with $-|\mathbb{P}|/2$ in bit-serial and 2's complement format. The mentioned three steps are repeated $|\mathbb{P}|$ times to obtain the final aggregation results. The sign bit of the result rows is selected for the binarization. After the encoding, the encoded query hypervector needs to be broadcast to the query hypervector register within the HSE of other Hamming similarity search banks. The encoded hypervector in a row is burst out from the encoding bank to the HyperOMS dispatcher via the inter-bank dataline. It helps to broadcast results at minimal cost without conflicting with DDR4 standard [138]. Finally, the dispatcher transmits the collected query hypervector to other Hamming search banks in a sequential manner.

4.4.4 Near-memory Hamming Similarity Search

The Hamming similarity search involves calculating the similarity between the query and reference hypervectors, which can be implemented either in-memory or near-memory. If we implement the Hamming similarity computation using the PuM technique, it incurs a long processing latency since 5 AAP commands need to be issued by the DRAM bank to compute the bit-wise XOR between two hypervectors. The DDR4-2400 4Gb $\times$ 8 needs around 390ns to perform the PuM-based XOR operation. On the other hand, the NMP-based design can provide shorter latency for the XOR computation and has higher internal memory bandwidth for pattern-matching workloads [118]. The long latency of PuM-based computation is not favorable for Hamming similarity search as the search step in HyperOMS accounts for the majority of the overall runtime. Hence, we choose to accelerate it using a near-memory HSE module as shown in Figure 4.11.

To exploit the internal bandwidth of DRAM, we locate one HSE next to the local row

buffer of the last and bottom subarray in each DRAM bank. Implementing only one HSE for each bank avoids adding excessive area overhead to the original DRAM. We can use one precharge and one activates command ($< 60\text{ns}$) to latch the data into the local sense amplifier (SA) and row buffer (RB). The HSE directly fetches the D -bit latched data from local RB at one time (D is the size of RB), and the processing latency can be hidden by the memory access time.

Meanwhile, the HSE computes the Hamming similarity of only one pair if it is implemented naively; the parallelism is proportional to the number of HSE. Also, the XOR computation of hypervector pairs can be computationally demanding due to its dimensionality. As such, we devise a strategy called *hypervector folding* to increase the parallelism within HSE and the design of HSE.

Hypervector Folding

The Hamming similarity is evaluated by XOR results between the reference hypervector and query hypervector. The naive way to get the Hamming similarity is to count the 0s of the XOR hypervector. However, the high dimensionality of the XOR hypervector with a few thousand bits makes it slow to count the accurate number of 1s. We propose the **hypervector folding** scheme to reduce the computation of the Hamming similarity and accommodate more parallelism. The parameters for the hypervector folding are static, and they are used for optimizing HSE design.

The basic idea of hypervector folding is to calculate only partial bits of the XOR hypervector (called subsequence) and approximate the actual Hamming similarity. Specifically, instead of calculating the Hamming similarity (θ_D) for the entire hypervector with dimensionality D , we only compute the Hamming similarity (θ_K) of the first K bits. Note that D is a multiple of K to make it evenly divisible and avoid padding. Let us assume that $K = D/2$. Figure 4.10(a) shows the relationship between θ_D and $\theta_{D/2}$, which are values obtained from randomly sampled query and reference pairs. Here, θ_D and $\theta_{D/2}$ are roughly proportional. The trend is similar in the

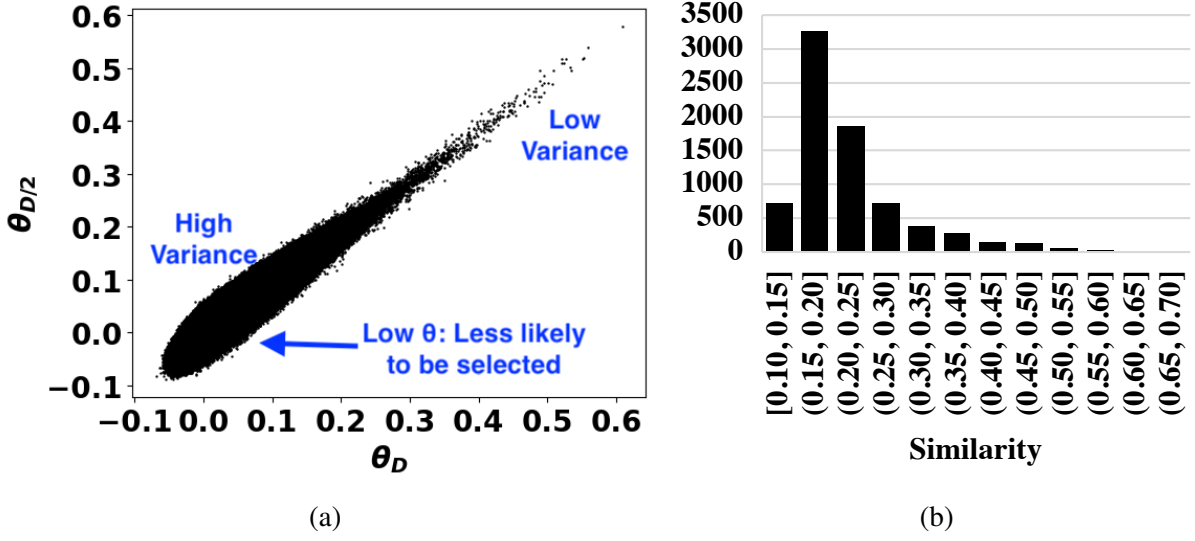


Figure 4.10: Statistics of Hamming similarity values. (a) Relevance between θ_D and $\theta_{D/2}$. (b) Similarity between query and matched reference spectrum (normalized to D).

$K = D/4$ or $D/8$ case. When θ_D is small, it shows high variance, while showing low variance when θ_D is large. Figure 4.10(b) shows a histogram of normalized Hamming similarity values of query and the matched reference pair in the spectrum charge +2 of the small scale dataset used in Section 4.5. It has been noted that for the other datasets as well, the matched reference and query pairs exhibit a normalized Hamming similarity of at least 0.1. Hence, we can approximate θ_D as $\theta_{D/2}$ since we pick a spectrum with a high similarity value.

The goal of hypervector folding is to predict θ_D using θ_K and to handle more pairwise similarity computations in parallel within HSE. The HSE only computes θ_K if it is less than the predefined threshold T . Otherwise, HSE computes θ_D . When θ_K successfully approximates θ_D , the same reference hypervector needs to be selected in either case. We define this success rate as the folding success rate (FSR). It is essential to choose appropriate values for K and T and maximize the FSR. We discuss the hyperparameter search experiment in Section 4.5.6.

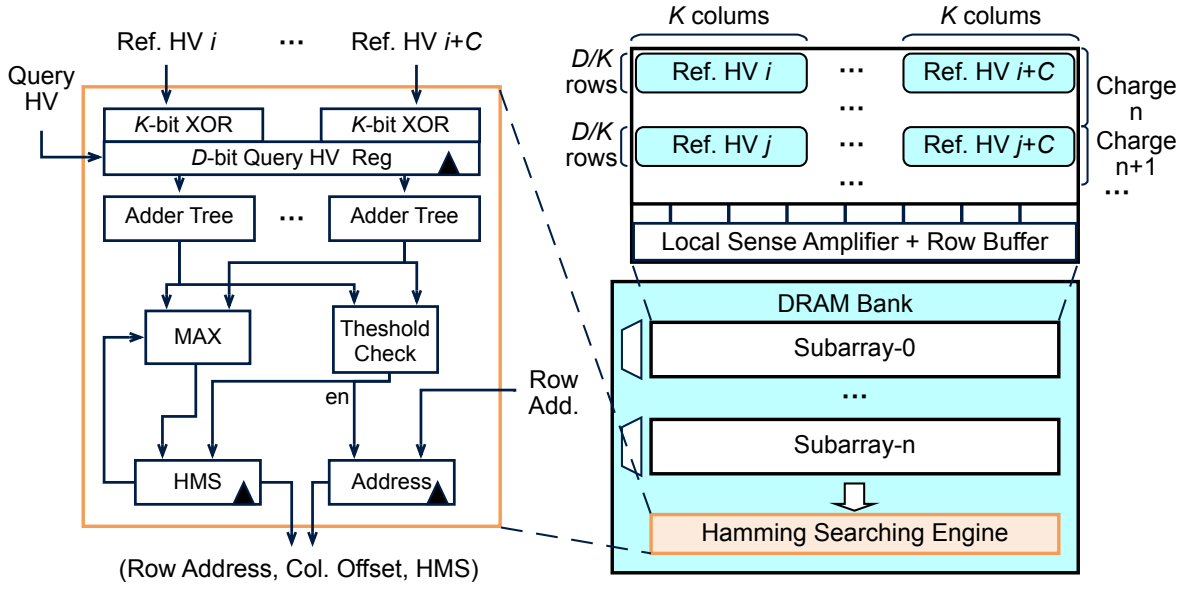


Figure 4.11: Near-memory HSE design.

Hamming Similarity Search Engine

The Hamming similarities between query hypervectors and reference hypervectors are computed in near-bank HSE (see blue-colored banks in Figure 4.7). In the memory subarray, each D -bit reference hypervector is arranged into D/K consecutive rows, with each row occupying K columns as shown in Figure 4.11. This arrangement simplifies reference indexing and allows for continuous memory access. To further streamline the process, we (1) organize the data by grouping references from the same spectrum charge (see Section 4.4.5) and (2) sort spectra based on their precursor m/z values within each spectrum charge. This enables all Hamming search banks to share the same address space of reference hypervectors, with the memory controller only needing to send identical starting and ending row addresses to all banks. Each bank then performs an independent search within the given address range. To achieve this, each HSE within each bank requires its independent memory controller and address generator to execute asynchronously. The additional overhead of the bank controller and address generator is negligible [141].

The HSE involves loading a row of data from a DRAM bank into a local RB with N_{col}

columns, where the RB contains $C (= N_{\text{col}}/K)$ subsequences, each with a length of K bits. HSE accesses the entire row data every t_{RC} ns and generates the Hamming similarity before new data comes in t_{RC} to avoid any stalls between consecutive DRAM row accesses.

