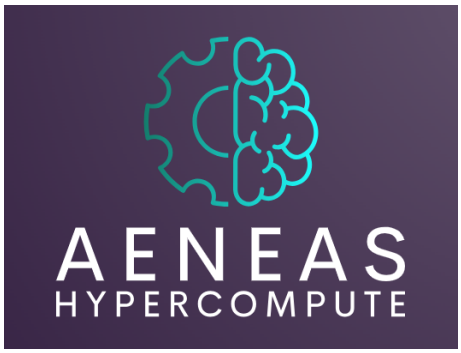


Aeneas HyperCompute Reference Manual

Version 3.0.0

Marco Angioli[1], Saeid Jamili[2]

[1]Contact: marco.angioli@uniroma1.it
[2]Contact: saeid.jamili@uniroma1.it

Contents

1 The Aeneas Hypercompute framework

The Aeneas Hypercompute framework offers a streamlined and automatic workflow for instantiating, managing, and evaluating Hyperdimensional Computing models—both in software and hardware. It allows users to quickly and easily test performance, execution time, resource usage, and energy consumption for different model configurations and design choices. The framework is highly customizable and supports all the most common techniques adopted in the literature for performing classification tasks. Aeneas is open source, available on Github and in constant evolution to provide users with a very simple and effective environment for testing and exploring SOA solutions in HDC.

Hyperdimensional Computing (HDC) is an emerging computing framework that mimics the biological brain by using high-dimensional distributed representations and is particularly well suited for edge devices since it offers several advantages in terms of latency, energy efficiency, robustness against noise and hardware faults, etc.

These models, in fact, are based on the blessing of the dimensionality property of the hyperspaces and just use simple arithmetic operations on vectors to combine different informations and perform complex tasks such as classifications, clustering and regressions.

However, when creating a Hyperdimensional Computing model, there are numerous design choices that can impact accuracy as well as hardware parameters such as energy consumption, area occupation, and execution time. For instance, the accuracy of a hypervector can vary slightly depending on the chosen dimension, whether it is 10.000 or 5.000, or the encoding technique employed. Unfortunately, these design choices are often made ad-hoc in the literature, following the state-of-the-art approaches, without proper justification or understanding of their impact on hardware. Consequently, hardware implementations frequently adopt the same design choices regardless of the specific classification problem or available resources. This lack of customization can lead to suboptimal hardware realizations.

Only two works are available in the literature for partial-automatic hardware model realizations and also these allow the user only to specify the vector size, area and power constraints without any freedom on sparsity, internal techniques used to encode, clip, train, retrain and perform the inference.

This work aims to fill this gap in the literature by providing a new complete, automatic and open-source framework that enables simple, fast and extremely customizable deployment of HDC models in software and hardware. It allows users to easily test and evaluate the impact of various design choices on hardware parameters. In doing so, users can discover and identify the optimal options for their specific classification problem. As an automatic tool, it provides both novice and expert designers with a useful resource. Designers can leverage the framework to efficiently deploy their desired HDC model on FPGAs or simply and rapidly discover the best design choices for a customized implementation.

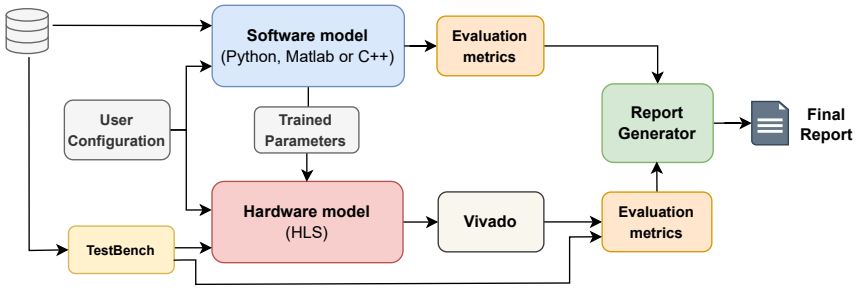


Figure 1: Overview of the Aeneas Hypercompute framework

The project's structure is shown in Fig 1 and is organized as follows. The user insert any dataset to be processed in the namesake folder. An intuitive and user-friendly graphical interface is provided to enable users to effortlessly specify all configuration details for the target HDC model. The system automatically generates all the configuration files necessary by the following entities. The HDC model is instantiated both in software and hardware environments, with accuracy evaluations conducted in each context. The hardware model is simulated, tested and synthesized. During this phase, crucial data encompassing energy consumption and resource requirements is collected. All accumulated results are subsequently relayed to a dedicated report-generation module. This module creates an HTML-based report incorporating the following information:

- Model configuration details;
- Duration of test;
- Model Accuracy (for both software and hardware implementations);
- Required Hardware resources;
- Energy consumption;

- Average execution time per classification.

