DPHDC: A Data Parallel framework for Hyperdimensional Computing

Pedro Silva André
Instituto Superior Técnico, Universidade de Lisboa, Portugal
pedro.silva.andre@tecnico.ulisboa.pt

***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 paper, 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.**

***Index Terms*—Parallel Computing, Hyperdimensional Computing, Vector Symbolic Architectures, Machine Learning, Heterogeneous Systems.**

I. Introduction

RECENT research and real-world application of Machine Learning (ML) algorithms, particularly of those based in Neural Networks (NN) and Deep Learning (DL), has illustrated the great potential and benefits of highly accurate classifiers in a plethora of applications, from natural language processing to image recognition. However, the success of these methods comes at the cost of typically high computational demands and power/hardware intensive testing and especially training even when using highly optimized ML frameworks [1]. 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 Internet of Things (IoT) devices [2]. The current impracticality of offline speech recognition on portable devices is a clear example of this limitation.

Hyperdimensional Computing (HDC) [3], also known as Vector Symbolic Architectures (VSA) [4], is a brain-inspired, Turing complete computing framework [4] that has emerged as a promising, more efficient and one-shot learning alternative to traditional ML techniques for classification tasks [2]. HDC applications are based on vectors, usually binary or bipolar, of very high and fixed dimensionality, denominated as hypervectors [3]. 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 [3]. For classification purposes, the similarity between hypervectors is one of the most important figures of merit, which allows to compare different high dimensional vectors [2], [3]. Compared with traditional ML methods, HDC classification applications typically present acceptable accuracy with lower execution costs [2]. For this reason, many recent research works apply HDC for classification in different application domains, including natural language processing [5], [6], speech recognition [7], DeoxyriboNucleic Acid (DNA) sequencing [8], gesture recognition [9] and character recognition [10], [11].

With the recent surge in research of HDC as a classification method [12], 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 NN and 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 [1], [13].

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 [14], 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 Arrays (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 [15].

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, 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 ML classification applications on heterogeneous devices, is proposed. The library was developed using the C++ programming

language and the SYCL standard. 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 Central Processing Units (CPU) and Graphics Processing Units (GPU) up to FPGAs [16]. 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 design that facilitates the expansion of the proposed framework;
- Extendable framework that allows for hyper-optimization of proposed design and algorithms to any device.

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.

The remainder of this paper is organized as follows. Section II presents the HDC background. This includes data representation, vector generation, operation definition and similarity measurement. A brief exposure of previous work related to HDC-based classification applications and general use HDC frameworks is done in Section III. In Section IV, the DPHDC library is highlighted. The library's high-level design, features and implementation details are introduced, accompanied by code samples and a complete example. The presentation and analysis of results is done in Section V, followed by comparisons with related work. Finally, Section VI concludes the paper and gives directions for future work.

II. Hyperdimensional Computing Background

Hyperdimensional Computing models are based on the algebraic and geometric properties of high-dimensional spaces [3]. As previously referred, points in these hyperspaces are represented by hypervectors, i.e., vectors with a large and fixed number of dimensions [3]. Depending on the HDC model, such vectors can be composed of several data types, from binary or bipolar to integer or complex numbers [17]. Despite this diversity in hyperspaces, the fundamental concepts of all HDC models are based on the same principles [13], [17]. 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 [2]. As such, binary and bipolar hypervectors will be assumed for the remainder of this Section, 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 [3]. 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 [3].

As shown in Figure 1, the HDC classification methodology usually starts by generating base hypervectors. Base hypervectors, also called

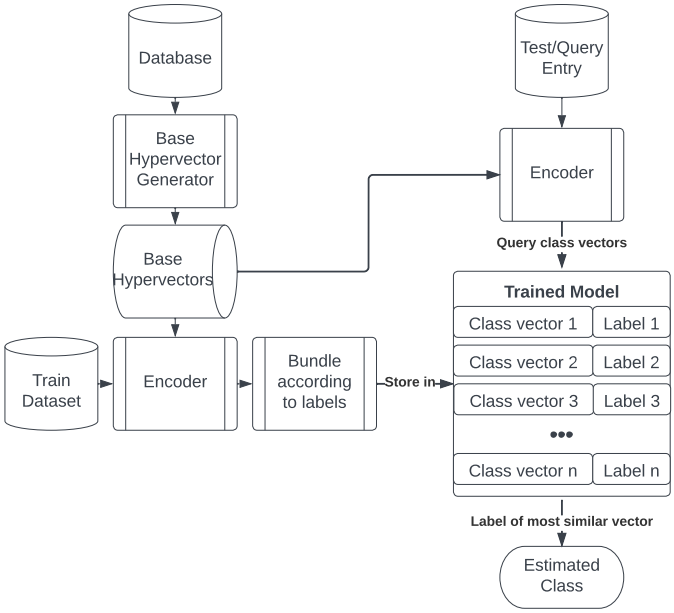

Fig. 1. Overview of supervised classification using HDC.

