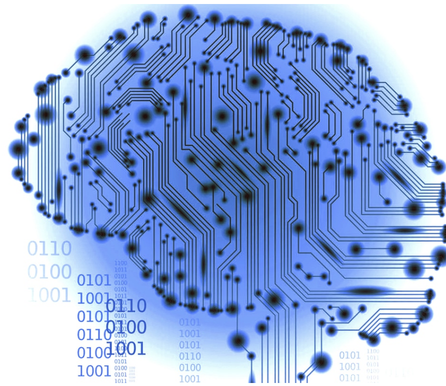
TÉCNICO
LISBOA



DPHDC: A Data Parallel framework for Hyperdimensional Computing

Pedro Silva André

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor(s): Prof. Aleksandar Ilić
Prof. Leonel Augusto Pires Seabra de Sousa

Examination Committee

Chairperson: Prof. José Fernando Alves da Silva
Supervisor: Prof. Aleksandar Ilić
Member of the Committee: Prof. Gabriel Falcão Paiva Fernandes

November 2022

Dedicated to my friend and father, Pedro Jacinto Toste André

May your stories always be the beginning of my dreams

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

This work was supported by the FCT (Fundação para a Ciência e a Tecnologia, Portugal) and EuroHPC Joint Undertaking through the Grant Nº 956213 (SparCity). Furthermore, it would not have been possible without the support of several people whom I would like to thank.

Firstly, I would like to express my gratitude and appreciation to Professor Aleksandar Ilic and Professor Leonel Sousa from Instituto Superior Técnico, whose constant dedication towards this work was unrivalled. Their contagious enthusiasm made all the work seem effortless and made this one of my life's more impactful and fulfilling projects. Their experience and knowledge were also crucial to ensure the quality of the thesis.

It is more appropriate to address everyone in Portuguese for the remainder of this section.

Também quero agradecer a todos os meus amigos e colegas que, através do seu apoio, tornaram não só a realização da dissertação, mas também de todo o curso, mais agradável e recompensante. Destes, creio ser importante destacar, sem qualquer ordem específica, aqueles que de mim se tornaram mais próximos: Martim Andrade, José Henriques, José Luís Parreira e David Martins.

Demonstro também o meu apreço pelo apoio e ajuda que recebi de todos os meus familiares. Em específico, gostaria de agradecer: Ao meu tio, Francisco André, e tia, Filomena André, que mesmo longe sempre quiseram estar presentes; À minha tia, Hermenegila André, que foi sempre uma presença constante durante toda a minha vida; À minha madrinha, Paula Borges, e à sua, mas também minha, avó, Maria Laudelina, que sempre me demonstraram que não é necessário ter-se o mesmo sangue para se fazer parte da família.

Por fim, gostaria de enaltecer a minha família mais próxima, cujo apoio e suporte constantes, a todos os níveis, tornaram esta dissertação possível. O meu mais sincero e sentido obrigado: À minha irmã, Carolina André, que sei que estará sempre lá para mim; À minha mãe, Maria André, de alcunha preferida Milu, que aconteça o que acontecer, sei que está sempre a pensar em mim e cujos cozinhados, confecionados com amor, fizeram com que a distância a casa parecesse ser menor; Ao meu pai e melhor amigo, também Pedro André, que sempre teve como maior prioridade o bem-estar e educação dos filhos e cuja dedicação e sacrificio para atingir estes objetivos não conhecem limites.

A todos vós, obrigado.

Resumo

Computação hiperdimensional (HDC) surgiu recentemente como uma alternativa menos taxativa a métodos tradicionais de aprendizagem automática, particularmente em ambientes com restrições a nível de energia e/ou recursos. Todavia, para ser possível explorar este potencial, é necessário propor algoritmos portáteis, eficientes e de uso geral, desenvolvidos de raiz com vista a tirar partido do paralelismo inerente às operações associadas a HDC. Nesta tese, é proposta uma biblioteca *open-source* e baseada em SYCL, DPHDC, com o fim de facilitar a implementação e acelerar a execução de tarefas de classificação baseadas em HDC em ambientes heterogéneos. DPHDC tem o objetivo de tirar o máximo partido da natureza altamente paralela das operações definidas por HDC, enquanto o seu *design* inovador é desenvolvido de forma a providenciar elevado desempenho e execução portátil em dispositivos com diferentes arquiteturas, como CPUs, GPUs e FPGAs. O armazenamento, movimento e comunicação eficiente de vetores com elevado número de dimensões, cruciais para qualquer aplicação baseada em HDC, também é abordado pela biblioteca proposta de forma a reduzir restrições de desempenho causadas por acesso excessivo à memória. Versatilidade, modularidade e facilidade de uso também foram prioridades durante o desenvolvimento do *design* orientado a objetos da biblioteca proposta. Quando comparada com a mais recente biblioteca dedicada à implementação de aplicações baseadas em HDC, aplicações desenvolvidas utilizando DPHDC são até 13x mais rápidas em CPU e até 10x mais rápidas em GPU, sendo também capazes de supportar uma maior gama de dispositivos e arquiteturas.

Palavras-chave: Computação Paralela, Computação Hiperdimensional, Arquiteturas de Vetores Simbólicos, Aprendizagem Automática, Sistemas Heterogéneos.

Abstract

Hyperdimensional computing has recently emerged as a lightweight classification alternative to traditional Machine Learning methods, particularly in environments with power and/or resource restrictions. However, in order to fully exploit this potential, general-use, portable and efficient data-parallel algorithms, developed from the ground-up to exploit the inherent parallelism of HDC operations, are yet to be proposed. In this thesis, DPHDC, a SYCL-based open-source framework to facilitate implementation and accelerate the execution of HDC-based classification tasks in heterogeneous environments, is proposed. The DPHDC framework aims at fully exploiting the highly parallel nature of HDC operations, while the novel design of the presented library is developed to provide high-performance and portable execution across devices of different architectures such as CPUs, GPUs and FPGAs. Efficient storage, movement and communication of high dimensional vectors, key to any HDC-based application, is also tackled by DPHDC in order to reduce performance bottlenecks. Versatility, modularity and ease of use were also taken into account while developing the intuitive object-oriented design of the framework. When compared to the most recent multi-device capable HDC frameworks, DPHDC is not only up to 13x faster on CPU and 10x faster on GPU but is also able to target more devices and architectures efficiently.

Keywords: Parallel Computing, Hyperdimensional Computing, Vector Symbolic Architectures, Machine Learning, Heterogeneous Systems.

Contents

Acknowledgments . . . vii

Resumo . . . ix

Abstract . . . xi

List of Tables . . . xv

List of Figures . . . xvii

Listings . . . xix

Acronyms . . . xxi

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Objectives . . . 3

1.3 Contributions . . . 3

1.4 Thesis Outline . . . 4

2 Background **5**

2.1 Machine Learning . . . 5

2.2 Hyperdimensional Computing . . . 6

2.2.1 Classification Methodology . . . 7

2.2.2 Similarity Metrics . . . 8

2.2.3 Randomness and Orthogonality . . . 9

2.2.4 Arithmetic Operations . . . 9

2.2.5 State-of-the-art of HDC-based Classifiers . . . 12

2.2.6 Other HDC use cases . . . 15

2.3 Aerospace Applications . . . 16

2.3.1 Current Machine Learning Applications . . . 16

2.3.2 Possible Hyperdimensional Computing Applications . . . 17

2.4 Related Work . . . 18

2.5 SYCL . . . 18

2.5.1 Accelerator Code Development and Submission . . . 19

2.5.2 Data Storage and Movement . . . 20

2.5.3 Vector Addition Example . . . 21

2.5.4 Implementations 24
2.6 Summary 25

3 Data Parallel Framework for Hyperdimensional Computing **27**
3.1 Design 27
3.1.1 Base Vector Generators 29
3.1.2 Encoder Methods 29
3.1.3 Training and Testing 31
3.1.4 Generality of Encoder Modules and Hypervector Storing 32
3.1.5 An Application Example 33
3.2 Development and Implementation 35
3.2.1 Data Management 36
3.2.2 Vector Generators 37
3.2.3 Encoder Methods 40
3.2.4 Training and Testing Methods 50
3.2.5 Python Front-end 56
3.2.6 Targeting FPGAs 56
3.3 Summary 57

4 Experimental Results **59**
4.1 Novel Encoder Alternative to HDNA Encoder I 60
4.2 Scalability with Vector Dimensionality 61
4.3 Comparison with Other Frameworks 63
4.4 FPGA Implementation 64
4.5 Comparison with Original Implementations 65
4.6 Performance Impact of Different DPHDC Methods 66
4.7 Summary 67

5 Conclusions **69**
5.1 Future Work 70

Bibliography **71**

A Code Listings **75**
A.1 Implementation of the VoiceHD Application using the DPHDC Python Front-end 75
A.2 Implementation of the Novel Encoder Alternative to HDNA Encoder I 76

List of Tables

2.1 Comparison of some notable HDC-based classifiers to equivalent ML methods. 15

4.1 DPHDC and TorchHD results of the Language Recognition, MNIST and VoiceHD applications using vectors with 10000 elements. 63

4.2 Execution time and accuracy of the hand gesture recognition application running on Intel Arria 10GX using DPHDC with a vector size of 2500. 64

List of Figures

2.1 Overview of supervised classification using HDC. 7
2.2 Probability of normalized Hamming distance between two randomly generated vectors of dimension d. 9
2.3 Overview of the VoiceHD encoding architecture. 13
2.4 Overview of the european language recognition application encoding architecture when using $N = 3$ (trigrams). 14
2.5 Overview of current major SYCL implementations. 24

3.1 Simplified UML diagram of the DPHDC library. 28
3.2 Flow chart of the encoding of data using DPHDC permutation-based encoders. 30
3.3 Flow chart of the encoding of data using DPHDC positional-based encoder. 32
3.4 Flow chart of the training and querying/testing classification methodologies using DPHDC. 32
3.5 Flow chart of a N-gram based encoder using DPHDC. 33
3.6 Flow chart of the implementation of the VoiceHD application using DPHDC. 34

4.1 Training and testing times of the HDNA example according to vector dimensionality. . . . 61
4.2 Training and testing times of the VoiceHD example according to vector dimensionality. . . 61
4.3 Training and testing times of the European language recognition example according to vector dimensionality. 61
4.4 Training and testing times of the MNIST example according to vector dimensionality. . . . 61
4.5 Classification accuracy of the HDNA example according to vector dimensionality. 62
4.6 Classification accuracy of the VoiceHD example according to vector dimensionality. 62
4.7 Classification accuracy of the European language recognition example according to vector dimensionality. 62
4.8 Classification accuracy of the MNIST example according to vector dimensionality. 62
4.9 DPHDC speedup compared with TorchHD in the Language Recognition, MNIST and VoiceHD applications using vectors with 10000 elements. 63
4.10 Ratio (%) between each method execution time and total application execution time using vectors containing 10000 elements. 66

Listings

2.1 Vector add example using SYCL. 21
3.1 VoiceHD speech recognition application implemented using DPHDC. 34
3.2 Generation of constant vectors. 38
3.3 Generation of random vectors. 38
3.4 Copy of vectors generated on the host to the class hypervectors buffer variable. 39
3.5 Mapping of data elements to the corresponding hypervector index. 40
3.6 Conversion of the provided dataset into hypervector indexes. 41
3.7 Implementation of the `encodeWithXOR` permutation method. 43
3.8 Circular shift right accelerator code. 45
3.9 Implementation of the `encodeWithBundle` method. 46
3.10 Thresholding of accumulators back into hypervectors. 48
3.11 Implementation of the `encodeWithXOR` positional module. 49
3.12 Identification of unique labels and association between encoded hypervector and respective class hypervector. 51
3.13 Implementation of the `reduceToLabelsBundle` method. 52
3.14 Implementation of the `queryModel` method. 54
3.15 Determination of the most similar class vector to every encoded hypervector using cosine similarity. 55
A.1 Implementation of the VoiceHD application using the Python DPHDC front-end. 75
A.2 Implementation of an equivalent HDNA application using the proposed novel encoder. . . 76

Acronyms

API Application Programming Interface

AI Artificial Intelligence

CPU Central Processing Unit

DPC++ Data Parallel C++

DPHDC Data Parallel framework for Hyperdimensional Computing

DL Deep Learning

DNA Deoxyribonucleic Acid

EMG Electromyography

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

HDC Hyperdimensional Computing

IoT Internet of Things

ML Machine Learning

NN Neural Networks

USM Unified Shared Memory

VSA Vector Symbolic Architectures

Chapter 1

Introduction

In the last decade, Artificial Intelligence (AI) and Machine Learning (ML) have exploded in popularity, mainly due to the advancements made in deep artificial neural networks [1], who have been winning numerous contests in pattern recognition [2]. Currently, ML is already being used in a vast number of applications and technologies, such as web search, data mining, image processing, predictive analytics, etc. [3]. In fact, data science and machine learning in particular are considered nowadays as key technologies that are rapidly transforming the scientific and industrial landscapes [4].

The aerospace industry will surely capitalize on big data and machine learning, which excels at solving the types of multi-objective, constrained optimization problems that arise in this field [4]. Indeed, several uses for these technologies in the aerospace field have already started to appear, like satellite pose estimation [5] and structure defect classification [6].

Despite the great leap forward achieved in the past years, the current state-of-the-art ML technologies (deep neural networks) still require considerable amounts of energy and processing power, which severely limits their use in real time and Internet of Things (IoT) applications [7, 8, 9]. The aforementioned limitation is usually more pronounced in the training stage [10], although the inference/testing step, despite being less energy and computationally demanding, can also be a limiting factor [11]. Furthermore, another factor that needs to be taken into consideration when using deep artificial neural networks is the large amount of data necessary to train the model, which may not be available for the application at hand [5].

Naturally, such limitation makes it difficult to use these technologies on devices and applications with power and/or resource restrictions, such as embedded systems, edge computing and IoT devices [12]. The current impracticality of offline speech recognition on portable devices is a clear example of this limitation.

1.1 Motivation

Hyperdimensional Computing (HDC) [13], also known as Vector Symbolic Architectures (VSA) [14], is a brain-inspired, Turing complete computing framework [14] that has emerged as a promising, more

efficient and one-shot learning alternative to traditional ML techniques for classification tasks [12]. HDC applications are based on vectors, usually binary or bipolar, of very high and fixed dimensionality, denominated as hypervectors [13]. This high dimensionality is generally in the order of thousands of dimensions, with 10000 elements per hypervector frequent in classification applications based on HDC. Hypervectors can be combined using two element-wise operations: add and multiply and can also be manipulated using permutations [13]. For classification purposes, the similarity between hypervectors is one of the most important figures of merit, which allows to compare different high dimensional vectors [13, 12]. Compared with traditional ML methods, HDC classification applications typically present acceptable accuracy with lower execution costs [12]. For this reason, many recent research works apply HDC for classification in different application domains, including natural language processing [15, 16], speech recognition [17], Deoxyribonucleic Acid (DNA) sequencing [18], gesture recognition [19] and character recognition [20, 21]. In addition, the resistance to noise, robustness and redundancy of HDC, derived from the equal weight attributed to all vector elements [13], may make it an ideal ML method for critical systems in aerospace applications, especially when considering the large amount of high energy radiation these systems can be exposed to and the hazard it presents [22]. The additional energy and processing power benefits that HDC presents can also be crucial to allow the deployment of ML technologies in satellites, given their power limitations [23].

With the recent surge in research of HDC as a classification method [24], the design and development of performant yet approachable libraries and tools that facilitate implementing and testing new HDC models is imperative. For traditional ML, especially in the Neural Networks (NN) and Deep Learning (DL) fields, libraries with similar goals, like Pytorch and TensorFlow, are developed, highly optimized and widely used by the community, showing how helpful such libraries can be. However, for HDC, they are yet to be developed. It is also important to note that given the unconventional architecture of HDC, such tools would also need to ensure code performance portability by efficiently exploring different types of accelerators and architectures, especially in heterogeneous environments and platforms [25, 26].

Considering the highly parallel nature of HDC algorithms, the efficient exploitation of parallelism across multiple architecturally different devices, while avoiding race conditions, is a significant challenge. Furthermore, since hypervectors contain a very high number of elements, optimization of data storage, movement and communication is another major hurdle that needs to be overcome in order to develop an efficient HDC framework.

Most existing approaches to create an easy-to-use, portable and high-performance HDC framework are typically based on an already existing ML framework, which is then adapted for HDC, i.e., are not built from the ground up for HDC applications. Since the ML frameworks were not designed and developed with HDC in mind, these solutions usually only provide ease of use and/or portability, while entailing performance costs. For example, the state-of-the-art framework that is general and can target multiple devices is TorchHD [27], a HDC framework based on Pytorch. Even though TorchHD can target a considerable amount of devices its portability is limited by the Pytorch backend (e.g. Field Programmable Gate Array (FPGA) and other low-powered devices are currently not supported). Furthermore, TorchHD was developed with a design philosophy focused firstly on ease of use and only then on performance

which typically leads to lower efficiency and performance. Other approaches tend to be device and/or application specific, focused on the hyper-optimization of specific HDC operations on a particular narrow class of devices [28].

1.2 Objectives

With the context mentioned above in mind, there is a need to develop a performance focused, highly portable and general HDC framework that is also approachable and easy to use. In order to close this gap, the objectives for this master thesis are as follows:

- Design and development of a general framework/library for implementation of HDC-based classifiers;
- Support for the most commonly available accelerators, i.e., Central Processing Units (CPUs), Graphics Processing Units (GPUs) and FPGAs, in the proposed framework;
- Higher levels of performance and parallel execution, when compared to existing comparable solutions, needs to be achieved by the developed library;
- Provision of an easy-to-use and intuitive programming interface for HDC application development;
- Benchmarking and testing of the proposed framework across different scenarios involving multiple devices and HDC applications.

1.3 Contributions

In order to achieve the aforementioned objectives, Data Parallel framework for Hyperdimensional Computing (DPHDC), a SYCL-based, performant, portable and robust open-source library designed and developed with the primary objective of efficiently running binary and bipolar based Hyperdimensional Computing Machine Learning classification applications on heterogeneous devices, is proposed in this dissertation. The library was developed using the C++ programming language and the SYCL standard, which is a royalty-free, open industry standard for programming in heterogeneous systems [29]. In order to make it more approachable, the proposed framework can also be compiled for use with the Python programming language. By being SYCL-based, DPHDC can currently target a multitude of devices, from most types of available CPU and GPU up to FPGAs [30]. It is expected that DPHDC will benefit the broader scientific community based on the following aspects:

- A general library built from the ground up to efficiently run HDC-based applications on multiple devices with vastly different architectures;
- Innovative design and development of the proposed framework that tackles the exploitation of parallelism across multiple architectures while minimizing memory bottlenecks;

- Intuitive programming interface that allows the easy implementation of HDC-based classification applications;
- Modular implementation that facilitates the expansion of the proposed framework.

To ensure the compatibility of the proposed library across different compilers and devices, unit tests were developed and are supplied with the framework to guarantee that developed functionalities work as intended. The proposed framework is extensively tested across several devices using notable HDC-based supervised classifiers, recreated employing DPHDC and provided with the library. The experimental results show that DPHDC is up to 13x faster when running on CPU and 10x faster when running on GPU than the state-of-the-art approach, while also being accessible given its intuitive design and accompanying examples.

1.4 Thesis Outline

The remainder of this dissertation is organized as follows:

- Chapter 2 presents the ML and HDC background. This includes data representation, vector generation, operation definition, similarity measurement and state-of-the-art. A brief exposure of previous work related to general-use frameworks dedicated to facilitate the implementation HDC-based classification applications is also done. Aerospace applications of ML are also explored, followed by a description of the SYCL standard, used to develop the proposed framework;
- In Chapter 3, the DPHDC library is presented. The library's high-level design, features, development and implementation details are explored, accompanied by code samples and examples;
- The presentation and analysis of obtained results and benchmarking tests is done in Chapter 4, followed by comparisons with related work;
- Chapter 5 concludes the dissertation and gives directions for future work.

Chapter 2

Background

Given that the research work conducted in the scope of this dissertation aims at proposing a general and optimized framework for HDC-based classifiers, all necessary background topics related with the developed solution are presented are discussed herein. As a result, firstly a brief overview of Machine Learning algorithms is provided. Naturally, the exploration of the mathematical and computing theoretical aspects of HDC, the use of HDC as a ML method and current state-of-the-art applications is the main focus of this Chapter. A brief study of current and possible applications of ML in the aerospace field is also performed in this Chapter while also providing an in-depth analysis of related HDC works. Finally, a concise exposure of the SYCL standard and its major features, necessary for understanding the development of DPHDC, is done.

2.1 Machine Learning

A ML algorithm can be defined as an algorithm in which performance doing a certain task (or tasks) increases with experience, i.e., with the execution of said task (or tasks) [31]. As such, these methods usually first need to go through a training phase before achieving decency at performing the wanted task [9]. Currently, several training techniques and ML algorithms and methods are used and there is not a one-size-fits-all type of algorithm that is best at solving all problems [1, 2, 3]. The ones chosen for a particular case depend on the type of operations that need to be done and the amount and type of data available for training the model [1, 2, 3].

Traditionally, most ML methods can be divided into one of three categories. These are:

- **Supervised Learning** - is the task of inferring a function that maps input data to an output label based on example pairs of data-label [9]. As such, it is easily concluded that these tasks are those which need external assistance, as they need external data to try approximate the intended function. Most classification tasks are supervised learning tasks, since the objective is to classify unlabeled data given sets of labeled examples. This is the most common ML tasked used [3];

- **Unsupervised Learning** - unlike supervised learning above the input data is unlabelled [9]. As

such, ML algorithms can mostly just find relationships between the inputted data [9]. It is mainly used for clustering and feature reduction [9];

- **Reinforcement Learning** - is the task concerned with optimizing agents actions in an environment in order to maximize some numerical reward [32]. The agent is not told what actions are optimal (in the sense that they maximize reward) and must learn itself by trying said actions [32].

Hyperdimensional Computing is usually utilized in the Machine Learning context as a classifier, mostly as a supervised classifier for recognition/classification tasks [12]. As such, in the scope of this dissertation, the main focus will be on supervised classifiers.

With the aforementioned focus on supervised classifiers, it is relevant to note that the current state-of-the-art algorithms used for supervised classification of data usually fall in the deep learning family of methods. The most commonly used ones are deep artificial neural networks, deep belief networks and convolutional neural networks (with this last one being particularly useful for images) [3]. As mentioned in Chapter 1, methods belonging to the deep learning family are usually computationally expensive, as they require the execution of many complex operations [1, 3]. As a consequence, considerable amounts of energy and/or time are required to execute them [7, 8, 9].

2.2 Hyperdimensional Computing

To face the high energy and computational cost of current state-of-the-art ML methods, Hyperdimensional Computing, also known as Vector Symbolic Architectures, as emerged as an efficient ML alternative, especially for supervised classification [12]. Based on the cognitive model developed by P. Kanerva in 1988 [33], HDC is a brain-inspired method in the sense that it is a paradigm that was designed with the ultimate goal of achieving brain-like computing (it can be defined as a brain-inspired computing model) [12]. Since it grew out of cognitive science, it tries to abstract the neural realization of the brain but understand its logical design, in order for that knowledge to be expressed in traditional mathematical and computing language [13]. HDC models are also Turing complete [14].

Hyperdimensional Computing/Vector Symbolic Architectures models are based on the algebraic and geometric properties of high-dimensional spaces [13]. As previously referred, points in these hyperspaces are represented by hypervectors, i.e., vectors with a large and fixed number of dimensions [13]. Depending on the HDC model, such vectors can be composed of several data types, from binary or bipolar to integer or complex numbers [34]. Despite this diversity in hyperspaces, the fundamental concepts of all HDC models are based on the same principles [34, 26]. Considering that binary and bipolar representations are generally more hardware friendly and, consequently, more performant, they have been the preferred type of model used in recent HDC-based classifiers [12]. As such, binary and bipolar hypervectors will be assumed for the remainder of this dissertation, unless otherwise stated.

It is worth noting that hypervectors are holographic, meaning that information is independently and identically distributed across all elements that compose a vector [13]. Such property explains the high robustness and resistance to noise of HDC models since each element of a hypervector encodes the

same amount of information as all other elements [13].