The encoded query bits are broadcast to each bank through the DQ bus and cached in the query hypervector register with the same size as N_{col} . The query hypervector is then XORed with the C loaded reference subsequences in parallel using K -bit XOR modules. To reduce latency, the XORed results pass through C adder trees, each consisting of K bit-serial adders. Although many adders are used, the design has low hardware complexity because most adders' bit width is low. The obtained partial Hamming similarities are checked to see if they satisfy the threshold condition in hypervector folding. If the similarity values for subsequences are all less than T , the row for the next pairs of reference hypervectors is loaded. Otherwise, for pairs that do not satisfy the remaining K -bit reference hypervector, it continues to be loaded and compute θ_D . The results are compared with the current best stored in the Hamming similarity (HMS) buffer, and the HMS buffer is updated with the maximum Hamming similarity. Meanwhile, the associated row address and column offset are recorded in the address buffer. In the following, we present a data organization optimization scheme that tackles the hardware under-utilization challenges caused by the reference hypervector filtration.

4.4.5 Data Organization Optimization

Efficient performance in the HyperOMS accelerator heavily relies on the data allocation since two filtration steps, spectrum charge and precursor m/z filter, which may cause workload imbalance and low bank utilization. The dispatcher distributes the encoded query hypervector to banks based on its spectrum charge and precursor m/z . Therefore, we optimize HyperOMS's data organization in two aspects: (1) the reference data mapping and (2) the query dispatching scheme.

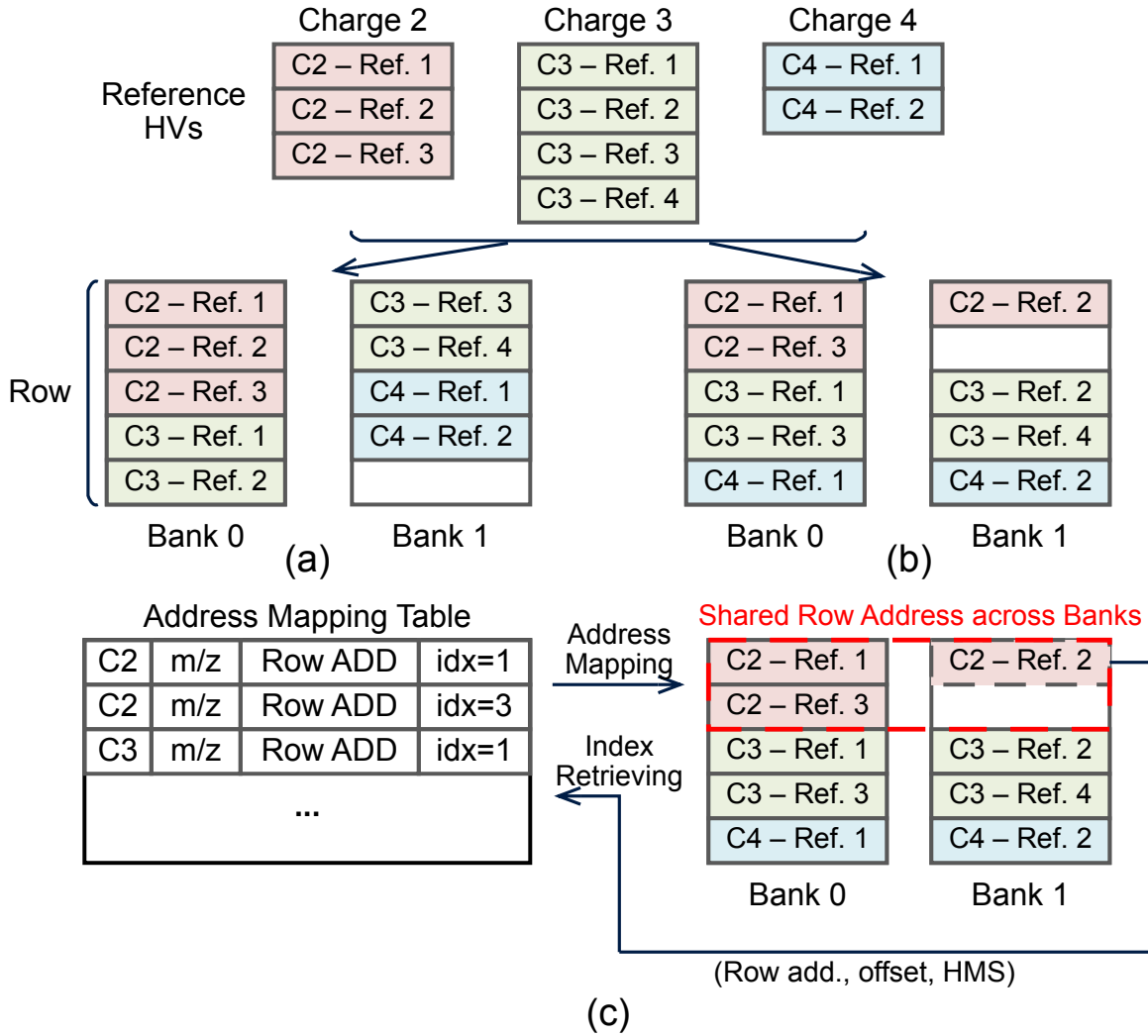


Figure 4.12: (a) Uniform mapping. (b) Proposed reference data mapping scheme. (c) Address mapping process and reference index retrieving process based on (b).

Reference Data Mapping

The naive way to map the reference hypervectors is uniformly distributing the reference data to each bank in a round-robin manner. Figure 4.12(a) gives an example of mapping nine reference hypervectors from spectrum charge +2 (C2) to +4 (C4) to two memory banks. For each spectrum charge, hypervectors are sorted based on their precursor m/z . The uniform mapping scheme allocates hypervectors to each bank based on spectrum charges. Here, each bank is unable to access references hypervectors from all spectrum charges, leading to low bank utilization. For

example, only one bank is activated during search for query with C2 or C4 because only Bank 0 or Bank 1 contains hypervectors of C2 or C4, yielding only 50% bank utilization.

We propose the **reference data mapping scheme** in Figure 4.12(b) to increase bank utilization. The reference hypervectors are assigned across the bank such that each DRAM bank stores hypervectors from all spectrum charges. By distributing the reference hypervectors to different banks, the proposed data mapping scheme enhances the bank utilization for queries from different spectrum charges. In this case, the total searching time is evenly amortized to each memory bank, thus reducing the processing latency. It also effectively reduces the overhead of the query dispatching policy.

Query Dispatching Policy

The dispatcher needs to properly allocate the given query to target banks to maximize bank utilization. Also, it needs to generate memory commands with addresses compatible with current DRAM standards [138] based on the query's spectrum charge and precursor m/z . As such, we design **query dispatching policy** to cooperate with the proposed reference data mapping scheme. As shown in Figure 4.12(c), the addresses and indexes for reference hypervectors are stored in the address mapping table within the dispatcher, which is designed to be compatible with current DRAM standards [138]. The address mapping table needs to store the information (spectrum charge, m/z , row address, and index) for the first memory bank (Bank 0) since the consecutive reference hypervectors across different banks share the same row address space. For example, Ref.1 and Ref.2 of spectrum charge +2 (C2) have identical row addresses. The dispatcher generates the starting and ending row addresses of Hamming search based on the filtering results of the query's spectrum charge and precursor m/z . The addresses with control signals are broadcast to all search banks via the C/A bus. All Hamming search banks perform Hamming similarity searches in parallel.

The Hamming similarity search returns the most similar reference hypervector stored in

each bank. The dispatcher needs to retrieve the index of reference hypervectors. Figure 4.12(c) shows how we retrieve the index. Assuming Bank 1 finds C2 Ref.2 as the best-matched result, Bank 1 returns the tuple (Row address, offset, HMS) that represents the associated row address, column offset, and Hamming similarity. The dispatcher first retrieves the reference index $idx = 1$ in the first bank by accessing the address mapping table. Since the dispatcher knows the bank address, then the index (idx) of C2 Ref.2 is computed using $idx + \text{Bank address} + \text{offset}$. The offset is used to index the two reference hypervectors organized over the same DRAM row in Figure 4.11. Similarly, the reference information from other banks can be inferred based on the returned row address and offset.

The dispatcher has an overhead from the buffer, as it needs to store the address mapping table. Assume that the size of reference data is N , each entry takes $3 + 32 + \lceil \log_2 N \rceil$ bits in the buffer, where 3 is for spectrum charge information, 32 is for m/z in a single FP32 number, and $\log_2 N$ is for index bits. Here, the row address does not need to be explicitly stored since it is identical to the entry address in the buffer. If we have B DRAM banks, the required buffer size is $\frac{N}{B} \times (3 + 32 + \lceil \log_2 N \rceil)$. For the large-scale dataset in Table 4.2 with around 3 million reference spectra, the estimated table size is 163KB for 128 banks. The buffer size decreases linearly as the number of banks increases, which is attractive for large scale processing.

4.5 Evaluation

4.5.1 Methodology

PIM Design. The specifications for DRAM and HyperOMS are summarized in Table 4.1. The large-scale dataset in Table 4.2 contains approximately 3 million reference spectra. We use a total of 128 DRAM banks with the DDR4-2400 standard [138], as the large-scale dataset requires $128 > \frac{3 \times 10^6}{2^{15}}$ to hold the reference data. Among the 128 banks, we assign one bank for encoding while the rest do Hamming similarity search. This is because encoding only consumes less than

Table 4.1: System specifications of HyperOMS accelerator.

DRAM Parameters	
DDR4-2400, 4Gb $\times$ 8, Rows = 2^{15} , Row size = 1KB 4 banks/bank group, 4 bank groups, tCLK=1.2GHz	
DRAM Timing	
nRC=55, nRP=16, nRAS=39, nCCDS=4, nCCDL=6 nWR=18, nWTRS=3, nWTRL=9, nRTP=12, nFAW=26	
HyperOMS	
Total 128 banks (8 chips, each chip with 16 banks)	
NMP Components	
HSE	Area per HSE: 42,680 μ m ² Total area: 5.462mm ² , Power: 5,463mW
Dispatcher	256KB SRAM, Word size = 64b Area: 0.234mm ² , Power: 77.4mW

5% of the total execution time. More DRAM banks can be added to support a larger dataset.

We use the 22nm DRAM process. The HSE components of HyperOMS are implemented using Verilog HDL and synthesized using the Synopsys Design Compiler, using the TSMC 28nm technology node. The clock is set to 1.2GHz, matching the IO clock frequency of DRAM. To provide a fair comparison, we scale the area and power to 22nm to align with the memory technology. Moreover, since DRAM generally uses fewer metal layers compared to the generic ASIC, we evaluate the overhead caused by the process difference between logic and DRAM using the method in [142]. The dispatcher uses a 256KB SRAM to store the address mapping table, and the area and power are estimated using CACTI-3DD [143]. We develop an in-house simulator to emulate the behavior of the HyperOMS accelerator. The latency and energy parameters for DRAM [131, 144] and additional ASIC components are loaded into the developed simulator to calculate runtime and energy consumption.