basis-hypervectors [18], 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 [5]. An encoder is then responsible for mapping each entry of the dataset to hyperdimensional space, using the aforementioned basis-hypervectors. 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 1). For testing/querying of the model to be possible, each test/unknown data 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 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 [3]. As can be inferred, the standard HDC classification methodology relies heavily on the encoder. Such a module, in order to take advantage of the mathematical properties of a hyperdimensional space [19], is composed of a combination of the three defined arithmetic operations of HDC: addition, multiplication and permutation [2], [3].

Addition ($+$ and $[.]$), also known as bundling, is 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 [3], [13]. For the bipolar case, the operation is defined by (1), where R and X_N are bipolar hyperdimensional vectors and $R(i)$ and $X_N(i)$ are, respectively, the element at position i of hypervectors R and X_N. 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 (1), 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 [2].

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

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 [2], [3].

Multiplication ($*$), also known as binding, is the element-wise exclusive OR (XOR) logical operation between two or more vectors [3], [6]. An example of this operation using binary hypervectors is presented in (2). As illustrated, 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 0 (or -1 in the case of bipolar vectors) in position i [3], [13].

$$\begin{aligned} X_1 &= 11110000 \\ X_2 &= 11001100 \\ X_3 &= 10101010 \\ \hline X_1 * X_2 * X_3 &= 10010110 \end{aligned} \quad (2)$$

Finally, a **permutation** (ρ) is a unary operator that randomly reorders the elements of a hypervector, generating a vector that is approximately orthogonal to the original one [3].

Encoders are application specific and depend directly on the type of data being mapped [20]. 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 [2]. The architecture and design of some specific encoders are briefly overviewed in Section III. For classification tasks, the goal of the encoder is to map similar data into similar hypervectors, or conversely dissimilar data into significantly distinct hypervectors [2], [3]. As also represented in Figure 1, this property is crucial for classification tasks because it is possible to encode known data types as class hypervectors (equivalent to training the model), which can then be compared with hypervectors encoded from an unknown data class, making it possible to infer the type of the unknown data (equivalent to testing the model) [2], [3]. 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 (3),

$$\cos(A, B) = \frac{A \cdot B}{|A||B|} = \frac{\sum_{i=1}^{d} A(i)B(i)}{\sqrt{\sum_{j=1}^{d} A(j)^2 \sum_{k=1}^{d} B(k)^2}} \quad (3)$$

where A and B are two hyperdimensional bipolar vectors with d elements, that exist in a space with d dimensions (typically, $d = 10000$) [6]. 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:

$$d_{Ham}(A, B) = \frac{1}{d} \cdot \sum_{i=1}^{d} (1 \text{ if } A(i) \neq B(i)), \quad (4)$$

where A and B are binary hypervectors, and d is the number of dimensions of both vectors [2]. 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.

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.

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 [3]. 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 [3]. 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 [3]. As a result, the generation of base hypervectors is usually randomly based [2].

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. 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.

III. Related Work

HDC and VSA models have evolved gradually since their initial ideas were proposed [19]. Nevertheless, these models have recently gained significant traction and attention from the ML learning scientific community [13], 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. In addition, a set of state-of-the-art HDC frameworks, including their advantages and limitations, are also overviewed in this Section.

A. HDC-based supervised classifiers

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 [2]. 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 [2]. 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 [2]. As mentioned in Section II, the encoding step is application and data specific and represents the crucial point where

pattern learning occurs [2], [20]. As a result, most of the variability between classification applications based on HDC comes from the encoder design [20].

In VoiceHD [7], a speech recognition application focused on the ISOLET dataset is proposed. 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. 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. 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 [7], [18]. 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.

VoiceHD was inspired by the approach taken in [9], where a gesture recognition application based on 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.

