POLITECNICO DI TORINO

Master's degree course in Electronic Engineering

Master's Degree Thesis

An Hyper Dimensional Classifier for Dynamic Vision Sensors

Relatori:
Prof. Marco Vacca
Prof. Guido Masera
Ing. Fabrizio Ottati

Candidato:
Raffaele Di Placido

July 2022

Aknowledgements - *Ringraziamenti*

English Version
With these few lines I want to thank all the people who have always been by my side during this journey. First of all, I want to thank my relatives, in particular my parents, my siblings, my uncles, my cousins and my two grandmothers.
Furthermore, i would like to thank ph.D student Fabrizio Ottati for guiding and helping me within this thesis.
Last but not least, a special thank goes to my friends, who have always raised me up after every dark moment I went through.

Versione Italiana
Con queste poche righe vorrei ringraziare tutte le persone che mi sono state accanto in questo lungo periodo.
Ringrazio innanzitutto i miei parenti, dai miei genitori, ai miei frateli, i miei zii, le mie cugine e le mie due nonne.
Ringrazio il dottorando Fabrizio Ottati per avermi guidato ed aiutato in questo percorso di tesi.
Un ringraziamento speciale ai miei amici, che mi hanno risollevato nei momenti più bui.

Summary

Hyperdimensional computing (**HDC**) is a neural inspired computing paradigm derived from the cognitive model proposed by Pentti Kanerva in 1988.
Hyperdimensional computing consists in representing data through **pseudo-random** ultra-wide vectors, referred as **hypervectors**, with independent and identically distributed (i.i.d) components. A typical dimension (D) of an hypervector is D = 10000. The high dimensionality comes from the attempt to replicate the human brain complex behavior which uses billions of synapses and neurons.
Hypervectors can be either **binary**, i.e. each of their dimensions is represented as a bit that can take a single value between '1' or '0', or **non-binary**, if their elements are integers or floating point.
In the binary case, an hypervector is defined as dense if the number of '1' and '0' is the same, while an hypervector is defined as sparse if '1'-bits are present with a lower percentage than '0'-bits.
The main advantages of Hyperdimensional computing are **robustness** and **efficiency**. Indeed, a vector with an higher dimensionality can tolerate an higher number of corrupted bit without losing information.
As regards efficiency, data represented with hyperdimensional computing are combined using a set of three simple operations: addition, multiplication (or **bitwise xor** for binary cases) and permutation, which require lower hardware resources. Hyperdimensional computing is employed to compute similarity between entities. In particular, different entities are associated to different hypervectors in the hyperdimensional space.
To map data into hypervectors, an **encoding phase** is needed. There are two main encoding methods, the **record-based method** and the **N-gram based** method. Both of them require a memory called **Continuous Item Memory** (**CiM**) which stores **Level Hypervectors** (**L**), i.e. hypervectors which represent the value that an input variable can assume.
During the **training phase** of the classifier, data from the same class are encoded and then combined together in order to create a **class hypervector**. Class hypervectors are stored in the **Associative Memory** (**aM**). During the **test phase**, a **query hypervector** is compared to all the class hypervectors inside the aM, which gives as a result the class with the highest similarity to the input vector.

Researches on human brain functionality inspired not only cognitive algorithms, but also new types of sensors. In this thesis, particular attention is given to a type of bio-inspired sensors called **Event cameras**.
Differently from standard frame-based cameras, images with event cameras are not captured at a constant frame. Instead, event cameras report changes in brightness in asynchronous way, which are referred as **events**. This results in acquiring images with low latency, high temporal resolution (μs), and high input dynamic (140 dB).
In this thesis, an hardware accelerator based on hyperdimensional computing is implemented to classify images acquired with event cameras, also known as **Dynamic Vision Sensors** (**DVS**).The implemented classifier was developed in collaboration with ing. Fabrizio Ottati. The proposed HDC model uses dense and binary hypervectors with dimension D = 8192. 8 levels of quantization are used for input data, i.e. values of data are discretized in 8 levels, all represented with 8 Level Hypervectors. A record-based approach is used for the encoding phase.
To reduce the dimensions of the Continuous Item Memory, a single **Seed Hypervector** is manipulated to obtain the other Level Hypervectors, instead of storing them in the CiM.
The design is validated on the **N-MNIST dataset**, while features of images are represented with **istograms of Averaged Time Surfaces** (**HATS**).
The HATS system used to extract features from the dataset, was implemented in software by ing. Fabrizio Ottati.
The whole HDC model is at first implemented in software, to validate the behavior of the model and to obtain first results.
After software validation, the RTL of the design is described in HDL and simulated using ModelSim simulator.
Only the encoder and the inference modules of the model are implemented in hardware. The training phase of the classifier is instead implemented in software. The training in software generates the class hypervectors which are then stored in the Associative Memory inside the Inference Module.
Since not all architectures can handle the whole hypervector given the high dimensionality, the hardware design is serialized in order to process parts of hypervectors at different times. Hence, the design is configurable, i.e. the number and the size of the hypervector chunks can be user defined. In particular, the model can be configured to process hypervectors divided from 8 parts with 1024 bits each, to 1024 parts with 8 bits each.

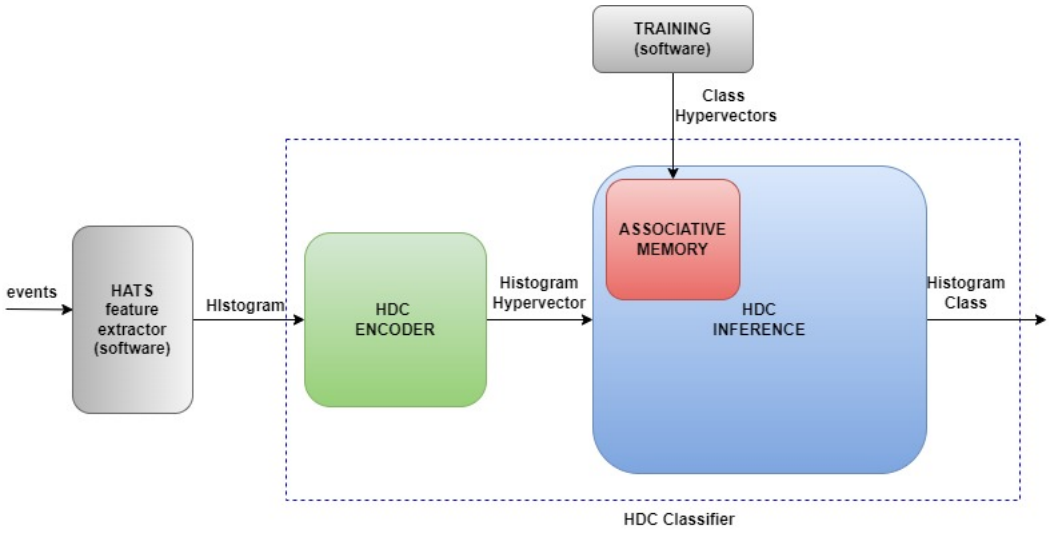


Figure 1: Overview on the Implemented Design

The design is also validated on FPGA using the **proFPGA prototyping system**. The target FPGA on which the design is implemented is the **Xilinx Virtex-7 2000T FPGA**. Thanks to Xilinx Vivado software, measurements of power, latency and resources utilization were extracted from the Xilinx Virtex-7 2000T FPGA with four different configuration of the design: 64x128, 32x256, 16x512, 8x1024. For all the configurations, an accuracy of 83% is obtained on the N-MNIST dataset.

Configuration	Time (ms)	Dynamic Power (W)	Static Power (W)	Total Power (W)
64x128	3.7	0.033	1.373	1.40
32x256	1.92	0.055	1.373	1.428
16x512	1.02	0.099	1.373	1.472
8x1024	0.57	0.181	1.377	1.558

Table 1: Measurements of required time and power for the inference of a single sample in each of the four configuration. The target FPGA is the Virtex 7 2000T. The clock period is 10 ns.

As can be seen, a configuration with a high number of parts with a reduced width requires a longer time to predict the class of a single sample, but it consumes less dynamic power.
The four configurations were also implemented on FPGAs from different families, in order to assess the low resources requirements of the hyperdimensional computing. In particular, the used FPGAs were: Spartan-7 xc7s25c, Spartan-7 xc7s100f, Artix-7 xc7a15T, Artix-7 xc7a200t, Kintex-7 xc7k70t, Kintex-7 xc7k480t, Virtex-7 xcv585t and Virtex-7 xc7vx1140t.

Contents

List of Figures

List of Tables

Chapter 1

Hyperdimensional Computing

1.1 Introduction on Hyperdimensional Computing

Hyperdimensional Computing (**HDC**) is a **paradigm** developed by **Pentti Kanerva** [14] based on researches on **brain capability**. As a matter of fact, the human brain and a standard computer have different approaches at the same tasks. While a computer is optimized to excel in **computationally heavy routine**, the human brain is much slower in calculation but flexible in **learning** activities. Hence, **brain-like computing** can be exploited in **cognitive operations**, such as **classifying data**.
Even if traditional computers and brains have different capabilities derived from their different structure, the construction of a brain-like architecture is not necessary to obtain a more "intelligent" system. However, studying the brain structure is fundamental to extract a cognitive model of computation.
Hyperdimensionality comes from the human brain structure composed of billions of **neurons** and **synapses**, which suggest that a large number of signals is fundamental for efficient cognitive functions. For this reason computing is done with words of nearly **10000 bits** that are called **hypervectors**. Hypervectors are defined as points of an hyperdimensional space,called **hyperspace**. Each of these points represents a real-world **entity** and similar entities have similar vectors.
As stated in [14], given the large amount of bits, two hypervectors are considered **similar** if they differ in **less then one third** of their bits, while they are **orthogonal** if they have **half or more non-identical** bits.