This comprehensive report serves as a holistic summary of the evaluation process, encapsulating key insights into the performance and characteristics of the HDC model across various dimensions.

As previously mentioned, the entire process is fully automated, eliminating the need for users to engage in any coding or manual interventions. It's worth highlighting that this open-source project provides an extensive collection of software and hardware resources. In the subsequent sections, we will provide a concise overview of these offerings.

1.1 Software HDC Library

We provide an extensive software library that is fully compatible with Python, C++, and Matlab. This library equips users with the capabilities to effortlessly handle hypervectors, execute arithmetic operations within the hyperdimensional realm, and effectively tackle classification challenges using an approach that is both intuitive and adaptable.

Table 1: Explorable Design Space in the HDC software library

Parameters	**Explorable space**
HV Dimension	Any, ex:[1.000, 2.000, 5.000, 10.000]
HV DataType	[Binary, Bipolar]
HV Density	[Dense, Sparse]
HV Sparsity	Any, ex: [0.01, 0.02, 0.05, 0.1, 0.2]
Similarity Measure	[Hamming, Cosine, DOTP]
LevelHVs Mode	[Linear, Approx. Linear, Thermometer, Non-linear]
Number of LevelHVs	Any, ex: [5, 10, 30, 50, ...]
Spatial Encoding Mode	[Record-based, N-gram based]
Clipping Technique	[Binary, Tripolar, Quantized, Integer]
Class HVs	[k-ClassHVs, One ClassHV]
Retraining ClassHV	[Yes, No]
Retrain Epochs	Any, ex: [5, 10, 20, 50]
Learning Rate Mode	[Data-dependent, Iteration Dependent, Hybrid]

This software package boasts an exceptionally flexible class, allowing users to assess the HDC model's performance under various configurations. The supported features are summarized in Table 1. In the following, we briefly explain what can be done with the library.

- **HV Generation:** You can specify any dimensionality for the Hypervectors, enabling flexibility in their design.
- **HV Data Type:** Starting from the hyperspace type, you can choose between binary or bipolar representations.
- **HV Density:** Hypervectors can be either dense or sparse, offering versatility in how data is stored.
- **HV Sparsity Factor:** You can control the density of ones within the Hypervectors, allowing fine-tuned adjustments.
- **Arithmetic in the Hyperspace:** All basic operations like binding, bundling, and permutation are supported for each HV type.
- **Similarity Measures:** You have the choice of using the Hamming distance, dot product, or cosine similarity for measuring similarity.
- **LevelHVs Generation:** You specify the number of levels, minimum and maximum values for the addressed dataset, and encoding technique for generating LevelHVs. Techniques include Linear encoding, Approximately Linear Encoding, Thermometer Encoding, and Non-linear Encoding.
- **Spatial Encoding:** Two primary techniques for spatial encoding are available: Record-based encoding (BaseHV+LevelHV) and N-gram based encoding (LevelHV+Permutation).
- **Temporal Sequence:** If needed, a temporal encoding based on the n-gram approach can be included in the encoding process.
- **HV Clip:** Different clipping strategies are available for encoding and classification phases, such as Binary clipping, Bipolar clipping, Quantized clipping, and No clipping.
- **Training Procedure:** You can train the model on a specific dataset by specifying the dataset, encoding technique, number of features, and classes. The routine encodes each feature vector as per your specifications and performs training tasks, superimposing HVs of the same class. ClassHVs can then be clipped using various techniques, with resulting ClassHVs displayed and stored.
- **Retraining Protocols:** To enhance accuracy, the model can be retrained for a set number of epochs. Various learning rate techniques are available, including learning rate specification and learning rate decay options based on data or iteration count.