CPU and GPU Baselines. The evaluation was performed on a system equipped with Intel i7-8700K with 64GB RAM and NVIDIA Geforce GTX 1080Ti. We used the optimized GPU implementation in Section 4.4.1. Since the GPU has limited memory, we split the reference and the query data into batches. We set the batch size to use the maximum amount of VRAM for

Table 4.2: Spectrum preprocessing settings.

Parameter Name	Dataset	
	Small scale	Large scale
Max peaks in spectra	50	
Min / max m/z	101 / 1500	
Bin size	0.05	0.04
Precursor m/z tolerance (narrow)	20ppm	5ppm
Precursor m/z tolerance (wide)	500Da	500Da

GPU-based solutions. We measured the energy consumption of the CPU and GPU using Intel Power Gadget and `nvidia-smi`, respectively.

Workloads. We evaluated HyperOMS on two real-world datasets. The datasets are commonly used for benchmarking OMS tool performance [47, 50] and include decoy spectra with the same ratio as the existing spectral libraries using the shuffle-and-reposition method [145] for FDR filtering, which removes false-positive identifications. Small scale dataset combines yeast spectral library [146] with the human HCD spectral library (total spectra: 1,162,392) as the reference libraries, and the iPRG2012 dataset [147] (total spectra: 15,867) as a query. For a larger-scale evaluation, we used the human spectral library [112] (total spectra: 2,992,672) and a HEK293 (Human Embryonic Kidney 293) dataset [105] (total spectra per query: 46,665 on average), as reference and query spectra, respectively. Note that HEK293 consists of multiple query files (b1906~b1938). We preprocessed query and reference spectra in a similar fashion to existing works [47, 50, 116], using the widely used configurations listed in Table 4.2.

Benchmarks. We compare the search quality of HyperOMS to existing search tools, including (1) SpectraST [116] on CPU and (2) the state-of-the-art OMS tool, ANN-SoLo, running on CPU [47] and GPU [50]. We count the number of identifications to compare the search quality in the same way as the state of the art. All search results are evaluated at a fixed 1% FDR threshold using Pyteomics [148].

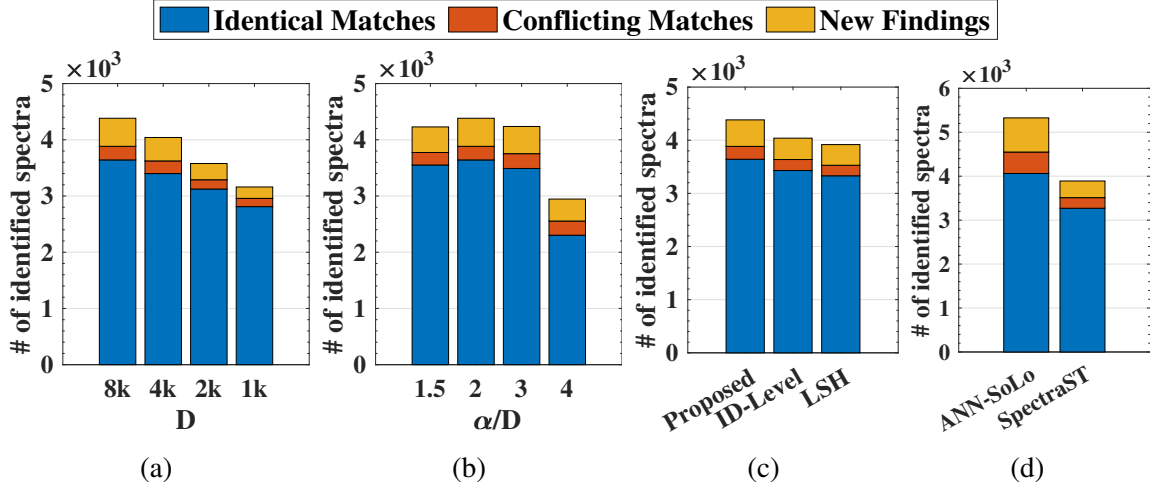


Figure 4.13: Search quality comparison on the small scale dataset. (a) HyperOMS with different hypervector dimensionality. (b) HyperOMS with different number of bit flips. (c) HyperOMS with various encoding strategies. (d) Results of baseline tools.

4.5.2 Impact of Encoding Configuration

Hypervector dimensionality. The hypervector dimensionality (D) plays a critical role in search quality. A low dimensionality limits separability. As demonstrated in Figure 4.13(a), the higher D leads to a higher number of identifications. However, the excessive D leads to an increase in computation and capacity demand. We set the D to 8192 (8k) which is equal to the number of columns in the DRAM subarray, which satisfies both the hardware efficiency and the search quality.

Flipped bits. The number of flipped bits (α) controls the balance between orthogonality and correlation between bins. A high α increases the orthogonality of each position, and a low α helps a more number of adjacent bins to have a correlation (spatial locality). We measured the number of identifications according to the ratio of flips to D , i.e., α/D . As shown in Figure 4.13(b), an adequate α/D leads to high search quality. But if we flip a small number of bits, HyperOMS cannot clearly differentiate the peak position. Also, since the peak shifts due to PTMs are not significant, spatial locality for a limited range is required. In the rest of our evaluation, we use $\alpha = D/2$, which shows the highest quality.

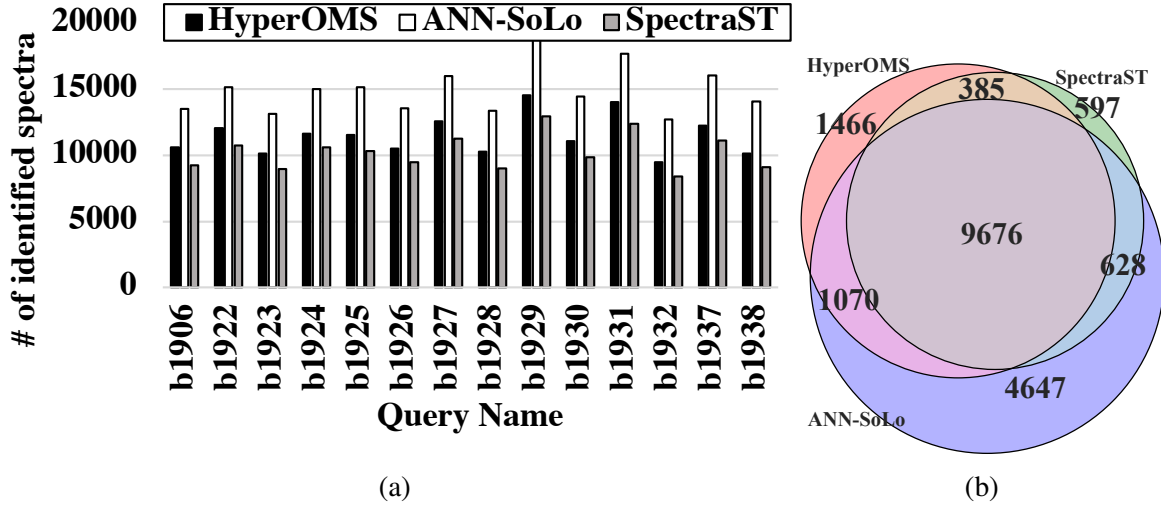


Figure 4.14: Search result analysis from HyperOMS and baseline tools on the large scale dataset. (a) Total number of identifications. (b) Venn diagram of b1927 query search result.

Encoding strategy. We compare the search quality of HyperOMS with the different binary encoding strategies. As discussed in Section 4.3.2, LSH [28, 136], ID-Level HDC encoding [15, 51] can be used alternatively to encode raw data to HD binary vectors. Figure 4.13(c) compares the search quality of HyperOMS with the (1) proposed encoding method, (2) ID-Level HDC encoding that can capture the position of feature and its value, and (3) random projection-based LSH approach. Our encoding offers the best search quality compared to baselines.

Quantization level. High quantization levels may not be flexible to the peak intensity changes due to noise and PTMs. Low Q leads to low resolution in intensity capturing of the encoder. The quantization level Q did not significantly affect the search quality unless it falls in the range of $[8, 32]$. Therefore, we use $Q = 16$.

4.5.3 Search Quality

We search the iPRG2012 dataset against the yeast spectral library. As no ground truth information is available when analyzing complex biological data, instead, we compare our search quality with a list of consensus identifications produced by multiple search tools during

the iPRG2012 study [147]. Figure 4.13(d) shows the search result of baseline tools. Among 7841 identifications in the iPRG2012 consensus result, HyperOMS is able to correctly identify 4141 spectra (Figure 4.13(b)). SpectraST and ANN-SoLo manages to identify 3891 and 5327 identifications, respectively.

We compare the performance of HyperOMS with the results from existing tools, including SpectraST and ANN-SoLo using the large-scale dataset. We use similar configurations for all tools, listed in Table 4.2. Figure 4.14 shows the number of identifications from the different search tools. HyperOMS offers a higher search quality than SpectraST, i.e., more identified spectra. ANN-SoLo managed to identify more spectra than our HyperOMS. Nevertheless, as described in Figure 4.14(b), HyperOMS can identify spectra that other tools can find (overlapped area). HyperOMS represents spectra in a way robust to PTMs, which is approximated form of the original data. Compared to ANN-SoLo, which uses floating-point representation, HyperOMS shows slightly degraded search quality.

The identification rate of HyperOMS can be improved by increasing the hypervector capacity. This can be done by (1) increasing the hypervector dimensionality D or (2) increasing the number of bits of each component in the hypervector. For example, increasing D from 8k (8192) to 16k (16384) can yield up to 10% more identifications. However, it raises the hardware cost, computational complexity, and energy consumption of the accelerator. Since our main goal is to maximize the speed and energy efficiency of the OMS while achieving reasonable quality in a biological sense, we use a $D = 8192$.