2.2.1 Classification Methodology

As shown in Figure 2.1, the HDC classification methodology usually starts by generating base hypervectors. Base hypervectors, also called basis-hypervectors [35], usually represent the simple data types that compose the more complex data types of the datasets to be analysed. For example, a base hypervector can be generated for each letter of the alphabet if the objective is to encode sentences [15]. An encoder is then responsible for mapping each entry of the dataset to hyperdimensional space, using the aforementioned basis-hypervectors.

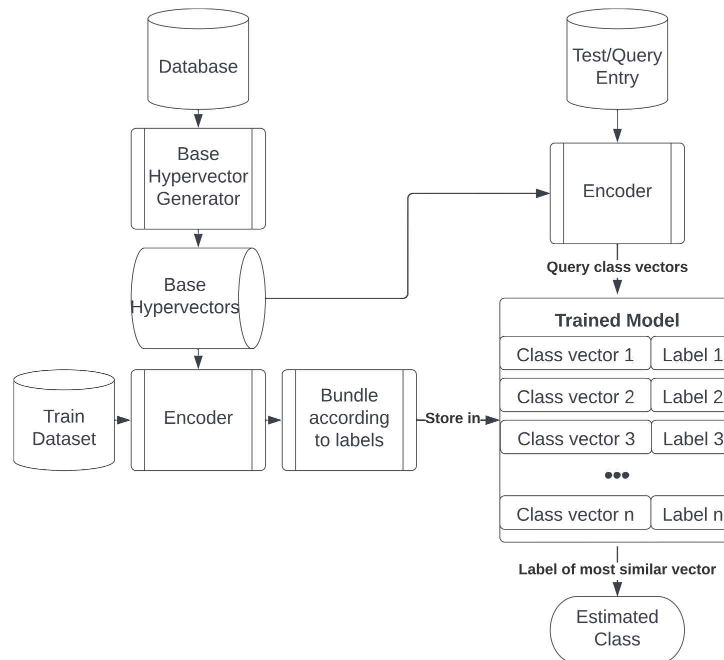


Figure 2.1: Overview of supervised classification using HDC.

If training the model, the mapped hypervectors are then bundled according to their respective label/-class, generating a hypervector per class present in the training dataset (see “Class vectors” in “Trained Model” of Figure 2.1). For testing/querying of the model to be possible, each test/unknown dataset entry needs to also be mapped to hyperspace (utilizing the same encoder used to create the model), generating a hypervector that can then be compared, using a similarity metric, with all the class vectors previously generated. The label/class of the most similar class vector is the estimated class of the data being queried [13].

As can be inferred, the standard HDC classification methodology relies heavily on the encoder. Encoders are application specific and depend directly on the type of data being mapped [36]. As a result, most of the recent HDC research has been focused on creating new encoders and improving feature extraction from data during this step [12]. The architecture and design of some specific encoders are briefly overviewed in Section 2.4.

For classification tasks, the goal of the encoder is to map similar data into similar hypervectors, or

conversely dissimilar data into significantly distinct hypervectors [13, 12]. This property of encoders makes the encoding step analogous to feature extraction of traditional ML methods [12]. The encoder module, in order to take advantage of the mathematical properties of a hyperdimensional space [33], is composed of a combination of the three defined arithmetic operations of HDC: addition, multiplication and permutation [13, 12].

As previously mentioned, Hyperdimensional Computing-based classifiers are usually powerful alternatives to traditional ML-based alternatives given their acceptable accuracy and more economical execution in comparison [12]. However, given the high number of elements needed to represent each hypervector and the inherent parallelism associated with HDC operations, the efficient implementation of HDC models, without the use of previously established frameworks/libraries, is a considerable challenge that researchers face.

2.2.2 Similarity Metrics

Given the necessity of comparing hypervectors in order to estimate the class/label of an unknown dataset entry when using HDC-based classifiers, the predominant similarity metrics used when dealing with binary and bipolar models are presented herein. It is important to mention that despite the existence of other similarity measurements, they are usually associated with different VSA models, being sporadically used when working with binary or bipolar values [13, 26].

The similarity between two hypervectors, when using bipolar values, is usually evaluated through the inner product by computing the cosine of the angle between them, as represented in (2.1),

$$\cos(A,B) = \frac{A \cdot B}{|A||B|} = \frac{\sum_{i=1}^{d} A(i)B(i)}{\sqrt{\sum_{j=1}^{d} A(j)^2 \sum_{k=1}^{d} B(k)^2}} \tag{2.1}$$

where A and B are two hyperdimensional bipolar vectors with d elements, that exist in a space with d dimensions (typically, $d = 10000$) [16]. Furthermore, the presented cosine definition generates a value between -1 and 1, i.e., $\cos(A,B) \in [-1,1]$; where a value of 1 indicates that both vectors are the same (all their elements are identical), while a result of -1 shows that both vectors are entirely dissimilar (all their elements are different).

A comparable similarity metric, applicable in binary HDC models, is the normalized Hamming distance, defined as:

$$d_{Ham}(A,B) = \frac{1}{d} \cdot \sum_{i=1}^{d} (1 \text{ if } A(i) \neq B(i)), \tag{2.2}$$

where A and B are binary hypervectors, and d is the number of dimensions of both vectors [12]. A distance of 1 means that the vectors are entirely dissimilar, while a distance of 0 means that the vectors are identical. Finally, it is important to note that, when working with binary and bipolar models, the similarity between two vectors is evaluated by, essentially, counting the number of different elements between them. Consequently, the similarity and distance terms can be used interchangeably, i.e., similar vectors are close to each other in the space they inhabit, and vice-versa [13].

2.2.3 Randomness and Orthogonality

Given the importance of similarity and dissimilarity between hypervectors for classification tasks, it is crucial to discuss the concept of orthogonality between vectors. Orthogonal vectors have a cosine value of 0 (equivalent to a Hamming distance of 0.5); i.e., the angle between these vectors is $\pm 90°$. In more practical terms for HDC-based classification, half of the elements from these vectors are different [12].

For classification purposes, it is ideal that base hypervectors, used to map all simple elements that compose the entries of a dataset, are orthogonal between themselves [13]. Thanks to hyperdimensionality, two independently and randomly generated vectors in hyperspace will be approximately orthogonal to one another, i.e., the cosine of the angle between them will be close to 0 and the Hamming distance close to 0.5 [13]. As can be seen on Figure 2.2, as d converges to infinity, the probability of two independently and uniformly randomly generated vectors having a normalized Hamming distance of 0.5 converges to 100%. This means that a new, randomly generated vector, will be, with a high degree of certainty, nearly orthogonal to the others that have been previously generated [13]. As a result, the generation of base hypervectors is usually randomly based [12].

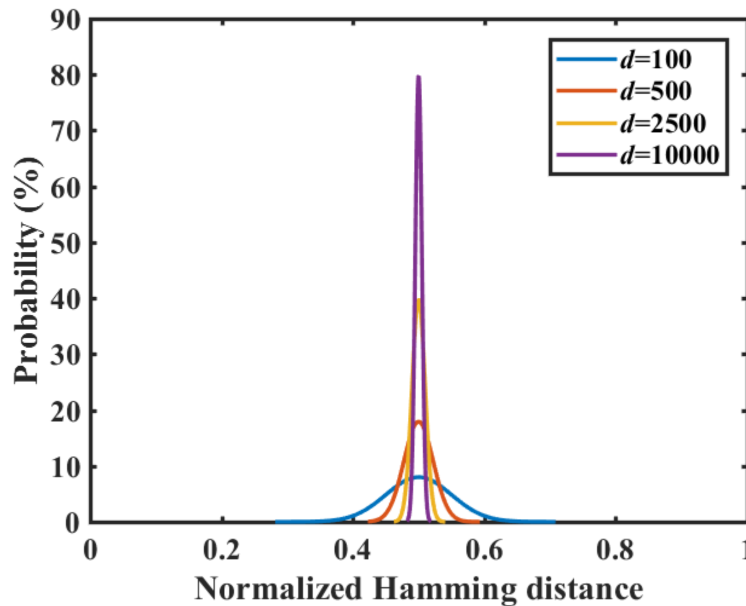


Figure 2.2: Probability of normalized Hamming distance between two randomly generated vectors of dimension d [12].

2.2.4 Arithmetic Operations

In order to better understand how the encoder module works and how data is manipulated when working with HDC models, the three defined arithmetic operations of HDC (addition, multiplication and permutation) are explored in this Section. It is important to note that all these operations are applied on a vector (or vectors) of d dimensions and return a vector of d dimensions as well, i.e., the hyperspace never changes. Furthermore, binary and bipolar hypervectors are used to define and exemplify these

operations. As such, it is important to refer that bipolar and binary representations are mathematically equivalent [37], i.e., apart from some minor differences, the operations are fundamentally equal for both cases. This fact also explains why both bipolar and binary HDC-based models are interchangeable, i.e, binary HDC models can replace bipolar HDC models and vice-versa.

Addition

Addition ($+$ and $[.]$), also known as bundling, is a commutative operation defined as an element-wise majority sum between two or more vectors, i.e., all elements at a certain position across all vectors are summed, and the result is then thresholded back to a bipolar or binary range through the use of a majority rule [13, 26]. For the bipolar case, the operation is defined by

$$R = [X_1 + X_2 + ... + X_N] \Rightarrow$$
$$\Rightarrow \begin{cases} R(i) = 1, & \text{if } X_1(i) + X_2(i) + ... + X_N(i) > 0 \\ R(i) = -1, & \text{if } X_1(i) + X_2(i) + ... + X_N(i) < 0 \end{cases}, \tag{2.3}$$

where R and X_N are bipolar hyperdimensional vectors, $R(i)$ and $X_N(i)$ are, respectively, the element at position i of hypervectors R and X_N and [.] indicates normalization using majority rule. After all elements from position i are summed, the resulting value ($X_1(i) + X_2(i) + ... + X_N(i)$) belongs to range $[-N, N]$, where N is the number of vectors being added. To apply a majority rule to threshold the values back to bipolar range, a comparison needs to be done. As illustrated by (2.3), if the summed value is in range $[-N, 0[$ (< 0), then $R(i)$ will be -1. Otherwise, if the summed value is in range $]0, N]$ (> 0), then $R(i)$ will be 1. When N is even, it is possible that the summed value is equal to 0, i.e., a draw happens. In this case several approaches can be used, from adding an extra randomly generated hypervector to the addition operation to ensure a draw never happens to favoring -1 or 1 in case of a draw [12].

Example (2.4),

$$\begin{aligned} X_1 &= 11110000 \\ X_2 &= 11001100 \\ X_3 &= 10101010 \;, \\ \hline [X_1 + X_2 + X_3] &= 11101000 \end{aligned} \tag{2.4}$$

illustrates that the binary case also follows a majority rule after all elements of a certain position are summed, except the final element value is not compared with 0 but with half the number of vectors being added together ($\frac{N}{2}$), since the range of values from a binary sum is $[0, N]$ and not $[-N, N]$ like in the bipolar case [13, 12].

Multiplication

Multiplication ($*$), also known as binding, is the element-wise exclusive OR (XOR) logical operation between two or more vectors [13, 16]. The operation, in the case of binary hypervectors, is defined as

$$R = X_1 * X_2 * ... * X_N \Rightarrow$$
$$\Rightarrow \begin{cases} R(i) = 1, & \text{if } X_1(i) + X_2(i) + ... + X_N(i) \text{ is odd} \\ R(i) = 0, & \text{if } X_1(i) + X_2(i) + ... + X_N(i) \text{ is even} \end{cases}, \tag{2.5}$$

where R and X_N are binary hyperdimensional vectors, $R(i)$ and $X_N(i)$ are, respectively, the element at position i of hypervectors R and X_N [13]. An example of this operation using bipolar hypervectors is presented in (2.6),

$$\begin{aligned} X_1 &= +1+1+1+1-1-1-1-1 \\ X_2 &= +1+1-1-1+1+1-1-1 \\ X_3 &= +1-1+1-1+1-1+1-1\,, \\ \hline X_1 * X_2 * X_3 &= +1-1-1+1-1+1+1-1 \end{aligned} \tag{2.6}$$

where the definition is identical to the binary case: if the number of elements with value 1 in an arbitrary position i of all vectors being multiplied is odd, then the resulting vector of the multiplication will contain an element with value 1 in position i. Otherwise, if the number of elements with value 1 is even, then the resulting vector will contain an element with value -1 (or 0 in the case of binary vectors) in position i [13, 26].

The multiplication operation is commutative and associative with addition [13, 12]. Furthermore, it is also its own inverse [13, 12], i.e.,

$$X = A * B \Leftrightarrow A = X * B \Leftrightarrow B = X * A. \tag{2.7}$$

Permutation

Finally, a permutation (ρ) is a unary operator that randomly reorders the elements of a hypervector, generally generating a vector that is approximately orthogonal to the original one [13]. Permutations can distribute over any component-wise operations, i.e., addition and multiplication [13]. A commonly used permutation is a circular shift, given the fact that these are usually hardware friendly. An example of a circular shift right permutation is presented in (2.8),

$$X = 11110000 \Rightarrow \rho_{csr}X = 01111000, \tag{2.8}$$

where X is a binary hypervector composed by 8 elements and ρ_{csr} indicates a circular shift right permutation of one element. As can be inferred, every permutation operation can be inverted, i.e., every

permutation has a corresponding inverse that can restore a permuted vector back to its non-permuted state. In the case of the circular shift right, exemplified in equation (2.8), the corresponding inverse permutation is a circular shift left of the same number of elements. This implies that the application of a circular shift right permutation on a hypervector followed by a circular shift left permutation (or vice-verse) results in the original vector, i.e.,

$$X = 11110000 \Rightarrow \rho_{csl}\rho_{csr}X = 11110000 = X, \tag{2.9}$$

where X is a binary hypervector composed by 8 elements, ρ_{csr} indicates a circular shift right permutation of one element and ρ_{csl} indicates a circular shift left permutation of one element.

2.2.5 State-of-the-art of HDC-based Classifiers

HDC and VSA models have evolved gradually since their initial ideas were proposed [33]. Nevertheless, these models have recently gained significant traction and attention from the ML learning scientific community [26], leading to a considerable expansion of proposed classifiers based on HDC. In order to understand the challenges and intricacies related with the implementation of these applications, several recent notable examples of HDC-based supervised classifiers are discussed herein.

As previously mentioned, most supervised classification applications based on HDC start by generating hypervectors to represent the building blocks of a particular data type, for example, generating a hypervector for each letter of the alphabet [12]. These vectors are usually randomly generated to guarantee quasi-orthogonality between themselves (although it is also common for some relation to exist between the vectors in this stage). The next step usually consists of encoding each data entry using the application-specific encoder, for example, encoding an hypervector for each sentence in a dataset [12]. In the case of supervised classification, the hypervectors associated with all known entries of a particular type are usually bundled (added/majority summed) together. This procedure is performed in order to be able to compare these generated class hypervectors with hypervectors encoded from unclassified data during the testing/querying steps [12]. As mentioned in Section 2.2.1, the encoding step is application and data specific and represents the crucial point where pattern learning occurs [12, 36]. As a result, most of the variability between classification applications based on HDC comes from the encoder design [36]. Most encoder modules fall under two categories: positional based encoders and N-gram based encoders.

Positional Based Encoders

In VoiceHD [17], a speech recognition application focused on the ISOLET dataset is proposed. As illustrated by Figure 2.3, the first step is to convert to the frequency domain of the analogue audio signal using Mel-frequency cepstral coefficients (MFCCs). Each frequency signal comprises 617 frequency buckets, each with a real intensity value from -1 to +1. To represent each frequency bucket, 617 random hypervectors are generated (see “ID Hypervectors” in Figure 2.3). The representation of frequency intensity is performed by quantifying the frequency range into 20 sub-ranges (-1 to -0.9, -0.9 to -0.8, etc.)

and associating a hypervector to each one (see “Level Hypervectors” in Figure 2.3). The generation of these sub-range hypervectors is not entirely random, except for the first one that is, indeed, randomly generated. All the other vectors are generated by shifting d/n bits from the previous vector, being n the total number of sub-ranges, i.e., vectors generated. As such, the initial hypervector (the randomly generated one representing the first sub-range of intensities) is diametrically opposed (entirely dissimilar) to the hypervector that represents the last sub-range. Vectors generated in this fashion are usually called level hypervectors [17, 35]. To encode a signal, the hypervector of the sub-range intensity is bound to the corresponding hypervector representing the frequency bucket, generating 617 vectors. These vectors are then bundled to generate the hypervector that represents the audio signal being encoded.

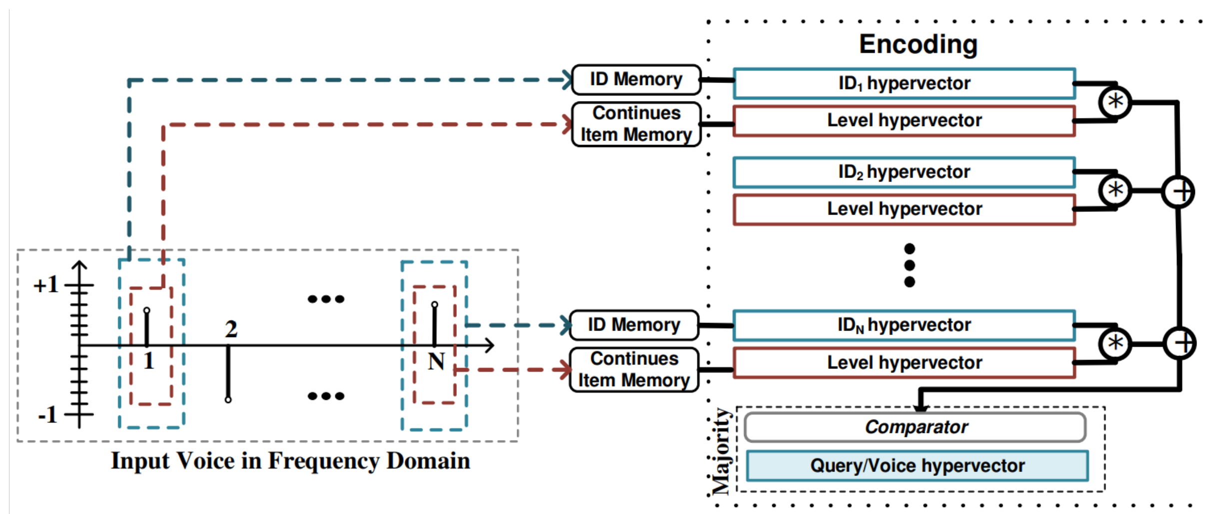


Figure 2.3: Overview of the VoiceHD encoding architecture [17].

VoiceHD was inspired by the approach taken in [19], where a gesture recognition application based on Electromyography (EMG) signals is proposed. The significant difference is that it is not an entire signal that is encoded, but the timestamps with an associated gesture of four EMG signals. Similar to VoiceHD, sub-range intensity hypervectors are generated along with four random hypervectors, one representing each signal. When encoding a timestamp/gesture, each signal hypervector is bound with the corresponding intensity hypervector, generating four vectors which are then bundled together to create the timestamp/gesture hypervector.

N-gram Based Encoders

HDC-based language recognition is also tackled in [15]. The objective was to be able to determine the language of sentences written in one of the 21 European languages. The proposed method starts by generating 27 random hypervectors (to be able to represent the 26 letters of the English alphabet, plus space). To encode a sentence, an N-gram encoder is used. Such an encoder divides the data to be encoded into fragments (groups of letters) of size N. In this particular case, this process consists of dividing the sentences to be encoded into the desired N-grams, like trigrams (N=3), tetragrams (N=4),

pentagrams (N=5), and so on. Once chosen, this value remains constant throughout the application's training and query/testing phases. As shown in Figure 2.4, the N-gram hypervector (H_N) is encoded, as illustrated in equation (2.10),

$$H_N = \prod_{i=1}^{N} (\rho) L_1 * \prod_{i=2}^{N} (\rho) L_2 * ... * \rho L_{N-1} * L_N \tag{2.10}$$

by binding the corresponding letter hypervectors ($L_{position}$) while permuting them according to their place in the N-gram. In particular, the first letter hypervector is permuted N times, the second is permuted $N-1$ times, and the last vector is not permuted. The permutation used (ρ) is always the same. Finally, as also portrayed by Figure 2.4, all the N-gram vectors corresponding to a specific sentence are bundled to generate the hypervector that represents that sentence/text segment.

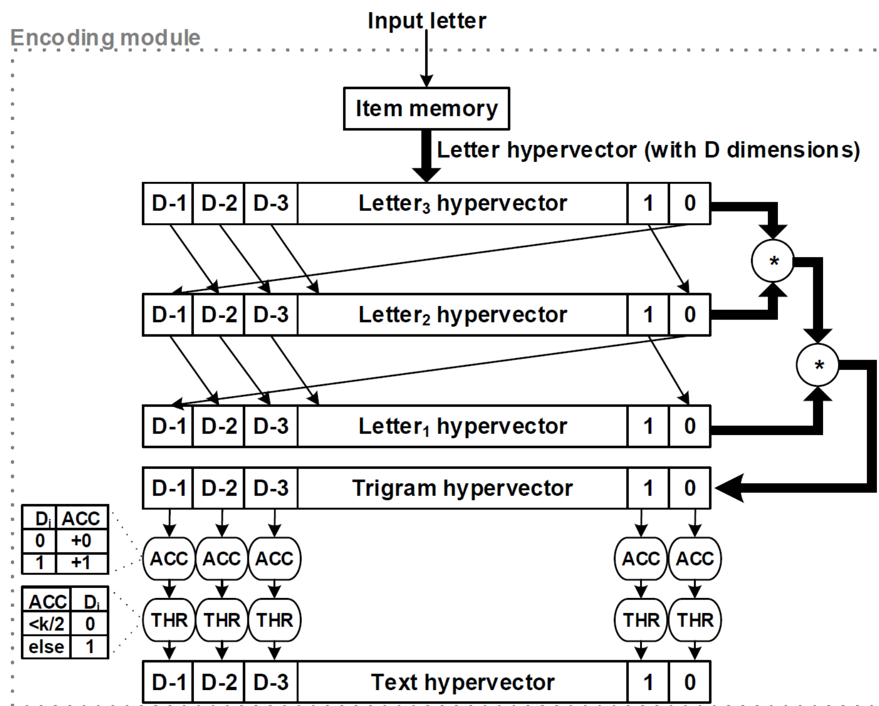


Figure 2.4: Overview of the european language recognition application encoding architecture when using $N = 3$ (trigrams) [15].

A very similar approach is taken by HDNA [18], where the proposed encoder I is also an N-gram-based encoder. DNA sequencing is the main goal of the work, and it is achieved by associating each gene with a class/species. Compared with the language recognition example presented, the only exception is that it is not sentences that are being encoded but genes. After generating 4 random hypervectors (one for each DNA base), the same N-gram logic utilized previously to encode a sentence can be used to encode a gene.

Comparison with Traditional ML Algorithms

In Table 2.1, an accuracy and execution comparison is performed between the state-of-the-art HDC-based classifiers presented so far and the equivalent state-of-the-art Machine Learning models, according to the original works. It is important to note that the gesture recognition using EMG example is not presented on Table 2.1 because a comparison with traditional ML methods in the original work is not performed. Despite this, the application manages an accuracy of 97.8% [19]. As can be observed in Table 2.1, when compared with state-of-the-art ML-based classifiers, HDC-based classification applications present acceptable accuracy while providing considerable speedups and energy efficiency gains, reinforcing the potential of HDC as a lightweight classifier for resource-constrained devices [12]. Despite the general lower accuracy of HDC models, in the case of DNA sequencing, HDNA managed to achieve 3.67% higher accuracy than state-of-the-art techniques when using the same dataset [18, 12].

Table 2.1: Comparison of some notable HDC-based classifiers to equivalent ML methods.