- **Test or Inference:** For testing or inference, you specify the test dataset, and the routine automatically performs iterative associative searches, classifying each pattern using the most similar ClassHV.

All the details about the Python library functions and their input parameters are described in section **??**.

1.2 HW Library

In addition to the software library, we also offer a comprehensive hardware library designed for High-Level Synthesis (HLS). This library provides all the configuration options available in the software library, along with additional settings for hardware parallelism and HV storage in memory. These hardware-specific options will be presented and explained in a dedicated section that follows. All the available configuration options for the hardware library are conveniently summarized in Table 2.

Table 2: Explorable Design Space in the HDC Hardware library

Parameters	**Explorable space**
HV Dimension	Any, ex:[1.000, 2.000, 5.000, 10.000]
Data Parallelism (FRAME)	Any in range [1, ..., *HD_DIM*]
Feature Parallelism	Any in range [1,n_features]
Class Parallelism	Any in range [1,n_classes]
HV DataType	[Binary, Bipolar]
HV Density	[Dense, Sparse]
HV Sparsity	Any, ex: [0.01, 0.02, 0.05, 0.1, 0.2]
Similarity Measure	[Hamming, Cosine, DOTP]
BaseHVs	[Memory, LFSR, COMB]
LevelHVs Mode	[Linear, Approx. Linear, Thermometer, Non-linear]
Number of LevelHVs	Any, ex: [5, 10, 30, 50, ...]
LevelHVs	[Memory, LFSR, COMB]
Spatial Encoding Mode	[Record-based, N-gram based]
Temporal Encoding Mode	[Yes, No]
Temporal Encoding Window	Any, ex: [2,3,5]
Clipping Technique	[Binary, Tripolar, Power Of Two, Quantized, Quantized Power Of Two, Integer]
Class HVs	[k-ClassHVs, One ClassHV]
Retraining ClassHV	[Yes, No]
Retrain Epochs	Any, ex: [5, 10, 20, 50]
Learning Rate Mode	[Data-dependent, Iteration Dependent, Hybrid]

1.3 Customizable Hardware Flow

The Aeneas HyperCompute Platform presents an inclusive, automated, and open-source framework that streamlines the effortless, rapid, and deeply customizable deployment of Hyperdimensional Computing (HDC) models onto hardware architectures. Parallel to the software realm, our hardware generation flow empowers users with the ability to fine-tune various configuration options for their HDC models. This guarantees a seamless evolution from software-defined prototypes to their tangible hardware counterparts, affording users the opportunity to assess how different design choices can impact vital hardware parameters, such as: energy consumption, area occupation, and execution time.

To accomplish this, the framework employs a comprehensive High-Level Synthesis (HLS) project that takes the user-specified options from the configuration file and automatically generates the corresponding HDC model in hardware.

1.3.1 Partial Processing in the Aeneas framework

The inherent parallelism in HDC classification problems naturally lends itself to hardware implementations. Each element within the HVs operates independently from the others, allowing for a potential full parallel processing. However, HDC is often utilized in embedded systems and edge devices, where resource constraints such as limited area and power consumption must be considered.

The Aeneas framework addresses this challenge by introducing additional flexibility to hardware implementations. This is achieved through a *partial processing* mechanism, which empowers users to define the degree of hardware parallelism at various stages of the classification task, as shown in Table 3.

Table 3: Hardware Parallelism Degrees

Parameters	**Explorable space**
Element Parallelism (FRAME)	Any in range $[1, D]$
Feature Parallelism	Any in range $[1, N]$
Class Parallelism	Any in range $[1, K]$

1.3.2 Element parallelism

Since the dimensionality of the HV (D) is commonly high, in resource-constrained environments, it is often impractical to replicate hardware components D times for full parallel processing. Our framework introduces the *FRAME* parameter that allows users to specify the number of HV elements processed

in parallel. This feature enables users to trade off processing speed for hardware resource utilization and energy consumption. In Figure 2, we illustrate an example of FRAME parallelism applied to a record-based encoding technique with four features and binary HVs. As highlighted, all the hardware required to produce one HV's element is replicated FRAME times. This approach represents a compromise between processing one element at a time (as in a sequential approach) and full parallel processing, offering the flexibility to tailor hardware parameters to meet specific problem requirements.