A ramification of lower search quality could be missing potentially relevant biomarker proteins in the context of a healthy versus diseased study, or missing data similarly impacting other downstream biological interpretations. Nevertheless, the HyperOMS identification rate is within the range of the state-of-the-art in MS identification. For example, we can typically expect an identification rate of 33–66% currently for human samples that we have used, and HyperOMS satisfies the expected range criterion. One advantage of HyperOMS is that there

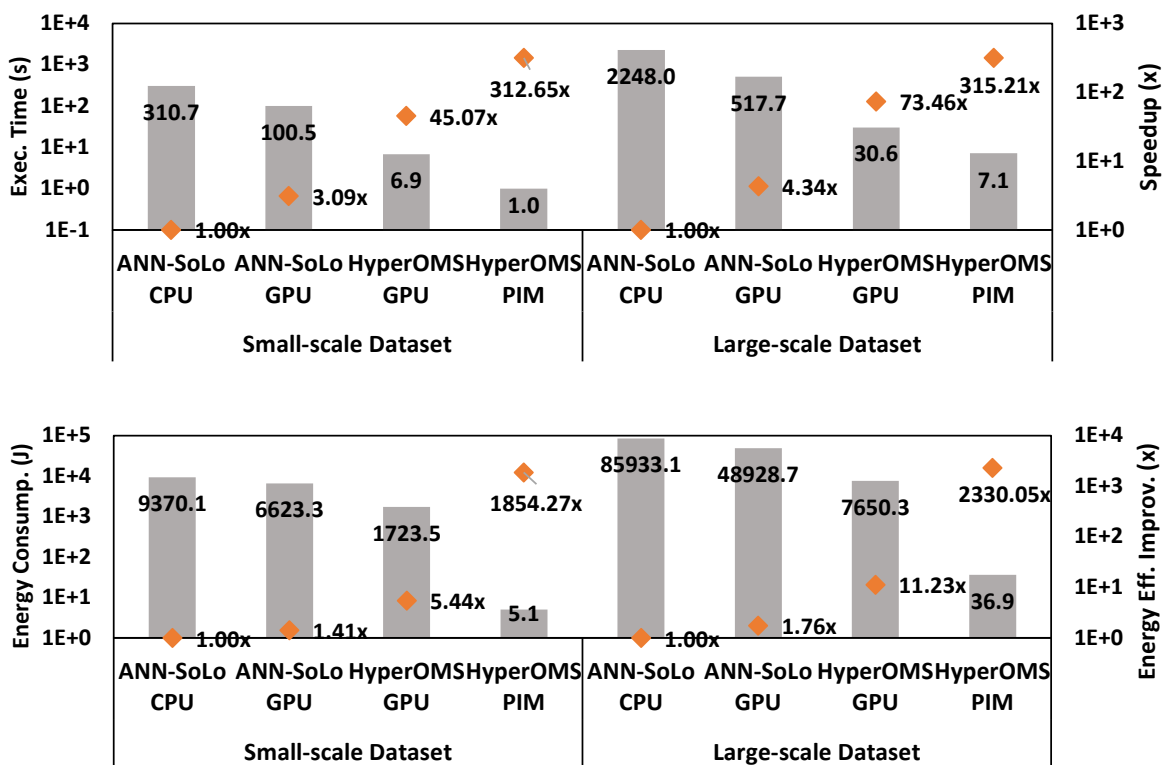


Figure 4.15: Performance and energy efficiency comparison.

is a search quality–efficiency trade-off that can be tuned using the hyperparameters. The user can decide between different search engines based on their requirements; HyperOMS runs much faster with superior energy efficiency compared to competing OMS tools (Section 4.5.4). It could be used to efficiently process extremely large proteomics datasets consisting of tens of thousands of query files, which are being generated increasingly often recently.

4.5.4 HyperOMS Performance Comparison on GPU

We compare the execution time and the energy consumption of HyperOMS on GPU to the state-of-the-art OMS tool ANN-SoLo, which offers a faster search speed with the GPU acceleration than other baseline [50]. Note that we measured the second run of the ANN-SoLo since the reference data is likely to be pre-encoded in reality. ANN-SoLo saves the pre-indexed information of the reference library on the first run and reuses it in the subsequent run. For the

large-scale dataset, we averaged the measurements from multiple queries (b1906 $\sim$ b1938).

Figure 4.15 compares the end-to-end runtime. ANN-SoLo builds the index on the CPU while the encoding of HyperOMS is done on the GPU. The HyperOMS encoding is parallelized over hypervector dimensions and datapoints. The encoding stage of HyperOMS, which corresponds to the index build of ANN-SoLo, is up to $8.6\times$ faster than ANN-SoLo. HyperOMS uses HD binary vector and easily parallelizable Hamming similarity computation, while ANN-SoLo uses FP32 vector. The search process of HyperOMS on GPU achieves, on average, $82\times$ speedup over ANN-SoLo on CPU and $11.2\times$ speedup ANN-SoLo on GPU. Ultimately, HyperOMS GPU offers significant speedup in all stages of the OMS. For the end-to-end execution, HyperOMS on GPU gains an average speedup of $15.7\times$ over the state-of-the-art OMS tool running on the same GPU.

HyperOMS running on the GPU requires more power than the ANN-SoLo, as it has high parallelism. However, the increased power consumption is compensated by reduced execution time, improving energy efficiency. Overall, HyperOMS results in $7.8\times$ and $5\times$ energy efficiency improvement over CPU and GPU on average, respectively (see Figure 4.15).

4.5.5 Performance Improvement of HyperOMS Accelerator

We evaluated the speed and efficiency of the HyperOMS accelerator. We set the hypervector folding parameter to $K = D/2$ and $T = 0.05$ based on the experiment in Section 4.5.6. The speed and energy consumption are shown in Figure 4.15. Compared to HyperOMS on GPU, our DRAM-based accelerator offers $6.9\times$ and $4.3\times$ speedup on small-scale and large-scale datasets, respectively. Our accelerator also significantly improves the energy efficiency by up to $341\times$, which is mainly powered by the energy-efficient datapath. During the search process, the query hypervector stays stationary in the HSE’s register while the reference hypervectors only need to be fetched from DRAM once. The in-situ similarity search in DRAM banks and HSE dramatically reduces the required time and energy for data movement. The hypervector folding scheme also

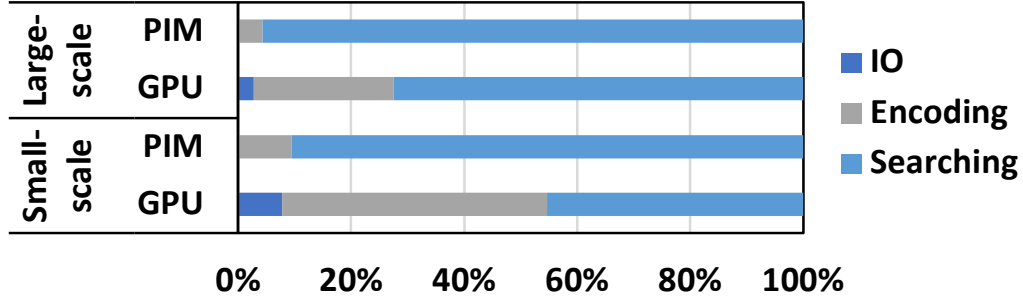


Figure 4.16: HyperOMS runtime breakdown.

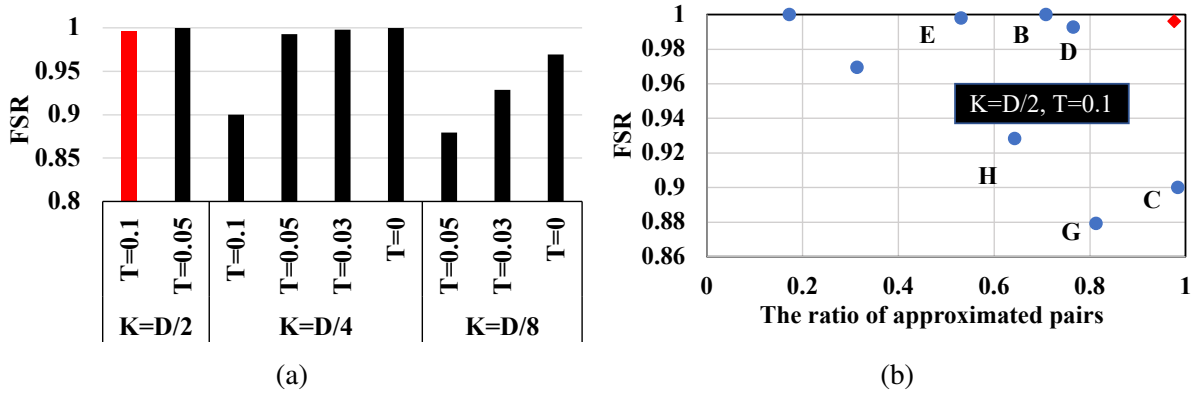


Figure 4.17: Hypervector folding parameter search. (a) Hypervector folding success rate (FSR) according to K, T . (b) Parameter selection.

saves about 50% energy spent on Hamming similarity computation.

Unlike conventional HDC-based algorithms, we tackle the large-scale pattern-matching problem. Hence, the search stage rather than the encoding stage occupies a larger percentage of the total runtime. As shown in Figure 4.16, our PIM design improves both stages. The accelerator offers $21\times$ speedup on average for the encoding stage compared to GPU. The Hamming similarity search is $2.1\times$ faster than GPU because we extended the achievable bandwidth for HSE.

4.5.6 Hypervector Folding Parameter Search

Figure 4.17(a) compares the FSR for different combinations of K and T . Increasing T makes the approximation in hypervector folding less accurate, resulting in a low FSR. Additionally,

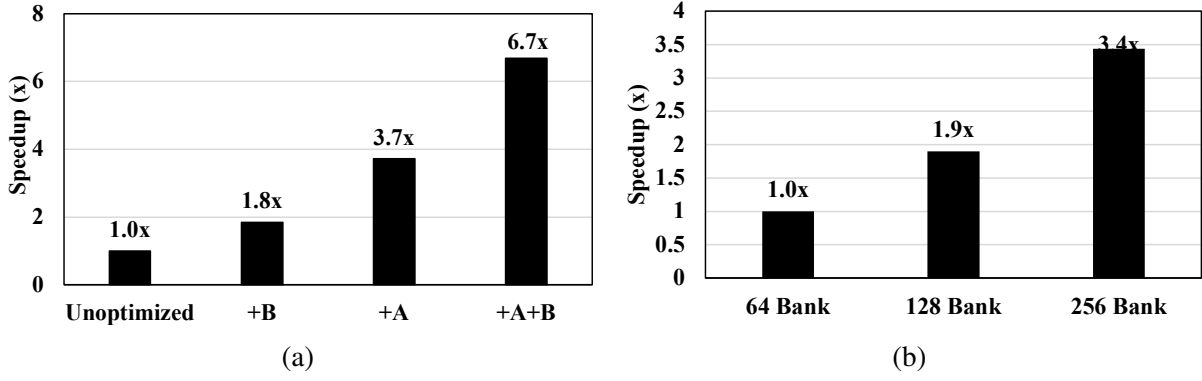


Figure 4.18: Speed comparison of HyperOMS with different configurations. (a) The cumulative effects of optimization strategies (A: Data organization optimization, B: Hypervector folding). (b) Impact of the number of banks.

as K decreases, we need to set a lower T to maintain a high FSR. However, for low T , the chances of computing θ_D increase. As shown in Figure 4.17(b), the best performance is achieved by setting K to $D/2$ and T to 0.05, with 97.6% of query-reference pairs approximated with 99.6% of FSR. When two subsequences are processed in parallel ($K = D/2$), the probability of successful folding is $0.976^2 = 0.952$, as both pairs need to satisfy the threshold check in HSE. Setting $K = D/4$ can improve throughput since more queries can be processed in parallel. However, the accelerator will be underutilized, with a probability of $1 - (0.765)^4 = 0.658$. Hence, we choose $K = D/2$ and $T = 0.05$.