HDC-based language recognition is also tackled in [5]. 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. The N-gram hypervector (H_N) is encoded, as illustrated in equation (5),

$$H_N = \prod_{i=1}^{N}(\rho)L_1 * \prod_{i=2}^{N}(\rho)L_2 * ... * \rho L_{N-1} * L_N \tag{5}$$

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, all the N-gram vectors corresponding to a specific sentence are bundled to generate the hypervector that represents that sentence.

A very similar approach is taken by HDNA [8], 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.

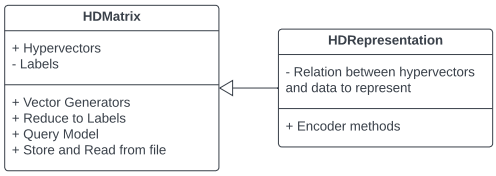

Fig. 2. Simplified UML diagram of the DPHDC library.

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. This reinforces that a general heterogeneous library for HDC-based classification could prove immensely valuable in speeding-up HDC research and application development, which is a specific topic tackled in this work.

B. Existing libraries and frameworks for HDC applications

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 [14], 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 [2].

To deal with these shortcomings, the proposed library 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.

IV. 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 [21].

To maximize hardware utilization across different architectures and minimize other performance bottlenecks, mainly related with data

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

memory accesses and communication, a unique design and optimized algorithms for HDC functions and operations are integrated in the proposed framework (as presented in Section IV-A). 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 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.

A. Design and Implementation

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 2, copes with this challenge by encapsulating two main classes: `HDMatrix` and `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 2, 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`.

As previously stated, binary and bipolar HDC models are the ones that show the most promise as a lightweight alternative to traditional ML techniques [2]. 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 [22].

To deal with the aforementioned potential memory bottlenecks, the `HDMatrix` and `HDRepresentation` classes use a SYCL boolean buffer with two dimensions to represent a collection of vectors. 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. Furthermore, a SYCL buffer variable is one that holds data that can automatically migrate between devices as needed. Since the buffer used has no association with any host data, data movement is minimized by allowing hypervector data to only exist on the accelerator device. The buffer class, through accessors, also facilitates the management of data dependencies and helps avoid data race conditions.

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 2), 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 II and exemplified in Section III-A.

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** and **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 III-A. 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.

Circular vectors were proposed in [18] 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 [18], i.e., any vector is closely related to its neighbours while being distinctly dissimilar to the vector that opposes it in the circle.

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 might negatively impact 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. 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`).

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.

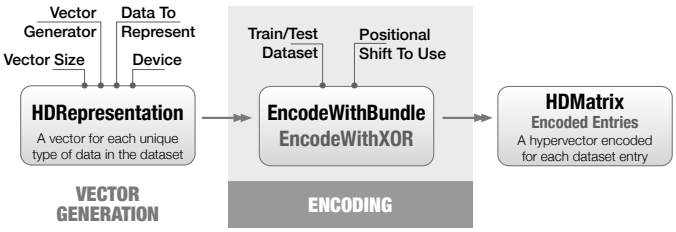

Fig. 3. Flow chart of the encoding of data using DPHDC permutation-based encoders.

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, 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 [2] 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.

As shown on Figure 4, the `encodeWithXOR` method is also overloaded to enable the encoding of data using positional vectors, like the encoder used by VoiceHD [7] and presented in Section III-A. 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.

All the presented encoder modules return a `HDMatrix` object containing one hypervector per dataset entry. As shown in Figure 6, in the case of training the model, all vectors (from "Encoded Train Entries" in Figure 6) 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 6, 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. Since model accuracy represents the success rate of the model, it can be defined as the ratio of correct estimations against the total number of queries performed.

Given the generic nature of these three encoder modules, combining them makes it possible to implement a wide range of encoders. As illustrated by Figure 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 said vector 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.

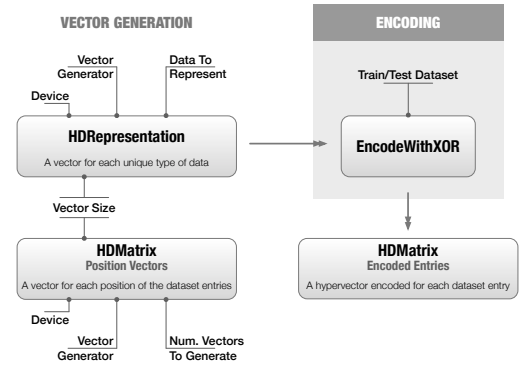

Fig. 4. Flow chart of the encoding of data using DPHDC positional-based encoder.