1.2 Advantages

Brain-like computational paradigm was mainly introduced for its innate capability to adapt to different learning algorithm. It is however the higher dimensionality which gives the most remarkable properties to the model. Hypervectors are indeed **redundant** and **random**, which makes Hyperdimensional Computing **robust to failure** and **efficient** in terms of required operations.

1.2.1 Robustness

Hypervectors are made with more than thousands of bits and this implies that even if a large number of bits is corrupted the hypervector can still hold its information without loss. In learning application, two different hyperectors are associated to two different entities, and due to redundancy those entities can be discerned even if **33% of bits** [14] in the hypervectors are incorrect, i.e. two hypervectors are still considered similar if they have less than one third of different bits.
Robustness can be further improved using the so called **holistic representation** [14], which results in equally spreading the information among all the bits. In standard binary representation the importance of a bit is bound to its position, thus the severity of failures depends on the position of the damaged bit; in the holistic representation the information loss depends only on the number of corrupted bits, leading to an higher tolerance for errors.
In [8], Sizhe Zhang et al. assessed the robustness to errors of hyperdimensional computing. Errors are simulated and studied in the associative memory [see 1.3.3], given the memory-centric approach of HDC. Errors are modelled using the single bit flip (**SBF**) approach, which randomly flips a single bit to generate an error.
Robustness is asserted by measuring accuracy variation while classifying three different datasets:

- ISOLET: collection of voice audio of the 26 letters of the English alphabet, from 150 different subjects;
- HAR: a collection of 12 types of human activities from 30 different subjects;
- CARDIO: a collection of fetal heart rates (FHR).

Three different dimensions for hypervectors are used: 10000, 5000 and 3000. In addition, data are represented with three different width: 16-bit, 8-bit, and 1-bit. Each HDC model is tuned by applying different retraining rates and epochs, in order to increase performances in accuracy.
Simulations are run with an error rate varying from 10^{-9} to 10^{-1} in a logarithmic scale.
Results in [8] state that accuracy starts to drop after an error rate of 10^{-6}, which is 9 times higher than the error rate tolerated by standard computing ($< 10^{-15}$) and

Configuration	16-bit			8-bit			1-bit		
	10000	5000	3000	10000	5000	3000	10000	5000	3000
ISOLET	94.42	94.48	93.97	94.42	94.55	94.03	93.134	88.64	85.12
HAR	93.77	93.58	93.93	91.71	92.75	93.26	87.16	86.46	85.20
CARDIO	93.43	92.29	91.55	92.96	92.96	86.38	86.85	82.16	77.93

Table 1.1: Error-Free accuracy (%) for HDC classification with three different datasets and configuration. Table adapted from [8].

similar to the one tolerated by Neural Networks (from 10^{-9} to 10^{-6}). The error rate has different impact according to the application: in ISOLET classification, accuracy drops after an error rate of 10^{-4}, while for HAR and CARDIO, there is a significant loss in accuracy with an error rate of 10^{-5}.
Robustness of HDC classifier is also related to the data-width as can be seen in Table 1.2. The reason of this is that for wider data, errors become more significant if they affect higher-order bits. Hence, when implementing an HDC design, the right data width should be used in order to increase robustness while maintaining a proper accuracy.

Critical Error Rate

	16-bit	8-bit	1-bit
ISOLET	10^{-4}	10^{-4}	10^{-2}
HAR	10^{-5}	10^{-5}	10^{-2}
CADIO	10^{-5}	10^{-5}	10^{-2}

Table 1.2: Error rate at which accuracy starts to drop for ISOLET, HAR and CARDIO datasets with different data width. Table extracted from [8].

1.2.2 Efficiency

HD computing consists of a sequence of simple operation which are more hardware friendly and less power angry then normal Machine Learning approaches.
The HD **standard algorithm** starts with the generation of a **random hypervector** which is then **bound** with other vectors using **permutation** and a **multiplication** [see 1.3.2]. **Accumulation** of related vectors is done with **majority voting** of bits in the same position, while **similarity** is computed with the **Hamming distance** in case of **binary hypervectors**, or with **cosine similarity** in **non-binary** cases [see 1.3.5].
To better define the efficiency of HDC, an example of performances comparison between a deep neural network and an hyperdimensional classifier is reported in [9], where Mohsen Imani et al. proposed **VoiceHD**, an hyperdimensional classifier for voice signals.

VoiceHD model consists in using dense binary hypervectors with dimension D = 10000 bits, to classify voice signals from the ISOLET dataset [see 1.2.1]. A unique hypervector is assigned to each frequency bin, hence for the 617 frequency bins in the ISOLET dataset, N = 617 hypervectors are needed. Voice signals are encoded using a record-based approach [see 1.3.4], in which each input frequency is translated into an hypervector and then *binded* with the correspondent channel hypervector.
The design is described in Matlab, with the model trained on ISOLET 1-2-3-4 and tested on ISOLET 5. VoiceHD is then compared with a neural network with three hidden layer: $L_1 : 617$, $L_2 : 1024$, $L_3 : 1024$, $L_4 : 512$, $L_5 : 26$. The measurements of power, execution time and accuracy are performed on CPU cores, in particular the Intel Core i7 processor (4-core, 2.8GHz, 16 GB memory).
Results reported by Mohsen Imani et al. show how the implemented binary hyperdimensional classifier reaches a 11.9x higher energy efficiency, 5.3x faster testing time and 4.6x faster training time compared to the deep neural network [Table 1.3].

		NN	VoiceHD
Training	Execution Time/Training Set	17min	3.7min
Testing	Energy Consumption/Single Query	454mJ	38mJ
	Execution Time/Single Query	4.61ms	1.14ms

Table 1.3: Energy consumption and execution time of Neural Network and VoiceHD on CPU. Table adapted from [9].

Other examples of HD computing efficiency compared to standard machine learning approach are reported in 1.4.

1.3 Model

Any computing system is composed of three main aspects [1]: **data representation**, **data trasformation**, **data retrieval**. Within the Hyperdimensional computing paradigm, data are represented with hypervectors and transformed using only **multiplication**, **addition** and **permutation** [see 1.3.2].
HD computing is mostly applied in classification problems, which are divided in two phases: **training**, i.e. the process to create entities which represent the main aspects of other entities that belong to the same class, and **testing**, i.e. associating an input entity to its class. In Hyperdimensional computing each class is represented with an hypervector denominated as **class hypervector**. The collection of class hypervectors is stored in the **Associative Memory** [see 1.3.3]. During testing phase, an input hypervector denominated **query hypervector** is compared to all class hypervectors through an **associative search** in order to predict with a

certain **accuracy** to which class the query hypervector belongs.

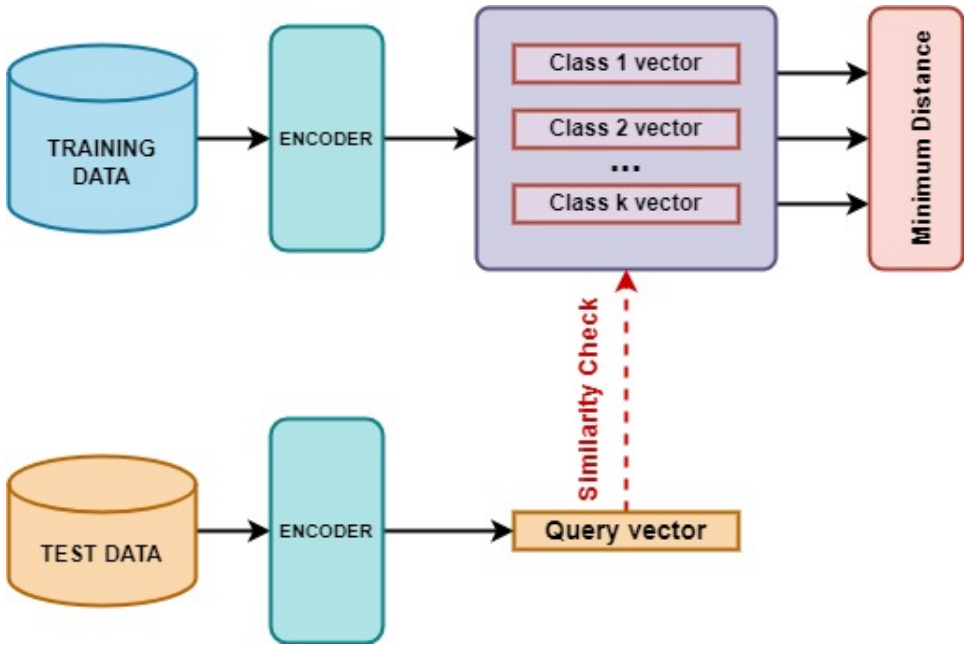


Figure 1.1: Scheme of classification with Hyperdimensional computing. Training data are encoded and processed to create class hypervectors stored in the Associative Memory [aM]. Test data are encoded to create a query vector and then compared with each class vector inside the aM. Image adapted from [1].