HDC classifier	Application	Hardware	Baseline accuracy*	HDC accuracy	Energy efficiency gain**	Speedup**
VoiceHD [17]	Speech Recognition	CPU	93.60%	88.40%	11.9x	5.3x
European Language [15]	Language Classification	CPU	97.90%	96.70%	-	-
HDNA [18]	DNA sequencing	CPU	94.53%	98.20%	2.05x	4.32x

* Using state-of-the-art ML algorithms

** When compared with state-of-the-art ML algorithms

The efficient implementation of HDC-based classifiers is usually more time-consuming and requires more technical knowledge, both of the target hardware and HDC in general, than their conceptual description. As a result, the speedups and energy gains presented in Table 2.1 could potentially be even higher if the applications developed could target an accelerator with a more parallel optimized architecture, like a GPU, given the highly parallel nature of HDC algorithms. Moreover, the further optimization of the CPU solutions presented in Table 2.1, by potentially using all available processing cores, would, most likely, lead to further gains. This reinforces that a general heterogeneous library for HDC-based classification, such as the one proposed in this thesis, could prove immensely valuable in speeding-up HDC research and application development while providing performance gains, which is a specific topic tackled in this work.

2.2.6 Other HDC use cases

As mentioned previously, Vector Symbolic Architectures are a Turing complete computing frameworks [14]. As a result, although currently predominantly used as supervised classifiers, these models can be used in general computing scenarios [13, 14, 38]. The use of HDC-based models outside classification-based tasks is still an under-explored area of research [14, 38]. Despite this, some promising use-cases have emerged, a relevant one being robust, fault-tolerant and noise resistant hash tables, also known as hash maps [13, 27, 39].

Given a key and a value, two (usually) random hypervectors can be generated to represent the key (K) and its associated value (V). The entry hypervector (E) can be generated by multiplying the key and

value hypervectors, i.e., $E = K * V$. All entry hypervectors can then be bundled together, generating a hash table hypervector. As a result, a hyperdimensional hash map is defined as

$$H(K_1, V_1, K_2, V_2, ..., K_n, V_n) = [\sum_{i=1}^{n}(K_i * V_i)], \tag{2.11}$$

where H is the generated hash map hypervector and K_i and V_i are the corresponding hypervector key-value pairs stored in the data structure [27, 39].

To recover a certain value V_o given its key pair K_o, an HDC hash table relies heavily on the fact that the multiplication operation in binary and bipolar HDC models is its own inverse, as explained in Section 2.2.4 and illustrated by equation (2.7). Firstly it is necessary to recover an approximate version of the value, $V_o' = K_o * H(K_1, V_1, K_2, V_2, ..., K_o, V_o, ..., K_n, V_n)$. The value obtained, V_o', is different from V_o due to the majority rule used by the sum operation. Despite this, the original value hypervector can be recovered by comparing V_o' with all values that were stored in the hash table ($V_1, V_2, ..., V_n$), using the similarity metrics presented in Section 2.2.2. The most similar hypervector to V_o' is, with a high degree of certainty, the value associated to key K_o stored in hash table H [27, 39]. A drawback from this approach is that if any entry hypervector ($E = K * V$) is similar to any other being stored in the same hash map, then the most similar value hypervector to the one being queried, V_o', might not be the value associated with the key provided K_o. Although there is a possibility for a scenario like this to happen, such is extremely unlikely given the high dimensionality of the hypervectors and their ensured quasi-orthogonality by random generation, as explained in Section 2.2.3. Given the holographic properties and high dimensionality of hyperdimensional vectors, such a hash table is inherently robust, resistant to noise and partial fault-tolerant since it is still possible to recover the original value after several elements of the key and or hash table hypervector are flipped [13].

2.3 Aerospace Applications

In the specific case of the Aerospace domain, as referred in Section 1.1, ML, and, by extension, HDC, are expected become important tools to address a plethora of hard-to-solve industry problems. For this reason, it is of the most importance to do a small survey of current and possible ML applications in this field, with intention of exemplifying use cases where HDC would be advantageous.

2.3.1 Current Machine Learning Applications

The aerospace industry is forced to uphold a very high level of safety and reliability, since it directly affects people's lives [40]. As a consequence, new and emerging technologies, like those in the ML field, are usually not immediately embraced but rather left to mature enough in order to assure their safety [40]. Despite this fact, there are already several proposed and in development aerospace projects that use ML.

In academic research, more specifically in [41], it was proposed using neural networks to both esti-

mate the state of a GEO (geostationary orbit) satellite and control it. This approach could have the potential to provide more accurate trajectory control when compared with traditional control techniques [41].

Furthermore, and entering into the industry of space, Neuraspace is a company that specializes in satellite collision avoidance using AI and ML [42]. Their solution is based on tracking and monitoring satellites in order to intercept and action critical conjunction data messages (CDMs). When compared to traditional satellite tracking and monitoring, Neuraspace claims this ML based approach presents some advantages: it does not need humans to operate, it is more efficient, takes less time and detects more high risks despite reporting less false alarms [42].

2.3.2 Possible Hyperdimensional Computing Applications

The previously referred applications are true testaments of the advancements ML has made in recent times, not only in accuracy but also in reliability and stability. Thus it truly can become an indispensable part of the aeronautics and space world in the future. For example, applying ML classification in aerospace is a topic yet to be fully investigated especially for the type of problems where HDC applications are more studied and effective.

A notable exception to this rule is aerospace structure defect classification. Given the importance of safety in the aerospace industry, this type of classification can have tremendous importance [40]. In [6], several extraction methods from (eddy) current signals and images are proposed along with classification methods (mostly based on neural networks) for said features. Multilayer perceptron and uncertainty managing batch relevance based artificial intelligence algorithms were found to be the best classifiers. Replacing these neural network based classifiers with HDC ones represents a novel approach to provide efficient and accurate results [6]. As with any ML approach, to apply HDC concepts to this problem it is also needed to guarantee the availability of large datasets for training, which might not be publicly available.

Another notable application area of HDC in aerospace is satellite pose estimation. It usually consists of estimating the orientation of an uncooperative target satellite using its images. Reliable pose estimation of uncooperative satellites is a key technology for enabling future on-orbit servicing and debris removal missions. Despite this, in [5] the results of a competition showed that the deep learning models used, which rely on large annotated datasets, have varying degrees of success. The main explanation presented was that while there is a considerable amount of datasets for various other applications of computer vision and pose estimation that allow for training state-of-the-art machine learning models, there is a lack of such datasets for spacecraft pose estimation. The main reason arises from the difficulty of obtaining a considerable amount of space-borne images of spacecraft with accurately annotated pose labels [5].

2.4 Related Work

In the existing state-of-the-art, only rare attempts are made on devising general-use and multi-device HDC frameworks dedicated to facilitating the implementation and accelerating the execution of HDC-based applications. The most prominent solution is TorchHD [27], an open-source python library for HDC applications. It is based on the machine learning framework PyTorch. Correspondingly, hypervectors are represented as tensors where each element is a 32-bit floating-point number. This representation allows the use of HDC models beyond binary and bipolar, but at the cost of reduced execution efficiency (performance) of built HDC models, given the significantly higher memory requirements when compared with boolean/bipolar values. This drawback stems from TorchHD's philosophy of prioritizing ease of use and feature set over performance, i.e., from being built on top of PyTorch, which operations are not optimized explicitly for HDC execution. The device architectures it can target are limited to the ones supported by PyTorch, which does not include low-power and resource constraint devices, where HDC-based classifiers show significant potential as lightweight alternatives to traditional ML-based methods [12].

To deal with these shortcomings, the library proposed in this thesis can efficiently target several devices, independently of architecture differences, while optimizing memory used, data transfers and exploiting parallelism across different platforms. Thanks to the novel design and development of the framework such is achieved without sacrificing ease-of-use.

2.5 SYCL

The development of a framework that can efficiently target several device architectures faces considerable implementation and program-ability challenges. Such difficulties when dealing with heterogeneous systems arise mainly from the need to use different, frequently vendor-specific, programming languages and models for each architecture and/or vendor being targeted. One solution to have a single design and implementation that is able to target architecturally distinct devices, from CPUs to GPUs and FPGAs, is to use and/or adapt higher level languages and frameworks that can already target the intended architectures. As mentioned in Section 2.4, this was the approach taken by TorchHD [27], which is based on the Pytorch ML framework for the Python programming language. This solution usually leads to an intuitive development and implementation that comes at the cost of suboptimal hardware utilization and performance. A novel approach to deal with this problem while taking full advantage of the hardware being targeted is to design and build a framework from the ground-up using the SYCL standard/specification [29]. Given the performance goals of the proposed framework, the latter approach presented was the one selected for the development and implementation of the library. As a result, an explanation of the SYCL 2020 specification (revision 5) [29] and its main concepts in the context of this work is performed herein.

SYCL (pronounced "sickle") is a royalty-free, open industry standard for programming in heterogeneous systems [29]. The design of SYCL allows standard C++ source code to be written such that it can run on either an accelerator or on the host (usually a CPU) [29]. As a result, a separation between host

code and accelerator code exists. The host code should be developed like any other application following the ISO C++ standard. It is important to mention that SYCL was developed with the goal in mind of allowing developers to use modern C++, i.e., versions 17 and later of the standard. Appropriately, the SYCL specifications are based on these versions [29].

2.5.1 Accelerator Code Development and Submission

Any code segment destined to be executed on an accelerator device is entitled a kernel. As mentioned previously, SYCL allows for the execution of standard C++ code on the target device, although some features cannot be used due to limitations imposed by the capabilities of the underlying heterogeneous platforms [29]. These features, that can be used outside of kernels, include virtual functions, virtual inheritance, throwing/catching exceptions, and run-time type-information [29].

SYCL Programs can be single-source, meaning that the same file contains both the code that defines the compute kernels to be executed on accelerator devices and also the host code that orchestrates execution of those compute kernels [43]. As a result, the separation between kernel code and host code is done by representing kernels in one of three different ways [43]:

- Lambda expressions;
- Named function objects (functors);
- Interoperability with kernels created via other languages or APIs.

This last option allows for the use of device/vendor specific functions and kernels already developed in other languages, like CUDA and OpenCL, inside SYCL. Of the options available, lambda expressions are the most used since they do not present any performance penalties over other representations while allowing capture rules that automatically pass data to kernels [43].

One of SYCL main goals is to provide a portable way to achieve efficient parallel execution across a wide range of device architectures [29]. As a result, kernels can be mainly invoked using one of two SYCL member functions: `single_task` and `parallel_for` [29]. `single_task`, as the name implies, executes the submitted kernel once. `parallel_for` executes the submitted kernel code several times. This last function can be used to launch data-parallel kernels or nd-kernels [29]. In the first case, a simple range is provided which leads to several kernels of the same type being concurrently executed across said range. This is ideal for kernels where the same operation is applied across the whole range and where each kernel being launched is independent, i.e., there is no data dependency between them. In this case, SYCL implementations can automatically select the work-group size to be used taking into account the device being targeted, without interaction from the user [29]. For more complex kernels and for finer performance control, the use of nd-kernels is recommended. Such can be achieved by providing a nd-range and a work-group size to the `parallel_for` function [29].

Invoked/launched kernels need to be associated with an accelerator on which they will be executed. Such a task is achieved by submitting kernels to a queue [29]. A queue is an abstraction provided by SYCL that allows scheduling kernels for execution on a certain device. The creation of a SYCL queue

starts by providing a device selector, which will provide the device that will execute the kernels submitted. As the name implies, a queue will ensure the chronological execution of kernels provided to it, i.e., the first kernels provided will be executed first while the remainder wait for device availability [29]. A queue can only be associated with one device at a time, including the host, although several queues associated with the same device can exist at the same time [29]. There also exists the possibility of targeting more than one device at the same time by using multiple queues, one associated with each device in the system [29].

2.5.2 Data Storage and Movement

With kernel development and submission tackled, it is important to mention how data storage and transfers between the host and accelerator devices can be managed using SYCL. SYCL offers two distinct technologies to deal with data movement and storage: Unified Shared Memory (USM) and buffers coupled with accessors [29].

Unified Shared Memory allows for explicit data allocation and movement using pointer-based syntax familiar to C/C++ developers [29]. It is possible to allocate data either on the host or target device. Data allocated in this fashion can then be explicitly copied from one device to another. The allocation of shared data across devices can also be done, although it is not recommended since the SYCL backend deals with the necessary data transfers by only performing them when a specific kernel (that needs that data on a different device then where it is currently) is called, which often leads to memory bottlenecks [29, 43]. USM presents many disadvantages when compared with the more modern approach of buffers/accessors, of which two stand out: data dependency across kernels needs to be managed manually by the developer in order to avoid data races and Unified Shared Memory needs to be explicitly freed by the user. As a result, USM is supported by SYCL mostly to facilitate the porting of applications already developed in other programming languages/standards that support this type of memory management [43]. It is generally recommended that applications built from the ground-up with SYCL use buffers/accessors [29, 43].

Buffers, as the name implies, are an abstraction provided by SYCL that allow an object to temporarily hold data while it needs to be used by an accelerator [29, 43]. When creating a buffer, three essential arguments need to be provided: the type of data the buffer will hold (`int`, `float`, `bool`, etc.), the dimensions of the buffer (it can be one-dimensional, two-dimensional or three-dimensional) and the range of the buffer [29, 43]. A pointer to host data can also be provided. If that is the case, the buffer object will take ownership of that data until it is destroyed (usually by getting out of scope). When destroyed, a buffer associated with host data will write back any changes performed by kernels to the original memory space [29, 43]. Otherwise, if no host data is provided to the buffer, data is automatically allocated on the device the buffer is first used on and is automatically deallocated when the buffer is destroyed [29, 43].

To be able to access data contained in a buffer each developed kernel needs to use accessor objects [29, 43]. The construction of an accessor object requires a buffer object which contains the data to be accessed and a specifier of the type of access the kernel needs: read only, write only or read and

write. By specifying the access type, the SYCL backend is able to construct a task graph, i.e., a graph of tasks to be performed (kernels to be executed) and the data dependencies between them [29, 43]. This allows for automatic data conflict management since the SYCL backend ensures that the execution of a kernel only starts when all other tasks that change data accessed by said kernel are completed [29, 43]. The creation of a task graph also allows buffers to migrate data between devices as efficiently as possible, given the kernel execution order inferred by the task graph [29, 43].

2.5.3 Vector Addition Example

To better illustrate and complement the SYCL concepts presented so far, please consider the example provided in Listing 2.1. The example performs the parallel addition of three integer vectors, considered as the “Hello World!” of parallel computing, on an available GPU on the system. The three vectors to be added are allocated and initialized in different ways in order to demonstrate the capabilities of the SYCL buffer class. They are allocated and initialized as follows:

- Vector 1 (`v_1`) is allocated and initialized on the host;
- Vector 2 (`v_2`) is allocated on the host but initialized on the accelerator;
- Vector 3 (`v_3`) is allocated and initialized on the accelerator;
- The resulting vector (`v_result`) is allocated on the host and is not initialized.

Furthermore, each position of these vectors is initialized according to its index. Vector 1 is filled with its own indexes, i.e., `v_1`= $[0, 1, 2, 3, ...]$, while vector 2 and vector 3 are filled, respectively, with the double and triple of the indexes, i.e., `v_2`= $[0, 2, 4, 6, ...]$ and `v_3`= $[0, 3, 6, 9, ...]$.

Listing 2.1: Vector add example using SYCL.

```
1  #define VECTOR_SIZE 10000
2  #include <CL/sycl.hpp>
3
4  int* addVectors(int* v1, int* v2);
5
6  int main() {
7      int* v1 = new int[VECTOR_SIZE];
8      int* v2 = new int[VECTOR_SIZE];
9      for(int i=0; i < VECTOR_SIZE; i++){
10         v1[i] = i;
11     }
12
13     int* v_result = addVectors(v1, v2);
14
15     delete[] v1;
16     delete[] v2;
17     delete[] v_result;
```

```
18      return 0;
19  }
20
21  int* addVectors(int* v1, int* v2){
22      int* v_result = new int[VECTOR_SIZE];
23
24      sycl::queue queue{sycl::gpu_selector()};
25      sycl::range<1> vector_range(VECTOR_SIZE);
26      sycl::buffer<int, 1> buff_v1(v1, vector_range);
27      sycl::buffer<int, 1> buff_v2(v2, vector_range);
28      sycl::buffer<int, 1> buff_v3(vector_range);
29      sycl::buffer<int, 1> buff_v_result(v_result, vector_range);
30
31      queue.submit([&](sycl::handler &h) {
32          sycl::accessor acc_v2(buff_v2, h, sycl::write_only);
33          sycl::accessor acc_v3(buff_v3, h, sycl::write_only);
34          h.parallel_for(vector_range, [=](sycl::id<1> i) {
35              acc_v2[i] = 2*i;
36              acc_v3[i] = 3*i;
37          });
38      });
39
40      queue.submit([&](sycl::handler &h) {
41          sycl::accessor acc_v1(buff_v1, h, sycl::read_only);
42          sycl::accessor acc_v2(buff_v2, h, sycl::read_only);
43          sycl::accessor acc_v3(buff_v3, h, sycl::read_only);
44          sycl::accessor acc_v_result(buff_v_result, h, sycl::write_only);
45          h.parallel_for(vector_range, [=](sycl::id<1> i) {
46              acc_v_result[i] = acc_v1[i] + acc_v2[i] + acc_v3[i];
47          });
48      });
49
50      return v_result;
51  }
```

As can be observed in Listing 2.1, the vector addition example provided starts by defining the vector size to use across all vectors (10000), in line 1, and by including the main sycl header file, in line 2, necessary for the development of any SYCL-based application. It is then possible to infer that the application implementation is divided into two functions: `main` and `addVectors`.

The `main` function represents the entry point of the developed application. Moreover, it is responsible for allocating vectors 1 and 2 (in lines 7 and 8, respectively), for initializing vector 1 (lines 9 through 11), for calling the `addVectors` function using both vectors already allocated (line 13) and, finally, for deallocating the resulting vector of the addition operation (`v_result`) and vectors 1 and 2 (lines 15 through 17).

From the description provided of the main function, it can be inferred that the major program logic is implemented in function `addVectors`. It starts by the allocation of the result vector returned by the

function, in line 22. This allocation is done on the host such that, after all kernels are executed, the result can be accessed by the host.

The creation of a queue is an indispensable step for any SYCL application. In the presented example, the queue is created in line 24 and is associated with a GPU available in the system through the use of the `gpu_selector` SYCL function. Naturally, other selectors, like `cpu_selector` or `fpga_selector`, could be used in order to target different accelerators [29, 43]. When multiple devices of the same type are present in the system where the application is going to be executed, as is the case with systems that contain both integrated and discrete graphics cards, custom device selectors can be created to target devices of a specific vendor, model and/or ID [29, 43].

This is followed by the declaration of the range (line 25) to be used when declaring the vector buffers and describing the kernels to be executed. As can be observed, this is a one-dimensional range with the same size as all vectors. In lines 26 through 29 the creation of all necessary buffers, crucial for data movement between devices, is performed. The buffers associated with the result vector and vectors 1 and 2 will take ownership of the provided host data, as mentioned previously, and will return it, with all the modifications performed by the execution of the kernels, once the buffers get out of scope, i.e., when they are destroyed at function return (line 50). In the case of the buffer associated with vector 3, since no host data is provided, the data associated with this buffer will be allocated on the GPU once it is first invoked inside a kernel and will be deallocated at buffer destruction (line 50 once again). This last approach can have performance benefits, since it avoids unnecessary data movement, at the cost of flexibility, as the host is not able to directly access this data.

In lines 31 through 38 the first kernel to be launched by the application, responsible for initializing vectors 2 and 3, is described. As can be observed, lambda functions were used to represent the device code and its submission. The declaration of accessors for write only access to vectors 2 and 3 is performed in lines 32 and 33. The use of the write only specifier can be explained by the operation being performed, i.e., since it is the goal of the kernel to initialize the vectors, there is no need to check what data is currently there, it is only necessary to write the intended initialization values. In line 34, the `parallel_for` SYCL member function is called using the range defined previously. Such will allow the device code (initialization of vectors 2 and 3), presented in lines 35 and 36, to be executed concurrently across the provided range, i.e., zero through vector size minus one.

The next kernel, presented in lines 40 through 48, is responsible for the vector addition and is conceptually identical to the one described so far. The major differences are the declaration of more accessors (lines 41 through 44) necessary for the addition of the three vectors, the change of the access modifiers of vectors 1, 2 and 3 to read only and the change to the device code itself in line 46. It is also important to mention that, given the data dependency of vectors 2 and 3 used in the second kernel, the execution of this kernel will only happen after the execution of the first one (which writes to vectors 1 and 2) is completed. As mentioned previously, this automatic avoidance of data races is made possible by the use of buffers and the task graph that SYCL automatically generates when using this type of data management technique.

After the execution of both kernels, the function returns the result vector (line 50). This function

return, as mentioned previously, also leads to all created buffers getting out of scope, which induces their destruction. By being destroyed, buffers return data ownership back to the host, allowing for the use of the resulting vector in the remainder of the program. When printing the result vector in the `main` function, after executing vector addition, the obtained result is as expected, i.e., $\texttt{v_result} = [0, 6, 12, 18, ...]$.

2.5.4 Implementations

So far, the key aspects of the SYCL specification were discussed and explored, without mentioning how applications developed using SYCL can be executed. This section intends to discuss this exact topic.

In order to run a SYCL application, it is first necessary to use a SYCL implementation that is capable of converting SYCL compliant source code into machine code that can be executed [43]. The main differentiating factors between implementations are the number of SYCL features implemented, accelerators that can be targeted and hardware utilization.

Currently, several SYCL implementations exist, although the most complete (in terms of SYCL features implemented), versatile and performant are: Intel oneAPI Data Parallel C++ (DPC++) [44], Codeplay ComputeCpp [45] and hipSYCL [46]. As can be seen in Figure 2.5, all implementations mentioned can target, among others, any CPU and can also target Intel, Nvidia and AMD GPUs, even if some implementations can only do so experimentally. It is important to mention that, currently, although the Intel oneAPI DPC++ [44] and hipSYCL [46] implementations are open-source, Codeplay ComputeCpp [45] is closed-sourced. It can also be observed in Figure 2.5 that each implementation uses specific Application Programming Interface (API) back-ends, like Nvidia CUDA, OpenCL, ROCm, etc., to generate the machine code to be executed on a specific accelerator.

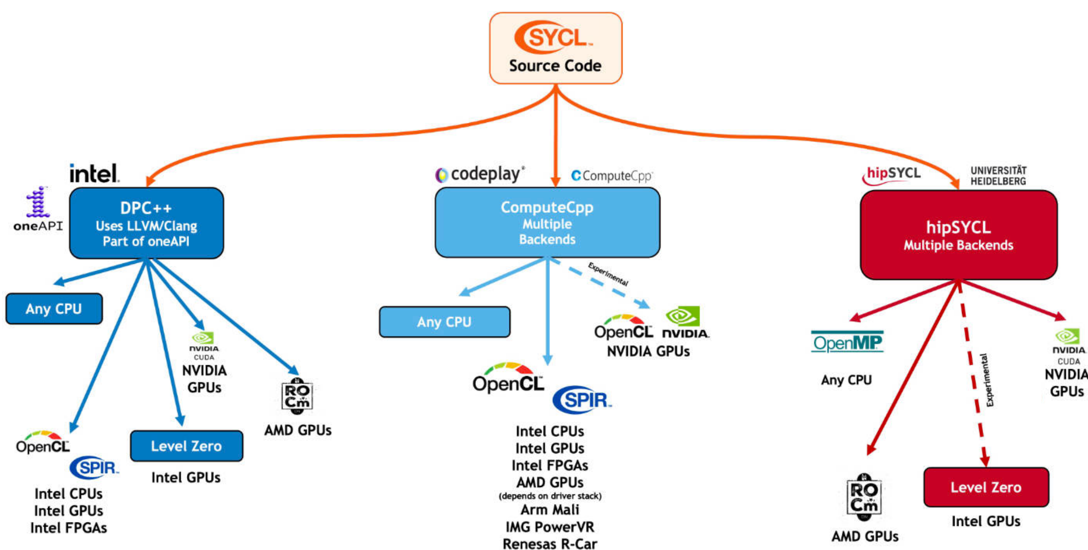


Figure 2.5: Overview of current major SYCL implementations [30].