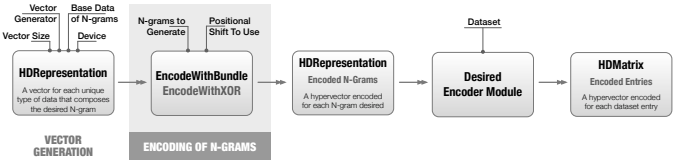

Fig. 5. Flow chart of a N-gram based encoder using DPHDC.

It is also important to note that any `HDMatrix` or `HDRepresentation` object can be stored in a binary file. 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.

As mentioned previously, the design of the library, presented up until now, was developed with the aim of maximizing parallel execution while remaining easy to use and versatile. By providing the encoder modules with all the data to encode simultaneously, it was possible to exploit as much parallelism as possible from all target architectures by using data-parallel SYCL kernels. As a result, some vector generator methods, all encoder methods, the `reduceToLabelsBundle` method and the query method are implemented using data-parallel SYCL kernels. The storage of vectors in a continuous memory space also improves parallel execution by exploiting memory locality.

As an example, the kernel implementation of the `encodeWithXOR` positional module is presented in Listing 1. Lines 02 trough 06 of Listing 1 are responsible for submitting the kernel to a SYCL queue (associated with an accelerator) and accessing buffer data through the use of accessors. This is followed by the beginning of the `parallel_for` kernel to be executed (line 08) and the declaration of the range of said kernel (line 07). 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 12). If the result of the operation is `true`, then one is accumulated (line 13). Otherwise, one is subtracted from the respective position in the accumulators variable (line 15). 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.

Listing 1. Main kernel code of the `encodeWithXOR` positional module.

```
1 using namespace sycl;
2 this->associated_queue.submit([&](handler &h) {
3     accessor acc_accumulators(buff_accumulators, h↩
```

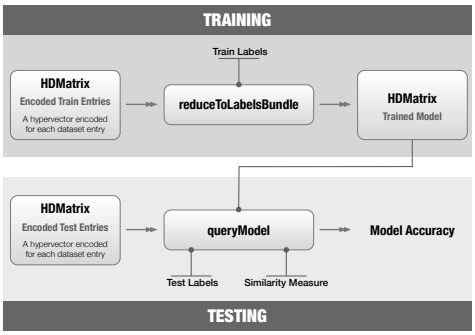

Fig. 6. Flow chart of the training and querying/testing classification methodologies using DPHDC.

```
        , read_write);
    accessor acc_data(buff_data, h, read_only);
    accessor acc_representation(this->vectors_buff
        , h, read_only);
    accessor acc_position_vectors(position_vectors
        .vectors_buff, h, read_only);
    range<3> r(n_entries_data, size_entry, 
        vector_size)
    h.parallel_for(r, [=](id<3> local_range) {
        size_t i = local_range[0];
        size_t j = local_range[1];
        size_t k = local_range[2];
        if (acc_representation[acc_data[i][j]][k] 
            ^ acc_position_vectors[j][k]) {
            acc_accumulators[i][k] += 1;
        } else {
            acc_accumulators[i][k] -= 1;
        }
    });
});

//Thresholding of accumulators back to binary (
    true or false)
```

Another optimization that is generally hidden from the user is that the `HDRepresentation` class contains a hash table that maps the base data to be represented by the base hypervectors to the index of the vector that represents it. 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.

As explained so far, all values needed to use and all values returned by the 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 [23]. 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 [7].

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.

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 [16]. 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 [2], 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.

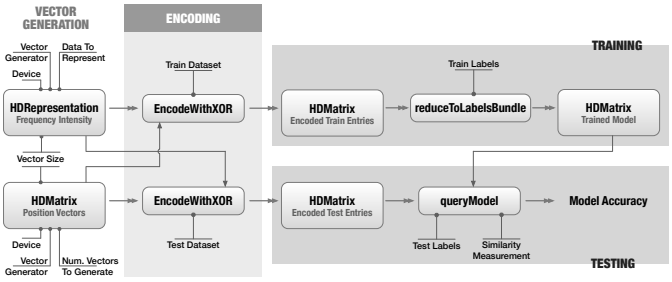

Fig. 7. Flow chart of the implementation of the VoiceHD application using DPHDC.