1.3.1 Data Representation

An **hypervector** [11] is a more than 1,000 dimension wide word which represents a point in the **hyperspace**. Elements inside hypervectors are **random** and independent and identically distributed (**i.i.d**).
Due to their **holistic representation**, information inside an hypervector is equally distributed in each component, hence each dimension holds the same amount of data.
Binary hypervectors are hypervectors with only binary elements, i.e. each dimension is represented with a single bit; binary hypervectors are referred as **dense hypervectors** if **'1'**-bits and **'0'**-bits are present with the same ratio, whilst binary hypervectors with a larger portion of '0'-bits are referred as **sparse hypervectors**.
In **Non-binary** representation, hypervectors are usually **ternary**, where each dimension can take a value only in the set $\{-1; 0; 1\}$, or **integer**.
Data representation directly affects efficiency and accuracy of the HD computing model. In [10], a full binary HD classifier with the possibility of retraining

called **BinHD** is proposed. The performances of BinHD are compared with a baseline HD algorithm using three different data representation. The first tested baseline configuration consists in using float data both for encoding and for the model (Float/Float). The second uses a non-binary encoder with a binary model (Float/Binary), while the third uses a binary encoder and a binary model (Binary/Binary). Performances are evaluated for four tasks: speech recognition (ISOLET), acitivity recognition (UCIHAR), Face recognition (FACE) and cardiotocograms classification (CARDIO).
As shown in Table 1.4, apart from the higher accuracy obtained with retraining in BinHD, in a full binary model the classification accuracy is lower with respect to other data representation. However, there is a reduction on the size of the model and the memory footprint of respectively 24.6x and 32x than the non-binary model (Float/Float), as shown in Table 1.5 and Table 1.6.

	Accuracy(%)			
Encoding/Model	ISOLET	UCIHAR	FACE	CARDIO
Float/Float HD	93.5	95.8	95.3	99.0
Float/Binary HD	88.1	91.3	91.9	93.8
Binary/Binary HD	85.6	87.3	83.5	90.2
BinHD	91.5	95.7	94.3	99.5

Table 1.4: Classification accuracy of BinHD and Baseline HD in different configurations. Table adapted from [10].

	Training Memory Footprint (MB)			
Encoding/Model	ISOLET	UCIHAR	FACE	CARDIO
Float/Float HD	251	249	898	77
Float/Binary HD	251	249	898	77
Binary/Binary HD	10	13	34	3
BinHD	10	13	34	3

Table 1.5: Training memory footprint of BinHD and Baseline HD in different configurations. Table adapted from [10].

1.3.2 Arithmetic

Hyperdimensional computing arithmetic has a set of operations that process **d-dimensional inputs** and produce **d-dimensional outputs** preserving the dimensionality [1]. All operations are also **point-wise**, hence elements in the same position are combined together without influence from other components.
The first of the three main operation is **addition**, which in the hyperdimensional model is also called **bundling**. In order to preserve the same type of representation after adding a set of hypervectors a **normalization** is performed. Normalization

	Model Size (KB)			
Encoding/Model	ISOLET	UCIHAR	FACE	CARDIO
Float/Float HD	1015.6	468.7	78.1	117.2
Float/Binary HD	31.7	14.6	2.4	3.7
Binary/Binary HD	31.7	14.6	2.4	3.7
BinHD	31.7	14.6	2.4	3.7

Table 1.6: Model size of BinHD and Baseline HD in different configurations. Table adapted from [10].

is usually obtained with a **weighted mean** in case of **non-binary** representation, or with a **majority voting** in **binary** case [eq. 1.1].

$$\begin{aligned} A &= 1\ \ 0\ \ 1\ \ 0\ \ 0\ \ 1\ \ 0\ \ 1, \\ B &= 0\ \ 0\ \ 0\ \ 1\ \ 0\ \ 0\ \ 0\ \ 1, \\ C &= 1\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 1, \\ \hline [A+B+C] &= 1\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 1. \end{aligned} \tag{1.1}$$

Hyperdimensional addition is characterized by an important property, which is that the resulting vector is **similar** to the addends, in the sense that the distance between the sum vector and the added vectors is minimum. This property is fundamental for classification since an hypervector produced by bundling a set of hypervectors can represent the whole set, being it similar to each vector inside the set itself.
Hyperdimensional **multiplication** [eq. 1.2] is also referred as **binding**, since it is used to bind vectors. The association $x = a$ can be indeed represented as $X * A$ in the hyperdimensional space, where X and A hypervectors associated respectively to x and a.
Contrary to addition, multiplication results in a vector that is **orthogonal** to all of the inputs. Multiplication is indeed used to represent structured data and for that reason cannot be similar to any of the bound vectors.

$$\begin{aligned} A &= 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 1\ \ 0\ \ 1, \\ B &= 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 0\ \ 0\ \ 1, \\ \hline A \bigotimes B &= 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 0\ \ 0. \end{aligned} \tag{1.2}$$

Other important properties of hyperdimensional multiplication [14] are:

- **Invertibility**: it is possible to retrieve one input vector knowing the other;

$$Z = X * Y \longrightarrow Y * Z = X \quad and \quad X * Z = Y \tag{1.3}$$

- **Distribution**: multiplication distributes over addition;

$$Z * [X + Y] = [Z * X + Z * Y] \tag{1.4}$$

- **Distance Preservation**: two different hypervectors with a certain distance multiplied by the same hypervector produce as outputs two hypervectors with the same distance as the inputs.

$$with \quad X_A = A * X \quad and \quad Y_A = A * Y \longrightarrow |X_A * Y_A| = |X * Y| \tag{1.5}$$

  with $|\cdot|$ representing the distance between hypervectors.

In case of a binary representation, hyperdimensional multiplication consists in a **bitwise XOR**, simplifying hardware requirements.
The third fundamental operation is **permutation** [eq. 1.6], i.e. reordering elements inside the hypervector. Given X as an input hypervector, the permutation is denoted as the product between X and a **permutation matrix** Π: $X_\Pi = \Pi X$, where X_Π is the permuted vector.

$$\begin{aligned} A &= 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1, \\ \hline \rho(A) &= 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0. \end{aligned} \tag{1.6}$$

Permutation is characterized by the same properties of multiplication [14]: invertibility, distribution, distance preservation and orthogonality with respect to inputs. Permutation is typically used to represent sequence of data together with addition [14]; in fact, addition alone cannot represent an ordered set of elements since it does not preserve the order. Hence, with hyperdimensional arithmetic the sequence AB is represented as: $S = \Pi A + B$. Larger sequences are obtained by progressively permuting elements, as for example the sequence $ABCDEF$ is represented as:

$$S = \Pi[\Pi[\Pi[\Pi[\Pi A + B] + C] + D] + E] + F \tag{1.7}$$

By exploiting the properties of addition, multiplication and permutation it is therefore possible to embed a **data record** inside a single hypervector [14]: given as an example three different channels *channel_1* [A], *channel_2* [B] and *channel_3* [C], and the three sensed value *value_1* [X], *value_2* [Y], *value_3* [Z], the actual order of sensed values and their respective channels can be condensed as:

$$S = \Pi[\Pi[X * A] + [Y * B]] + Z * C \tag{1.8}$$

1.3.3 Memory

As explained in the previous section, with Hyperdimensional computing significant entities are represented with hypervectors that need to be stored in order to be exploited in algorithms. The type of memories in which those vectors are stored depends on their function and their origin. Hypervectors which represent simple entities are memorized in what is called an **Item Memory** (**iM**).
The item memory can be defined as a record of significant vectors with the main characteristic of being **autoassociative** [14], which means that it is a **content addressable** memory addressed with vectors that are similar to the stored ones. Hence a vector is stored using itself as the address. In particular, it is possible to obtain a noise-free version of a noisy vector using it as an address to the Item Memory, if the noise is not large enough to make those vectors completely different.
The **class hypervectors** used to represent classes in classification problems are instead stored in the **Associative Memory** (**aM**). The number of stored vectors inside the Associative Memory corresponds to the number of classes, and classification is done by comparing a **query hypervector** to all the elements inside the memory: the class vector with the lowest difference with respect to the query vector represents the class where the query vector most likely belongs [Figure 1.1].
In certain encoding algorithm, a third type of memory is required, called **Continuous Item Memory** (**CiM**). This memory stores vectors which do not represent entities but different levels of values that a variable can assume. Indeed, when a real quantity is mapped in the hyperdimensional space, it can assume only a limited number of values depending on the provided resolution. The CiM indeed stores L = *number of possible values* vectors called **Level Hypervectors**.
Levels hypervectors are not mutually orthogonal, since vectors which represent close values must be similar, while the minimum and maximum values must be orthogonal. This relation between level hypervectors can be easily obtained by taking a random vector to represent the lowest value, called **seed hypervector**, and splitting one of is halves in **L - 1** parts [2]; those parts are then progressively inverted in order to have the level hypervector associated to the maximum value with half of his bits complementary to the minimum level hypervector, thus obtaining orthogonality.