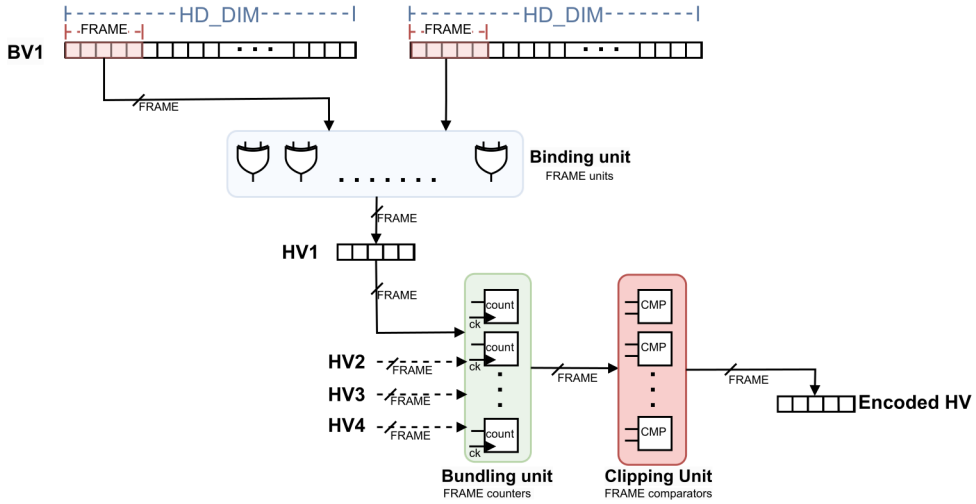


Figure 2: FRAME parallelism. FRAME elements are processed per time

As shown in Figure 2, each iteration processes FRAME elements of the hypervector. FRAME elements are taken from the base vector (BV) and the corresponding level vector (LV), and then they are binded, bundled, and clipped. These operations are performed by dedicated hardware units that, in which, thanks to the partial processing only FRAME functional units are instantiated, rather than HD_DIM. The total number of iterations is equal to the hypervector size divided by the FRAME size. So, the trade-off between FRAME size and the number of iterations is as follows:

- A smaller FRAME size requires less hardware resources and less energy consumption, but it increases the number of iterations;
- A larger FRAME size requires more hardware resources, but it reduces the execution time.

The optimal FRAME size depends on the specific application and the available hardware resources. This level of flexibility empowers users to make informed trade-offs between execution speed and hardware resource

usage, adapting the framework to their specific application requirements. Note that, in the previous example, we have supposed to have the BV and the LV stored in memory and simply take FRAME elements from them. However, as will be discussed in the following sections, the Aeneas Framework also allow generating these vectors on-fly or recovering them through a dedicated combinational logic. In both these cases, just FRAME elements would be generated per iteration.

1.4 Feature Parallelism

Within the Aeneas framework, users have the ability to determine the concurrent processing of features. While the inherent parallelism of HDC could theoretically replicate processing logic for all features simultaneously, practical constraints arise, especially in embedded environments dealing with datasets containing numerous features (e.g. ISOLET's 617 features).

In Aeneas, users are granted the flexibility to process features in multiple ways: sequentially, one at a time, or in parallel. They can specify any desired number within the $[1, N]$ range. Each concurrently processed feature handles FRAME elements at a time.

1.5 Class Parallelism

Furthermore, our framework permits users to specify the number of classes compared in parallel during the inference phase, a vital feature when dealing with intricate multi-class datasets. Like the feature parallelism, users can compare the query HV with each class sequentially, one at a time, or in parallel, specifying any number within the $[1, K]$ range. Each concurrently compared class also processes FRAME elements at a time.