B. An application example

To better understand the DPHDC library's workflow, this Section presents a short overview of how the VoiceHD [7] application was implemented and is presented in Listing 2. Listing 2 is written in the C++ programming language. As mentioned previously, the Python front-end of the library is almost identical to the C++ variant. As a result, the same application developed using the Python programming language would follow the same steps.

As can be observed in Figure 7, the first step involves generating a level hypervector-based representation for the 20 frequency sub-ranges (lines 4 trough 7 of Listing 2) and a matrix with 617 random positional hypervectors, one for each frequency bin (line 9 of Listing 2).

Listing 2. VoiceHD [7] application implemented using DPHDC.

```
#include <dphdc.hpp>
using namespace std;
int main{
    int v_size = 10000;
    //Generating representation
    vector ints_represent = {-10, -9, ... , 9, 
        10};
    dphdc::HDRepresentation<int> 
        freq_inten_represent(v_size, full_level, 
        device, ints_represent);
    //Generating position vectors
    dphdc::HDMatrix position_vectors(v_size, 617, 
        random, device);

    vector<vector<int>> train_data = readTrainData
        (dataset_path);
    dphdc::HDMatrix encoded_matrix = 
        freq_inten_represent.encodeWithXOR(
        train_data, position_vectors);

    vector<string> train_labels = readTrainLabels(
        dataset_path);
    dphdc::HDMatrix trained_model = encoded_matrix
        .reduceToLabelsBundle(train_labels);
```

```
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 2). 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 2).

As indicated by Figure 7, using the labels provided with the ISOLET dataset (line 14 of Listing 2), it is possible to derive the trained model by bundling all vectors with the same label together (line 15 of Listing 2).

Finally, it is possible to query/test the model (lines 18 trough 20 of Listing 2) with all encoded test entries (encoded in lines 17 and 18 of Listing 2) to get the estimated label for each.

V. 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 III-A):

- **VoiceHD** [7] a speech recognition application;
- **European language recognition** [5];
- **HDNA** [8] N-gram based encoder, used for DNA sequencing;
- A binding positional encoder for recognizing the handwritten digits of the **MNIST** dataset, inspired by [11];
- **Hand gesture recognition** using EMG signals [9].

It is important to note that all applications are replicated in DPHDC as originally proposed, except HDNA [8], which encoder is implemented slightly differently from the one presented in Section III-A with the aim of improving its performance and accuracy. 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. This results in a slight increase in classification accuracy when applied to the empirical bats dataset, used to benchmark the application using DPHDC. Furthermore, all presented results of the European Language recognition example [5] consider N-grams of size 3 (trigrams).

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 subsections 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.

A. Scalability with vector dimensionality

To explore how the execution times and accuracy of DPHDC-based applications behave with vector dimensionality, the HDNA [8], VoiceHD [7], European Language Recognition [5] and MNIST [11] examples were executed using different vector sizes. Training and testing times are presented in Figures 8, 9, 10 and 11, while classification accuracies are shown in Figures 12, 13, 14 and 15, 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 [8] (Figure 8), VoiceHD [7] (Figure 9) and European language recognition [5] (Figure 10), both training and testing are faster on the GPU when vector sizes approach 10000, another indication that HDC algorithms 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 [2]. 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 8, 9, 10 and 11), 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 8). 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 [2]. 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.