4.5.7 Effectiveness of Optimization Scheme

We show the effectiveness of our optimization strategies on our 128-bank accelerator design. Figure 4.18(a) shows speedup according to the optimization schemes. The unoptimized design denotes the HyperOMS accelerator using the uniform reference data mapping in Figure 4.12(a) and without the hypervector folding.

Hypervector folding. We first examine the performance gain by applying the hypervector folding scheme. For the $K = D/2$ case, 48.8% of Hamming similarity computation is saved, offering nearly $2\times$ speedup as the HSE can accommodate more parallelism. As shown in

Figure 4.17, decreasing K can lower the probability of passing the threshold check module. We compare the execution time and the energy consumption of the HSE with $K = D/2$ and $K = D/4$. $K = D/4$ allows the HSE to process four pairs in parallel. All pairs need to pass the threshold check to maintain parallelism. In contrast, when $K = D/2$, the probability of all pairs passing the threshold check module is higher than the $K = D/4$ case (see Section 4.5.6), yielding 26% shorter execution time and 25% higher energy efficiency.

Data organization optimization. Secondly, we observe the effectiveness of the proposed data organization optimization. The dispatcher efficiently distributes the query to memory banks and helps to maintain nearly full hardware utilization. It leads to $3.7\times$ speedup on 128-bank configuration. After applying all optimization strategies, the cumulative speedup over the unoptimized design is $6.7\times$.

4.5.8 Scalability and Overhead Analysis

Scalability. Our DRAM-based HyperOMS accelerator provides scalable memory capacity and bandwidth to effectively handle the increasing sizes of spectral data. As shown in Figure 4.18(b), the HyperOMS accelerator runs faster when the number of banks increases. For example, the accelerator with the 128 bank configuration is $1.9\times$ faster than the 64 bank case, almost achieving a linear speedup. Since the dispatcher buffer size is mainly determined by the reference data size and inversely proportional to the number of banks as analyzed in Section 4.4.5, when we raise the number of banks while keeping data size unchanged, the required buffer size decreases, which is beneficial for OMS with a large-sized database. Considering that the HyperOMS is bounded by memory bandwidth and needs a large memory capacity to cope with expanding the database, the proposed accelerator is a promising solution.

Area overhead. We use the $4\text{Gb} \times 8$ DDR4 DRAM chip in [149] as the baseline to study the area overhead of HyperOMS’s peripheral circuits. The DRAM chip size [149] is scaled to 22nm. The 5.462mm^2 HSEs contribute to an additional 1.67% overhead of the overall chip size.

The low area overhead results from the low-complexity bit-serial adder trees. The other area overhead is from the isolation transistors to realize fast row copy across subarrays. According to [139], the additional area is less than 1%.

Energy overhead. The energy overhead of HyperOMS is evaluated by comparing the energy consumed by each HSE with DRAM’s row activation energy. Each HSE only incurs additional 3.62% energy compared to the row activation energy; each HSE only needs 16 cycles to compute the Hamming similarity and corresponding the threshold check, implying that the HSE is idle over 60% of the time. Thus, the infrequent circuit switching leads to a low energy overhead.

4.6 Conclusion

In this chapter, we proposed HyperOMS, which accelerates OMS in MS-based proteomics by leveraging HDC. HyperOMS algorithm encodes spectra into binary hypervectors, considering spatial and value locality of peaks in spectra. It maximizes the computation efficiency by replacing floating-point operations in OMS with Boolean operations. We further accelerate HyperOMS on a DRAM-based PIM accelerator. The proposed accelerator exploits PuM and NMP technologies to deal with latency-area overhead trade-offs based on each stage’s characteristics. In addition, we optimize the data organization and propose hypervector folding strategy that can increase the hardware utilization and accommodate more parallelism in the proposed accelerator. Our evaluation results show that HyperOMS offers comparable search quality to existing OMS tools. HyperOMS PIM accelerator provides up to $6.9\times$ speedup and $341\times$ better energy efficiency over HyperOMS running on the GPU. Compared to the state-of-the-art GPU-based OMS tool, the proposed accelerator is up to $100\times$ faster and $1,337\times$ more energy efficient.

This chapter contains material from “Massively Parallel Open Modification Spectral Library Searching with Hyperdimensional Computing”, by Jaeyoung Kang, Weihong Xu, Wout

Bittremieux, and Tajana Rosing, which appears in International Conference on Parallel Architectures and Compilation Techniques (PACT), 2022. The dissertation author was the primary investigator and author of this paper.

This chapter contains material from “DRAM-based Acceleration of Open Modification Search for Mass Spectrometry-based Proteomics”, by Jaeyoung Kang, Weihong Xu, Wout Bittremieux, Niema Moshiri, and Tajana Rosing, which is under submission. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Summary and Future Work

The deluge of data and the advanced big data processing algorithms, such as deep learning, have facilitated the extraction of meaningful insights at the cost of significantly increased energy consumption. To alleviate this pressure, the focus of this dissertation is on studying a lightweight, energy-efficient, and hardware-friendly alternative computing paradigm, namely brain-inspired hyperdimensional computing. In particular, we address three challenges in HDC: (1) hardware availability, (2) limited support on structured data, and (3) the eligibility of HDC in large-scale scenarios. We first build OpenHD, an user-friendly, versatile, and high-performing HDC software infrastructure with GPU acceleration (Chapter 2) to ease the development of HDC algorithms. Next, we design a graph-based ML algorithm with HDC using OpenHD and enhance its speed and energy efficiency with FeFET (Chapter 3). Finally, we show the capabilities of HDC on large-scale problems; we design an HDC-based OMS algorithm and DRAM-based accelerator for mass spectrometry-based proteomics (Chapter 4). The following sections summarize the contributions of this dissertation and delineate future directions.

5.1 Dissertation Summary

Chapter 2 presented a GPU-based HDC acceleration framework called OpenHD. The absence of a software infrastructure makes it challenging to develop HDC applications from scratch. The proposed framework maps HDC applications to GPU and automatically performs GPU-specific optimizations for maximized efficiency and performance. Users can implement an HDC application with NumPy-style Python syntax and accelerate it with the GPU through the JIT compilation technique. On the backend, it performs optimizations that address the memory access challenges of HDC on GPU: efficiently leveraging GPU memory hierarchies and reusing data. We also introduced a novel parallel training technique, PARTRAIN, to improve hardware utilization and maximize the advantages of data-parallel processing. Our experimental results on the low-powered embedded GPU, NVIDIA Jetson TX2 [35], show that OpenHD has a $9\times$ smaller model and is $11.7\times$ faster than the DNN-based solution while offering comparable accuracy. For clustering, OpenHD offers comparable accuracy to the K-means algorithm with $53\times$ faster execution time. The proposed parallel training method reduces the required epochs to obtain comparable accuracy by $4\times$. Our acceleration framework achieves $4.5\times$ and $146\times$ speedup on average for HDC classification and clustering, respectively, over the state-of-the-art GPU-based HDC implementations [38].

Using the OpenHD framework, we designed and tested an HDC-powered graph ML solution, RelHD in Chapter 3. It reflects the relationship between nodes and node features in a single hypervector. Our evaluation results show that RelHD achieves up to $32\times$ faster speed compared to GNN-based algorithms on GPU while maintaining comparable accuracy. We accelerate RelHD using a scalable FeFET-based PIM with optimized data allocation and scheduling. It achieves a significant improvement in speed and energy efficiency, with an average enhancement of $33\times$ and $59,287\times$, respectively, compared to OpenHD-based RelHD. Furthermore, RelHD offers $10\times$ speedup and $986\times$ energy efficiency improvements over the

state-of-the-art crossbar memory-based GNN accelerator [46].

We demonstrated the capability of HDC in large-scale scenarios by proposing an HDC-based approach for the large-scale proteomics problem in Chapter 4. We introduced HyperOMS, a massively parallel open modification spectral library search algorithm for OMS of MS-based proteomics. HyperOMS maps spectra to HD space while reflecting the spatial and value locality between peaks. It simplifies the execution pipeline in existing OMS solutions and replaces the floating-point operations with boolean operations making the algorithm hardware-friendly. We further accelerate HyperOMS using a DRAM-based accelerator with various optimization strategies to maximize parallelism. Experimental results show that HyperOMS provides comparable search quality to existing OMS tools [47, 50, 116]. Running HyperOMS on GPU results in a $17\times$ speedup and a $6.4\times$ improvement in energy efficiency compared to the current state-of-the-art GPU-based OMS solution [50]. The proposed accelerator further accelerates HyperOMS on GPU, resulting in up to $6.9\times$ speedup and $341\times$ energy efficiency improvement.

5.2 Future Directions

HDC-based algorithms can run efficiently on parallel hardware platforms such as ASIC and PIM. However, in order to achieve optimal levels of parallelism and performance, it is essential to hand-optimize the kernel based on various factors, such as the characteristics of the data, changes in hyperparameters, and the HDC model architecture. This requires the user to be aware of the domain knowledge of the target hardware. We have proposed OpenHD, which simplifies the development and deployment of GPU-accelerated HDC applications by providing a Python-style software interface that does not require specialized expertise in GPU. OpenHD can be expanded to enable users to develop HDC applications in a hardware-agnostic way. Using a multi-level intermediate representation (MLIR) [150] or heterogeneous parallel virtual machine (HPVM) [151, 152], a compiler backend can automatically parallelize HDC

operations, minimize data movements, and efficiently utilize on-chip memory for large-sized hypervectors. In addition, it can be extended to support heterogeneous computing. As a result, users can reuse the same OpenHD-based code and effortlessly map HDC applications to different hardware systems, including an HDC ASIC/PIM accelerator.