2.6 Summary

In the present chapter, a brief presentation of ML methods and algorithms was performed followed by an exposition and explanation of the Hyperdimensional Computing framework. A study of current and possible ML applications in the aerospace field was also carried out. Finally, an analysis of related work and the SYCL standard, coupled with an example, was completed.

Chapter 3

Data Parallel Framework for Hyperdimensional Computing

The proposed Data Parallel framework for Hyperdimensional Computing (DPHDC) is developed with the aim of efficiently and robustly running classification tasks based on HDC across devices of different architectures while fully exploiting their processing capabilities[1]. For this purpose, DPHDC was developed using the C++ programming language and the cross-platform SYCL abstraction layer. To provide versatility and ease of use, the proposed framework is also extended with a Python-based front-end, since the Python programming language is commonly used for machine learning research [47].

To maximize hardware utilization across different architectures and minimize other performance bottlenecks, mainly related with data memory accesses and communication, a unique design and optimized algorithms for HDC functions and operations are integrated in the proposed framework (as presented in Sections 3.1 and 3.2, respectively). Another goal of DPHDC is to also ensure an intuitive and easy to use interface, both for beginner and experienced users, despite the vast range of devices it can support.

It is also worth emphasizing that DPHDC is designed to be compatible with any SYCL-capable compiler. By being SYCL-based, DPHDC can currently target any CPU, including low-powered RISC ARM processors, most GPUs and FPGA cards. To ensure that DPHDC works with a particular setup, unit tests were developed and are provided with the library. These can be executed to ensure that the behaviour of all implemented functionalities is as expected.

3.1 Design

As previously referred, at the heart of any HDC-based application are hypervectors. Each one of these large vectors, either binary or bipolar, occupies a significant amount of memory. As a result, the representation, storage and communication of hypervectors needs to be handled in a way that minimizes memory footprint and data movement. The object-oriented design of the DPHDC library, illustrated in Figure 3.1, copes with this challenge by encapsulating two main classes: `HDMatrix` and

[1] DPHDC is publicly available at https://github.com/PedroSAndre/DPHDC

`HDRepresentation`. Both objects of type `HDMatrix` and `HDRepresentation` represent an arbitrary number of arbitrarily large binary or bipolar hypervectors, with vectors in the same matrix or representation having all the same size. With this intuitive hypervector-centric approach, all functionalities related with HDC classification applications are invoked as methods of these types of objects. As shown in Figure 3.1, these methods can be divided into: i) vector generators, ii) encoders, iii) reducer to labels, iv) query and v) storing and reading objects of type `HDMatrix` and `HDRepresentation`.

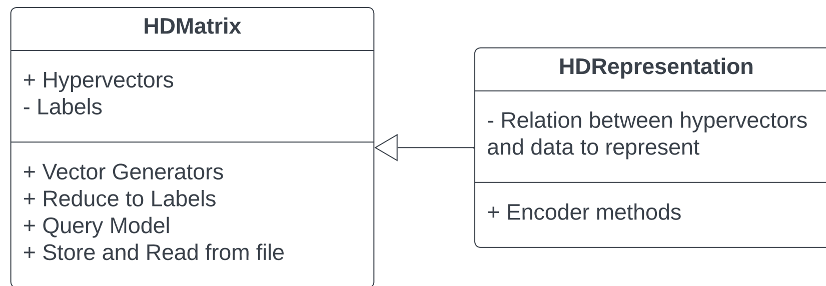


Figure 3.1: Simplified UML diagram of the DPHDC library.

As previously stated, binary and bipolar HDC models are the ones that show the most promise as a lightweight alternative to traditional ML techniques [12]. As a result, to optimize the library's memory requirements, DPHDC was designed and developed with a binary/bipolar HDC first approach. It is also relevant to note that binary and bipolar representations are mathematically equivalent which makes it possible to use the same environment for both models [37].

An `HDMatrix` object can also have an associated label to each hypervector in the matrix by use of the `labels` vector variable. This allows for an `HDMatrix` object to store the data class that each vector represents, allowing bundling hypervectors according to their labels (training the model) and query of an already trained model. As shown in the simplified UML diagram of the proposed framework (Figure 3.1), the `HDRepresentation` class is an extension of the `HDMatrix` class, thus it inherits all methods and variables from it. The `HDRepresentation` class was designed and implemented in order to solve the problem of associating data provided by the user to hypervectors. Providing this functionality is crucial for encoder methods to work since, as data is being read from the dataset, it is necessary to know which base hypervector is associated with each element of the dataset before applying the desired HDC-based arithmetic operations. As a result, each vector in a `HDRepresentation` is associated with an element that can be found on a particular dataset. For example, each vector can be associated with a letter (char), a string, an integer number, a floating point number, etc. A vector can even be associated with an object of a user defined class or struct, as long as it has a comparison method. Ideally, an `HDRepresentation` will contain a hypervector for representing each different type of data that can be found in the dataset to be mapped into hyperdimensional space. This representation of base data using hypervectors allows the encoding of similar information into similar parts of the hyperspace, using the encoder modules, as described in Section 2.2 and exemplified in Section 2.2.5.

3.1.1 Base Vector Generators

To generate a set of base or general hypervectors belonging to an `HDRepresentation` or `HDMatrix` object, respectively, **vector generator** methods must be used. Currently, four different types of vectors can be generated by DPHDC:

- **constant vectors**;
- **random vectors**;
- **level vectors**;
- **circular vectors**.

Constant vectors contain the same element in all dimensions, i.e., all vectors are composed of zeros (negative ones in the bipolar case) or ones. As the name implies, random vectors have all their elements randomly generated.

Level vectors were already described in Section 2.2.5. The first vector is randomly generated, while the remaining vectors are obtained by flipping $d/2/N$ elements from the previous vector generated (where d is the vector size and N the number of vectors being generated) such that the first and last vectors are quasi-orthogonal. DPHDC currently offers two possibilities to generate level vectors: **half-level vectors**, where the first and last vectors in the matrix are quasi-orthogonal, i.e., the traditional definition of level vectors, and **full-level vectors**, where the first and last vectors in the matrix are nearly diametrically opposed. The creation of full-level vectors is achieved by flipping d/N elements from the previous vector generated, contrary to half-level hypervectors, where $d/2/N$ elements are flipped instead. It is important to mention that, in order for the first and last hypervectors to be quasi diametrically opposed or orthogonal, it is necessary that once a certain element is flipped it is never flipped again, i.e., supposing, as an example, that the first element is flipped while generating hypervector number five from hypervector number four, then this element will never be flipped again during the generation of the remaining hypervectors.

Circular vectors were proposed in [35] with the intent of mapping circular data types to the hyperspace, like angles. These are a set of hypervectors whose distances are proportional to that of a set of equidistant points on a circle [35], i.e., any vector is closely related to its neighbours while being distinctly dissimilar to the vector that opposes it in the circle.

It is important to mention that hypervectors are generated, using the above described methods, whenever a `HDMatrix` or `HDRepresentation` object is created. As a result, when creating an object of these types, it is necessary to provide the constructor method with the vector generator method to use. Such is achieved through the use of an enumerator type defined by the library.

3.1.2 Encoder Methods

After generating a base representation, the next step usually in HDC processing consists of **encoding** each dataset entry into the hyperspace. This is achieved by providing the data to be encoded to the

encoder methods. The dataset structure to be provided to the encoder methods is a standard vector of vectors (equivalent list of lists in Python). Each sub-vector represents a dataset entry containing all the base data elements that compose it. It is important to mention that all of the encoder methods presented encode each dataset entry independently and deterministically, returning an `HDMatrix` object containing one encoded hypervector per dataset entry provided.

The `encodeWithBundle` method iterates through the provided data, fetching the associated hypervector (stored in the `HDRepresentation` object calling the method) for each base data element read and proceeds to bundling all base hypervectors that belong to the same entry. If binding is the desired operation instead of bundling, in the same context, the `encodeWithXOR` method should be used. As illustrated by the flow chart presented in Figure 3.2, these methods also take a permutation argument, which indicates which permutation to apply to the representation vectors when advancing to the next position of an entry of data. Currently, the options are no permutation or a circular shift permutation. A circular permutation is generally hardware friendly [12] and, as a result, has been the preferred permutation used in recent HDC-based applications. Furthermore, all permutations generally have the same outcome: the generation of a quasi-orthogonal vector to the one that gave origin to it, making a circular shift permutation as useful as any other. Mathematically, each entry hypervector generated by the `encodeWithBundle` method can be described by

$$E(d_1, d_2, ..., d_M) = [\sum_{i=1}^{M}(\prod_{j=1}^{i-1}(\rho)R(d_i))], \tag{3.1}$$

and generated using the `encodeWithXOR` permutation method by

$$E(d_1, d_2, ..., d_M) = \prod_{i=1}^{M}(\prod_{j=1}^{i-1}(\rho)R(d_i)), \tag{3.2}$$

where M is the number of data elements in the dataset entry provided, d_i is the data element at position i, $R(d_i)$ is the hypervector responsible for representing data element d_i (stored in the `HDRepresentation` object), ρ is always the same permutation to be applied to $R(d_i)$, and $E(d_1, d_2, ..., d_M)$ is the resulting entry hypervector.

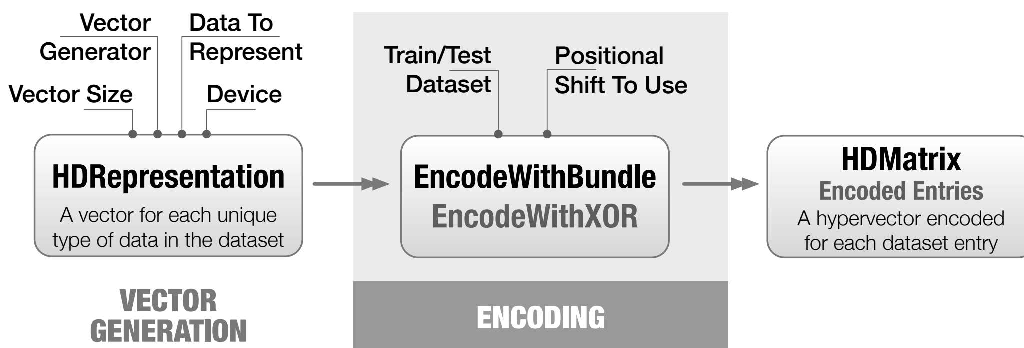


Figure 3.2: Flow chart of the encoding of data using DPHDC permutation-based encoders.

As shown on Figure 3.3, the `encodeWithXOR` method is also overloaded to enable the encoding of

data using positional vectors, like the encoder used by VoiceHD [17] and presented in Section 2.2.5. In this case, a matrix of positional hypervectors, consisting of one vector per each element of a dataset entry, previously generated by the user, needs to be provided to the encoder. Similarly to the permutation based encoders, presented immediately above, the `encodeWithXOR` positional method starts by reading the provided data and fetching the corresponding hypervectors that represent each piece of data in each entry. These vectors are then multiplied with the positional hypervectors, according to their position in the data entry. Finally, all multiplied vectors corresponding with the same entry are bundled together to generate the entry hypervector. This encoding of each entry performed by this encoder method can be described mathematically by

$$E(d_1, P_1, d_2, P_2, ..., d_M, P_M) = [\sum_{i=1}^{M}(R(d_i) * P_i)], \tag{3.3}$$

where, once again, M is the number of data elements in the dataset entry provided, d_i is the data element at position i, $R(d_i)$ is the hypervector responsible for representing data element d_i (stored in the `HDRepresentation` object), P_i is the provided hypervector for encoding elements at position i and $E(d_1, P_1, d_2, P_2, ..., d_M, P_M)$ is the resulting entry hypervector. As can be observed, equation (3.3) is identical to equation (2.11), i.e., the definition of an hyperdimensional hashing table, presented in Section 2.2.6. Despite the fact that the design of the DPHDC framework was constructed with the objective of allowing the easy and performant implementation of HDC-based classifiers, it is important to mention that most other general-use HDC-based algorithms and data-structures can also be implemented using the library, as is the case of the hyperdimensional hash-map. Given the fact that the `encodeWithXOR` method is overloaded, and, as a result, one of two operations can be performed with it, for the remainder of this dissertation, unless otherwise stated, the `encodeWithXOR` permutation method refers to the one that encodes each entry following the definition presented in equation (3.2), while the `encodeWithXOR` positional method refers to the one in which each entry is encoded as defined in (3.3).

3.1.3 Training and Testing

As mentioned previously, all the presented encoder modules return a `HDMatrix` object containing one hypervector per dataset entry provided. As shown in Figure 3.4, in the case of training the model, all these encoded hypervectors (from "Encoded Train Entries" in Figure 3.4) associated with the same label can be bundled together to generate the trained hypervectors using the `reduceToLabelsBundle` method that only requires the labels associated with each entry to be provided. The associated labels should be provided as a vector of strings (a list of strings in the case of the Python DPHDC front-end). The testing/querying methodology is also illustrated in Figure 3.4, where the encoded vectors can be queried against an already existing trained model to estimate each dataset entry's class. This is achieved by using the `queryModel` method, requiring a similarity measurement to be specified, i.e., Hamming distance or cosine similarity. The estimated labels of each dataset entry provided are then returned by this method. Furthermore, the model accuracy illustrated in Figure 3.4, represents the

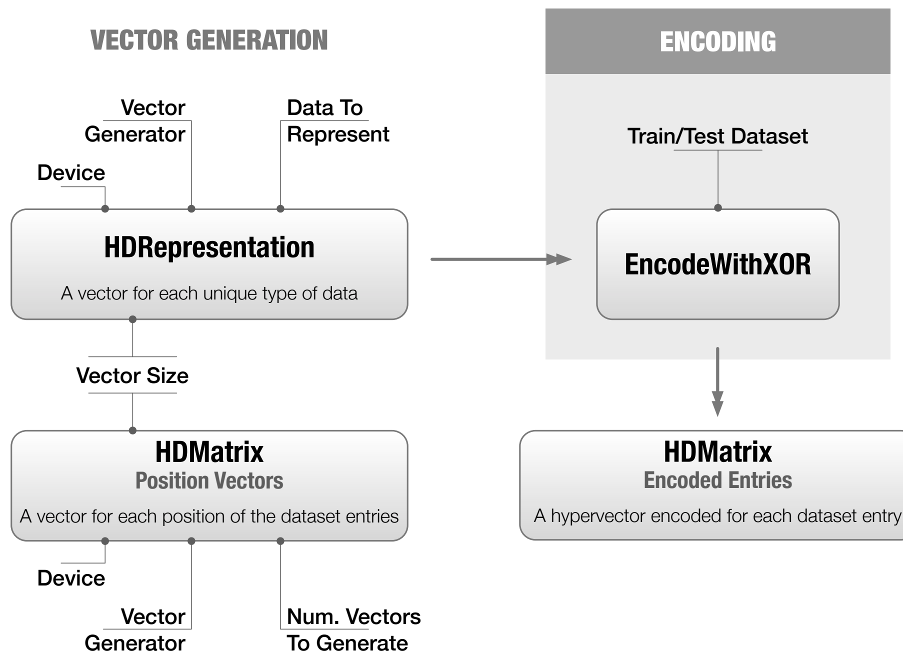


Figure 3.3: Flow chart of the encoding of data using DPHDC positional-based encoder.

success rate of the model and it can be defined as the ratio of correct estimations against the total number of queries performed.

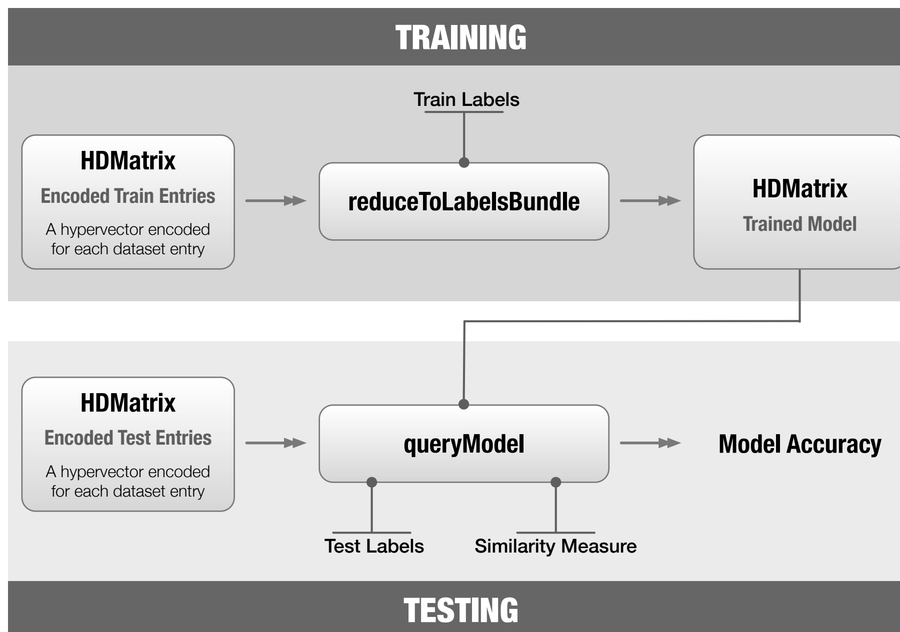


Figure 3.4: Flow chart of the training and querying/testing classification methodologies using DPHDC.

3.1.4 Generality of Encoder Modules and Hypervector Storing

Given the generic nature of the three encoder modules provided by the proposed framework, combining them makes it possible to implement a wide range of encoders and functionalities. As illustrated by Figure 3.5, an example of this capability can be evidenced when implementing an N-gram-based

encoder. This is achieved by first generating an `HDRepresentation` object that represents the basic data that composes the desired N-grams. A vector of vectors containing all the N-grams desired should also be generated. By providing this vector of vectors to the `encodeWithXOR` permutation method called using the `HDRepresentation` object generated previously, an `HDMatrix` (that can be converted into an `HDRepresentation`) representing all the desired N-gram hypervectors is generated and can be used with all the available encoder modules.

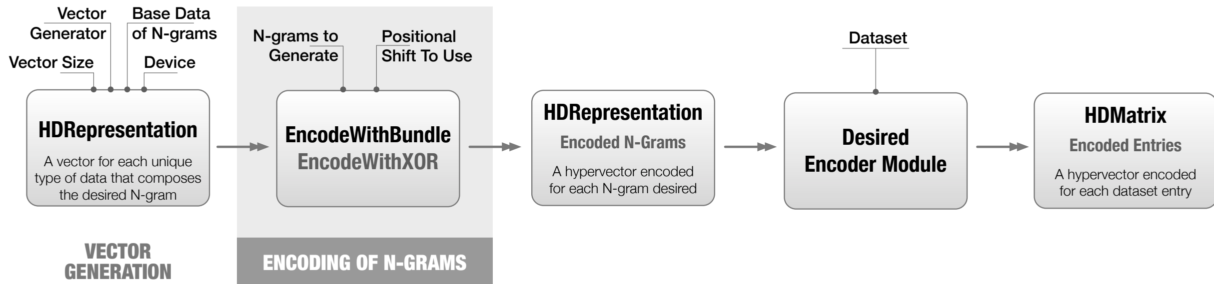


Figure 3.5: Flow chart of a N-gram based encoder using DPHDC.

It is also important to note that any `HDMatrix` or `HDRepresentation` object can be stored in a binary file for optimal size utilization. This not only allows for training and querying operations to happen on different systems, but it also allows for a model to be trained once on one device and queried many times and in many situations on multiple distinct devices. The storing of hypervectors can be achieved by calling method `storeMatrix` if dealing with an `HDMatrix` object or method `storeRepresentation` if dealing with an `HDRepresentation` object. The resulting files have different extensions, namely, `.dphdcm` when an `HDMatrix` object is stored and `.dphdcr` when an `HDRepresentation` object is stored. Such a distinction is necessary since `HDRepresentation` objects also need to store the data element that each hypervector is supposed to represent. It is then possible to read stored hypervectors by creating a new object of type `HDMatrix` or `HDRepresentation` and providing the appropriate stored file to the constructor method.

While designing the library, modularity was also taken into account. Even though DPHDC currently includes all functionalities to handle most HDC classification needs, both `HDMatrix` and `HDRepresentation` are intuitive abstractions, making it easy for users to add new methods, such as new types vector generators, new encoders and/or new similarity measurements.

3.1.5 An Application Example

To better understand the DPHDC library's workflow, this Section presents a short overview of how the VoiceHD [17] application (described in Section 2.2.5) was implemented and is presented in Listing 3.1. Listing 3.1 is written in the C++ programming language. As mentioned previously, the Python version of the library is almost identical to the C++ variant. As a result, the same application implemented using Python (presented in Appendix A.1) follows the same steps.

The first step to develop any application using DPHDC consists in including the library header (line 1

of Listing 3.1), necessary for the application to recognize and use DPHDC classes and methods. As can be observed in Figure 3.6, the next step, in this example, involves generating a level hypervector-based representation for the 20 frequency sub-ranges (lines 4 through 7 of Listing 3.1) and a matrix with 617 random positional hypervectors, one for each frequency bin (line 9 of Listing 3.1).

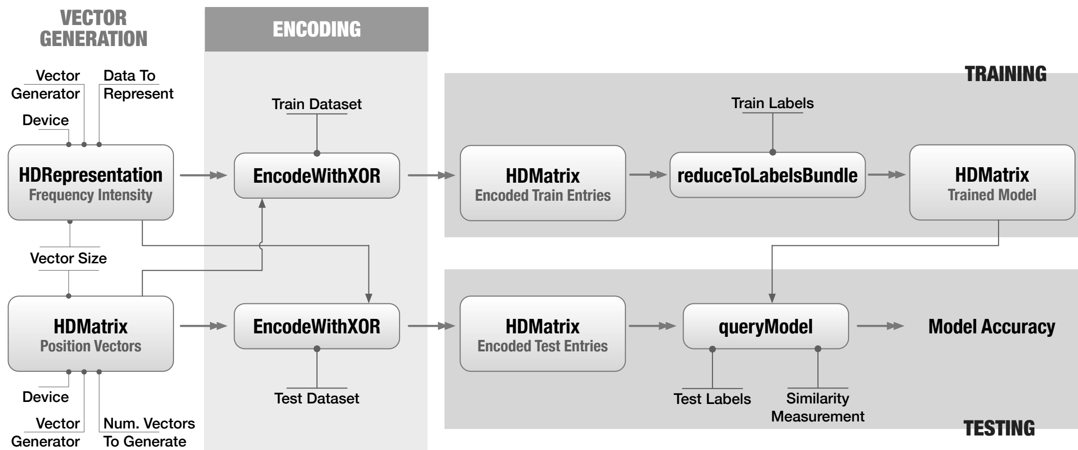


Figure 3.6: Flow chart of the implementation of the VoiceHD application using DPHDC.

Listing 3.1: VoiceHD [17] speech recognition application implemented using DPHDC.

```
1  #include <dphdc.hpp>
2  using namespace std;
3  int main{
4      int v_size = 10000;
5      //Generating representation
6      vector ints_represent = {−10, −9, ... , 9, 10};
7      dphdc::HDRepresentation<int> freq_inten_represent(v_size, full_level, device, ↩
           ints_represent);
8      //Generating position vectors
9      dphdc::HDMatrix position_vectors(v_size, 617, random, device);
10
11     vector<vector<int>> train_data = readTrainData(dataset_path);
12     dphdc::HDMatrix encoded_matrix = freq_inten_represent.encodeWithXOR(train_data, ↩
           position_vectors);
13
14     vector<string> train_labels = readTrainLabels(dataset_path);
15     dphdc::HDMatrix trained_model = encoded_matrix.reduceToLabelsBundle(train_labels);
16
17     vector<vector<int>> test_data = readTestData(dataset_path);
18     dphdc::HDMatrix encoded_test_entries = freq_inten_represent.encodeWithXOR(test_data, ↩
           position_vectors);
19     vector<string> test_labels = readTestLabels(dataset_path);
20     vector<string> test_estimated_labels = trained_model.queryModel(encoded_test_entries, ↩
           hamming_distance);
21     float accuracy = getAccuracy(test_estimated_labels, test_labels);
22     return 0;
```

```
23 }
```

With both constructed, the training data can be encoded using the `encodeWithXOR` positional method (line 12 of Listing 3.1). It is important to note that the functions that read the datasets and labels and the organization of data into DPHDC recognizable formats (vector of vectors for data and vector of strings for labels (or lists if using the Python front-end) need to be developed by the user (line 11 of Listing 3.1).