1.3.4 Encoding

Encoding consists in translating input data into hypervectors. Accordingly to the type of data to be converted, two types of encoding methods are mainly used [1].
Record-based Encoding [Figure 1.2] maps both **data values** and **data positions**. Each position is represented with a **position hypervector** ID_i where $1 \leq i \leq N$ with N = *number of positions*. Values are instead discretized and represented using **level hypervectors** L_i where $1 \leq i \leq m$ with m = *number of quantization levels*. While position hypervectors are orthogonal among them,

level hypervectors preserve the similarity of the value they represent, i.e. L_1 is similar to L_2 but orthogonal to L_m. Position hypervectors are stored in the Item Memory whilst level hypervectors are stored in the Continuous Item Memory [see 1.3.3]. With record based encoding, a feature vector **h** with **N** features each with **m** possible values, is encoded as:

$$H = L_i \oplus ID_1 + L_i \oplus ID_2 + ... + L_i \oplus ID_{N-1} + L_i \oplus ID_N, \qquad (1.9)$$
$$L_i \in \{L_1, L_2, ..., L_m\}, \text{ with } 1 \leq i \leq m.$$

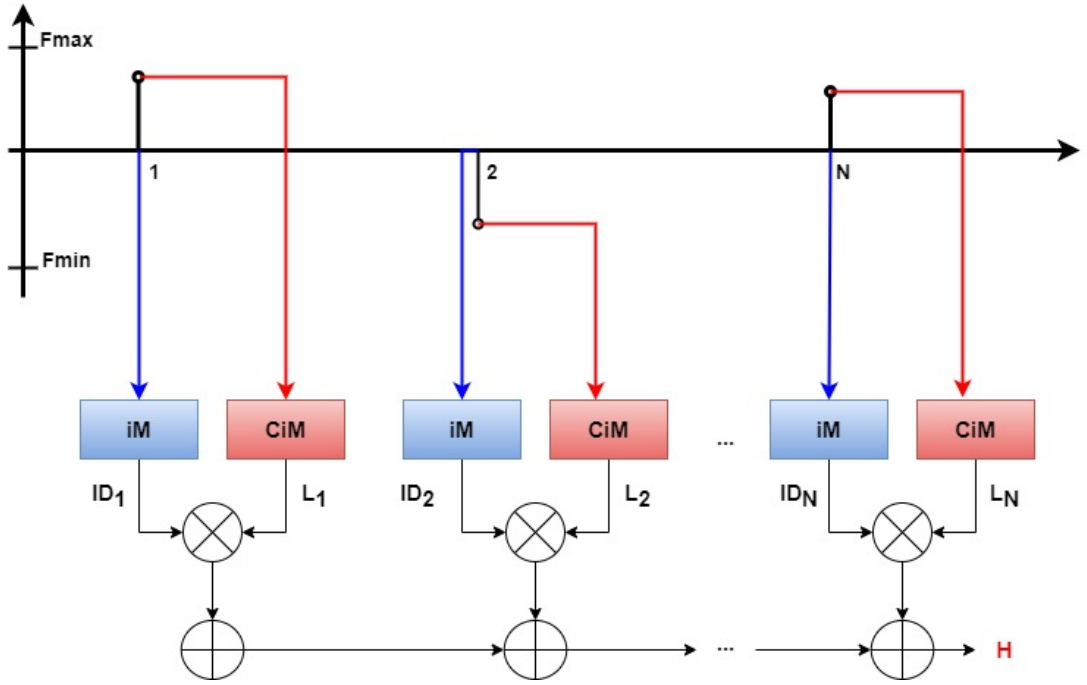


Figure 1.2: Record based encoding. Position hypervectors read from the Item Memory (iM) are indicated as ID, while level hypervectors read from the Continuous Item Memory (CiM) are indicated as L. Image adapted from [1].

The alternative encoding method is referred as **N-gram Encoding** [Figure 1.3]. In this case the position of a feature inside the feature vector is not encoded using a position hypervector but it is instead derived from the level hypervector through **permutation**. In this case a level hypervector L_i associated to the value of the feature at position m is permuted $m-1$ times before being bound to the other level hypervectors. Differently from record-based encoding, the hypervectors **H** which maps a feature vector is obtained by **binding** the level hypervectors, i.e. using

multiplication, instead of **bundling** them [eq. 1.10].

$$H = L_i \oplus \Pi L_2 \oplus ... \oplus \Pi^{N-1} L_N \qquad (1.10)$$
$$L_i \in \{L_1, L_2, ..., L_m\}, \text{ with } 1 \leq i \leq N.$$

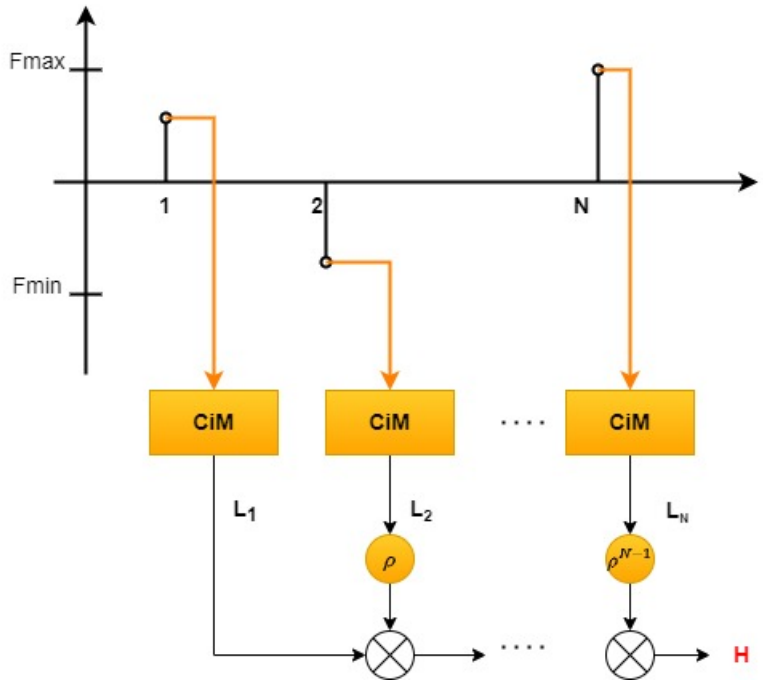


Figure 1.3: N-gram based encoding. Level hypervectors read from the Continuous Item Memory [CiM] are indicated with L. Image adapted from [1].

1.3.5 Distance Evaluation

As stated in section [1.2.2],the similarity between hypervectors is computed using the **Hamming distance** for binary hypervectors, and **cosine similarity** for non-binary hypervectors.
In case of two binary hypervectors, they are considered similar if their Normalized Hamming [eq.1.11] distance is low, with a distance equal to 0 indicating two identical hypervectors. If instead the normalized Hamming distance has a value greater or equal than 0.5, the two vectors are considered orthogonal [1].

$$Ham(A, B) = \frac{1}{d} \sum_{i=1}^{d} 1_{A(i) \neq B(i)} \qquad (1.11)$$

Indicating with **d** the dimension of the hypervector, for higher d the probability for vectors to have similar values becomes smaller, while the probability for vectors to be orthogonal becomes higher [14] . This phenomenon has important implications since with higher grade of orthogonality tolerance to errors increases, since it is possible to distinguish hypervectors even with a large number of damaged bit [1, 14]. Hamming distance cannot be applied in presence of non-binary hypervectors. In that case the **cosine similarity** [eq. 1.12] is used as a similarity metric. Cosine similarity measures only the orientation between vectors, i.e. the angle between them, without taking into account their magnitude. Differently from Hamming distance, an higher value of cosine similarity indicates lower difference between vector, with a cosine similarity equal to **1** meaning identical vectors or an angle of 0° in terms of orientation. If the cosine similarity is *0* then hypervectors are dissimilar, i.e. they form an angle of 90° thus being orthogonal.

$$cos(A, B) = \frac{A \cdot B}{|A||B|} \tag{1.12}$$

As can be seen by the formulation of the two previously illustrated distance metrics, the cosine similarity requires more resource with respect to the Hamming distance in case of hardware implementation. Indeed for Hamming distance only XOR ports and a counter of ones are needed, while a multiplier is needed for cosine similarity.

1.4 Applications and examples from Literature

In [1] some application of hyperdimensional computing for classification problems are presented. In *European Language Recognition*, different languages are classified through encoding **N-grams**, i.e sequence of N letters [Figure 1.4]. At first, each of the *27* letters of the alphabet is represented with a random seed hypervector; then N-grams are generated with permutation and multiplication of letter hypervectors inside a series formed of N elements.Finally, the text is encoded by adding all present **N-grams hypervectors**. By comparing the **text hypervector** with class hypervectors that map each studied languages, it is possible to predict the language in which the text is written.

As observed in [11], the HD approach reach a better accuracy with respect the baseline algorithm if the N-gram is formed by N = 2 letters. For N $\geq$ 3, the baseline machine learning algorithm displays higher accuracy. As can be seen in Table 1.7, for N-grams of three or less letters, HD model also requires a larger amount of memory with respect to the baseline method. However, while the baseline approach memory requirements grow exponentially with the complexity of the N-grams, HD computing manage to maintain the same hardware structure. Indeed, for N-grams with 5 letters, the baseline algorithm requires 500x larger memory size with respect to the HD model.