Another potential future direction involves developing a large-scale HDC-based system for MS data analysis. As the amount of MS data has grown exponentially, it is essential to reduce data movement and process data in parallel in order to enhance system energy efficiency [153]. This can be realized on a large-scale heterogeneous system with compute-capable memory and storage. Technologies such as CXL (Compute Express Link), which enable memory-centric interconnects, could serve as foundational components for proof-of-concept. The MS data analysis pipeline includes data preprocessing, clustering, and spectral library searching. Recent research [153] has demonstrated that near-storage MS data preprocessing can improve both speed and energy efficiency. The work in [154] enhanced the speed and efficiency of MS clustering on GPUs. In Chapter 4, we presented an HDC-based spectral library search solution with OMS capability and accelerated it on a DRAM-based PIM accelerator. By developing a near-data accelerator for [154], the entire MS analysis pipeline can be done in the unit of hypervectors. Ultimately, this would reduce development time and enhance the energy efficiency of the drug discovery process significantly.

Bibliography

- [1] N. P. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, *et al.*, “Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings,” *arXiv preprint arXiv:2304.01433*, 2023.
- [2] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, “A domain-specific supercomputer for training deep neural networks,” *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.
- [3] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-I. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” *SIGARCH Comput. Archit. News*, vol. 45, p. 1–12, jun 2017.
- [4] C.-J. Wu, R. Raghavendra, U. Gupta, B. Acun, N. Ardalani, K. Maeng, G. Chang, F. Aga, J. Huang, C. Bai, *et al.*, “Sustainable ai: Environmental implications, challenges and opportunities,” *Proceedings of Machine Learning and Systems*, vol. 4, pp. 795–813, 2022.
- [5] D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean, “Carbon emissions and large neural network training,” *arXiv preprint arXiv:2104.10350*, 2021.
- [6] “Semiconductor Research Corporation Decadal Plan.” <https://www.src.org/about/decadal-plan/>, 2021.
- [7] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey, “Recalibrating global data center energy-use estimates,” *Science*, vol. 367, no. 6481, pp. 984–986, 2020.

- [8] P. Kanerva, *Sparse distributed memory*. MIT press, 1988.
- [9] S. Datta, R. Antonio, *et al.*, “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.
- [10] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, “Exploring hyperdimensional associative memory,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2017.
- [11] Y. Kim, M. Imani, N. Moshiri, and T. Rosing, “Geniehd: Efficient dna pattern matching accelerator using hyperdimensional computing,” in *Proceedings of the 23rd Conference on Design, Automation and Test in Europe, DATE '20*, (San Jose, CA, USA), pp. 115–120, EDA Consortium, 2020.
- [12] Z. Zou, H. Chen, P. Poduval, Y. Kim, M. Imani, E. Sadredini, R. Cammarota, and M. Imani, “Biohd: an efficient genome sequence search platform using hyperdimensional memorization,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 656–669, 2022.
- [13] A. Rahimi, P. Kanerva, and J. M. Rabaey, “A robust and energy-efficient classifier using brain-inspired hyperdimensional computing,” in *International Symposium on Low Power Electronics and Design*, pp. 64–69, Association for Computing Machinery, 2016.
- [14] A. Mitrokhin, P. Sutor, C. Fermüller, and Y. Aloimonos, “Learning sensorimotor control with neuromorphic sensors: Toward hyperdimensional active perception,” *Science Robotics*, vol. 4, May 2019.
- [15] M. Imani, D. Kong, A. Rahimi, and T. Rosing, “Voicehd: Hyperdimensional computing for efficient speech recognition,” in *International Conference on Rebooting Computing (ICRC)*, pp. 1–8, IEEE, 2017.
- [16] O. Räsänen and S. Kakouros, “Modeling dependencies in multiple parallel data streams with hyperdimensional computing,” *IEEE Signal Processing Letters*, vol. 21, no. 7, pp. 899–903, 2014.
- [17] Y. Guo, M. Imani, J. Kang, S. Salamat, J. Morris, B. Aksanli, Y. Kim, and T. Rosing, “Hyperrec: Efficient recommender systems with hyperdimensional computing,” in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, p. 384–389, ACM, 2021.
- [18] F. Asgarinejad, A. Thomas, and T. Rosing, “Detection of epileptic seizures from surface eeg using hyperdimensional computing,” in *2020 42nd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, pp. 536–540, IEEE, 2020.

- [19] Y. Wu, G. Wayne, A. Graves, and T. Lillicrap, “The kanerva machine: A generative distributed memory,” in *International Conference on Learning Representations (ICLR)*, 2018.
- [20] 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.
- [21] S. Ahmad and J. Hawkins, “Properties of sparse distributed representations and their application to hierarchical temporal memory,” *arXiv preprint arXiv:1503.07469*, 2015.
- [22] 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.
- [23] B. Khaleghi, H. Xu, J. Morris, and T. Š. Rosing, “tiny-hd: Ultra-efficient hyperdimensional computing engine for iot applications,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 408–413, IEEE, 2021.
- [24] B. Khaleghi, J. Kang, H. Xu, J. Morris, and T. Rosing, “Generic: highly efficient learning engine on edge using hyperdimensional computing,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 1117–1122, 2022.
- [25] 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, 2018.
- [26] 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. 16.1.1–16.1.4, 2016.
- [27] 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.
- [28] 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 *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, Oct. 2020.
- [29] J. Morris, K. Ergun, B. Khaleghi, M. Imani, B. Aksanli, and T. Rosing, “Hydrea: Towards more robust and efficient machine learning systems with hyperdimensional computing,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 723–728, IEEE, 2021.

- [30] J. Kim, H. Lee, M. Imani, and Y. Kim, “Efficient brain-inspired hyperdimensional learning with spatiotemporal structured data,” in *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 1–8, 2021.
- [31] D. Kleyko, A. Rahimi, D. A. Rachkovskij, E. Osipov, and J. M. Rabaey, “Classification and recall with binary hyperdimensional computing: Tradeoffs in choice of density and mapping characteristics,” *IEEE transactions on neural networks and learning systems*, vol. 29, no. 12, pp. 5880–5898, 2018.
- [32] J. Kang, M. Zhou, A. Bhansali, W. Xu, A. Thomas, and T. Rosing, “Relhd: A graph-based learning on feft with hyperdimensional computing,” in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pp. 553–560, IEEE, 2022.
- [33] P. Neubert and S. Schubert, “Hyperdimensional computing as a framework for systematic aggregation of image descriptors,” in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, (Los Alamitos, CA, USA), pp. 16933–16942, IEEE Computer Society, jun 2021.
- [34] “NVIDIA Jetson Nano.” <https://developer.nvidia.com/embedded/jetson-nano>.
- [35] “NVIDIA Jetson TX2.” <https://developer.nvidia.com/embedded/jetson-tx2>.
- [36] “NVIDIA Jetson Xavier NX.” <https://developer.nvidia.com/embedded/jetson-xavier-nx>.
- [37] “NVIDIA Jetson Orin.” <https://developer.nvidia.com/embedded/jetson-orin>.
- [38] J. Kang, B. Khaleghi, Y. Kim, and T. Rosing, “Xcelhd: An efficient gpu-powered hyperdimensional computing with parallelized training,” in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 220–225, 2022.
- [39] L. Wu, P. Sun, R. Hong, Y. Fu, X. Wang, and M. Wang, “Socialgcn: An efficient graph convolutional network based model for social recommendation,” *arXiv preprint arXiv:1811.02815*, 2018.
- [40] Q. Tan, N. Liu, and X. Hu, “Deep representation learning for social network analysis,” *Frontiers in big Data*, vol. 2, p. 2, 2019.
- [41] K. Jha, S. Saha, and H. Singh, “Prediction of protein–protein interaction using graph neural networks,” *Scientific Reports*, vol. 12, no. 1, p. 8360, 2022.
- [42] F. Yang, K. Fan, D. Song, and H. Lin, “Graph-based prediction of protein-protein interactions with attributed signed graph embedding,” *BMC bioinformatics*, vol. 21, no. 1, pp. 1–16, 2020.

- [43] M. Ghorbani, A. Kazi, M. S. Baghshah, H. R. Rabiee, and N. Navab, “Ra-gcn: Graph convolutional network for disease prediction problems with imbalanced data,” *Medical Image Analysis*, vol. 75, p. 102272, 2022.
- [44] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in *International Conference on Learning Representations (ICLR)*, 2017.
- [45] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph attention networks,” in *International Conference on Learning Representations (ICLR)*, 2018.
- [46] N. Challapalle, K. Swaminathan, N. Chandramoorthy, and V. Narayanan, “Crossbar based processing in memory accelerator architecture for graph convolutional networks,” in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2021.
- [47] W. Bittremieux, P. Meysman, W. S. Noble, and K. Laukens, “Fast open modification spectral library searching through approximate nearest neighbor indexing,” *Journal of Proteome Research*, vol. 17, pp. 3463–3474, Sept. 2018.
- [48] F. Bugyi, D. Szabó, G. Szabo, A. Revesz, V. F. Pape, E. Soltész-Katona, E. Tóth, O. Kovács, T. Langó, K. Vékey, *et al.*, “Influence of post-translational modifications on protein identification in database searches,” *ACS omega*, vol. 6, no. 11, pp. 7469–7477, 2021.
- [49] D. J. Geiszler, A. T. Kong, D. M. Avtonomov, F. Yu, F. da Veiga Leprevost, and A. I. Nesvizhskii, “Ptm-shepherd: analysis and summarization of post-translational and chemical modifications from open search results,” *Molecular & Cellular Proteomics*, vol. 20, 2021.
- [50] W. Bittremieux, K. Laukens, and W. S. Noble, “Extremely fast and accurate open modification spectral library searching of high-resolution mass spectra using feature hashing and graphics processing units,” *Journal of Proteome Research*, vol. 18, pp. 3792–3799, Aug. 2019.
- [51] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, “F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing,” in *International Symposium on FPGAs*, pp. 53–62, ACM, 2019.
- [52] S. Salamat, M. Imani, and T. Rosing, “Accelerating hyperdimensional computing on fpgas by exploiting computational reuse,” *IEEE Transactions on Computers*, vol. 69, pp. 1159–1171, aug 2020.
- [53] P. Kanerva, “Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors,” *Cognitive Computation*, vol. 1, pp. 139–159, Jan. 2009.