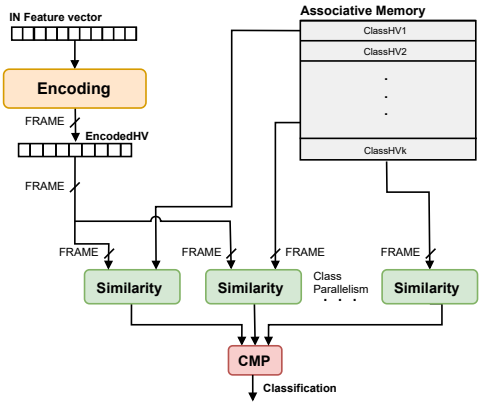


Figure 3: Class Parallelism applied to a Record-based encoding

1.6 Partial Processing during inference

In resource-constrained contexts, it is often desirable to perform the training procedure offline, on software, and then deploy only the trained model on the target device. In the case of hyperdimensional computing (HDC), this means exporting the base vectors (BVs), level vectors (LVs), and class hypervectors (ClassHVs) produced by the training procedure (for any further info on the training procedure check the corresponding Section). These vectors can then be reused to encode and perform the associative search during inference. The Aeneas framework provides three different ways to obtain the BVs, LVs, and ClassHVs during inference:

- Store the vectors in an external memory and read them (FRAME elements at a time, if desired). This is the simplest approach, but it can be inefficient if the vectors are large or if the external memory is slow;
- Generate the vectors on the fly using random number generators and dedicated logic;
- Retrieving the hypervectors from combinatorial logic tables.

While the first option is trivial, this section delves into an elaborate explanation of the operational principles behind the latter two methods.

1.6.1 Storing the HVs in an external memory

The easiest and most common way to recall and use the trained HVs during the inference procedure, is storing them into external memories. At each iteration, when the system receive a new data as input, FRAME elements of the corresponding BV and LV are read from the memory and they are combined to produce FRAME elements of the encoded HV. Finally, another access in memory is performed to read the FRAME element of each class and perform the associative search. However, the memory requirements of this approach linearly increase with the number of features, the number of classes and the *HD_DIM*. Table **??** summarizes the required memory size as a function of these parametes. For example, the ISOLET dataset that shows 617 features, 30 levels and 26 classes will require a total of 6.730.000 elements store in memory, making this approach unfeasible.

1.7 Generating HVs on-fly through random generators

Another possibility provided by the Aeneas framework is exploiting Linear Feedbak Shift Register (LFSR) random generators and dedicated logic to

HVs	Required memory
BaseVectors	*DS_FEATURE_SIZE* $\times$ *HD_DIM*
LevelVectors	*HD_LV_LEN* $\times$ *HD_DIM*
ClassHVs	*DS_CLASS_SIZE* $\times$ *HD_DIM*

Table 4: Caption

produce the required HVs on the fly, during inference, rather than storing them in memory.

As described in detail in the theory documentation, in HDC, the basic independent elements/concepts in the problem are mapped in random HVs. This is the case for all the BaseHVs in the record-based encoding procedure, and of the first and last LevelHVs in the case of approximately linear encoding.

Usually, as we already discussed, all the HVs are generated during train, exported and saved in memory. However, for the BaseHVs, for example, this would require storing *DS_FEATURE_SIZE* $\times$ *HD_DIM* elements in the memory. The Aeneas framework proposes an original method for reproducing these vectors on-fly, during inference. While training the model, all the random HVs are generated using LFSR units capable of generating BV_RND_GEN_W_BITS bits per iteration. The LFSRs are initialized with a random seed and, then, they are randomized at each iteration, producing random HVs. At the end of training, the adopted random generators' configuration is exported csv files and used to instantiate the LFSRs in hardware. No HVs are stored in memory, they are simply reproduced on fly through this architecture.

The Aeneas framework offers an original approach that harnesses Linear Feedback Shift Register (LFSR) random generators and specialized logic to dynamically create the necessary HVs during inference, on-fly. This contrasts with the conventional method of precomputing and storing HVs in memory.

As elaborated in the theoretical documentation, in the context of HDC, the basic independent elements or concepts within the problem domain are represented by random HVs. This is the case for all the BaseHVs in the record-based encoding procedure, and of the first and last LevelHVs in the case of approximately linear encoding.

Usually, as we have previously discussed, all HVs are generated during training, and subsequently saved in memory for utilization during inference. However, for instance, in the case of BaseHVs, this approach necessitates reserving memory for a substantial number of elements—amounting to *DS_FEATURE_SIZE* $\times$ *HD_DIM*. To address this problem, the Aeneas framework introduces a novel method for generating these vectors on-fly, during inferece.

Throughout the model's training, all the requisite random HVs are gener-

ated using specialized LFSR units capable of producing BV_RND_GEN_W_BITS bits per iteration. These LFSRs are initiated with random seeds, and subsequently randomized at every iteration, generating random HVs.