Manuel Schmuck et al. [2] implemented an HD classifier for *hand gesture recogni-*

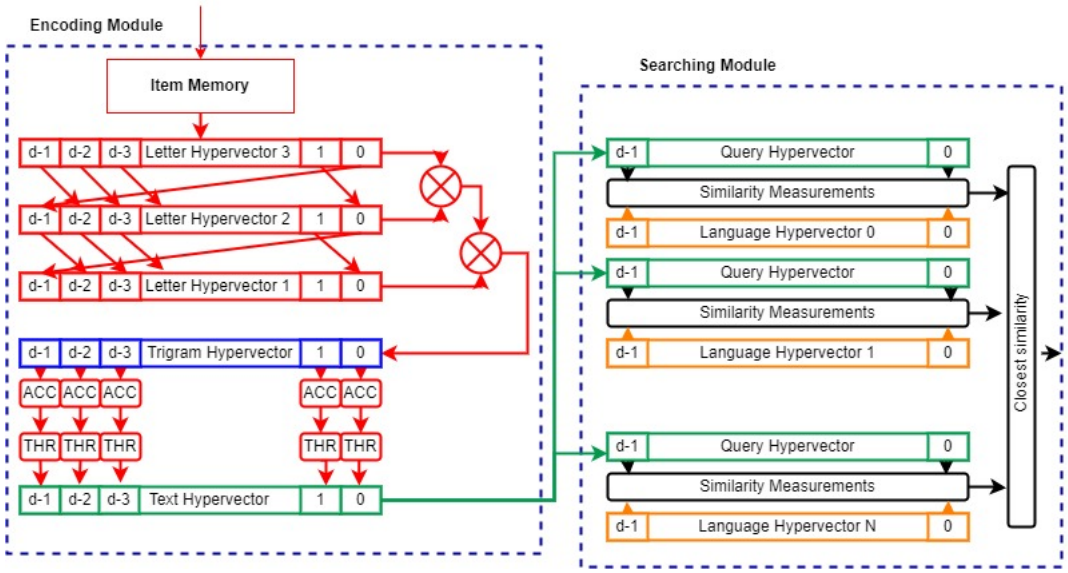


Figure 1.4: Scheme of language classification using trigrams. Two main module implemented: an encoding module for text hypervectors construction, and a searching module were query vectors are compare to the language hypervectors stored in the associative memory. Image adapted from [1].

N-grams	Accuracy(%)		Memory (KB)	
	HD	Baseline	HD	Baseline
N = 2	93.2	90.9	670	39
N = 3	96.7	97.9	680	532
N = 4	97.1	99.2	960	13837
N = 5	95.0	99.8	700	373092

Table 1.7: Classification accuracy and memory requirements of HD and baseline classifiers. Table adapted from [11].

tion. Inputs are **EMG** analog signals taken with a four-channel **wearable sensor**. Each channel is independently encoded with a record-based approach, where channels are used as **position hypervectors** and discretized EMG signals are mapped using **level hypervectors**. The sequence of received signals is instead encoded with an N-gram approach. Hence, encoding is split in two modules [Figure 1.5] to keep track not only on the **spatial correlation** of channels but also the **temporal correlation** of samples. [12] focuses more on simplifying the computational model for hyperdimensional computing algorithms that use **floating point** elements. Hence, the **QuantHD** framework is proposed. QuantHD uses a **complete binary HD model**, thus enhancing efficiency, and faces the low accuracy problem typical of binary models using an **iterative training approach**. In particular,

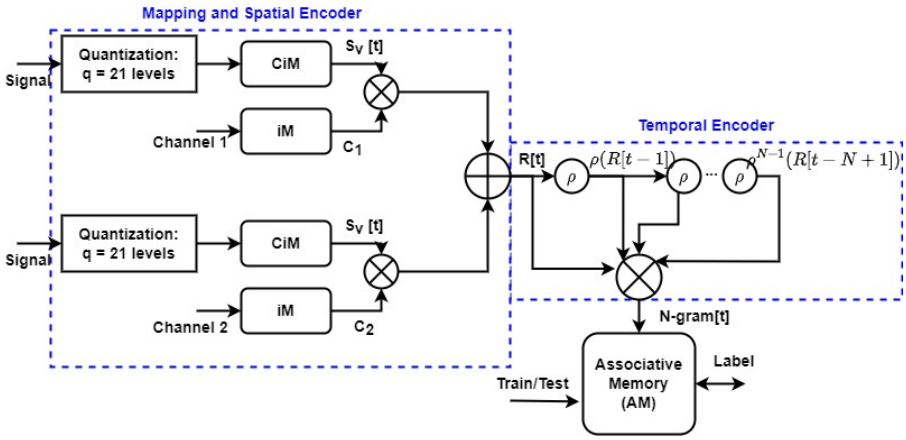


Figure 1.5: Scheme of HD architecture for EMG signals classification. Three modules implemented: a spatial encoder that maps each signal to its channel;a temporal encoder to keep track of the sequence of samples and an associative memory for training or testing. Image adapted from [2].

QuantHD improves of the **17%** the HD classification accuracy in **face recognition** taks [Table 1.8].
In QuantHD, after the initial standard training with non-binary hypervectors follows a **quantization** phase, which transforms non-quantized hypervectors in binary hypervectors; hence a **retraining** process is done. During retraining, a first classification is executed on binary hypervectors in order to understand which hypervector $\mathbf{H^q}$ is misclassified. After that, the non-quantized counterpart $\mathbf{H}$ of the misclassified hypervector is subtracted from the non-quantized class hypervector of the mispredicted class $\mathbf{C_{miss}}$ and added to its real class hypervector $\mathbf{C_{match}}$ according to a **learning rate** α [eq. 1.13]. Retraining is reiterated until the classification error $\Delta\mathbf{E}$ is less than a threshold $\boldsymbol{E}$, thus both the non-quantized and quantized models adapt to the dataset.

$$C_{miss} = C_{miss} - \alpha H \quad and \quad C_{match} = C_{match} + \alpha H \quad 0 < \alpha < 1 \tag{1.13}$$

The performances of QuantHD are evaluated on four different datasetets, and then compared with a baseline HD algorithm, a **multi-level perceptron** (**MLP**) and a **binary neural network** (**BNN**) [Table 1.9]. Training results are reported on ARM Cortex A53 CPU, while testing results on Kintex-7 FPGA KC705.
In [15] Mohsen Imani et al. presented **BRIC**, a binary hyperdimensional classifier which goal is to reduce energy consumed during encoding phase. BRIC maps an input feature vector $\mathbf{F} = \{f_1, f_2, .., f_n\}$, with $N = number\ of\ features$, using **Random Projection**, which consists in multiplying the feature vector with a **random projection matrix** of $D = dimension\ of\ hypervectors$ bipolar vectors of length N. Indicated as $\mathbf{P}$, with $\mathbf{P}_i \in \{-1,1\}^n$, the **projection vectors** inside the

	Baseline HD		QuantHD		
	Non-Quantized	Binary	Non-Quantized	Binary	Ternary
ISOLET	91.1%	88.1%	95.8%	94.6%	95.3%
UCIHAR	93.8%	77.4%	98.1%	96.5%	97.2%
PAMPAP2	88.9%	85.7%	92.7%	91.3%	92.7%
FACE	95.9%	68.4%	96.2%	94.6%	95.4%
CARDIO	93.7%	90.9%	97.4%	95.3%	97.7%
EXTRA	70.2%	66.7%	74.1%	72.6%	74.0%

Table 1.8: Comparison of QuantHD accuracy with non-quantized, binary and ternary model with baseline HD computing with non-quantized and binary model. Table adapted from [12].

	Accuracy(%)			CPU Training (s)			FPGA Inference (μs)			Model Size		
	MLP	BNN	HD	MLP	BNN	HD	MLP	BNN	HD	MLP	BNN	HD
ISOLET	95.8	96.1	95.8	2.08	17.69	0.31	27.39	5.24	0.40	1.81MB	56.7KB	65.0KB
UCIHAR	97.3	95.9	97.2	1.04	8.32	0.12	21.43	5.18	0.37	1.68MB	52.7KB	30.0KB
PAMAP2	95.8	94.2	92.7	0.61	4.75	0.07	13.07	3.78	0.35	0.68MB	21.3KB	15.0KB
FACE	96.1	96.1	95.4	0.56	4.30	0.04	17.68	5.11	0.34	1.77MB	55.3KB	5.0KB

Table 1.9: Comparison of accuracy, efficiency and model size between MLP, BNN and QuantHD. Table adapted from [12].

matrix, the **sign** of the inner product between a component of P and F gives out a single dimension h_i of the output encoded hypervector H [eq. 1.14].

$$h_i = sign(P_i \cdot F) \quad with \quad H = \{h_1, h_2, ..., h_D\} \tag{1.14}$$

The introduced optimization uses a **sparse random projection matrix**,i.e. a projection matrix with a percentage of **non-zero** elements below **50%**, thus avoiding more than one half of operations required for encoding. However, a random sparse matrix requires random addressing to access non-zero elements, hence requiring additional energy. To avoid this problem the principle of **locality** is exploited: random non-zero elements are placed near to each other inside the matrix, with each row of the matrix being a shifted version of the previous one. In this way, accessing the matrix becomes predictable, since the elements inside the row **w** correspond to the elements at row *0* shifted **w-times** by a fixed number of position.
Being the vectors inside the projection matrix ternary, $P_i \in \{-1,0,1\}$, there is no actual need of multiplication, since the inner product is performed only with addition/subtraction, further improving encoding efficiency. As stated in [15], thanks to the usage of a locality-based projection matrix and a full pipelined FPGA implementation, BRIC reaches a 64.1x and 43.8x energy efficiency in training and testing with respect to the baseline HD computing, with a speed up of 9.8x in training phase and 6.1x in testing phase.
The efficiency of hyperdimensional computing is strongly related to dimensions,

indeed a lower dimensionality leads to a faster algorithm and to a greater energy saving. Reducing vectors dimension however reduces performances in terms of accuracy. For all the previous reasons, Chenyu Huang et al. [13] proposed **MHD**, a newer encoding block for HD computing where the type of encoding approach and hypervectors dimension are configurable. MHD is composed of two modules, a **main stage** and a **decider stage**, and has four types of **classifier**: one using record-based encoding and a dimension of 10.000, one using N-gram based encoding with 10.000 dimensions, one using record-based encoding and a dimension of 2000 and finally one using N-gram based encoding with 2000 dimensions. The main stage trains with all encoding methods and all dimensions in parallel, producing the four classifier, then the decider stage assign a classifier to an input vector based on **confidence**. Since a model with higher dimensions always gives better results, the decider stage is configured to prefer the classifier with the lowest dimensionality if the accuracy has an acceptable grade of degradation compared to efficiency improvement. MHD implementation provides five configuration: from 1-level of hierarchy, i.e. with a single type of encoding at a fixed dimension, to 8 levels of hierarchy, with 2 types of encoding and four possible dimensions. Its performances are tested in speech recognition task [Table 1.10].