- [54] A. Rahimi, S. Datta, D. Kleyko, E. P. Frady, B. Olshausen, P. Kanerva, and J. M. Rabaey, “High-dimensional computing as a nanoscalable paradigm,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 9, pp. 2508–2521, 2017.
- [55] Y. Kim, M. Imani, and T. Rosing, “Efficient human activity recognition using hyperdimensional computing,” in *Proceedings of the 8th International Conference on the Internet of Things*, pp. 1–6, 2018.
- [56] F. R. Najafabadi, A. Rahimi, P. Kanerva, and J. M. Rabaey, “Hyperdimensional computing for text classification,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.
- [57] A. Joshi, J. Halseth, and P. Kanerva, “Language recognition using random indexing,” 2014.
- [58] M. Imani, J. Morris, S. Bosch, H. Shu, G. De Micheli, and T. Rosing, “Adapthd: Adaptive efficient training for brain-inspired hyperdimensional computing,” in *IEEE BioCAS*, pp. 1–4, 2019.
- [59] 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)*, IEEE, Mar. 2019.
- [60] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, “PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation,” *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.
- [61] B. Peksag, “astor.” <https://github.com/berkerpeksag/astor>, 2020.
- [62] M. Imani, J. Morris, *et al.*, “Bric: Locality-based encoding for energy-efficient brain-inspired hyperdimensional computing,” in *56th Annual Design Automation Conference*, pp. 1–6, 2019.
- [63] M. Harris, “Optimizing parallel reduction in cuda.” NVIDIA Developer Technology, 2007.
- [64] H. Jin, Q. Song, and X. Hu, “Auto-keras: An efficient neural architecture search system,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1946–1956, ACM, 2019.
- [65] D. Arthur and S. Vassilvitskii, “K-means++: The advantages of careful seeding,” in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA ’07, (USA)*, pp. 1027–1035, Society for Industrial and Applied Mathematics, 2007.
- [66] D. Dua and C. Graff, “UCI machine learning repository,” 2017.
- [67] A. Reiss and D. Stricker, “Introducing a new benchmarked dataset for activity monitoring,” in *2012 16th International Symposium on Wearable Computers*, pp. 108–109, 2012.

- [68] A. Ultsch and J. Löttsch, “The fundamental clustering and projection suite (FCPS): A dataset collection to test the performance of clustering and data projection algorithms,” *Data*, vol. 5, p. 13, Jan. 2020.
- [69] J. Morris, M. Imani, S. Bosch, A. Thomas, H. Shu, and T. Rosing, “Comphd: Efficient hyperdimensional computing using model compression,” in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, 2019.
- [70] NVIDIA, “TensorRT.” <https://developer.nvidia.com/tensorrt>, 2020.
- [71] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, “Tvm: An automated end-to-end optimizing compiler for deep learning,” in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI’18*, (USA), pp. 579–594, USENIX Association, 2018.
- [72] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, “Cupy: A numpy-compatible library for nvidia gpu calculations,” in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.
- [73] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, “Automatic c-to-cuda code generation for affine programs,” in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction, CC’10/ETAPS’10*, (Berlin, Heidelberg), pp. 244–263, Springer-Verlag, 2010.
- [74] S. K. Lam, A. Pitrou, and S. Seibert, “Numba: A llvm-based python jit compiler,” in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM ’15*, (New York, NY, USA), Association for Computing Machinery, 2015.
- [75] H. C. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.
- [76] D. A. Beckingsale, T. R. Scogland, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, and B. S. Ryuji, “RAJA: Portable performance for large-scale scientific applications,” in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, IEEE, Nov. 2019.
- [77] M. Nazemi, A. Esmaili, A. Fayyazi, and M. Pedram, “Synergiclearning: neural network-based feature extraction for highly-accurate hyperdimensional learning,” in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2020.

- [78] Z. Zhou, B. Shi, Z. Zhang, Y. Guan, G. Sun, and G. Luo, "Blockgmn: Towards efficient gmn acceleration using block-circulant weight matrices," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 1009–1014, 2021.
- [79] D. Kleyko, M. Davies, E. P. Frady, P. Kanerva, S. J. Kent, B. A. Olshausen, E. Osipov, J. M. Rabaey, D. A. Rachkovskij, A. Rahimi, and F. T. Sommer, "Vector symbolic architectures as a computing framework for nanoscale hardware," *CoRR*, vol. abs/2106.05268, 2021.
- [80] Y. Ma, M. Hildebrandt, V. Tresp, and S. Baier, "Holistic representations for memorization and inference.," in *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 403–413, 2018.
- [81] P. Poduval, H. Alimohamadi, A. Zakeri, F. Imani, M. H. Najafi, T. Givargis, and M. Imani, "Graphhd: Graph-based hyperdimensional memorization for brain-like cognitive learning," *Frontiers in Neuroscience*, vol. 16, 2022.
- [82] I. Nunes, M. Heddes, T. Givargis, A. Nicolau, and A. Veidenbaum, "Graphhd: Efficient graph classification using hyperdimensional computing," in *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1485–1490, 2022.
- [83] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, p. 855–864, ACM, 2016.
- [84] D. Reis, K. Ni, W. Chakraborty, X. Yin, M. Trentzsch, S. D. D unkel, T. Melde, J. M uller, S. Beyer, S. Datta, *et al.*, "Design and analysis of an ultra-dense, low-leakage, and fast fefet-based random access memory array," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 5, no. 2, pp. 103–112, 2019.
- [85] K. Ni, X. Yin, A. F. Laguna, S. Joshi, S. D unkel, M. Trentzsch, J. M uller, S. Beyer, M. Niemier, X. S. Hu, *et al.*, "Ferroelectric ternary content-addressable memory for one-shot learning," *Nature Electronics*, vol. 2, no. 11, pp. 521–529, 2019.
- [86] Q. Huang, D. Reis, C. Li, D. Gao, M. Niemier, X. S. Hu, M. Imani, X. Yin, and C. Zhuo, "Computing-in-memory using ferroelectrics: From single-to multi-input logic," *IEEE Design & Test*, 2021.
- [87] M. Jerry, S. Dutta, A. Kazemi, K. Ni, J. Zhang, P.-Y. Chen, P. Sharma, S. Yu, X. S. Hu, M. Niemier, *et al.*, "A ferroelectric field effect transistor based synaptic weight cell," *Journal of Physics D: Applied Physics*, vol. 51, no. 43, p. 434001, 2018.
- [88] X. Yin, C. Li, Q. Huang, L. Zhang, M. Niemier, X. S. Hu, C. Zhuo, and K. Ni, "Fecam: A universal compact digital and analog content addressable memory using ferroelectric," *IEEE Transactions on Electron Devices*, vol. 67, no. 7, pp. 2785–2792, 2020.

- [89] A. Kazemi, M. M. Sharifi, Z. Zou, M. Niemier, X. S. Hu, and M. Imani, “Mimhd: Accurate and efficient hyperdimensional inference using multi-bit in-memory computing,” in *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2021.
- [90] A. Kazemi, M. M. Sharifi, A. F. Laguna, F. Müller, R. Rajaei, R. Olivo, T. Kämpfe, M. Niemier, and X. S. Hu, “In-memory nearest neighbor search with fefet multi-bit content-addressable memories,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1084–1089, IEEE, 2021.
- [91] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore, “Automating the construction of internet portals with machine learning,” *Information Retrieval*, vol. 3, pp. 127–163, Jul 2000.
- [92] P. Kanerva, J. Kristoferson, and A. Holst, “Random indexing of text samples for latent semantic analysis,” in *In Proceedings of the 22nd Annual Conference of the Cognitive Science Society*, pp. 103–6, Erlbaum, 2000.
- [93] M. Sahlgren, “An introduction to random indexing,” in *In Methods and Applications of Semantic Indexing Workshop at the 7th International Conference on Terminology and Knowledge Engineering (TKE)*, 2005.
- [94] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” *Advances in neural information processing systems*, vol. 30, 2017.
- [95] J. Kang, B. Khaleghi, T. Rosing, and Y. Kim, “Openhd: A gpu-powered framework for hyperdimensional computing,” *IEEE Transactions on Computers*, vol. 71, no. 11, pp. 2753–2765, 2022.
- [96] M. Fey and J. E. Lenssen, “Fast graph representation learning with PyTorch Geometric,” in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- [97] X. Peng, S. Huang, Y. Luo, X. Sun, and S. Yu, “Dnn+neurosim: An end-to-end benchmarking framework for compute-in-memory accelerators with versatile device technologies,” in *2019 IEEE International Electron Devices Meeting (IEDM)*, pp. 32.5.1–32.5.4, 2019.
- [98] C. L. Giles, K. D. Bollacker, and S. Lawrence, “Citeseer: An automatic citation indexing system,” in *Proceedings of the Third ACM Conference on Digital Libraries*, p. 89–98, ACM, 1998.
- [99] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, “Collective classification in network data,” *AI Magazine*, vol. 29, p. 93, Sep. 2008.
- [100] T. Yang, D. Li, Y. Han, Y. Zhao, F. Liu, X. Liang, Z. He, and L. Jiang, “Pimgcn: A reram-based pim design for graph convolutional network acceleration,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 583–588, 2021.

- [101] S. W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W.-m. Hwu, “Large graph convolutional network training with gpu-oriented data communication architecture,” *arXiv preprint arXiv:2103.03330*, 2021.
- [102] D. Xu, W. Cheng, D. Luo, X. Liu, and X. Zhang, “Spatio-temporal attentive rnn for node classification in temporal attributed graphs,” in *Proceedings of the 29th International Joint Conference on Artificial Intelligence*, 2019.
- [103] A. M. Frank, N. Bandeira, Z. Shen, S. Tanner, S. P. Briggs, R. D. Smith, and P. A. Pevzner, “Clustering millions of tandem mass spectra,” *Journal of proteome research*, vol. 7, no. 01, pp. 113–122, 2008.
- [104] H. Lam, “Building and searching tandem mass spectral libraries for peptide identification,” *Molecular & Cellular Proteomics*, vol. 10, pp. R111.008565–R111.008565, Dec. 2011.
- [105] J. M. Chick, D. Kolippakkam, D. P. Nusinow, B. Zhai, R. Rad, E. L. Huttlin, and S. P. Gygi, “A mass-tolerant database search identifies a large proportion of unassigned spectra in shotgun proteomics as modified peptides,” *Nature Biotechnology*, vol. 33, pp. 743–749, June 2015.
- [106] W. Bittremieux, D. L. Tabb, F. Impens, A. Staes, E. Timmerman, L. Martens, and K. Laukens, “Quality control in mass spectrometry-based proteomics,” *Mass Spectrometry Reviews*, vol. 37, pp. 697–711, Sept. 2018.
- [107] C. T. Walsh, S. Garneau-Tsodikova, and G. J. Gatto, “Protein posttranslational modifications: The chemistry of proteome diversifications,” *Angewandte Chemie International Edition*, vol. 44, pp. 7342–7372, Nov. 2005.
- [108] S. Na and E. Paek, “Software eyes for protein post-translational modifications,” *Mass Spectrometry Reviews*, vol. 34, pp. 133–147, Apr. 2015.
- [109] C. Adams, K. Boonen, K. Laukens, and W. Bittremieux, “Open modification searching of sars-cov-2–human protein interaction data reveals novel viral modification sites,” *Molecular & Cellular Proteomics*, vol. 21, no. 12, 2022.
- [110] M. Heck and B. A. Neely, “Proteomics in non-model organisms: A new analytical frontier,” *Journal of Proteome Research*, vol. 19, no. 9, pp. 3595–3606, 2020. PMID: 32786681.
- [111] Y. Perez-Riverol, J. Bai, C. Bandla, D. García-Seisdedos, S. Hewapathirana, S. Kamatchinathan, D. J. Kundu, A. Prakash, A. Frericks-Zipper, M. Eisenacher, *et al.*, “The pride database resources in 2022: a hub for mass spectrometry-based proteomics evidences,” *Nucleic acids research*, vol. 50, no. D1, pp. D543–D552, 2022.
- [112] M. Wang, J. Wang, J. Carver, B. S. Pullman, S. W. Cha, and N. Bandeira, “Assembling the community-scale discoverable human proteome,” *Cell Systems*, vol. 7, pp. 412–421.e5, Oct. 2018.