At the end of the training, the specific configuration of the adopted random generators is exported to CSV files and used to instantiate the LFSRs in the hardware implementation. All the random HVs are generated through LFSRs and, then, eventually elaborated through dedicated logic (as for generating intermediate-level HVs in the approximate encoding).

Importantly, this innovative architecture obviates the need to store HVs in memory; instead, they are dynamically reproduced during inference, as and when required.

Table **??** reports a comparison between the memory requirements of the standard approach and the resources required by LFSR.

This approach effectively optimizes memory usage while maintaining the capability to generate the essential HVs for accurate inference using the HDC model.

1.8 Retrieving HVs through combinational logic

The Aeneas framework also provides users with an innovative method for retrieving the HVs used in train in an easy, fast and convenient way. This technique is based on the use of combinational logic and, in particular, on the synthesis of truth tables. During the train procedure, we generate *DS_FEATURE_SIZE* BaseHVs, *HD_LV_LEN* LevelHVs and DS_CLASS_SIZE ClassHVs. Performing inference using the partial processing technique will require FRAME elements of each of these HVs per time. Let's suppose that we are performing the train offline, using an *HD_DIM* = 20 and that we aim to deploy the model in hardware, using a FRAME size equal to 4. Knowing this information, we can factorize the HVs in $HD\_DIM/FRAME$ vectors at the end of the training. These vectors can be, then, uniquely associated to two indices:

- the index of the frame (that will go from 0 to $(HD\_DIM/FRAME - 1)$)
- an index specific to the HV type. This will be:
  - the feature ID for BaseHVs;
  - the quantization index for the LevelHVs;
  - the class number for ClassHVs.

in this way, each frame window of a given HV will be uniquely encoded with two indexes. An example of this on LevelHvs is reported in Fig. 4, showing how LV0 can be fragmented and indexed.

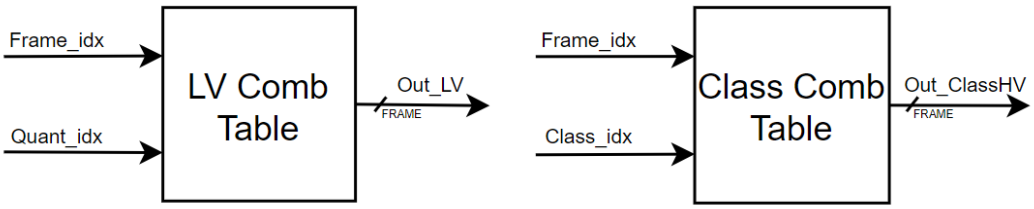
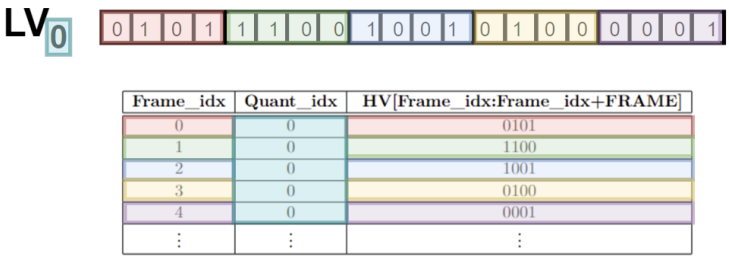


Figure 5: Combinational block units for retrieving HVs



Frame_idx	Quant_idx	HV[Frame_idx:Frame_idx+FRAME]
0	0	0101
1	0	1100
2	0	1001
3	0	0100
4	0	0001
⋮	⋮	⋮

Figure 4: Retrieving HVs through combinational logic: comb table applied to one example LevelHv

After the factorization, the table is given as input to a synthesis tool (in this case, Vivado) and synthesized in hdl. The resources required to represent the whole HVs with this approach are surprisingly low and are reported, for same example cases, in Table 5. The block diagram of the comb units so created is reported in figure 5

Table 5: Dataset: CARDIO, 30 levels, 4 classes

DIM	FRAME	#LUTs ClassHVs	Memory ClassHVs	#LUTs LevelHVs	Memory LevelHVs
[-1m] 1000	50	65	500B	280	3.75 MB
10000	50	664	5 KB	2728	37.5 MB