As reported in [13], MHD reaches a 6.6x energy improvement with respect to

Configuration	Encoder	1-level	2-level		4-level		6-level		8-level	
		Encoding I	Encoder I	Encoder II	Encoder I	Encoder II	Encoder I	Encoder II	Encoder I	Encoder II
Speech Recognition	Dimensions	10000	10000		2000 10000		2000 4000 10000		2000 4000 6000 10000	
	Model Size	67.5KB	70KB		72.5KB		75KB		77.5KB	
Classification Accuracy		93.6%	95.9%		95.9%		95.9%		95.9%	

Table 1.10: Model size and classification accuracy of MHD in different configurations. Table adapted from [13].

single level baseline HD computing model, with a speed-up if 6.3x.

In [16] a methodology based on **Processing in-Memory** (**PIM**) is proposed to accelerate HD computing performances. PIM indeed performs a set of tasks inside the memory without using processing cores and avoiding the bottle-neck of memory access. To exploit the advantage of **in-memory** architecture, the proposed method called **SearcHD** uses fully binary HD computing algorithm. In this way, each step of HD computing, i.e. encoding, training and inference, is implemented in-memory without non-binary operation.

In HD computing, during training, classes are created by adding together vectors belonging to that specific class, and this creates vectors with non binary elements that need to be normalized to be employed in a binary model. In SearcHD the operation of adding vectors while creating a class is replaced with **Bitwise Substitution**. This means that given two vectors A and B, some bits of B are copied in A at the same indices. If A and B are similar, less bits of A are changed while **cloning**. Since some bits of A are changed while B remains the same, A is called

a **binary accumulator** and B is called the **operand**.
To cope for the loss in accuracy with HD computing using binary model, **vector quantization** is proposed, where multiple vectors are associated to the same class. Hence, training keeps information of the same class in different hypervectors. Therefore for k classes there are N **model vectors**. Those N vectors for each class are initialized as random input hypervectors belonging to that class. Training is done by updating the model vector inside a class that has the lowest hamming distance with the input vector of that label. In other words, the input vector is compared to all the model vectors of the class indicated by its label, and after the most similar model vector is found, the model vector is modified using bitwise substitution. This algorithm allows to encode only useful parts of hypervectors, i.e. the one in the most similar piece, without the need of encoding all data. After the model is updated for all the training dataset, classification is done by comparing the query hypervector to all the k x N class hypervectors. According to what is reported in [16], SearchHD can reach a 31.1x higher energy efficiency and 12.8x speed-up with respect to baseline HD computing. In addition, using eight hypervectors per class and 16 hypervectors per class gives an higher accuracy of respectively 9.2% and 12.7% with respect the baseline implementation.

Chapter 2

Event-Based Cameras

2.1 Event Cameras

Researches on brain structure and capabilities inspired the implementation of novel algorithm and sensors which aim to reproduce neural capacity to interpret and perceive reality.
Particular attention was given to the realization of the so called **silicon retinas** [3], i.e. **bio-inspired** cameras, since sight is the primary interface between brain and environment. Event cameras are also referred as **Dynamic Vision Sensors** (**DVS**).
Instead of sampling images at a constant frame rate, DVS measure **brightness changes** in each of its **pixel** in **asynchronous** way, with each pixel independently sampled from others. The output of a DVS is therefore a stream of events for each pixel, with an event **e** defined as a data structure containing **x** and **y** coordinates of the pixel, the time of occurrence **t**, and the polarity **p** of the change, i.e. if brightness increased or decreased.An event is reported any time the relative pixel measures an higher or lower brightness than a stored **log intensity**, with respect to a given threshold. The log intensity stored in the pixel is then updated with the latest measured value.
Event transmission from pixel to out of the camera is done through a shared digital bus **address-event representation** (**AER**) with a readout rate ranging from *2MHz* to *1200MHz* [3].

2.1.1 Advantages of Event Cameras

Being completely asynchronous, event-based systems are faster than standard frame-based cameras; indeed, any time a change in brightness is detected it is reported, without waiting for a clock signal. In addiction, pixels inside DVS are completely independent thus further reducing latency, since each pixel can report

its measurements without being temporized by a global entity.
Events are read with a clock frequency of *1MHz*, therefore with an high temporal resolution of $1\mu s$.
The consequence of transmitting only changes in brightness is the reduction of redundancy [3], considering that a constant intensity value is not reported twice.
Without reporting redundant events, the number of transmissions is reduced, thus leading to a significant reduction of consumed power. The medium power used by a standard camera is about *10mW*, while an event cameras uses only $10\mu W$ [3].
Event cameras surpass standard cameras also in the input dynamic range. While an high-quality frame-based camera reaches a maximum of *60dB* of dynamic, a standard DVS as a dynamic range of above *120dB*, thus being able to receive signals both in dark or bright environments. This higher dynamic comes directly from pixel operating independently and in the logarithmic scale [3].
As illustrated in this section, the properties which make event cameras attractive are their **high temporal resolution** and **low latency**, together with **low power consumption** and an **high dynamic range**. However, this novel approach to data sensing requires proper frameworks which can exploit the complete potential of event cameras. Algorithm used for frame-based cameras are in fact not applicable, since DVS outputs have a totally different structure with respect to the sequence of dense images at a fixed frequency like the standard cameras.
In addition, brightness sensing in DVS is done via recording changes, hence data meaning depends on current and past information. A proper quantization methods for events is also not present, therefore reducing noise becomes harder than in standard cases.

2.1.2 Typologies of Event Cameras Design

Event cameras are also referred as DVS, but an actual DVS is a first type of design of an event cameras. The circuit of a DVS is composed by a **continuous-time photoreceptor** capacitively coupled to a **readout circuit** [3]. Any time a new event is sampled, the readout circuit value is reset [Figure 2.1].
A main problem with applications using DVS is that they require an absolute value of brightness to interpret information reported from the DVS. Hence, new types of event-based camera were implements in order to simultaneously measure brightness absolute values and changes.

An example of these types of cameras is represented by the **Asynchronous Time-based Image Sensor** [**ATIS**]. In ATIS, the simpler DVS acts as a subpixel which measures brightness change and which signals another subpixel into reading absolute intensity. The DVS subpixel is referred as **change detection** (**CD**), while the absolute intensity subpixel is referred as **exposure measurement** (**EM**). When a

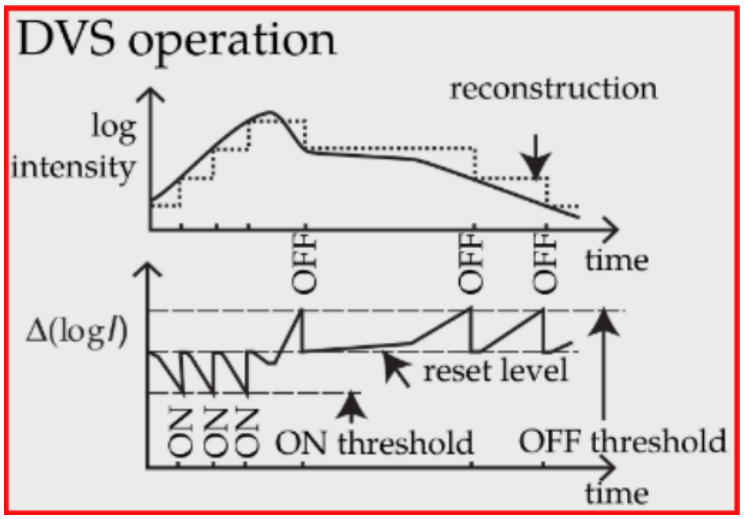


Figure 2.1: Overview on light change to event conversion performed by a DVS. Image taken from [3].

change is detected, the CD brings a capacitor to an high voltage, then a photodiode discharge the capacitor accordingly to brightness: a brighter light results in a faster discharge of the capacitor [3]. With the ATIS, a dynamic range of more than *120dB* is achieved, but the cost in terms of area is doubled, since each pixel contains two modules.
A more efficient design is represented by the **Dynamic and Active Pixel Vision Sensor** (**DAVIS**). DAVIS reduces pixel size by combining in a single pixel a standard **active pixel sensor** (**APS**) and a DVS [Figure 2.2]. This is done by using a single photodiode shared between absolute intensity sensor and the DVS, while the readout circuit introduces an area overhead of about *5%* [3].
Since the APS is a standard type of pixel, it can be programmed to capture frames at a constant rate. However this configuration is rarely used since it is not coherent with event cameras behavior.
Even if they present different designs and specifics, ATIS and DAVIS are also referred as DVS since they include a DVS as a submodule.

2.2 Event Representation and Processing Methods

Event cameras introduced a new way to represent data extracted from images, i.e. *events*. Acquisition of events is asynchronous, sparse and with high resolution and low latency. Hence, event processing should preserve the advantages introduced by the low latency.