As indicated by Figure 3.6, using the labels provided with the ISOLET dataset (line 14 of Listing 3.1), it is possible to derive the trained model by bundling all vectors with the same label together (line 15 of Listing 3.1).

Finally, it is possible to query/test the model (lines 18 through 21 of Listing 3.1) with all encoded test entries (encoded in lines 17 and 18 of Listing 3.1) to get the estimated label for each. Function `getAccuracy` presented in line 21 of Listing 3.1 compares the estimated labels/classes with the actual labels/classes of each dataset entry and returns the percentage of correct estimations, i.e., the model accuracy.

It is also important to mention that when creating new objects of type `HDMatrix` or `HDRepresentation` (as illustrated in lines 7 and 9 of Listing 3.1), two DPHDC defined enumerators need to be provided as arguments: one for which vector generator to use and another for the accelerator (`device`) that should handle all operations related with the created object. In the case of the vectors to be generated, the available options for the enumerator are, as described in Section 3.1.1:

- `all_false` (all elements will be zero/minus one) or `all_true` (all elements will be ones) for generation of constant hypervectors;
- `random` for generation of random hypervectors;
- `half_level` for generation of half-level hypervectors;
- `full_level` for generation of full-level hypervectors;
- `circular` for generation of circular hypervectors.

For device selection, the enumerator options available are: `cpu`, `gpu`, `cuda` (the `gpu` selector will default to cuda if no other GPU is available), `fpga` and `fpga_emulator` (an emulator for FPGA that runs on the CPU for guaranteeing that the developed code works as intended before starting the lengthy FPGA compilation process). If none of the presented selectors allow the targeting of the intended device, then a SYCL queue, associated with the desired device, can be provided instead of the enumerator, ensuring that all devices compatible with SYCL are also compatible with the proposed framework.

3.2 Development and Implementation

The description of the high-level design of the proposed library, presented so far in Section 3.1, is crucial to get a complete view of the feature set and capabilities of the proposed DPHDC framework, as well as provide an understanding as to how an application can be developed using the DPHDC.

It is important to follow such a description with details behind of the development and implementation of the proposed library in order to better understand certain design decisions, what algorithms were developed and how their execution is optimized for performing HDC-based operations across heterogeneous devices. As a result, these aspects of the proposed work are presented herein. Furthermore, as mentioned previously, the library was developed and implemented using the SYCL standard, more specifically, the SYCL 2020 specification (revision 5) [29] which, as explained in Section 2.5, is based on modern C++ (versions 17 and higher). Consequently, all code listings presented for the remainder of this thesis, unless otherwise stated, are written in modern C++ programming language using the SYCL standard.

3.2.1 Data Management

Since `HDMatrix` and `HDRepresentation` classes are the main data structures of the DPHDC library, it is important to discuss how hypervectors are stored within these classes and how their offloading to accelerators is handled. `HDMatrix` and `HDRepresentation` classes use a SYCL boolean buffer with two dimensions to represent a collection of hypervectors, which allows for the easy migration of data, as explained in Section 2.5.2. The first dimension indicates the number of vectors in the matrix, and the second the hypervector size. By using a boolean buffer, it is possible to minimize the memory space occupied by each hypervector, since just two values are necessary to represent either binary or bipolar values. Furthermore, the buffer used has no association with any host data, which leads to minimal data transfers between host and accelerator by allowing hypervector data to only exist on the accelerator device, as explained in Section 2.5.2. The access to the data contained within a buffer, through the use accessors, also facilitates the management of data dependencies and helps avoid data race conditions, as also explained in Section 2.5.2. As a result, the proposed framework minimizes memory requirements while avoiding unnecessary memory transfers and copies, crucial to avoid memory related performance bottlenecks.

Another important consideration taken during the development of the proposed framework is the storage of hypervectors in a matrix format. As mentioned previously, given the highly parallel nature of HDC operations, DPHDC was developed with the aim of maximizing parallel execution while remaining easy to use and versatile. While developing "conventional" single threaded algorithms, the individual and separate storage of hypervectors might be considered an intuitive approach. Such a design would lead to suboptimal exploration of the parallelism capabilities of all modern hardware architectures, since, as an example, only by providing the encoder modules with all the data to encode simultaneously, it is possible to exploit as much parallelism as possible from all target architectures by using data-parallel SYCL kernels. The same logic applies for all other developed methods that utilize the accelerator device, i.e., it is much more efficient in a parallel problem, like HDC operations, to launch a smaller amount of bigger, higher dimensional kernels, where more parallelism is exploited on the accelerator, than to launch a higher amount of smaller, lower dimensional kernels, where a lot of the parallelism is offloaded to the host. As a result, all methods that utilize accelerator capabilities (some vector generator methods,

encoder methods, the reduceToLabelsBundle method and the query method) are implemented using data-parallel SYCL kernels, as described in Sections 3.2.2, 3.2.3 and 3.2.4. Furthermore, the storage of vectors in a continuous memory space also improves parallel execution by exploiting memory locality.

Accelerator Selection and Targeting

As mentioned in Section 3.1.5, each HDMatrix and HDRepresentation object is associated with an accelerator. Such is achieved by storing a SYCL queue, associated with the desired accelerator, inside each class implemented in the proposed library. This queue is a private variable named associated_queue in both classes, since, as illustrated in Figure 3.1 and explained in Section 3.1, the HDRepresentation class inherits all variables and methods from the HDMatrix class. As expected, the associated_queue is used to launch kernels by any method that takes advantage of an accelerator. Since the execution always happens on the accelerator associated with object calling the method, it is recommended that all other hypervectors necessary to execute a certain method are associated with the same device, in order to avoid excessive memory transfers and, consequently, performance bottlenecks. As mentioned in Section 2.5.1, there is no problem in having several SYCL queues associated with the same device since kernels are always guaranteed to execute in the same order they were launched in.

When constructing a collection of hypervectors, as explained in Section 3.1.5, either a device selector enumerator is provided, which allows for the initialization of the associated_queue variable, or a queue, which is copied to associated_queue, can also be passed as an argument. The accelerator associated with any DPHDC object can also be easily changed, after construction, by calling the setAssociatedQueue method. This highlights the versatility and portability of the proposed library, as accelerators can be changed on the fly by just using one simple method, although the migration of hypervectors might entail a performance penalty.

3.2.2 Vector Generators

Most of the vector generators currently offered by DPHDC are implemented on the host device and then copied to the SYCL buffer so that they can be used on any device. Such a decision stems from the fact that functions used to generate pseudo-random values are usually device-specific, thus offloading the vector generators methods to the accelerator will have a negative impact on the portability of the proposed framework. Furthermore, generating basic hypervectors on the host is usually an efficient process that would not benefit significantly if performed on an accelerator. It is also important to mention that host functions that generate pseudo-random values usually assure that the probability of each element of a vector being false or true is independent. This property is vital in HDC since for two randomly generated hypervectors to be quasi-orthogonal each value on the vector that is randomly generated must have an equal and independent chance of being either 0 (false) or 1 (true).

The only vector generator methods implemented directly on the accelerator are the constant hypervectors generators. Since there is no need to use a pseudo-random function, the assignment of the same value to all elements on the host to then copy to the accelerator would be counterproductive. If

the user chooses to generate constant hypervectors, through the use of a DPHDC defined enumerator at the construction of a `HDMatrix` or `HDRepresentation` object, then the helper private method `constantVectorGenerator`, presented in Listing 3.2, is called to ensure that the hypervectors are filled with the desired value, either one (`true`) or zero/minus one (`false`). As can be observed in lines 1 and 2, method `constantVectorGenerator` uses the `associated_queue` variable to launch a data-parallel kernel (line 4) that assigns `value` (either true of false, passed by the constructor) to all elements of the `vector_buff` SYCL boolean buffer variable (the hypervector matrix associated with the object calling this method) (line 5). Since the associated hypervectors are stored in a SYCL buffer, it is necessary the creation of an accessor variable, in line 3, using the write-only specifier, to ensure that no kernel that depends on this `vector_buff` is launched before the generation of the vectors is concluded, as explained in Section 2.5.2.

Listing 3.2: Generation of constant vectors.

```
1 void HDMatrix::constantVectorGenerator(bool value) {
2     this->associated_queue.submit([&](cl::sycl::handler &h) {
3         cl::sycl::accessor acc(this->vectors_buff, h, cl::sycl::write_only);
4         h.parallel_for(this->vectors_buff.get_range(), [=](cl::sycl::id<2> i) {
5             acc[i[0]][i[1]] = value;
6         });
7     });
8 }
```

The remaining vector generation possibilities offered by DPHDC are all randomly-based and, as a consequence, are generated on the host. When the desired outcome is that all hypervectors generated have each value independently and randomly assigned, then the `randomVectorGenerator` method, presented in Listing 3.3, is used. As can be observed in lines 2 through 4, modern C++ abstractions are used to guarantee uniform and independent generation of each one of the hypervectors values, which is crucial for classification applications based on HDC, as explained in Section 2.2.3. It is then necessary to obtain the number of vectors and vector size to use for the generation (saved during the constructor) by reading the range associated with the buffer that stores the hypervectors (line 6). This is followed by the creation of the host hypervector matrix itself (line 7), which, after assignment of the random values (lines 9 through 11), is copied to the buffer (`vectors_buff`) through the use of the `copyBoolVector` method, presented in Listing 3.4.

Listing 3.3: Generation of random vectors.

```
1 void HDMatrix::randomVectorGenerator() {
2     std::random_device dev;
3     std::mt19937 rng(dev());
4     std::bernoulli_distribution dist;
5
6     cl::sycl::range<2> buffer_range = this->vectors_buff.get_range();
```

```
7       std::unique_ptr<bool[]> vector_on_host(new bool[buffer_range[0] * buffer_range[1]]);
8
9       for (unsigned int i = 0; i < buffer_range[0] * buffer_range[1]; i++) {
10          vector_on_host.get()[i] = dist(rng);
11      }
12
13      this->copyBoolVector(vector_on_host.get());
14  }
```

As can be observed in Listing 3.4, this auxiliary method starts by creating a temporary buffer for the vector on the host (line 3) which is then accessed, through the created accessor objects (lines 5 and 6), by the data-parallel kernel (lines 7 through 9) responsible for copying the data from the host hypervector to the hypervector buffer associated with the current object. It is relevant to question why the hypervector generated on the host is copied to the `vectors_buff` SYCL buffer when a buffer containing this data is already created in line 3 to perform this exact operation (`vector_on_host_buff`). The answer lies in the fact that after the execution of the `copyBoolVector` method, the `vector_on_host_buff` variable gets out of scope, which leads to its destruction. The direct assignment of `vectors_buff` variable to the `vector_on_host_buff` variable would also not be possible, since `vector_on_host_buff` is associated with host data, more specifically, with the `vector_on_host` variable, created in line 7 of Listing 3.3, which also gets out of scope when the `randomVectorGenerator` function ends, leading to the deallocation of the memory reserved for this variable. As a result, when the `vectors_buff` buffer is inevitably destroyed, it will try to write the data it is holding back to the `vector_on_host` address, which has been deallocated, leading to a segmentation fault error.

Listing 3.4: Copy of vectors generated on the host to the class hypervectors buffer variable.

```
1   void HDMatrix::copyBoolVector(const bool *vector_to_copy) {
2       {
3           cl::sycl::buffer<bool, 2> vector_on_host_buff(vector_to_copy, this->vectors_buff.↩
                get_range());
4           this->associated_queue.submit([&](cl::sycl::handler &h) {
5               cl::sycl::accessor acc_vector_device(this->vectors_buff, h, cl::sycl::write_only↩
                    );
6               cl::sycl::accessor acc_vector_host(vector_on_host_buff, h, cl::sycl::read_only);
7               h.parallel_for(this->vectors_buff.get_range(), [=](cl::sycl::id<2> i) {
8                   acc_vector_device[i[0]][i[1]] = acc_vector_host[i[0]][i[1]];
9               });
10          });
11      }
12  }
```

All remaining currently available hypervector generators offered by DPHDC generate the first vector randomly, using the same modern DPHDC approaches used in the `randomVectorGenerator` method, and are followed by simple algorithms, specific for each case, that ensure the sequential flipping of bits in order to ensure that the remaining hypervectors are generated as described in Section 3.1.1. After

the host creation of the desired matrix, the vector on the host is also copied to the `vectors_buff` buffer using the `copyBoolVector` method, as illustrated in line 13 of Listing 3.3.

3.2.3 Encoder Methods

After the desired generation of hypervectors is performed, the next step usually consists of providing a dataset to an encoder module in order to start mapping data into hyperdimensional space. As mentioned in Section 3.1, the creation of the `HDRepresentation` class, necessary for the use of the encoder methods defined in Section 3.1.2, was a solution to the problem of associating base hypervectors with the data they represent. This association is of crucial importance, since encoder methods manage to encode each dataset entry by applying the desired HDC arithmetic operations to the base hypervectors of each data element that is read from said entry.

Hypervector and Data Element Association

The representation of any data type by a hypervector, as explained in Section 3.1, is possible thanks to the fact that the `HDRepresentation` class contains a template argument, `TypeOfDataToRepresent`, which, as the name implies, is the type of data that each hypervector will represent. As mentioned previously, this can be any normal C++ data type, i.e., integers, floats, etc., or even a user defined data type, as long as it has a comparison method, necessary for the association between data elements and hypervectors. To establish this relation, each `HDRepresentation` object contains, within himself, a hast-table (with variable name `data_translation`) that maps the index of each hypervector to the data element it represents. The creation of a new `HDRepresentation` implies also that a vector (or list in case of the Python DPHDC front-end) containing all data elements to represent is provided. Such is necessary in order for the automatic generation of the same amount of hypervectors as the number of data elements provided and for the mapping of each data element to the corresponding hypervector index, using the class hash-map variable, as can be observed in Listing 3.5, where the auxiliary `generateHashTable` function is presented. As can be observed, the hash-table used consists in a standard C++ `unordered_map` (line 2). This map then associates each data element to represent (line 3), to the corresponding index of the hypervector generated plus one (lines 4 through 6). The reason for adding one to the index shall become apparent once the hash-map is used to fetch the hypervector indexes from provided data.

Listing 3.5: Mapping of data elements to the corresponding hypervector index.

```
1 template<class TypeOfDataToRepresent>
2 void generateHashTable(std::unordered_map<TypeOfDataToRepresent, int> &map,
3                        const std::vector<TypeOfDataToRepresent> &elements_to_represent) {
4     for (unsigned int i = 1; i <= elements_to_represent.size(); i++) {
5         map[elements_to_represent[i - 1]] = i;
6     }
7 }
```

When one of the three encoder modules starts, the first step consists of translating the data provided into the respective hypervector indexes such that the encoder operations can be executed. Such an operation, implemented by `convertData` method (presented in Listing 3.6), returns an array where the data elements provided are replaced with corresponding hypervector index (`converted_data`) and the biggest size of all the dataset entries provided (`max_entry_size`) (line 2). This latter value is obtained using a modern C++ approach, as can be seen in lines 3 through 5. The creation, allocation and initialization (with value -1) of the `converted_data` array follows (lines 7 through 11). For the conversion of data, it is necessary to mention that, whenever a standard C++ `unordered_map` is queried with a data element not stored inside it, it returns zero. Since zero is a valid hypervector index, this could lead to unrecognized data elements being mapped to the first hypervector stored in the buffer. This is the reason why plus one is summed to every index while filling the hash-map, it allows for unrecognized data elements to be stored in the array as -1. Such is achieved by subtracting minus one for every index plus one obtained from the hash map (line 16) while converting the dataset provided (lines 13 through 21). This feature is a safety net in case the standard way of checking if the hash-map contains a certain element (line 15) fails (which was a frequent occurrence during testing while using the Intel DPC++ compiler). A double redundancy in the conversion of data elements to hypervector indexes is crucial to ensure that such a process is accurate and, consequently, that the encoder output is also accurate.

Listing 3.6: Conversion of the provided dataset into hypervector indexes.

```
1  template<class TypeOfDataToRepresent>
2  std::unique_ptr<int[]> HDRepresentation<TypeOfDataToRepresent>::convertData(const std::↩
       vector<std::vector<TypeOfDataToRepresent>> &data, int &max_entry_size) {
3      max_entry_size = std::max_element(data.begin(), data.end(), [](const std::vector<↩
           TypeOfDataToRepresent> &lhs, const std::vector<TypeOfDataToRepresent> &rhs) -> bool ↩
           {
4          return lhs.size() < rhs.size();
5      })->size();
6
7      std::unique_ptr<int[]> converted_data(new int[max_entry_size * data.size()]);
8
9      for (size_t i = 0; i < max_entry_size * data.size(); i++) {
10         converted_data.get()[i] = -1;
11     }
12
13     for (size_t i = 0; i < data.size(); i++) {
14         for (size_t j = 0; j < data[i].size(); j++) {
15             if (this->data_translation.count(data[i][j])) {
16                 converted_data.get()[i * max_entry_size + j] = this->data_translation[data[i↩
                       ][j]] - 1;
17             } else {
18                 converted_data.get()[i * max_entry_size + j] = -1;
19             }
20         }
```

```
21      }
22
23      return converted_data;
24  }
```

It is important to also mention that this conversion of dataset entries to hypervector, by using a hash-map, is performed on the host due two main reasons. The first is that the creation of the hash map and conversion of data by checking the hash-table on the host is usually a process that manages to exploit most capabilities of host architecture. Secondly, the efficient execution of this process on an accelerator requires the use of architecture specific optimizations on the kernels developed, which would most likely make the proposed library portability goals becoming impossible to achieve.

Binding Permutation Encoder Method

After the provided dataset is converted into the hypervector indexes, with unknown data mapped as -1, it is possible to start the encoder modules themselves. The development and implementation of the `encodeWithXOR` permutation method, presented in Listing 3.7, is the most straightforward to explain, since no sum operation is performed at any stage during this encoder execution. As explained up until now, the encoder module starts by converting the provided data (lines 3 and 4). Next, the encoded entries `HDMatrix` object, returned by the method, is constructed (initialized with all elements containing value `false`) (line 6), followed by the creation of the buffer to hold the converted indexes (line 9). In line 10, given the fact that the converted indexes array will not be modified by any device code (the information will just be read), it is possible to optimize the eventual associated buffer destruction by instructing the SYCL backend that there is no need to write the buffer data back to host, using the `set_write_back` SYCL method. This optimization considerably reduces the transfers between host and accelerator performed, especially when dealing with big datasets.

All HDC based operations are, as explained in Section 2.2.4, elementwise operations whose parallelism can be exploited using data-parallel kernels. This is not the case with permutations, as they are not element-wise operations. Given the fact both `encodeWithBundle` and `encodeWithXOR` permutation methods have the possibility of using permutations during the mapping of data into hyperdimensional space, it is important to explore how the permutations offered by DPHDC (circular shifts) are implemented. Before tackling the implemented solution itself, it is relevant to mention that one promising path to achieve this goal would be to use vendor specific functions that already efficiently implement these shifting operations, although such would be incompatible with the portability goals of the library, as mentioned previously. As a result, to perform circular shifting operations during the encoding process while taking full advantage of the parallel resources of the device being targeted, a specialized algorithm was developed and is presented in lines 12 through 39 of Listing 3.7 and in Listing 3.8.

The first step of this algorithm consists in the creation two copies of the base hypervectors buffer stored in the `HDRepresentation` object calling this method (lines 12 and 13 of Listing 3.7). This double copy of the base hypervectors is necessary for two reasons: the first copy guarantees the original base hypervectors are never changed in order for them to be used again, with other methods, if necessary,

while the second copy is necessary for the developed kernels responsible for the shifting to work, as will be explained shortly. The `copyBuffer` used in lines 12 and 13 is another auxiliary function part of the DPHDC library responsible for returning an exact copy of the buffer provided to it using a simple copy data-parallel kernel.

The main logic of the developed algorithm is expressed through the for loop presented in lines 15 through 39 of Listing 3.7. This loop goes iterates through every “column” of the provided dataset, i.e., in the first iteration all the first elements of all dataset entries are processed, in the second iteration all the second elements of all dataset items are processed, etc. The processing of these elements consists in applying the desired HDC operation to the base hypervector representing the element being processed and saving the result into the `encoded_vectors_matrix` variable, using a data-parallel kernel in order to exploit the parallelism capabilities of the target hardware. In the case of the `encodeWithXOR` permutation method, this “column” processing data-parallel kernel is presented in lines 16 through 27 of Listing 3.7, where it can be observed in line 24 that every element of every base hypervector fetched using the provided converted data into indexes is continuously multiplied with the corresponding encoded vector (one for each dataset entry). More specifically, in lines 20 through 27 of Listing 3.7 variable `i` indicates the dataset entry being processed, variable `j` indicates the dataset entry position being processed and variable `k` indicates the hypervector element being multiplied. It is important to mention that the `if` statement present in line 23 assures that a base hypervector is only multiplied into the encoded vector matrix to be returned if the provided data element has an association with a base hypervector stored in the `HDRepresentation` object calling this method, i.e., the index in the converted data indexes array is valid (bigger or equal to zero).

After each column of the dataset is processed, the desired permutation, provided through enumerator `permutation_to_use`, is executed, as illustrated by lines 29 through 38 of Listing 3.7. It is important to mention again that, aside from circular permutations, the `permutation_to_use` enumerator can also have the value `no_permutation`, which, as the name implies, guarantees that no permutations are performed between the processing of each dataset column (lines 36 and 37). This feature increases the versatility and general-use potential of the library while avoiding the need to implement extra, unnecessary methods.

Listing 3.7: Implementation of the `encodeWithXOR` permutation method.

```
1 template<class TypeOfDataToRepresent>
2 HDMatrix HDRepresentation<TypeOfDataToRepresent>::encodeWithXOR(const std::vector<std::↩
      vector<TypeOfDataToRepresent>> &data, permutation::permutation permutation_to_use) {
3     int max_element_size;
4     std::unique_ptr<int[]> converted_data = this->convertData(data, max_element_size);
5
6     HDMatrix encoded_vectors_matrix(this->vectors_buff.get_range()[1], data.size(), dphdc::↩
          vectors_generator::all_false, this->associated_queue);
7
8     {
9         cl::sycl::buffer<int, 2> buff_data(converted_data.get(), cl::sycl::range<2>(data.↩
```

```
           size(), max_element_size));
10         buff_data.set_write_back(false);
11 
12         cl::sycl::buffer<bool, 2> buff_copy_of_representation = this->copyBuffer(this->↩
           vectors_buff);
13         cl::sycl::buffer<bool, 2> buff_copy_of_representation_duplicate = this->copyBuffer(↩
           this->vectors_buff);
14 
15         for (unsigned int j = 0; j < max_element_size; j++) {
16             this->associated_queue.submit([&](cl::sycl::handler &h) {
17                 cl::sycl::accessor acc_encoded_vectors(encoded_vectors_matrix.vectors_buff, ↩
                   h, cl::sycl::read_write);
18                 cl::sycl::accessor acc_representation(buff_copy_of_representation, h, cl::↩
                   sycl::read_only);
19                 cl::sycl::accessor acc_data(buff_data, h, cl::sycl::read_only);
20                 h.parallel_for(cl::sycl::range<2>(data.size(), this->vectors_buff.get_range↩
                   ()[1]), [=](cl::sycl::id<2> local_range) {
21                                 size_t i = local_range[0];
22                                 size_t k = local_range[1];
23                                 if (acc_data[i][j] >= 0) {
24                                     acc_encoded_vectors[i][k] ^= acc_representation[↩
                                           acc_data[i][j]][k];
25                                 }
26                             });
27             });
28 
29             switch (permutation_to_use) {
30                 case permutation::shift_right:
31                     this->shiftRight(buff_copy_of_representation, ↩
                           buff_copy_of_representation_duplicate);
32                     break;
33                 case permutation::shift_left:
34                     this->shiftLeft(buff_copy_of_representation, ↩
                           buff_copy_of_representation_duplicate);
35                     break;
36                 case permutation::no_permutation:
37                     break;
38             }
39         }
40     }
41 
42     return encoded_vectors_matrix;
43 }
```

The development and implementation of the circular shift right method (`shiftRight`), presented in Listing 3.8, consists in a data-parallel kernel that performs a shifted copy from the duplicate buffer provided (lines 6 through 14). The duplicate buffer is then updated with the shifted values in line 17, using the auxiliary DPHDC method `copyBuffer`, which can also be used to copy, using a data-parallel kernel,

all values from the first argument provided (a SYCL buffer) to it into the second argument provided (a SYCL buffer of the same size, as expected). The `shiftLeft` method is implemented almost identically to the `shiftRight` presented so far, with the only difference being in the indexes used to perform the shifted copy.