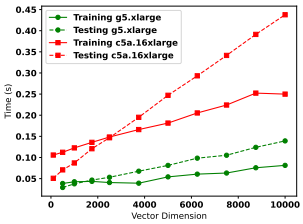

Fig. 8. Training and testing times of the HDNA example according to vector dimensionality.

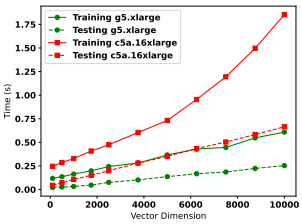

Fig. 9. Training and testing times of the VoiceHD example according to vector dimensionality.

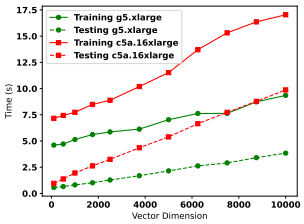

Fig. 10. Training and testing times of the European language recognition example according to vector dimensionality.

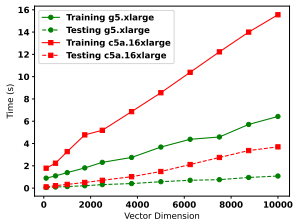

Fig. 11. Training and testing times of the MNIST example according to vector dimensionality.

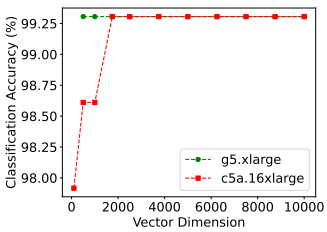

Fig. 12. Classification accuracy of the HDNA example according to vector dimensionality.

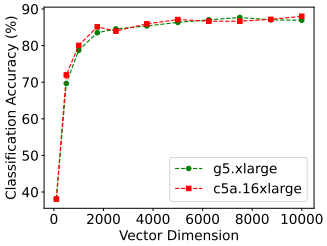

Fig. 13. Classification accuracy of the VoiceHD example according to vector dimensionality.

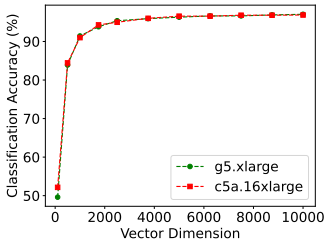

Fig. 14. Classification accuracy of the European language recognition example according to vector dimensionality.

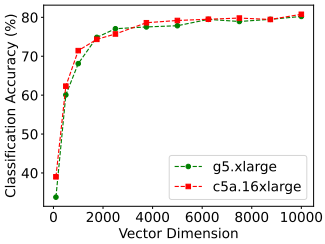

Fig. 15. Classification accuracy of the MNIST example according to vector dimensionality.

TABLE I
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

B. Comparison with other frameworks

Given the existence of the TorchHD library [14] that facilitates the implementations of HDC-based applications (as presented in Section V-B), it is essential to compare performances when targeting devices that are compatible with both libraries. To compare with the TorchHD library [14], the Language Recognition [5], VoiceHD [7] and MNIST [11] examples presented in Section III-A (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 I. As expected, the accuracies obtained on all applications running both on CPU and GPU are similar across both frameworks. 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 I 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.

C. FPGA implementation

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** [9] 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

TABLE II
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

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 II. Even though no execution times are presented in the original work [9], 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.

VI. Conclusion

In this manuscript, a SYCL-based open-source heterogeneous library to facilitate implementation and accelerate HDC-based classification applications was proposed. Through the efficient storage of information and movement of data coupled with effective exploitation of parallelism across multiple different architectures, DPHDC is up to 13x faster on CPU and 10x faster on GPU than currently available general-purpose and multi-device HDC frameworks. 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.

One main direction for future work is expanding the library to include more HDC model types. This research would require templating the buffer data type used to represent hypervectors and adjusting the implemented methods to be compatible with the new model being added. 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.

References

[1] A. Rahimi, S. Datta, D. Kleyko, *et al.*, "High-dimensional computing as a nanoscalable paradigm," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 9, pp. 2508–2521, 2017. DOI: 10.1109/TCSI.2017.2705051.

[2] L. Ge and K. K. Parhi, "Classification using hyperdimensional computing: A review," *IEEE Circuits and Systems Magazine*, vol. 20, no. 2, pp. 30–47, 2020. DOI: 10.1109/MCAS.2020.2988388.

[3] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, Jun. 2009, ISSN: 1866-9964. DOI: 10.1007/s12559-009-9009-8. [Online]. Available: https://doi.org/10.1007/s12559-009-9009-8.

[4] D. Kleyko, M. Davies, E. P. Frady, *et al.*, "Vector symbolic architectures as a computing framework for nanoscale hardware," 2021. DOI: 10.48550/ARXIV.2106.05268. [Online]. Available: https://arxiv.org/abs/2106.05268.