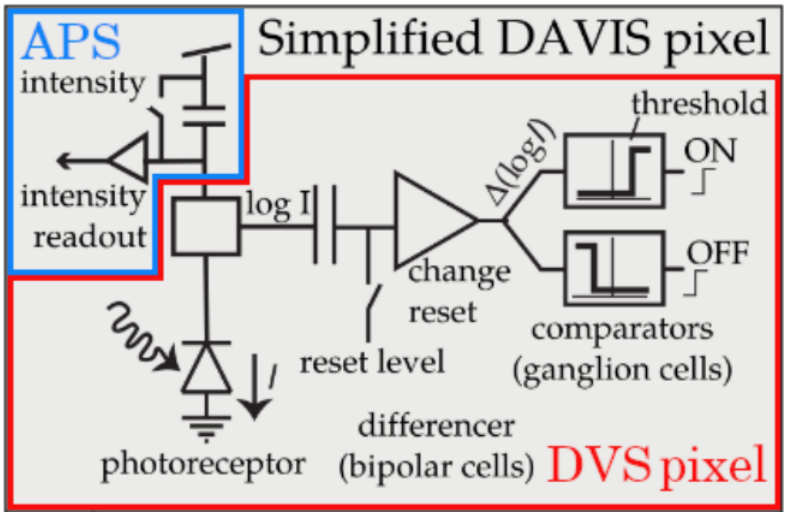


Figure 2.2: Simplified DAVIS pixel design. Image taken from [3].

Event processing methods can be divided according to the number of processed elements, or according to how events are processed.
Based on the number of events processed at each iteration, event processing algorithms are divided in two categories: **event-by-event basis** methods process single events at time, thus having a minimum latency; **group or packets of events** methods wait for a certain amount of events to be recorded before processing them altogether. In the later case, latency increases due to waiting for events. There is however a blurred division line between those two methods, since a *group of events* algorithm can still use groups of a single element, and an *event-by-event* algorithm needs a set of information coming from previous events for reliable estimations [3].
The way in which events are processed divides event processing algorithms in **model-based** methods and **model-free** methods.

2.2.1 Types of Event Representation

In this section, a list of event representation method will be reported,as illustrated in [3].
Individual events representation report events as single points. It is used in *event-by-event* algorithms like **probabilistic filters** or **Spiking Neural Networks** (**SNN**).
With **Event packet** representation, a number N_e of events e_k in the same spatio-temporal zone are combined to obtain an output $\mathcal{E} = \{e_k\}_{k=1}^{N_e}$. The number of events in a packet depends on the algorithm.
Events in the same spatio-temporal zone are converted in *2D* images with the **Event Frame** representation. In this way, frame produced from events can serve

as inputs for standard image processing algorithms. The advantage of using event cameras to produce standard frames resides in the definition of more defined edges and in information about not only the presence, but also absence of events.
Differently from *Event Frame*, **Time surface** (**TS**) representationmaps events in a 2D configuration which intensity is correlated to motion. TS records the motion history of a certain area with a set of values, with higher values corresponding to more recent events. For this reason, time-surfaces are also referred as **Motion History Images**.
In TS, information is compressed since each pixel stores only a single time value, called **timestamp**. Timestamp represents the time in which the last event occurred at that pixel.
Voxel Grid represent events with 3D histograms. Therefore, temporal information is not altered by projecting it in a 2D representation. Event polarity can be stored in a single voxel or shared wight neighbour voxel in what is called a **interpolated voxel grid**, with the second method resulting in higher accuracy.
3D point set represents group of events in 3D space, using the time information as the third dimension, while **Point sets on image plane** represents events as evolving set of 2D points.
Additionally to recording events, **Motion-compensated event image** representation makes also motion estimations. While passing through pixels, a moving edge generates events that can be properly warped in order to obtain sharp images, i.e. images with better defined edges.
A motion-invariant representation of events is given by **Reconstructed images** where image reconstruction is obtained by integrating events over time. A set of events represents indeed a series of brightness variation over time, thus integrating events ideally produces an absolute value of brightness. In reality, pixels do not record the total amount of brightness at a certain time, but they store changes. Hence, the result of events integration is an indication on how much brightness changed at the end of the measuring window. The original brightness at the start of measurements must be added to brightness change in order to correctly reconstruct images.
In event representation and processing an important aspect regards polarities. Polarities are generated by motion, since a moving object simultaneously increases and decrease brightness in a scene, according to its direction. Hence, studying polarities can be useful for motion estimation, but superfluous in other cases. It is also important to say that values of opposite polarities recorded at the same time are not complementary, i.e. it is not possible to extract the value associated to a positive polarity starting from the value of the negative polarity registered at the same time.

2.2.2 Event Processing Methods

As stated in 2.2, event processing methods can be either **event-by-event** or **for group of events**, according to the quantity of processed events. Which of these two approaches should be used depends on event representation and hardware resources [3].
Event-by-event methods need to be able to process asynchronous data with a minimum latency, and to acquire information from multiple sources. Methods which satisfy these requirements can be found in **deterministic filters** or **artificial neural network** (**ANN**). One application of event-by-event model-free, i.e. unsupervised [see 2.2], method is in object classification using **spiking neural network** (**SNN**), as reported in [17].
Events are mostly processed in groups since each event stores only a small fraction of information and because processing many events at the same time is helpful in reducing noise. In addition, *event-by-event* methods requires external additive information for a correct evaluation of images, while **Group of Events** based method are usually self-sufficient [3]. Given the presence of many representation for packets of events, many group-based algorithm are available.
Events represented like a standard image, like with *event frames* representation, can be processed using classical image-based learning algorithm, like **SVM** or **Random Forest**.
Time surfaces are capable to detect edges, hence they are deployed in motion detection or shape recognition. TS are also used as features extractor or as inputs for **convolutional neural network** (**CNN**).
Voxel grids are mainly used for **optical flow estimation** or **deep neural network** (**DNN**). Since *voxel grids* are tridimensional, algorithms using this type of representation require larger memory space and higher computational effort.
Event-by-event and *group of events* based processing are also used together; in fact, some image classification algorithms use processing for group of events during training and event-by-event processing during testing, as in [18].

2.3 Algorithms and Applications of DVS

One of the first and simplest application based on *DVS* can be found in **feature detection and tracking**. Thanks to event cameras, tracking is faster and less consuming, although requiring customized algorithm. In tracking, it is fundamental to record data variation at any time without missing; this is however a tougher task with event cameras, since they report fast changing values. Tracking depends also on the observed object and its features: for simple tracking problems, the moving object is considered as a source of events with a non-defined shape; for more complex algorithm, the shape of the tracked object is user defined. Tracking algorithm are used in traffic monitoring and surveillance, microrobotics and particle tracking

in fluids.
Another example of event-driven vision application is in the **Optical Flow Estimation**. Optical flow estimation consists in computing velocity of objects without information about shape and motion. In standard approach using frame-based cameras, optical flow is done by comparing sequences of images; with event-cameras this is obviously not possible. Some approaches transform events in suitable data for classical frame-based algorithm, while others rely on more complex structures like neural networks.
Event cameras are also applied for **3D reconstruction**. Problems regarding depth estimation can be divided in many categories, based on the approach and used devices. **Instantaneous stereo** reconstruction consists in estimating depth instantaneously, i.e. in a narrow interval, recording and then matching events taken from two or more synchronized cameras. A possible approach to event matching is represented by **normalized cross-correlation** done on *event frame* or *time surfaces*.
The same approach of events recording and event matching as in instantaneous stereo is done with the **Multi-Perspective Panoramas** method. However in this case cameras are not synchronized.
Monocular Depth Estimation uses a single cameras instead of many. For this reason, event correlation cannot be used. Instead, in multi-perspective panoramas the events recorded by a single moving cameras are integrated over time for depth estimation. With this approach, depth estimation is no more instantaneous.
All the aforementioned depth estimations were passive, i.e. they simply record events. **Depth Estimation using Structures Light** is instead an active approach, which consists in projecting light in the scene and then measuring reflection.
Event cameras can be used also for **image reconstruction** [see 2.2.1], or even for **Recognition**. Early event-based object recognition algorithms consisted on a static camera capturing events generated by movements of the tracked object that is modeled as a simple shape. In case of a more complex object, its geometry is extracted by matching its features with template shapes. Feature matching is typically done with classifier, like the **Nearest Neighbor**. The main problem of this process is to determine which features to match. Features can be specified a priori or extracted from the available data, with the second case leading to better results.
A type of algorithm that automatically extracts features from data is **clustering**, i.e. an unsupervised learning algorithm which groups data around centers according to distance. The produced groups of data are called **clusters**, and each center of those clusters represents a feature.
Another approach to classification, using data recorded from event cameras, consists in transforming events into tensors, which are entities more suitable for hierarchical models like neural network. In the latest case, events are represented with *time surfaces*, which can averaged to obtain a more reliable representation.
An example of classification algorithm using averaged time surfaces is the **HATS**

[5]. HATS has also the peculiarity of receiving images from a DVS mounted on a moving vehicle; this results in a more difficult classification task with respect to the case of images taken from a static camera.