Listing 3.8: Circular shift right accelerator code.

```
void HDMatrix::shiftRight(cl::sycl::buffer<bool, 2> &buffer_to_shift, cl::sycl::buffer<bool, 2> &duplicate_of_buffer) {
    size_t vector_size = buffer_to_shift.get_range()[1];
    this->associated_queue.submit([&](cl::sycl::handler &h) {
        cl::sycl::accessor acc_buffer_shift(buffer_to_shift, h, cl::sycl::write_only);
        cl::sycl::accessor acc_duplicate(duplicate_of_buffer, h, cl::sycl::read_only);
        h.parallel_for(this->vectors_buff.get_range(), [=](cl::sycl::id<2> local_range) {
            size_t i = local_range[0];
            size_t k = local_range[1];
            if (k == 0) {
                acc_buffer_shift[i][k] = acc_duplicate[i][vector_size - 1];
            } else {
                acc_buffer_shift[i][k] = acc_duplicate[i][k - 1];
            }
        });
    });

    this->copyBuffer(buffer_to_shift, duplicate_of_buffer);
}
```

Bundling Permutation Encoder Method

The `encodeWithBundle` encoder method, presented in Listing 3.9, by also being permutation based follows the same algorithm described so far for the `encodeWithXOR` permutation based method, with a few adaptations to perform the intended encoding. As expected, the main difference resides in the data-parallel kernel developed to encode each column of the dataset. To explain this kernel, it is first necessary to introduce accumulator variables. Accumulator variables, in the context of the DPHDC library, are two-dimensional SYCL buffers that, unlike hypervector buffers, hold signed 16-bit integers and not booleans. These accumulators are necessary when the bundling HDC operation is used, since it is necessary to first add all hypervectors (which cannot be represented using booleans) before thresholding back to boolean values according to a majority rule. As the name implies, these types of variables accumulate all hypervectors into themselves before generating the resulting encoded hypervectors through thresholding. As a result, in the context of the encoder modules, they have the same dimensions as the encoded hypervectors of the `HDMatrix` object returned by the method. In line 9 of Listing 3.9, the accumulators used within the `encodeWithBundle` method is created and initialized using the auxiliary DPHDC method `generateInitializeAccumulators`, which returns several accumulators, initialized with all ze-

ros, with size equal to the range provided as an argument (in this case, `data.size()` is the number of dataset entries and `this->vectors_buff.get_range()[1]` is the hypervector size).

As mentioned previously, the main difference between the `encodeWithXOR` permutation based method and the `encodeWithBubdle` method, is the main kernel developed, i.e., lines 20 through 26 of Listing 3.7 and lines 21 through 31 of Listing 3.9, respectively. As can be observed, in the case of the `encodeWithBundle` method, after fetching the base hypervector associated with the data element that is being processed, an evaluation of each one of its elements is performed. If the value is `true`, then one is added to the corresponding accumulator position. Otherwise, one is subtracted from said position. This partial bundling operation is valid for bipolar HDC models, but, since bipolar and binary models are mathematically equivalent [37], it is also valid for binary models, hence why DPHDC is able to implement both models by just using one data type to represent hypervectors. After all additions are performed, intercalated by the desired permutations (lines 16 through 44 of Listing 3.9), it is necessary to normalize the accumulators back into hypervectors. Such is done in line 46 of Listing 3.9, by use of the `normalizeAccumulator` method, presented in Listing 3.10.

Listing 3.9: Implementation of the `encodeWithBundle` method.

```
1  template<class TypeOfDataToRepresent>
2  HDMatrix HDRepresentation<TypeOfDataToRepresent>::encodeWithBundle(const std::vector<std::vector<TypeOfDataToRepresent>> &data, permutation::permutation permutation_to_use) {
3      int max_element_size;
4      std::unique_ptr<int[]> converted_data = this->convertData(data, max_element_size);
5
6      HDMatrix encoded_vectors_matrix(this->vectors_buff.get_range()[1], data.size(), dphdc::vectors_generator::none, this->associated_queue);
7
8      {
9          cl::sycl::buffer<short int, 2> buff_accumulators = this->generateInitializeAccumulators(cl::sycl::range<2>(data.size(), this->vectors_buff.get_range()[1]));
10
11         cl::sycl::buffer<bool, 2> buff_copy_of_representation = this->copyBuffer(this->vectors_buff);
12         cl::sycl::buffer<bool, 2> buff_copy_of_representation_duplicate = this->copyBuffer(this->vectors_buff);
13         cl::sycl::buffer<int, 2> buff_data(converted_data.get(), cl::sycl::range<2>(data.size(), max_element_size));
14         buff_data.set_write_back(false);
15
16         for (unsigned int j = 0; j < max_element_size; j++) {
17             this->associated_queue.submit([&](cl::sycl::handler &h) {
18                 cl::sycl::accessor acc_accumulators(buff_accumulators, h, cl::sycl::read_write);
19                 cl::sycl::accessor acc_representation(buff_copy_of_representation, h, cl::sycl::read_only);
```

```
20                  cl::sycl::accessor acc_data(buff_data, h, cl::sycl::read_only);
21                  h.parallel_for(cl::sycl::range<2>(data.size(), this->vectors_buff.get_range↩
                        ()[1]), [=](cl::sycl::id<2> local_range) {
22                                  size_t i = local_range[0];
23                                  size_t k = local_range[1];
24                                  if (acc_data[i][j] >= 0) {
25                                      if (acc_representation[acc_data[i][j]][k]) {
26                                          acc_accumulators[i][k] += 1;
27                                      } else {
28                                          acc_accumulators[i][k] -= 1;
29                                      }
30                                  }
31                              });
32              });
33
34              switch (permutation_to_use) {
35                  case permutation::shift_right:
36                      this->shiftRight(buff_copy_of_representation, ↩
                            buff_copy_of_representation_duplicate);
37                      break;
38                  case permutation::shift_left:
39                      this->shiftLeft(buff_copy_of_representation, ↩
                            buff_copy_of_representation_duplicate);
40                      break;
41                  case permutation::no_permutation:
42                      break;
43              }
44          }
45
46          this->normalizeAccumulator(buff_accumulators, encoded_vectors_matrix.vectors_buff);
47      }
48
49      return encoded_vectors_matrix;
50  }
```

It is important to mention that the reason for using the bipolar definition of the bundling operation is to avoid keeping track of the number of vectors added before thresholding, a fact that facilitates tremendously the thresholding process. As illustrated by equation 2.3, in the present case it is only necessary to compare all the elements of the accumulator with zero (line 8 of Listing 3.10). If the value is bigger than zero, then the corresponding hypervector position will contain `true` (line 9 of Listing 3.10). Otherwise, its value will be `false` (line 11 of Listing 3.10).

From line 8 of Listing 3.10 it is also possible to infer another decision that was taken during the development of the library: how DPHDC handles bundling draws. As explained in Section 2.2.4, several approaches exist to deal with bundling draws. It is recommended that the user provides an odd number of vectors to be added, in order to ensure that a bundling draw never happens. This can be achieved, in this case, by adding a random known data element to the end of every dataset entry that contains

an even number of elements. If it is not the intention of the user to provide an extra hypervector per each even bundling operation being performed, then, in every bundling operation performed by the DPHDC library, the draw is decided in favor of the `true`/1 value. Such a decision stems from the fact that favoring one value over the other does not significantly affect the results of HDC operations and, as a result, the accuracy of HDC-based classifiers, given the usually high amount of vectors summed before thresholding [12]. As a result, the DPHDC framework provides an efficient implementation of bundling operations that automatically deals with bundle draws, by favoring one value, while still allowing the user to override this behaviour in favor of random draw breaks. This illustrates how the proposed library was developed with the intention of providing ease-of-use while not compromising on flexibility and available features.

Listing 3.10: Thresholding of accumulators back into hypervectors.

```
1  void HDMatrix::normalizeAccumulator(sycl::buffer<short int, 2> &accumulators, sycl::buffer<↩
       bool, 2> &destination) {
2      this->associated_queue.submit([&](cl::sycl::handler &h) {
3          cl::sycl::accessor acc_result(destination, h, cl::sycl::write_only);
4          cl::sycl::accessor acc_accumulators(accumulators, h, cl::sycl::read_only);
5          h.parallel_for(accumulators.get_range(), [=](cl::sycl::id<2> local_range) {
6              size_t i = local_range[0];
7              size_t k = local_range[1];
8              if (acc_accumulators[i][k] >= 0) {
9                  acc_result[i][k] = true;
10             } else {
11                 acc_result[i][k] = false;
12             }
13         });
14     });
15 }
```

Binding Positional Encoder Method

The remaining encoder method, the `encodeWithXOR` positional based encoder, presented in Listing 3.11, follows the same basic steps as the two other encoder modules presented so far, with a few significant changes, since no permutations operations are performed during this method. The most significant one is that no parallelism needs to be expressed on the host, i.e., no host `for` loop is used, which in the other two encoder methods was necessary for executing shift operations. As a result, a single, three-dimensional data-parallel kernel can be used to encode all dataset entries (lines 14 through 30 of Listing 3.11). Another consequence of the absence of permutations is that there is no need to create copies of the base hypervectors buffer. Since the base hypervectors buffer will not be modified (it will be just read), it can be used directly in this encoder module (line 17 of Listing 3.11). The same accumulator and thresholding technique used in the `encodeWithBundle` method is also used in this method in order to accumulate all the resulting vectors from the multiplication of base hypervectors with the po-

sition hypervectors provided by the user (as an `HDMatrix` object) (the creation and initialization of the accumulators variable takes place in line 9 of Listing 3.11). Finally, the main kernel code of this method (lines 14 through 30 of Listing 3.11) can be described as follows: for each simple data element of each dataset entry provided, the kernel fetches the associated hypervector and performs the XOR operation with the associated position vector on every element (line 24). If the result of the operation is `true`, then one is accumulated to the corresponding accumulator (line 25). Otherwise, one is subtracted from the respective position in the accumulators variable (line 27). The encoded entries, i.e., the output of the `encodeWithXOR` positional module, is obtained when the accumulators variable is threshold back to `true` or `false` according to majority rule using the `normalizeAccumulator` method, already explained and presented in Listing 3.10.

Listing 3.11: Implementation of the `encodeWithXOR` positional module.

```
1  template<class TypeOfDataToRepresent>
2  HDMatrix HDRepresentation<TypeOfDataToRepresent>::encodeWithXOR(const std::vector<std::↩
       vector<TypeOfDataToRepresent>> &data, HDMatrix &position_vectors) {
3      int max_element_size = 0;
4      std::unique_ptr<int[]> converted_data = this->convertData(data, max_element_size);
5
6      HDMatrix encoded_vectors_matrix(this->vectors_buff.get_range()[1], data.size(), ↩
           vectors_generator::none, this->associated_queue);
7
8      {
9          cl::sycl::buffer<short int, 2> buff_accumulators = this->↩
               generateInitializeAccumulators(cl::sycl::range<2>(data.size(), this->↩
               vectors_buff.get_range()[1]));
10
11         cl::sycl::buffer<int, 2> buff_data(converted_data.get(), cl::sycl::range<2>(data.↩
               size(), data[0].size()));
12         buff_data.set_write_back(false);
13
14         this->associated_queue.submit([&](cl::sycl::handler &h) {
15             cl::sycl::accessor acc_accumulators(buff_accumulators, h, cl::sycl::read_write);
16             cl::sycl::accessor acc_data(buff_data, h, cl::sycl::read_only);
17             cl::sycl::accessor acc_representation(this->vectors_buff, h, cl::sycl::read_only↩
                   );
18             cl::sycl::accessor acc_position_vectors(position_vectors.vectors_buff, h, cl::↩
                   sycl::read_only);
19             cl::sycl::range<3> r(n_entries_data, size_entry, vector_size)
20             h.parallel_for(r, [=](cl::sycl::id<3> local_range) {
21                size_t i = local_range[0];
22                size_t j = local_range[1];
23                size_t k = local_range[2];
24                if (acc_representation[acc_data[i][j]][k] ^ acc_position_vectors[j][k]) {
25                    acc_accumulators[i][k] += 1;
26                } else {
```

```
27                         acc_accumulators[i][k] -= 1;
28                     }
29                 });
30             });
31 
32             this->normalizeAccumulator(buff_accumulators, encoded_vectors_matrix.vectors_buff);
33         }
34 
35         return encoded_vectors_matrix;
36 }
```

As hopefully became apparent with the explanation of the development and implementation of the encoder methods offered by DPHDC, despite the fact that these methods were primarily designed to facilitate the implementation of classification applications using HDC, their use can be extended to general-computing HDC-based algorithms, like hash-tables as explained in Section 3.1.2, or to the implementation of multistep encoders, as explained in Section 3.1.4. This is only possible due to the fact that, at their core, encoder methods are converting the provided data into base hypervectors which are then manipulated using the HDC defined arithmetic operations. As a result, it is possible for the user to create its own dataset and associated `HDRepresentation` object to run a desired set of HDC operations. The performant implementation of non-strictly classification HDC-based algorithms using DPHDC is a testament to generality of the design and development of the proposed library.

3.2.4 Training and Testing Methods

After the encoding of all entries in a dataset, a typical HDC-based classifier usually performs one of two actions. It either bundles all encoded entry hypervectors that correspond with the same class in order to generate a trained model (training phase) or it queries said encoded entries against an already trained model (testing/querying phase). As a result, the implementation of the DPHDC methods developed to deal with both of these procedures will be explored in this Section, as has been done so far for all methods that take advantage of the parallel computing capabilities of an accelerator device. It is important to mention that in both the training and testing cases, there is no need to associate the encoded hypervectors or trained model to a data element, like it was necessary for encoder modules. Consequently, both encoded hypervectors and trained models are represented by an `HDMatrix` object (as discussed in Section 3.1) and the methods associated with training (`reduceToLabelsBundle`) and testing (`queryModel`) are part of the `HDMatrix` class, as illustrated in Figure 3.1 and presented in the beginning of Section 3.1.

Training

In order to bundle all encoded hypervectors from a previously provided dataset according to the respective class/label, it is necessary to use the `reduceToLabelsBundle` method by providing the corresponding class/label in the form of a vector of strings of characters, one string per dataset entry encoded.

This method will return an `HDMatrix` object with the corresponding class hypervectors and associated labels (stored in the `labels` private variable), i.e., it will return a trained HDC model.

The first step in this process consists in finding the unique labels in the classes provided, and associating each encoded hypervector to a respective class hypervector. This is achieved through the use of the DPHDC auxiliary method `processReduceLabels`, presented in Listing 3.12. The `provided_labels` method consists of a simple algorithm (lines 5 through 11 of Listing 3.12) that takes advantage of the `find` function available in modern versions of the C++ programming language. For every label/class provided (line 5 of Listing 3.12), first it is necessary to check if the label/class is already stored in the `unique_labels` vector (lines 6 and 7 of Listing 3.12). If such is not the case, then this label/class is inserted at the end of the vector (line 8 of Listing 3.12). Finally, it is also necessary to store the corresponding index that the label/class being processed has in the `unique_labels` variable (line 11 of Listing 3.12), in order for bundling of all vectors of the same class to be possible, as will be explained once the main kernel of the `processReduceLabels` method is explored. These indexes are stored in vector `vector_correspondence`, returned by the method.

Identically to the conversion of provided datasets into corresponding base hypervector indexes, as can be inferred from the presentation done so far, this process of finding unique labels is performed on the host. The reasons for such a decision are similar as well, i.e., this process is usually well adapted to the host (normally a CPU) architecture with an impractical translation into data-parallel kernels. Furthermore, given the fact that the labels/classes provided are strings of characters, which can have a significant memory footprint, by performing the conversion to indexes on the host it is possible to reduce the size of the data transfers to the accelerator, diminishing the potential of memory bottlenecks affecting the performance of the proposed library. This is another reason why the conversion of the provided datasets into corresponding base hypervector indexes in the encoder methods is performed on the host.

Listing 3.12: Identification of unique labels and association between encoded hypervector and respective class hypervector.

```
1  std::vector<unsigned int> HDMatrix::processReduceLabels(const std::vector<std::string> &↩
       provided_labels, std::vector<std::string> &unique_labels) const {
2      unique_labels = {};
3      std::vector<unsigned int> vector_correspondence(provided_labels.size());
4  
5      for (unsigned int i = 0; i < provided_labels.size(); i++) {
6          auto iterator = std::find(unique_labels.begin(), unique_labels.end(), ↩
               provided_labels[i]);
7          if (iterator == unique_labels.end()) {
8              unique_labels.push_back(provided_labels[i]);
9              iterator = std::find(unique_labels.begin(), unique_labels.end(), provided_labels↩
                   [i]);
10         }
11         vector_correspondence[i] = iterator - unique_labels.begin();
12     }
13 
```

```
14     return vector_correspondence;
15 }
```

After obtaining the unique labels from those provided and the respective corresponding indexes (lines 2 and 3 of Listing 3.13), the `reduceToLabelsBundle` method proceeds to create the `HDMatrix` `trained_model` object (line 4 of Listing 3.13), returned by the method, and save these unique labels in the `labels` variable of said object (line 5 of Listing 3.13). The storage of these labels/classes is crucial for when the model is queried, since it is the label of the most similar hypervector to the one being queried that is returned.

When observing the implementation of the `reduceToLabelsBundle` method, presented in Listing 3.13, it can be inferred that the same accumulator and thresholding technique used with the `encodeWithBundle` method and `encodeWithXOR` positional method is used to deal with the bundling operation. As a result, accumulators are created and initialized in line 8 of Listing 3.13 and normalized back into the trained model in line 26 of Listing 3.13. The main kernel of the method (lines 16 through 22 of Listing 3.13) consists only in the addition of one hypervector, i.e., it is a one dimensional data-parallel kernel, contrary to most kernels presented so far. The remaining parallelism/dimension, i.e., iterating over all encoded hypervectors that need to be summed to the respective class accumulator, is expressed on the host through the use of a `for` loop (line 11 of Listing 3.13).

At first glance, such an approach seems inadequate given the fact that one of the main objectives of the library is the maximum exploitation of the parallelism capabilities of the accelerator. The reason for using this approach stems from the need to avoid race conditions. When using the buffer and accessor technique for managing data, the SYCL backend guarantees that data races never occur due to the posterior launching of kernels, as explained in Section 2.5.2, but it does not guarantee that race conditions will not happen while executing the kernels themselves. As a result, kernels need to be designed and developed in a way that avoids data races. In the present case, the number of hypervectors is being reduced, i.e., the number of encoded hypervectors (equal to the number of dataset entries) is higher than the number of class hypervectors (equal to the number of unique classes provided) to be returned by this method. The implementation of a two-dimensional data-parallel kernel to replace the current approach would inevitably lead to data races, since different execution threads would be trying to read and write to the same accumulator position at the same time. The presented solution was the one that allowed to exploit as much parallelism as possible from the accelerator device while guaranteeing a deterministic result, as the sequential launch of the data-parallel kernels responsible for the hypervector addition is generally able to saturate the accelerator, even when targeting highly parallel architectures like GPUs. The use of atomic operations in conjunction with a two-dimensional data-parallel kernel was also considered, but it was deemed not ideal due to the fact that it lead to worse performance than the current approach across the target architectures of the library.

Listing 3.13: Implementation of the `reduceToLabelsBundle` method.

```
1 HDMatrix HDMatrix::reduceToLabelsBundle(const std::vector<std::string> &labels_provided) {
```

```
2     std::vector<std::string> unique_labels;
3     std::vector<unsigned int> vector_correspondence = this->processReduceLabels(
          labels_provided, unique_labels);
4     HDMatrix trained_model(this->vectors_buff.get_range()[1], unique_labels.size(), dphdc::
          vectors_generator::none, this->associated_queue);
5     trained_model.labels = unique_labels;
6
7     {
8         cl::sycl::buffer<short int, 2> buff_accumulators = this->
              generateInitializeAccumulators(cl::sycl::range<2>(unique_labels.size(), this->
              vectors_buff.get_range()[1]));
9
10        unsigned int aux;
11        for (size_t i = 0; i < vector_correspondence.size(); i++) {
12            aux = vector_correspondence[i];
13            this->associated_queue.submit([&](cl::sycl::handler &h) {
14                cl::sycl::accessor acc_encoded_vectors(this->vectors_buff, h, cl::sycl::
                      read_only);
15                cl::sycl::accessor acc_accumulators(buff_accumulators, h, cl::sycl::
                      read_write);
16                h.parallel_for(cl::sycl::range<1>(buff_accumulators.get_range()[1]), [=](cl
                      ::sycl::id<1> k) {
17                    if (acc_encoded_vectors[i][k]) {
18                        acc_accumulators[aux][k] += 1;
19                    } else {
20                        acc_accumulators[aux][k] -= 1;
21                    }
22                });
23            });
24        }
25
26        trained_model.normalizeAccumulator(buff_accumulators, trained_model.vectors_buff);
27    }
28    return trained_model;
```

Querying/Testing

The `queryModel` method is responsible for querying the trained model by calling the method with the provided encoded entries and using the similarity measurement desired: Hamming distance (for binary models) or cosine similarity (for bipolar models). Since bipolar and binary HDC models are mathematically equivalent [37], so far it has not been necessary to make a distinction between them during the presentation of the development and implementation process. With the similarity measurement used the major difference between these two models, when choosing the similarity metric to use while querying the trained hypervectors, the HDC model being used is also inferred. As a result, the implementation of the `queryModel` method, presented in Listing 3.14, is actually hiding two hidden methods behind a `switch` statement: `hammingDistanceIndexVector`, responsible for querying the model using Hamming

distance as the similarity metric (line 5 of Listing 3.14), and `cosineIndexVector`, responsible for querying the model using cosine similarity as the similarity metric (line 8 of Listing 3.14). The selection of the method to use is done based on the `method_to_use` enumerator provided by the user.

Listing 3.14: Implementation of the `queryModel` method.

```
std::vector<std::string> HDMatrix::queryModel(HDMatrix &encoded_test_entries, ↩
    distance_method::distance_method method_to_use) {
    std::vector<unsigned int> indexes_vector;
    switch (method_to_use) {
        case distance_method::hamming_distance:
            indexes_vector = this->hammingDistanceIndexVector(encoded_test_entries);
            break;
        case distance_method::cosine:
            indexes_vector = this->cosineIndexVector(encoded_test_entries);
            break;
    }
    std::vector<std::string> to_return(indexes_vector.size());

    for (unsigned int i = 0; i < indexes_vector.size(); i++) {
        to_return[i] = this->labels[indexes_vector[i]];
    }

    return to_return;
}
```

The calculation of cosine similarity between two vectors, defined by equation (2.1), can be boiled down to a weighted sum, i.e., when two elements are identical, one is added to the current similarity value (which starts at zero), otherwise one is subtracted. The `cosineIndexVector` method, presented in Listing 3.15, uses this exact approach to calculate the cosine similarity value of each encoded hypervector provided with each class hypervector. Firstly, the `distance_vectors_temp` array, which will store all similarity values, is created, allocated (line 5 of Listing 3.15) and all its elements are initialized at zero using a normal data-parallel kernel implemented in the auxiliary DPHDC method `fill2DBuffer` (line 9 of Listing 3.15). The main algorithm responsible for the calculation of all similarity values follows (lines 11 through 27 of Listing 3.15). As can be observed, if both the elements at a certain position from the encoded test entries and trained model hypervectors are identical (line 19 of Listing 3.15) then one is added to the corresponding similarity value, stored in the `distance_vectors_temp` array (line 20 of Listing 3.15). Otherwise, one is subtracted from this value (line 22 of Listing 3.15), as previously described. It is easy to verify that some parallelism is expressed on the host in order to avoid race conditions, as explained when exploring the implementation of the `reduceToLabelsBundle` method. While the host iterates through each hypervector element (line 11 of Listing 3.15), each launched data-parallel kernel compares that exact element (`i`) in every encoded hypervector being queried with the corresponding value of every class vector, summing or subtracting to the corresponding similarity value stored at the `distance_vectors_temp` array (lines 16 through 24 of Listing 3.15).