[5] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ser. ISLPED '16, San Francisco Airport, CA, USA: Association for Computing Machinery, 2016, pp. 64–69, ISBN: 9781450341851. DOI: 10.1145/2934583.2934624. [Online]. Available: https://doi.org/10.1145/2934583.2934624.

[6] A. Joshi, J. T. Halseth, and P. Kanerva, "Language geometry using random indexing," in *Quantum Interaction*, J. A. de Barros, B. Coecke, and E. Pothos, Eds., Cham: Springer International Publishing, 2017, pp. 265–274, ISBN: 978-3-319-52289-0.

[7] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, 2017, pp. 1–8. DOI: 10.1109/ICRC.2017.8123650.

[8] M. Imani, T. Nassar, A. Rahimi, and T. Rosing, "Hdna: Energy-efficient dna sequencing using hyperdimensional computing," in *2018 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI)*, 2018, pp. 271–274. DOI: 10.1109/BHI.2018.8333421.

[9] A. Rahimi, S. Benatti, P. Kanerva, L. Benini, and J. M. Rabaey, "Hyperdimensional biosignal processing: A case study for emg-based hand gesture recognition," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, 2016, pp. 1–8. DOI: 10.1109/ICRC.2016.7738683.

[10] D. Kleyko, E. Osipov, A. Senior, A. I. Khan, and Y. A. Şekerciogğlu, "Holographic graph neuron: A bioinspired architecture for pattern processing," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 6, pp. 1250–1262, 2017. DOI: 10.1109/TNNLS.2016.2535338.

[11] A. X. Manabat, C. R. Marcelo, A. L. Quinquito, and A. Alvarez, "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.

[12] D. Kleyko, D. A. Rachkovskij, E. Osipov, and A. Rahimi, "A survey on hyperdimensional computing aka vector symbolic architectures, part II: Applications, cognitive models, and challenges," *ACM Computing Surveys*, Aug. 2022. DOI: 10.1145/3558000. [Online]. Available: https://doi.org/10.1145%5C%2F3558000.

[13] D. Kleyko, D. A. Rachkovskij, E. Osipov, and A. Rahimi, "A survey on hyperdimensional computing aka vector symbolic architectures, part i: Models and data transformations," *ACM Comput. Surv.*, May 2022, Just Accepted, ISSN: 0360-0300. DOI: 10.1145/3538531. [Online]. Available: https://doi.org/10.1145/3538531.

[14] M. Heddes, I. Nunes, P. Vergés, D. Desai, T. Givargis, and A. Nicolau, "Torchhd: An open-source python library to support hyperdimensional computing research," 2022. DOI: 10.48550/ARXIV.2205.09208. [Online]. Available: https://arxiv.org/abs/2205.09208.

[15] J. Kang, B. Khaleghi, T. Rosing, and Y. Kim, "Openhd: A gpu-powered framework for hyperdimensional computing," *IEEE Transactions on Computers*, pp. 1–1, 2022. DOI: 10.1109/TC.2022.3179226.

[16] M. Wong, N. Liber, S. Bassini, *et al.*, *Sycl - c++ single-source heterogeneous programming for acceleration offload*, Jan. 2014. [Online]. Available: https://www.khronos.org/sycl/.

[17] K. Schlegel, P. Neubert, and P. Protzel, "A comparison of vector symbolic architectures," *Artificial Intelligence Review*, vol. 55, no. 6, pp. 4523–4555, Aug. 2022, ISSN: 1573-7462. DOI: 10.1007/s10462-021-10110-3. [Online]. Available: https://doi.org/10.1007/s10462-021-10110-3.

[18] I. Nunes, M. Heddes, T. Givargis, and A. Nicolau, *An extension to basis-hypervectors for learning from circular data in hyperdimensional computing*, 2022. DOI: 10.48550/ARXIV.2205.07920. [Online]. Available: https://arxiv.org/abs/2205.07920.

[19] P. Kanerva, *Sparse distributed memory*. MIT press, 1988.

[20] E. Hassan, Y. Halawani, B. Mohammad, and H. Saleh, "Hyper-dimensional computing challenges and opportunities for ai applications," *IEEE Access*, vol. 10, pp. 97 651–97 664, 2022. DOI: 10.1109/ACCESS.2021.3059762.

[21] K. J. Millman and M. Aivazis, "Python for scientists and engineers," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 9–12, 2011. DOI: 10.1109/MCSE.2011.36.

[22] P. Kanerva, "Computing with high-dimensional vectors," Stanford EE Computer Systems Colloquium, 2017. [Online]. Available: https://web.stanford.edu/class/ee380/Abstracts/171025.html.

[23] W. Jakob, J. Rhinelander, and D. Moldovan, *Pybind11 – seamless operability between c++11 and python*, https://github.com/pybind/pybind11, 2017.