- [113] UCSD - Center for Computational Mass Spectrometry, “MassIVE: Mass Spectrometry Interactive Virtual Environment.” <https://massive.ucsd.edu/>.
- [114] A. T. Kong, F. V. Leprevost, D. M. Avtonomov, D. Mellacheruvu, and A. I. Nesvizhskii, “MSFragger: ultrafast and comprehensive peptide identification in mass spectrometry-based proteomics,” *Nature Methods*, vol. 14, pp. 513–520, Apr. 2017.
- [115] A. Devabhaktuni, S. Lin, L. Zhang, K. Swaminathan, C. G. Gonzalez, N. Olsson, S. M. Pearlman, K. Rawson, and J. E. Elias, “TagGraph reveals vast protein modification landscapes from large tandem mass spectrometry datasets,” *Nature Biotechnology*, vol. 37, pp. 469–479, Apr. 2019.
- [116] H. Lam, E. W. Deutsch, J. S. Eddes, J. K. Eng, N. King, S. E. Stein, and R. Aebersold, “Development and validation of a spectral library searching method for peptide identification from MS/MS,” *PROTEOMICS*, vol. 7, pp. 655–667, Mar. 2007.
- [117] L. Jiang and F. Zokaee, “Exma: A genomics accelerator for exact-matching,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 399–411, IEEE, 2021.
- [118] L. Wu, R. Sharifi, M. Lenjani, K. Skadron, and A. Venkat, “Sieve: Scalable in-situ dram-based accelerator designs for massively parallel k-mer matching,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 251–264, 2021.
- [119] W. Huangfu, X. Li, S. Li, X. Hu, P. Gu, and Y. Xie, “Medal: Scalable dimm based near data processing accelerator for dna seeding algorithm,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 587–599, 2019.
- [120] M. Zhou, W. Xu, J. Kang, and T. Rosing, “Transpim: A memory-based acceleration via software-hardware co-design for transformer,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1071–1085, IEEE, 2022.
- [121] W. Xu, S. Gupta, N. Moshiri, and T. Rosing, “Rapidix: High-performance reram processing in-memory accelerator for sequence alignment,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.
- [122] S. Roy, M. Ali, and A. Raghunathan, “Pim-dram: Accelerating machine learning workloads using processing in commodity dram,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 11, no. 4, pp. 701–710, 2021.
- [123] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, “Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 27–39, 2016.

- [124] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, *et al.*, “Hardware architecture and software stack for pim based on commercial dram technology: Industrial product,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 43–56, IEEE, 2021.
- [125] P. Poduval, M. Issa, F. Imani, C. Zhuo, X. Yin, H. Najafi, and M. Imani, “Robust in-memory computing with hyperdimensional stochastic representation,” in *2021 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pp. 1–6, 2021.
- [126] R. Chandrasekaran, K. Ergun, J. Lee, D. Nanjunda, J. Kang, and T. Rosing, “Fhdnn: Communication efficient and robust federated learning for aiot networks,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 37–42, 2022.
- [127] J. E. Elias and S. P. Gygi, “Target-decoy search strategy for increased confidence in large-scale protein identifications by mass spectrometry,” *Nature Methods*, vol. 4, pp. 207–214, Feb. 2007.
- [128] Y. Li, H. Chi, L. Xia, and X. Chu, “Accelerating the scoring module of mass spectrometry-based peptide identification using GPUs,” *BMC Bioinformatics*, vol. 15, no. 1, p. 121, 2014.
- [129] M. Yang, T. Chen, X. Zhou, L. Zhao, Y. Zhu, and L. Wang, “A complete cpu-fpga architecture for protein identification with tandem mass spectrometry,” in *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 295–298, 2019.
- [130] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim, Y. Cho, J. G. Kim, J. Choi, H.-S. Shin, J. Kim, B. Phuah, H. Kim, M. J. Song, A. Choi, D. Kim, S. Kim, E.-B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. Song, J. Youn, K. Sohn, and N. S. Kim, “25.4 a 20nm 6gb function-in-memory DRAM, based on HBM2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications,” in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, IEEE, Feb. 2021.
- [131] M. F. Ali, A. Jaiswal, and K. Roy, “In-memory low-cost bit-serial addition using commodity dram technology,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 1, pp. 155–165, 2019.
- [132] F. Gao, G. Tziantzioulis, and D. Wentzlaff, “Computedram: In-memory compute using off-the-shelf drams,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO ’52*, (New York, NY, USA), p. 100–113, Association for Computing Machinery, 2019.
- [133] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, “Simdram: a framework for bit-serial simd processing using dram,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 329–345, 2021.

- [134] M. S. Truong, E. Chen, D. Su, L. Shen, A. Glass, L. R. Carley, J. A. Bain, and S. Ghose, “Racer: Bit-pipelined processing using resistive memory,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 100–116, 2021.
- [135] L. Liu, A. F. Laguna, R. Rajaei, M. M. Sharifi, A. Kazemi, X. Yin, M. Niemier, and X. S. Hu, “A reconfigurable fefet content addressable memory for multi-state hamming distance,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2023.
- [136] M. Norouzi, D. J. Fleet, and R. R. Salakhutdinov, “Hamming distance metric learning,” in *Advances in Neural Information Processing Systems* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.
- [137] A. Kertesz-Farkas, U. Keich, and W. S. Noble, “Tandem mass spectrum identification via cascaded search,” *Journal of Proteome Research*, vol. 14, pp. 3027–3038, Aug. 2015.
- [138] D. S. Standard and D. SDRAM, “Standard jesd79-4b,” *JEDEC*, Jun, 2017.
- [139] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, “Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 568–580, IEEE, 2016.
- [140] B. Kim, J. Chung, E. Lee, W. Jung, S. Lee, J. Choi, J. Park, M. Wi, S. Lee, and J. H. Ahn, “Mvid: Sparse matrix-vector multiplication in mobile dram for accelerating recurrent neural networks,” *IEEE Transactions on Computers*, vol. 69, no. 7, pp. 955–967, 2020.
- [141] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, “A case for exploiting subarray-level parallelism (salp) in dram,” in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 368–379, IEEE, 2012.
- [142] Y.-B. Kim and T. W. Chen, “Assessing merged dram/logic technology,” *Integration*, vol. 27, no. 2, pp. 179–194, 1999.
- [143] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, “Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory,” in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 33–38, IEEE, 2012.
- [144] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization,” in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 185–197, 2013.
- [145] H. Lam, E. W. Deutsch, and R. Aebersold, “Artificial decoy spectral libraries for false discovery rate estimation in spectral library searching in proteomics,” *Journal of Proteome Research*, vol. 9, pp. 605–610, Dec. 2009.

- [146] N. Selevsek, C.-Y. Chang, L. C. Gillet, P. Navarro, O. M. Bernhardt, L. Reiter, L.-Y. Cheng, O. Vitek, and R. Aebersold, “Reproducible and consistent quantification of the *saccharomyces cerevisiae* proteome by SWATH-mass spectrometry,” *Molecular & Cellular Proteomics*, vol. 14, pp. 739–749, Mar. 2015.
- [147] R. J. Chalkley, N. Bandeira, M. C. Chambers, K. R. Clauser, J. S. Cottrell, E. W. Deutsch, E. A. Kapp, H. H. N. Lam, W. H. McDonald, T. A. Neubert, and R.-X. Sun, “Proteome Informatics Research Group (iPRG)_2012: A study on detecting modified peptides in a complex mixture,” *Molecular & Cellular Proteomics*, vol. 13, pp. 360–371, Jan. 2014.
- [148] A. A. Goloborodko, L. I. Levitsky, M. V. Ivanov, and M. V. Gorshkov, “Pyteomics—a python framework for exploratory data analysis and rapid software prototyping in proteomics,” *Journal of The American Society for Mass Spectrometry*, vol. 24, pp. 301–304, Jan. 2013.
- [149] K. Sohn, T. Na, I. Song, Y. Shim, W. Bae, S. Kang, D. Lee, H. Jung, S. Hyun, H. Jeoung, *et al.*, “A 1.2 v 30 nm 3.2 gb/s/pin 4 gb ddr4 sdram with dual-error detection and pvt-tolerant data-fetch scheme,” *IEEE journal of solid-state circuits*, vol. 48, no. 1, pp. 168–177, 2012.
- [150] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, “Mlir: A compiler infrastructure for the end of moore’s law,” *arXiv preprint arXiv:2002.11054*, 2020.
- [151] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, “Hpvm: Heterogeneous parallel virtual machine,” in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 68–80, 2018.
- [152] A. Ejje, A. Councilman, A. Kothari, M. Kotsifakou, L. Medvinsky, A. R. Noor, H. Sharif, Y. Zhao, S. Adve, S. Misailovic, *et al.*, “Hpvm: Hardware-agnostic programming for heterogeneous parallel systems,” *IEEE Micro*, vol. 42, no. 5, pp. 108–117, 2022.
- [153] W. Xu, J. Kang, and T. Rosing, “A near-storage framework for boosted data preprocessing of mass spectrum clustering,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 313–318, 2022.
- [154] W. Xu, J. Kang, W. Bittremieux, N. Moshiri, and T. Rosing, “Hyperspec: Ultrafast mass spectra clustering in hyperdimensional space,” *Journal of Proteome Research*, vol. 22, no. 6, pp. 1639–1648, 2023. PMID: 37166120.