After calculating all similarity values, the `distance_vectors_temp` array is converted into a C++ vector (line 29 of Listing 3.15), in order for it to be possible to use the `max_element` function of the standard C++ library to obtain the index of the closest class hypervector to every encoded hypervector provided (lines 32 through 35 of Listing 3.15), which are then returned to the `queryModel` method (line 37 of Listing 3.15). The `queryModel` method then proceeds to convert the returned indexes `indexes_vector` into the estimated labels to be returned, using the classes stored in the `labels` variable (lines 13 through 15 of Listing 3.14).

Listing 3.15: Determination of the most similar class vector to every encoded hypervector using cosine similarity.

```
1  std::vector<unsigned int> HDMatrix::cosineIndexVector(HDMatrix &encoded_test_entries) {
2      cl::sycl::range<3> total_range(encoded_test_entries.vectors_buff.get_range()[0], this->↩
           vectors_buff.get_range()[0], this->vectors_buff.get_range()[1]);
3
4      {
5          std::unique_ptr<long long int[]> distance_vectors_temp(new long long int[total_range↩
               [0] * total_range[1]]);
6          {
7              cl::sycl::range<2> first_two_range(total_range[0], total_range[1]);
8              cl::sycl::buffer<long long int, 2> buff_distance_vectors(distance_vectors_temp.↩
                   get(), first_two_range);
9              this->fill2DBuffer<long long int>(buff_distance_vectors, 0);
10
11             for (size_t k = 0; k < total_range[2]; k++) {
12                 this->associated_queue.submit([&](cl::sycl::handler &h) {
13                     cl::sycl::accessor acc_distance_vectors(buff_distance_vectors, h, cl::↩
                           sycl::read_write);
14                     cl::sycl::accessor acc_encoded_test_entries(encoded_test_entries.↩
                           vectors_buff, h, cl::sycl::read_only);
15                     cl::sycl::accessor acc_model_entries(this->vectors_buff, h, cl::sycl::↩
                           read_only);
16                     h.parallel_for(first_two_range, [=](cl::sycl::id<2> local_range) {
17                         size_t i = local_range[0];
18                         size_t j = local_range[1];
19                         if (acc_encoded_test_entries[i][k] == acc_model_entries[j][k]) {
20                             acc_distance_vectors[i][j] += 1;
21                         } else {
22                             acc_distance_vectors[i][j] -= 1;
23                         }
24                     });
25                 });
26             }
27         }
28
29         distance_vectors = convertArrayToVector(distance_vectors_temp);
30     }
```

```cpp
31
32     std::vector<unsigned int> to_return(distance_vectors.size());
33     for (unsigned int i = 0; i < distance_vectors.size(); i++) {
34         to_return[i] = std::max_element(distance_vectors[i].begin(), distance_vectors[i].end↩
               ()) - distance_vectors[i].begin();
35     }
36
37     return to_return;
38 }
```

Hamming distance can be calculated in a similar fashion to cosine similarity, i.e., one is summed to the respective Hamming distance value if the elements are different. Otherwise the distance value does not change. As a result, the `hammingDistanceIndexVector` is almost identical to the `cosineIndexVector` method, with the only differences being: in lines 19 through 23 of Listing 3.15, where instead it is only checked if the values are different and, if they are, one is summed to the corresponding distance value; And line 34, where function `min_element` is used instead of `max_element`, since the vector with the smallest Hamming distance is the most similar.

3.2.5 Python Front-end

As explained so far, all values needed to use and all values returned by the proposed library are standard C++ variables. This design feature not only allows for easy interoperability with other C++ frameworks, but it also allows for the compilation of the framework as a Python binary module using the pybind11 library [48]. The resulting Python DPHDC front-end is practically identical to its C++ counterpart, replacing the standard C++ variables used with the library as inputs and outputs with Python standard library variables. For example, encoder modules in the Python front-end receive dataset entries as a list of lists (equivalent to a vector of vectors in C++). Such a feature not only increases the approachability of the library but can also allow the easy creation of hybrid models with the use of traditional Python-based ML libraries, like the hybrid model proposed in VoiceHD [17].

3.2.6 Targeting FPGAs

Finally, as already mentioned, DPHDC can currently target a wide array of devices, including CPUs, GPUs and FPGAs. Almost all currently available CPUs and GPUs are compatible with the SYCL standard [30]. In the case of FPGAs, generally, when developing an application specific for this type of accelerator, it is necessary to describe its design. With a SYCL-compatible code base, DPHDC complies with the requirements necessary to target FPGAs. This feature makes it possible to run applications without prior knowledge of FPGA design and optimization techniques. Given the potential of HDC running for low-powered architectures [12], like FPGA, such functionality allows the use of these devices without any prior experience, allowing researchers and users to focus on improving encoder design and feature extraction.

3.3 Summary

In this Chapter, the novel design of the proposed Data Parallel framework for Hyperdimensional Computing was explored, complete with available features and an application example. The two main classes of the proposed library were presented, followed by the associated developed methods, their uses, required arguments and outputs.

This was followed by a presentation of the main development and implementation aspects of the proposed framework, focused on explaining how data movement and storage, developed highly parallel algorithms and design were optimized for peak performance across devices with significantly different architectures.

Chapter 4

Experimental Results

In order to thoroughly test and benchmark the DPHDC library and all its capabilities, it was necessary to implement state-of-the-art applications of supervised classification using Hyperdimensional Computing in the proposed framework. A list of applications deployed and provided with the library are as follows (a majority of which are also described in Section 2.2.5):

- **VoiceHD** [17] a speech recognition application;
- **European language recognition** [15];
- **HDNA** [18] N-gram based encoder, used for DNA sequencing;
- A binding positional encoder for recognizing the handwritten digits of the **MNIST** dataset, inspired by [21];
- **Hand gesture recognition** using EMG signals [19].

It is important to note that all applications are replicated in DPHDC as originally proposed, except HDNA [18], which encoder is implemented slightly differently from the one presented in Section 2.2.5 with the aim of improving its performance and accuracy. The presentation of this proposed encoder for HDNA [18] is performed in Section 4.1. Furthermore, all presented results of the European Language recognition example [15] consider N-grams of size 3 (trigrams), since this is the value that provides the best accuracy to performance ratio and is the main focus of the original work [15].

Two distinct Amazon Web Services (AWS) instances were used to benchmark the DPHDC library: c5a.16xlarge and g5.xlarge. The c5a.16xlarge comprises 64 vCPUs, part of an AMD EPYC 7R32 CPU, and 128GiB of RAM. When running the DPHDC library on this instance type, the target device is the CPU itself. On the other hand, the g5.xlarge instance is composed of 4 vCPUs, 16 GiB of RAM and an NVIDIA A10G tensor core GPU, which is the target device when running DPHDC-based examples on this instance.

All results related to DPHDC presented in the following sections were obtained using Intel's DPC++ compiler, a SYCL-compatible compiler. Training time is defined as the time necessary to encode all training entries and then reduce them according to the training labels, creating the model. On the other

hand, testing time is defined as the time necessary to encode all test entries, query them with the model and check the accuracy.

4.1 Novel Encoder Alternative to HDNA [18] Encoder I

As mentioned throughout Chapter 3, the encoder methods of the DPHDC library were designed and developed in order to provide an intuitive way for users to implement the desired encoder for any application necessary. To achieve such a goal, said encoder methods were developed to be as general as possible, allowing for the chaining of encoders methods in order to achieve the desired encoder. An N-gram encoder, as explained in Section 3.1.4, is implemented in this fashion, by first applying the `encodeWithXOR` permutation encoder method to generate the N-grams and then only using the `encodeWithBundle` method to encode the data. The HDNA [18] encoder I, responsible for encoding genes into the hyperspace, is also an N-gram encoder, as explained in Section 2.2.5. In an effort to try to simplify (and, consequently, speedup) the HDNA [18] encoder I, a novel encoder is proposed. It is important to notice that the generation of the base hypervectors to be used (one for each DNA base) remains random and that the similarity metric used also remains the same (Hamming distance), i.e., only the encoder module changes in the proposed approach.

The proposed encoder relies on the fact that a permutation generates an hypervector that is quasi-orthogonal to the one that gave origin to it, just like two hypervectors when multiplied. It is goal to be able to obtain the high accuracy associated with N-gram encoders while reducing the amount of operations necessary, by utilizing the mathematical properties of the permutation operation. The achieved solution encodes each entry of the dataset using equation (3.1), i.e., using the `encodeWithBundle` encoder method. As a result, instead of using an N-gram based encoder, all hypervectors corresponding to a particular gene are bundled in order to generate the gene hypervector, being shifted according to the position they occupy in the gene, i.e., the hypervector corresponding to the first base of the gene is not permuted before being bundled, the second one is permuted once, the third one permuted twice, etc. Conceptually, by applying a unique permutation to each position, a distinct hypervector is consistently generated for each DNA base/gene position pair. A similar result is achieved when using a binding positional based encoder (`encodeWithXOR` positional encoder method), like VoiceHD [17], except the novel approach requires the use of much less memory, since there is no need to generate, store and use positional hypervectors. The proposed encoder leads to a slight increase in classification accuracy when applied to the empirical bats dataset, used to benchmark the application using DPHDC. More specifically, the novel encoder proposed manages to achieve an accuracy of 99.3% while the original encoder manages to achieve an accuracy of 98.2%, a 1.1% increase. The implementation of the proposed encoder, using DPHDC, is presented in Appendix A.2. Finally, it is important to mention again that for all results presented in this Chapter, whenever results related with the HDNA [18] application are presented or studied, the proposed novel encoder presented here is the one being used.

4.2 Scalability with Vector Dimensionality

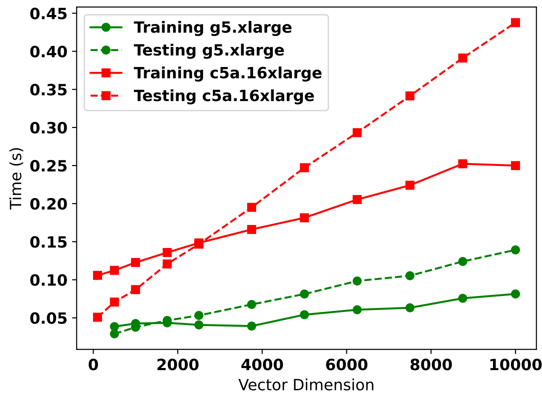


Figure 4.1: Training and testing times of the HDNA example according to vector dimensionality.

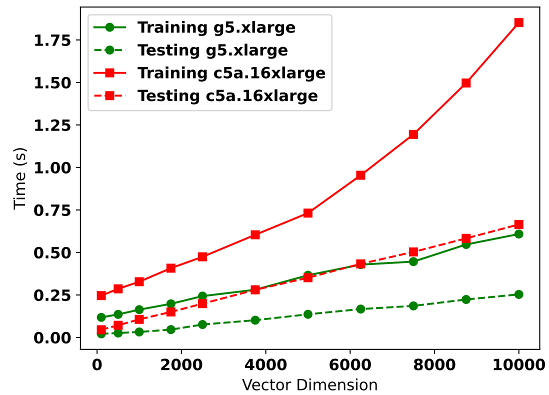


Figure 4.2: Training and testing times of the VoiceHD example according to vector dimensionality.

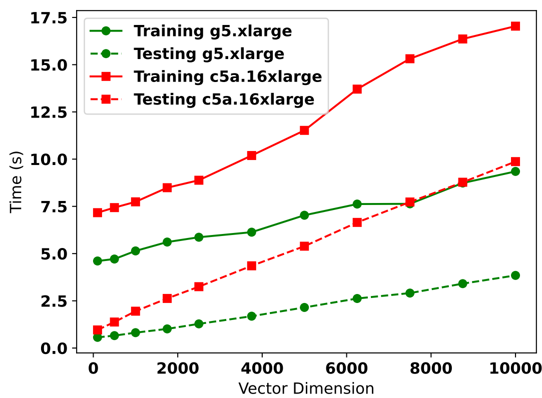


Figure 4.3: Training and testing times of the European language recognition example according to vector dimensionality.

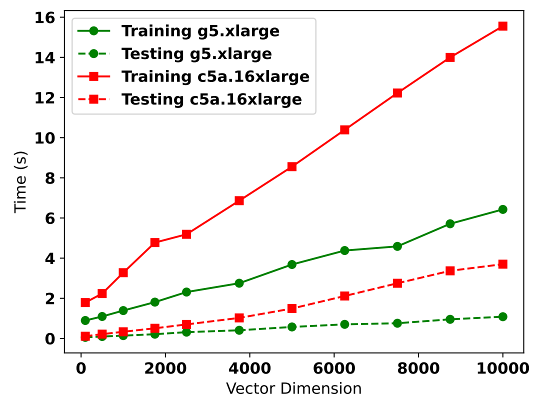


Figure 4.4: Training and testing times of the MNIST example according to vector dimensionality.

To explore how the execution times and accuracy of DPHDC-based applications behave with vector dimensionality, the HDNA [18], VoiceHD [17], European Language Recognition [15] and MNIST [21] examples were executed using different vector sizes. Training and testing times are presented in Figures 4.1, 4.2, 4.3 and 4.4, while classification accuracies are shown in Figures 4.5, 4.6, 4.7 and 4.8, respectively.

Given the highly parallel nature of HDC operations, of which most are element-wise, it is to be expected that training and testing favors the highly data-parallel architecture. Such an observation can be made in all examples presented since the execution that targeted a GPU (g5.xlarge) was always faster than the execution that targeted a CPU (c5a.16xlarge). In the case of HDNA [18] (Figure 4.1), VoiceHD [17] (Figure 4.2) and European language recognition [15] (Figure 4.3), both training and testing are faster on the GPU when vector sizes approach 10000, another indication that HDC algorithms

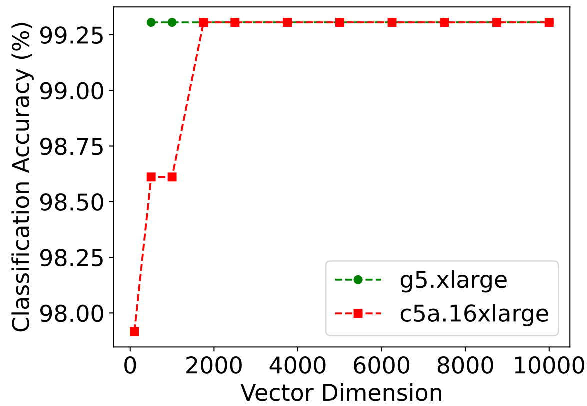


Figure 4.5: Classification accuracy of the HDNA example according to vector dimensionality.

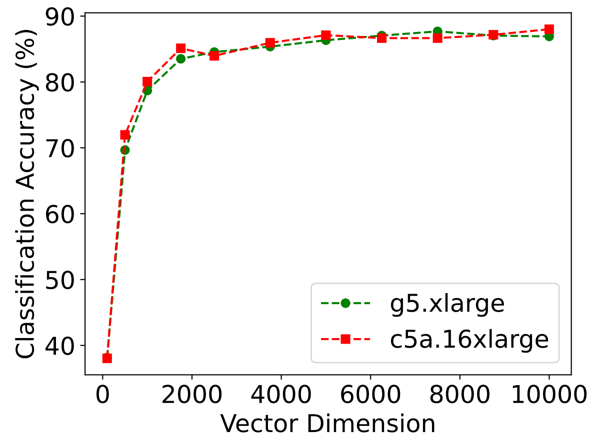


Figure 4.6: Classification accuracy of the VoiceHD example according to vector dimensionality.

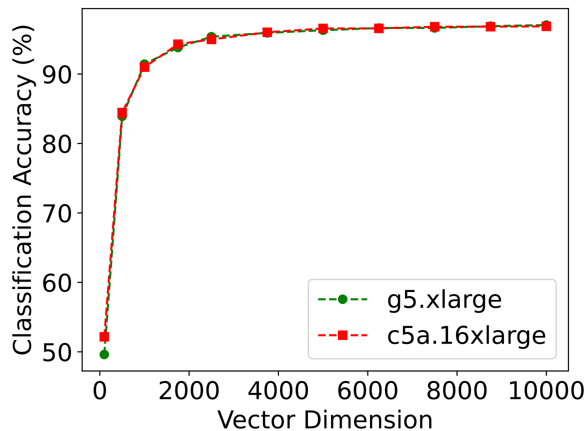


Figure 4.7: Classification accuracy of the European language recognition example according to vector dimensionality.

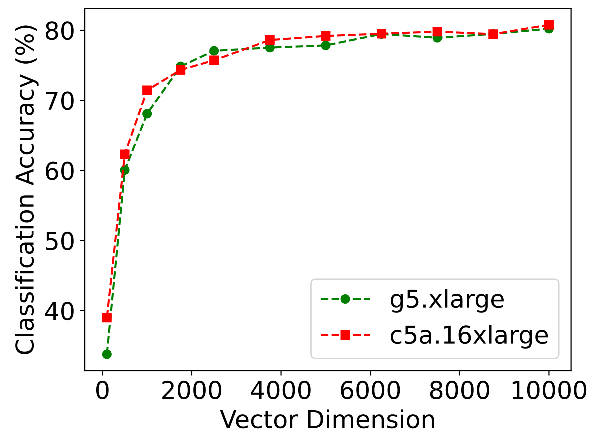


Figure 4.8: Classification accuracy of the MNIST example according to vector dimensionality.

benefit from massively parallel architectures. Despite being slightly slower, the execution on the CPU still takes advantage of all its threads and resources. The importance of this fact cannot be understated since running HDC applications on IoT devices based on low-powered CPU architectures shows excellent promise as an alternative to traditional ML classification algorithms [12]. Although in all examples presented, testing on the CPU offers a higher cost than on the GPU, thanks to the ability of the DPHDC library to store hypervectors in binary files, it is possible to train a model on a state-of-the-art GPU and then query it on a low-powered device in order to take advantage of the lightweight nature of HDC classification applications. Furthermore, an almost linear relation between training/testing time and vector dimensionality can be inferred (Figures 4.1, 4.2, 4.3 and 4.4), indicating an absence of memory bottlenecks when using vectors up to 10000 elements in size (which is the most commonly used value in HDC-based classification applications). Preliminary testing also demonstrated that running the DPHDC library by using the Python front-end entails a minimal performance decrease (lower than 5% when testing the VoiceHD application) while providing the same accuracy results.

For all deployed applications and devices, training expectedly takes more time than testing. However, it is also possible to observe that the opposite can also be true, i.e., testing times are greater than training times in the HDNA example (Figure 4.1). An explanation for this phenomenon is that the HDNA empirical bats dataset contains a small number of entries in the training dataset while containing many data classes. This dataset characteristics forces the query module to perform many comparisons while querying each test entry, leading to shorter training times when compared with slightly higher testing times.

The classification accuracies obtained are within the margin of error of the respective original works, as expected. Compared with traditional ML methods, the accuracies presented are acceptable while offering higher efficiency [12]. When the vector sizes are small, there are not enough elements to guarantee quasi-orthogonality between two randomly generated hypervectors, leading to a steep drop in accuracy. As vector dimensionality increases, so does classification accuracy until it flattens in a constant value similar to the ones presented in each work being replicated.

4.3 Comparison with Other Frameworks

Table 4.1: DPHDC and TorchHD results of the Language Recognition, MNIST and VoiceHD applications using vectors with 10000 elements.

Application	Target Device	AWS Instance	DPHDC Classification Accuracy (%)	DPHDC Training Time (s)	DPHDC Testing Time (s)	TorchHD Classification Accuracy (%)	TorchHD Training Time (s)	TorchHD Testing Time (s)	DPHDC Training Speedup	DPHDC Testing Speedup
Language	CPU	c5a.16xlarge	96.87	17.04	9.87	97.30	224.74	19.49	13.19	1.97
Language	GPU	g5.xlarge	97.06	9.35	3.85	-	85.57	-	9.15	-
MNIST	CPU	c5a.16xlarge	80.78	15.55	3.70	82.76	187.74	23.50	12.07	6.35
MNIST	GPU	g5.xlarge	80.23	6.43	1.08	82.86	45.16	11.22	7.03	10.36
VoiceHD	CPU	c5a.16xlarge	88.01	1.85	0.66	85.25	15.55	3.75	8.40	5.63
VoiceHD	GPU	g5.xlarge	86.91	0.61	0.25	85.06	3.77	2.05	6.19	8.10

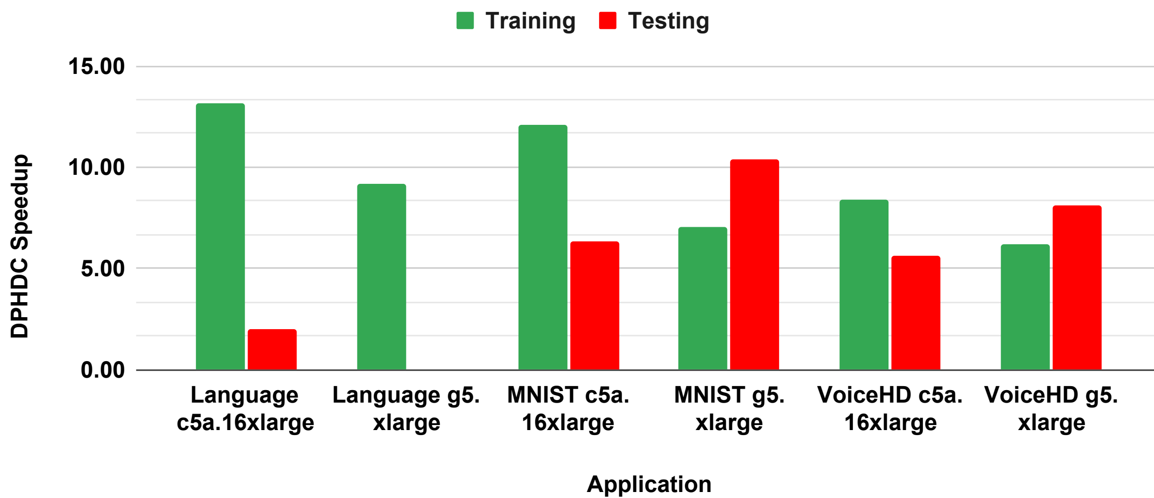


Figure 4.9: DPHDC speedup compared with TorchHD in the Language Recognition, MNIST and VoiceHD applications using vectors with 10000 elements.

Given the existence of the TorchHD library [27] that facilitates the implementations of HDC-based applications (as presented in Section 4.3), it is essential to compare performances when targeting devices that are compatible with both libraries. To compare with the TorchHD library [27], the Language Recognition [15], VoiceHD [17] and MNIST [21] examples presented in Section 2.2.5 (also provided with the TorchHD library), were executed using the same two AWS instances.

The results obtained using the latest version of the TorchHD library (version 3.3.0) are presented in Table 4.1. As expected, the accuracies obtained on all applications running both on CPU and GPU are similar across both frameworks. As shown in Figure 4.9, despite the more general device focus of the presented solution, DPHDC outperforms TorchHD in all applications tested by, on average, 11.2x on CPU training, 4.7x on CPU testing, 7.5x on GPU training and 9.2x on GPU testing. From Table 4.1 it can also be inferred that, in general, as training time increases, so does the speedups obtained by DPHDC. This can be explained by the data-parallel optimizations embedded in DPHDC, that become more prevalent as the amount of data to encode increases.

4.4 FPGA Implementation

Table 4.2: Execution time and accuracy of the hand gesture recognition application running on Intel Arria 10GX using DPHDC with a vector size of 2500.

Subject	Classification Accuracy (%)	Training Time (s)	Testing Time (s)
Subject 1	92.18	0.0544	0.132
Subject 2	91.07	0.0469	0.132
Subject 3	95.78	0.0491	0.132
Subject 4	87.65	0.0458	0.131
Subject 5	90.68	0.0351	0.129

Given the potential of HDC classification applications as lightweight replacements for traditional ML methods on low-powered and dedicated devices like FPGAs, it is crucial to assess the performance of the proposed DPHDC framework when targeting this class of devices. To achieve this goal, the lightweight **hand gesture recognition** [19] spatial encoder was used. The dataset comprises data from 5 subjects, of which 70% of the entries were used for training and 30% for testing. A down-sampling of 250 was performed on all subjects before running the application. It is important to note that the 70/30 division of the dataset, coupled with its small size and the extra cost that inference has on HDC models (due to the necessity of comparing hypervectors) leads to testing times that are higher than training times when targeting any type of accelerator. The example was compiled for the Intel Arria 10 GX FPGA, available at Intel DevCloud. This compilation process took several hours since a unique design and bitstream for the specific FPGA card are generated based on the compiled code.

The obtained experimental results are presented in Table 4.2. Even though no execution times are presented in the original work [19], the results show that low-powered real-time classification based on FPGA using DPHDC is possible without the additional cost of creating an application-specific low-level design. The generated design manages to achieve a clock frequency of 230MHz while using 35% of

available ALUTs, 25% of available FFs, 93% of the available RAM, 4% of available MLABs and 26% of available DSPs.

4.5 Comparison with Original Implementations

Given the generally high cost of designing, developing and implementing an HDC-based classifier, a major contribution that DPHDC can provided to the wider scientific community is the ability it confers researchers and engineers to quickly implement HDC-based classifiers that can efficiently target most common types of available accelerators. As a result, it is important to compare applications developed using DPHDC with the respective original implementations, if available, to explore how DPHDC could benefit HDC research and real-world use.

As mentioned previously, most recent research concerning HDC-based classifiers has been focused on improving encoder design with the goal of improving accuracy [12]. Consequently, most of the articles of the applications presented and studied in Section 2.2.5, all of which were implemented using DPHDC, do not present performance results and usually mention few implementation details. Both the European Language Recognition application [15] and the gesture recognition application based on EMG [19] are examples of this. Even tough the original implementation of both of these examples can be found publicly, they are implemented using the proprietary MATLAB language, can only target the host (i.e., the CPU) and were not designed with performance in mind.

On the other hand, both the articles describing VoiceHD [17] and HDNA [18] present performance results and briefly mention how they were implemented. In the case of VoiceHD [17], the original implementation, which is also only able to target the host, is publicly available. By being Python based, it was possible to directly compare the original implementation with the DPHDC implementation. When running on the low-powered Intel Core i5-10210U Processor (4 cores, 1.60GHz of base frequency) with 8GiB of RAM, the total execution time (training and testing) when using the original implementation is $381.2s$, while using the DPHDC implementation it is $8.6s$, a 44x speedup. Unfortunately, to the best of the author knowledge, the implementation of HDNA [18] is not publicly available, making a direct comparison with DPHDC impossible.

The focus on improving encoder design and general HDC encoding methodology [12] explains the why all of the presented implementations are based on high-level languages, as they make it considerably quicker and easy to build an HDC-based application from the ground-up in order to test and evaluate an encoder accuracy, at the cost of performance. As such, the DPHDC library, with its easy to use and versatile design, is expected to significantly contribute to a faster implementation and execution of HDC-based classifiers across a wide array of device architectures, allowing researchers and engineers to focus on improving the classifiers themselves.

4.6 Performance Impact of Different DPHDC Methods

In order to measure the impact that each method of the proposed framework, whose implementation is described in Section 3.2, has on the overall performance of applications implemented using DPHDC, the measurement of the execution time of each method was performed and compared with the total execution time of the application. The obtained results, presented in Figure 4.10, were generated on a system containing an Intel Core i5-10210U Processor (4 cores, 1.60GHz of base frequency) with 8GiB of RAM and a NVIDIA GeForce MX350 graphics card. As a result, the HDNA [18], VoiceHD [17], MNIST [21] and European Language Recognition [15] were executed in this environment while targeting the presented CPU and the GPU.

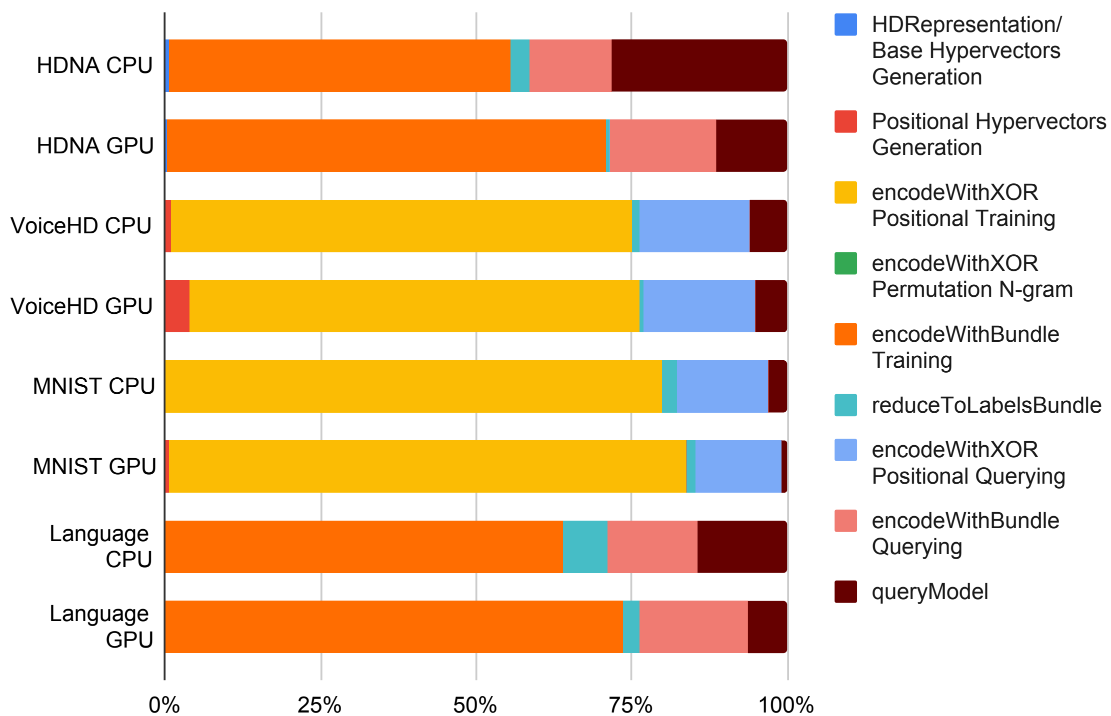


Figure 4.10: Ratio (%) between each method execution time and total application execution time using vectors containing 10000 elements.

As expected, the results presented in Figure 4.10 illustrate that in all applications the encoder methods are the ones that take the most time to complete, both for training and testing. On the other end, the generation of hypervectors is the performance-wise least significant part of all applications tested. The remaining methods associated with the HDC classification methodology, i.e., `reduceToLabelsBundle` and `queryModel`, have a significant (but considerably smaller than the encoders) impact on the overall execution across all presented cases.

Furthermore, it is possible to infer that the execution weight of the encoder methods depends on the dataset size, as applications dealing with smaller datasets (HDNA [18] and VoiceHD [17]) have smaller ratios of encoder method utilization when compared with applications dealing with larger datasets (MNIST

[21]). This conclusion is expected, since as the datasets get larger, the number of entries increase and, as a result, more data needs to be encoded to the hyperspace, while the time necessary to execute the other methods associated with the HDC classification methodology does not grow as much with dataset size. This also explains why encoder methods used during testing have a smaller overall impact than encoder methods used during training, since test datasets are usually considerably smaller than training datasets. It can also be observed that the impact of the `queryModel` method on HDNA [18] is considerably greater than on the remaining applications. The explanation for this stems from the characteristics of the bats empirical dataset used to test the application, since the dataset is relatively small and contains a considerable amount of classes, leading to smaller encoding times and higher query times, as already mentioned in Section 4.2. The consistency of the presented results across applications and devices also demonstrates the scalability and portability of the DPHDC library, as every method was developed with the intention of exploiting as much parallelism as possible, independent of data being encoded, device being targeted and dataset size.

4.7 Summary

In the present Chapter, the results of the benchmarking and testing of the proposed DPHDC library were presented. A study of the scalability of the library with hypervector size was performed across both CPU and GPU architectures, followed by a comparison with the state-of-the-art of frameworks focused on the implementation of HDC-based classifiers. Next, the results of targeting FPGA cards using the proposed framework are explored. A comparison with the original implementations of the applications developed and a study of the impact of each of the library methods follows. Finally, the presentation of a proposed novel encoder as an alternative to HDNA encoder I is also performed.

Chapter 5

Conclusions

In this dissertation, the design, development, implementation, benchmarking and testing of a SYCL-based open-source heterogeneous library to facilitate the implementation and accelerate HDC-based classification applications was proposed.

The presented object-oriented design of the proposed Data Parallel framework for Hyperdimensional Computing (DPHDC) provided data classes and methods to perform all necessary actions related with HDC-based classification scenarios. The intuitive programming interfaces made available in the C++ and Python programming languages ensured an easy implementation process when using the proposed library, while the developed encoder methods allow for the implementation of most desired encoder. Furthermore, despite the focus on classification applications, it was also possible to infer that general HDC algorithms could also be implemented using the proposed solution.

The development and implementation of SYCL data-parallel kernels, optimized for CPU, GPU and FPGA devices, made it possible to efficiently target these device architectures efficiently by exploiting the inherent parallelism of HDC operations. The optimized storage of information and minimization of data movement were also crucial to reduce the framework's memory requirements and decrease the risk of memory bottlenecks during execution, which would negatively affect performance.

The testing of the proposed DPHDC revealed that the HDC-based classification applications developed using this tool achieved the same accuracy and results as the counterparts they are based upon, which indicated the correct functioning of the proposed framework. Moreover, the benchmarking of the library exposed that DPHDC is up to 13x faster on CPU and 10x faster on GPU than the state-of-the-art general-purpose and multi-device HDC framework, while allowing the targeting of more device architectures, like FPGAs.

In sum, all goals were achieved as the proposed DPHDC framework allows for easy deployment and high performance of applications based on HDC classification with the potential to allow researchers and the broader scientific community to focus on encoder and application design without worrying about implementation details across a wide range of compute devices.

5.1 Future Work

Given the fact that the proposed framework can currently only support binary and bipolar HDC models, given their increased hardware friendliness and potential as an alternative to traditional ML algorithms, one main direction for future work is expanding the library to include more HDC model types, i.e, allowing the use of hypervectors that are not strictly binary or bipolar. This research would require templating the buffer data type used to represent hypervectors and adjusting the implemented methods to be compatible with the new VSA models being added.

Furthermore, DPHDC was developed with the goal in mind of being able to efficiently target as many device architectures as possible. As a result, the development of hyper-optimized device-specific versions of the library could also be relevant and readily achievable, given the intuitive object-oriented design of the framework. To achieve this goal, the developed device code would have to be modified to include device/vendor specific functions and functionalities.

Finally, given the current general lack of adequate publicly available datasets necessary for developing a classifier in the aerospace field [5], the development and/or implementation, using the proposed library, of applications based on HDC and related with this field could prove valuable, given the potential of robust and noise-resistant lightweight classifiers in this subject area, as illustrated in Section 2.3.

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[2] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural networks* 61 (2015), pp. 85–117.

[3] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[4] Steven L Brunton et al. “Data-driven aerospace engineering: reframing the industry with machine learning”. In: *AIAA Journal* 59.8 (2021), pp. 2820–2847.

[5] Mate Kisantal et al. “Satellite pose estimation challenge: Dataset, competition design, and results”. In: *IEEE Transactions on Aerospace and Electronic Systems* 56.5 (2020), pp. 4083–4098.

[6] Gianni D’Angelo and Salvatore Rampone. “Feature extraction and soft computing methods for aerospace structure defect classification”. In: *Measurement* 85 (2016), pp. 192–209.

[7] Priyadarshini Panda, Abhronil Sengupta, and Kaushik Roy. “Conditional deep learning for energy-efficient and enhanced pattern recognition”. In: *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2016, pp. 475–480.

[8] Savath Saypadith and Supavadee Aramvith. “Real-time multiple face recognition using deep learning on embedded GPU system”. In: *2018 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE. 2018, pp. 1318–1324.

[9] Joe Lemley, Shabab Bazrafkan, and Peter Corcoran. “Deep Learning for Consumer Devices and Services: Pushing the limits for machine learning, artificial intelligence, and computer vision.” In: *IEEE Consumer Electronics Magazine* 6.2 (2017), pp. 48–56.

[10] Daniel Justus et al. “Predicting the computational cost of deep learning models”. In: *2018 IEEE international conference on big data (Big Data)*. IEEE. 2018, pp. 3873–3882.

[11] Yue Wang et al. “Energynet: Energy-efficient dynamic inference”. In: (2018).

[12] Lulu Ge and Keshab K. Parhi. “Classification Using Hyperdimensional Computing: A Review”. In: *IEEE Circuits and Systems Magazine* 20.2 (2020), pp. 30–47. DOI: 10.1109/MCAS.2020.2988388.

[13] Pentti Kanerva. "Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors". In: *Cognitive Computation* 1.2 (June 2009), pp. 139–159. ISSN: 1866-9964. DOI: 10.1007/s12559-009-9009-8. URL: https://doi.org/10.1007/s12559-009-9009-8.

[14] Denis Kleyko et al. "Vector Symbolic Architectures as a Computing Framework for Nanoscale Hardware". In: (2021). DOI: 10.48550/ARXIV.2106.05268. URL: https://arxiv.org/abs/2106.05268.

[15] Abbas Rahimi, Pentti Kanerva, and Jan M. Rabaey. "A Robust and Energy-Efficient Classifier Using Brain-Inspired Hyperdimensional Computing". In: *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ISLPED '16. San Francisco Airport, CA, USA: Association for Computing Machinery, 2016, pp. 64–69. ISBN: 9781450341851. DOI: 10.1145/2934583.2934624. URL: https://doi.org/10.1145/2934583.2934624.

[16] Aditya Joshi, Johan T. Halseth, and Pentti Kanerva. "Language Geometry Using Random Indexing". In: *Quantum Interaction*. Ed. by Jose Acacio de Barros, Bob Coecke, and Emmanuel Pothos. Cham: Springer International Publishing, 2017, pp. 265–274. ISBN: 978-3-319-52289-0.

[17] Mohsen Imani et al. "VoiceHD: Hyperdimensional Computing for Efficient Speech Recognition". In: *2017 IEEE International Conference on Rebooting Computing (ICRC)*. 2017, pp. 1–8. DOI: 10.1109/ICRC.2017.8123650.

[18] Mohsen Imani et al. "HDNA: Energy-efficient DNA sequencing using hyperdimensional computing". In: *2018 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI)*. 2018, pp. 271–274. DOI: 10.1109/BHI.2018.8333421.

[19] Abbas Rahimi et al. "Hyperdimensional biosignal processing: A case study for EMG-based hand gesture recognition". In: *2016 IEEE International Conference on Rebooting Computing (ICRC)*. 2016, pp. 1–8. DOI: 10.1109/ICRC.2016.7738683.

[20] Denis Kleyko et al. "Holographic Graph Neuron: A Bioinspired Architecture for Pattern Processing". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.6 (2017), pp. 1250–1262. DOI: 10.1109/TNNLS.2016.2535338.

[21] Alec Xavier Manabat et al. "Performance Analysis of Hyperdimensional Computing for Character Recognition". In: *2019 International Symposium on Multimedia and Communication Technology (ISMAC)*. Aug. 2019, pp. 1–5. DOI: 10.1109/ISMAC.2019.8836136.

[22] Tyler Michael Lovelly. "Comparative Analysis of Space-Grade Processors". PhD thesis. University of Florida, 2017.

[23] Sonja Caldwell. *Power State of the Art NASA report*. Oct. 2021. URL: https://www.nasa.gov/smallsat-institute/sst-soa/power#3.8.

[24] Denis Kleyko et al. "A Survey on Hyperdimensional Computing aka Vector Symbolic Architectures, Part II: Applications, Cognitive Models, and Challenges". In: *ACM Computing Surveys* (Aug. 2022). DOI: 10.1145/3558000. URL: https://doi.org/10.1145%5C%2F3558000.

[25] Abbas Rahimi et al. “High-Dimensional Computing as a Nanoscalable Paradigm”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.9 (2017), pp. 2508–2521. DOI: 10.1109/TCSI.2017.2705051.

[26] Denis Kleyko et al. “A Survey on Hyperdimensional Computing Aka Vector Symbolic Architectures, Part I: Models and Data Transformations”. In: *ACM Comput. Surv.* (May 2022). Just Accepted. ISSN: 0360-0300. DOI: 10.1145/3538531. URL: https://doi.org/10.1145/3538531.

[27] Mike Heddes et al. “Torchhd: An Open-Source Python Library to Support Hyperdimensional Computing Research”. In: (2022). DOI: 10.48550/ARXIV.2205.09208. URL: https://arxiv.org/abs/2205.09208.

[28] Jaeyoung Kang et al. “OpenHD: A GPU-Powered Framework for Hyperdimensional Computing”. In: *IEEE Transactions on Computers* (2022), pp. 1–1. DOI: 10.1109/TC.2022.3179226.

[29] The Khronos® SYCL™ Working Group. *SYCL™ 2020 Specification (revision 5)*. 2022. URL: https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

[30] Michael Wong et al. *Sycl - C++ single-source heterogeneous programming for acceleration offload*. Jan. 2014. URL: https://www.khronos.org/sycl/.

[31] Susmita Ray. “A quick review of machine learning algorithms”. In: *2019 International conference on machine learning, big data, cloud and parallel computing (COMITCon)*. IEEE. 2019, pp. 35–39.

[32] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[33] Pentti Kanerva. *Sparse distributed memory*. MIT press, 1988.

[34] Kenny Schlegel, Peer Neubert, and Peter Protzel. “A comparison of vector symbolic architectures”. In: *Artificial Intelligence Review* 55.6 (Aug. 2022), pp. 4523–4555. ISSN: 1573-7462. DOI: 10.1007/s10462-021-10110-3. URL: https://doi.org/10.1007/s10462-021-10110-3.

[35] Igor Nunes et al. *An Extension to Basis-Hypervectors for Learning from Circular Data in Hyperdimensional Computing*. 2022. DOI: 10.48550/ARXIV.2205.07920. URL: https://arxiv.org/abs/2205.07920.

[36] Eman Hassan et al. “Hyper-Dimensional Computing Challenges and Opportunities for AI Applications”. In: *IEEE Access* 10 (2022), pp. 97651–97664. DOI: 10.1109/ACCESS.2021.3059762.

[37] Pentti Kanerva. “Computing with High-Dimensional Vectors”. Stanford EE Computer Systems Colloquium. 2017. URL: https://web.stanford.edu/class/ee380/Abstracts/171025.html.

[38] Prasanna Date et al. *Neuromorphic Computing is Turing-Complete*. 2021. DOI: 10.48550/ARXIV.2104.13983. URL: https://arxiv.org/abs/2104.13983.

[39] Mike Heddes et al. *Hyperdimensional Hashing: A Robust and Efficient Dynamic Hash Table*. 2022. DOI: 10.48550/ARXIV.2205.07850. URL: https://arxiv.org/abs/2205.07850.

[40] Adelina Miteva. “Safety in Aerospace Engineering”. In: *complex systems* 1 (), p. 12.

[41] Youtao Gao, Zhicheng You, and Bo Xu. "Integrated Design of Autonomous Orbit Determination and Orbit Control for GEO Satellite Based on Neural Network". In: *International Journal of Aerospace Engineering* 2020 (2020).

[42] Neuraspace. URL: https://www.neuraspace.com/.

[43] James Reinders et al. *Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL*. Springer Nature, 2021.

[44] *Data Parallel C++ language*. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/data-parallel-c-plus-plus.html#gs.frlrcs.

[45] *ComputeCpp™*. URL: https://developer.codeplay.com/products/computecpp/ce/home/.

[46] *hipSYCL: Multi-backend implementation of SYCL for cpus and gpus*. URL: https://github.com/illuhad/hipSYCL.

[47] K. Jarrod Millman and Michael Aivazis. "Python for Scientists and Engineers". In: *Computing in Science & Engineering* 13.2 (2011), pp. 9–12. DOI: 10.1109/MCSE.2011.36.

[48] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. 2017.

Appendix A

Code Listings

A.1 Implementation of the VoiceHD [17] Application using the DPHDC Python Front-end

The implementation of the VoiceHD [17] application, using the DPHDC Python front-end is presented in Listing A.1. As mentioned in Section 3.1.5, the approach to use DPHDC using the Python front-end is almost identical to when using the C++ version, as the design and all features presented in Chapter 3 are carried over to the Python version. These similarities can be observed by comparing Listing A.1 with Listing 3.1, i.e., the implementation of VoiceHD using the C++ front-end of DPHDC and the Python front-end of DPHDC, respectively.

Listing A.1: Implementation of the VoiceHD [17] application using the Python DPHDC front-end.

```
1  from readDataset import readDataset
2  import pydphdc
3
4  VECTOR_SIZE = 10000
5
6  def generateRangeVector() -> list[int]:
7      to_return = []
8
9      for i in range(20):
10         to_return.append(i)
11
12     return to_return
13
14 def main():
15     train_data_labels = readDataset(FILE_PATH)
16     test_data_labels = readDataset(FILE_PATH)
17
18     intensity_representation = pydphdc.HDRepresentationInt(VECTOR_SIZE, pydphdc.full_level, ↩
           DEVICE_SELECTOR, generateRangeVector())
```

```
19      position_vectors = pydphdc.HDMatrix(VECTOR_SIZE, len(train_data_labels[0][0]), pydphdc.↩
            random, DEVICE_SELECTOR)
20
21      associative_memory = intensity_representation.encodeWithXOR(train_data_labels[0],↩
            position_vectors).reduceToLabelsBundle(train_data_labels[1])
22
23      encoded_test_entries = intensity_representation.encodeWithXOR(test_data_labels[0], ↩
            position_vectors)
24      accuracy = associative_memory.testModel(encoded_test_entries, test_data_labels[1], ↩
            pydphdc.hamming_distance) * 100
25
26  if __name__ == "__main__":
27      main()
```

A.2 Implementation of the Novel Encoder Alternative to HDNA [18] Encoder I

It is the objective of this appendix section to present the implementation of the proposed novel encoder used as an alternative to HDNA [18] encoder I. To this end, the implementation of an equivalent application to HDNA [18], using the proposed gene encoder, is presented in Listing A.2. It is relevant to mention that the `readDataset` function, present in lines 9 and 10, is, as the name implies, an auxiliary function developed with the objective of reading the dataset entries and corresponding labels.

Listing A.2: Implementation of an equivalent HDNA [18] application using the proposed novel encoder.

```
1  #include <dphdc.hpp>
2  #include <readExeInputs.hpp>
3  #include <ResultsHandler.hpp>
4  #include "readDataset.hpp"
5
6  int main(int argc, char **argv) {
7      int vector_size = 10000;
8
9      std::pair<std::vector<std::vector<char>>, std::vector<std::string>> train_data = ↩
           readDataset(FILE_PATH);
10     std::pair<std::vector<std::vector<char>>, std::vector<std::string>> test_data = ↩
           readDataset(FILE_PATH);
11
12     cl::sycl::queue q{SELECTED_DEVICE()};
13     std::vector<char> dna_bases = {'A', 'C', 'G', 'T'};
14     dphdc::HDRepresentation<char> hd_representation_dna_bases(vector_size, dphdc::↩
           vectors_generator::random, q, dna_bases);
15
16     dphdc::HDMatrix associative_memory = hd_representation_dna_bases.encodeWithBundle(↩
           train_data.first, dphdc::permutation::shift_right).reduceToLabelsBundle(train_data.↩
```

```
        second);
17
18    dphdc::HDMatrix encoded_test_entries = hd_representation_dna_bases.encodeWithBundle(↩
        test_data.first, dphdc::permutation::shift_right);
19    results_handler.success_rate = associative_memory.testModel(encoded_test_entries, ↩
        test_data.second, dphdc::distance_method::hamming_distance) * 100;
20
21    return 0;
22 }
```