Chapter 3

Hierarchical Representation of Time-Surfaces

As explained in section 2.2.1, events cameras offer higher temporal resolution, higher dynamic range and less energy consumption with respect to standard frame-based cameras [3, 4, 5]. However, the full potential of these sensors is not fully exploited due to the lack of proper algorithms and architectures for classification on event-objects.
A classifier for DVS based on a low-level operator referred as **Local Memory Time Surfaces** is proposed in [4]. In order to improve robustness to errors and noises, Local Memory Time Surfaces can be combined to obtain an higher order representation called **Histograms of Averaged Time Surfaces** (**HATS**), as proposed in [5].
In this chapter, the time-surfaces representation is explained, followed by the illustration of Local Memory Time Surfaces and Histograms of Averaged Time Surfaces, as presented respectively in [4] and [5]. Afterward, a comparison of performances between image classification using standard approaches and HATS taken from [5] are reported.

3.1 Time Surfaces Inspired Models

3.1.1 Time Surfaces

The motion of an object in front an event-cameras, or the movement of the camera itself, produces events in pixels of the cameras. The set of those events represents the object position and dynamics. Hence, time-surfaces can be exploited to represent sets of events in order to keep track of the target object activity.
An event is mathematically defined as:

$$e_i = [\mathbf{x}_i, t_i, p_i]^T, \qquad i \in N \tag{3.1}$$

where $\mathbf{x}_i = [x_i, y_i]^T$ represents the location of the event inside the pixel grid, t_i indicates the time of arrival of the event and p_i is the polarity, with $p_i \in \{-1,1\}$. Negative polarity is indicated as -1, while 1 indicates positive polarity.
Defined $\mathbf{u} = [u_x, u_y]^T$ as a **square neighborhood** of pixels centered in $\mathbf{x}_i$, a **time context** $T_i(\mathbf{u}, p)$ around the event e_i is defined as:

$$T_i(\mathbf{u}, p) = \max_{j \leq i} \{t_j | \mathbf{x}_j = (\mathbf{x}_i + \mathbf{u}), p_j = p\} \tag{3.2}$$

where $u_x \in \{-R, ..., R\}$ and $u_y \in \{-R, ..., R\}$, with R **radius** of the neighborhood. After the definition of the time contest $T_i(\mathbf{u}, p)$, the time surface $S_i(\mathbf{u}, p)$ can be computed by applying an exponential decay kernel on values of $T_i(\mathbf{x}, p)$:

$$S_i(\mathbf{u}, p) = e^{-(t_i - T_i(\mathbf{u},p))/\tau} \tag{3.3}$$

with τ time constant of the kernel. The exponential decay not only extend the representation of activities to past events, but also highlights recent events by weighting values of events according to their arrival time.
Time-surface prototypes are a set of time surfaces which represents features of the observed scenes. Time-surface prototypes are obtained from time-surface through clustering [4].
When an event is reported, its time-surface is computed and then compared with each time-surface prototype. An output event is then generated from the closer prototype to the input event surface.
The starting condition of time-surface clustering consists in a set of **N** time-surface prototypes $\mathbf{C}_n$, with $n \in [1, N]$, where each prototype is described as in equation 3.2. Any time an event e_i is reported, its correspondent time-surface S_i is computed. The cluster $\mathbf{C}_k$ to which S_i belongs is computed as the cluster with the lowest euclidean distance from S_i. After that, $\mathbf{C}_k$ undergoes to an updating process as described in the following equation:

$$\mathbf{C}_k \leftarrow \mathbf{C}_k + \alpha(S_i - \beta C_k) \tag{3.4}$$

where:

$$\alpha = \frac{0.01}{1 + \frac{p_k}{20000}} \tag{3.5}$$

$$\beta = cos(\mathbf{C}_k, S_i) = \frac{\mathbf{C}_k \cdot S_i}{\|\mathbf{C}_k\| \cdot \|S_i\|} \tag{3.6}$$

with p_k number of the time-surfaces already assigned to $\mathbf{C}_k$.
With the clustering algorithm, sequences of input events are transformed into **prototype activations** [4]

$$feat_i = [x_i, y_i, t_i, k_i]^T \tag{3.7}$$

with k_i index of the cluster center $\mathbf{C}_k$.

3.1.2 Local Memory Time Surfaces

The result of the clustering algorithm illustrated in section 3.1.1 is a time-surface prototype which has the same structure of a time-surface. Hence, the same clustering algorithm used on time-surfaces can be applied to time-surface prototypes. For this reason, in [4], a hierarchical model for time-surface representation is proposed [Figure 3.1].
The model consists in repeating the clustering algorithm **L** times, in which the output of one clustering works as the input of the following. In particular, after each level of the hierarchy, referred as **Layers**, the output carries more information with respect to the previous output. Before entering the following Layer, the produced time-surface prototypes are filtered using the same exponential decay kernel function expressed in equation 3.3.
Layers of the hierarchical model are characterized by three parameters:

- $\mathbf{R}_l$: radius of the time-surface neighborhood;
- τ_l: time constant of the exponential kernel;
- $\mathbf{N}_l$: number of cluster centers, i.e. number of of time-surface prototypes.

At each level, R_l, τ_l and N_l, are multiplied by a factor K, different for each parameter:

$$R_{l+1} = K_R \cdot R_l \tag{3.8}$$
$$\tau_{l+1} = K_\tau \cdot \tau_l \tag{3.9}$$
$$N_{l+1} = K_N \cdot N_l \tag{3.10}$$

The advantage of a multi-layer model is in the accumulation of information among each layer. In this way, two entities with very close characteristics can be discerned by underlying small difference with the repeated clustering [4].
This Hierarchical model for time-surfaces representation is also called **Local Memory Time Surfaces** since past events need to be stored in memory elements to be used in time-surfaces prototypes computations.

3.1.3 Histograms of Averaged Time Surfaces

In [5], the concept of Local Memory Time Surfaces is further improved for implementing a more compact and robust time-surface representation.
The grid of pixels in the camera are divided into cells $\{C_l\}_{l=1}^{L}$ of size KxK. Histograms are then extracted from each cell, by adding together the time-surfaces representing events inside the cell, as reported in equation 3.11:

$$\overline{\mathbf{h}}_C(\mathbf{u}, p) = \sum_{e_i \in C} T_{e_i}(\mathbf{u}, p) \tag{3.11}$$

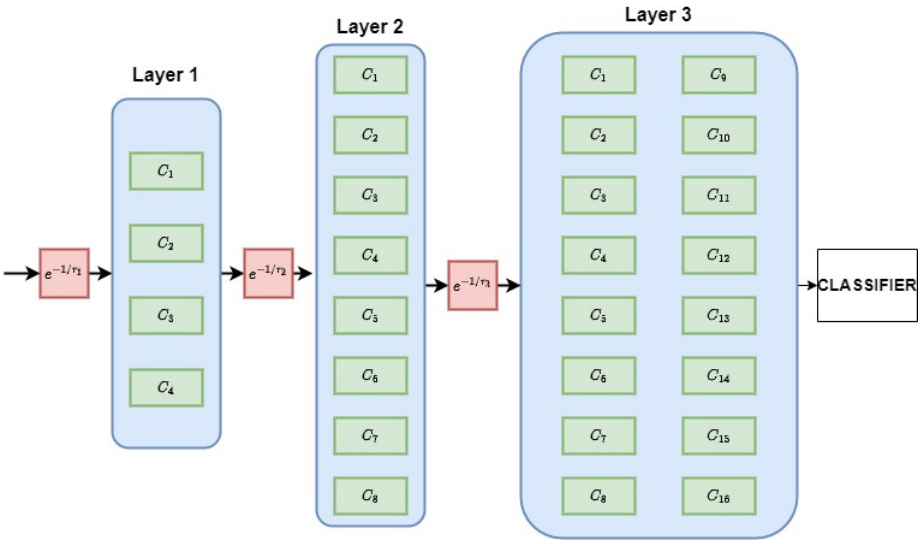


Figure 3.1: Overview of the Hierachical Time-surfaces model. When an input feature matches a prototype (green boxes) inside the layer, an output is reported. Each input is convoluted with an exponential decay kernel (red boxes) before entering the next layer (blue boxes). Image adapted from [4].

with $\overline{\mathbf{h}}_C$ representing the histogram for cell C, and e_i the events inside C.
The output of an event cameras depends also on contrasts, since objects with higher contrast produce more events than object with lower contrast. In [5], a solution to avoid dependence on contrast for histograms representing cells is proposed. Histograms are indeed normalized according to the number of events $|C|$ occurred in the spatio-temporal neighborhood used to compute the histogram, as reported in equation 3.12.

$$\mathbf{h}_c(\mathbf{z}, p) = \frac{1}{|C|}\overline{\mathbf{h}}_C(\mathbf{u}, p) = \frac{1}{|C|}\sum_{e_i \in C} T_{e_i}(\mathbf{u}, p) \tag{3.12}$$

As result, a stream of events is represented, with the approach used in [5], as a vector of cell histograms for each polarity p, position $\mathbf{u}$ and cells C (3.13).

$$\mathbf{H}(e_i) = [\mathbf{h}_{C_1}, ..., \mathbf{h}_{C_L}]^T \tag{3.13}$$

The descriptor introduced in [5] is referred as **HATS**, from Histograms of Averaged Time Surfaces.

3.1.4 Shared Memory Units

As stated in section 3.1.2, past events need to be stored in memory to be accessed for computation. Since events are asynchronous, memory needs to be accessed

asynchronously and with low latency in order to retrieve events in the neighborhood needed for the computation of the histogram representing the input event. In [5] a possible solution to the memory access issue is obtained by using shared memory units. In particular, in [5] it is noticed that for a neighborhood radius **R** equal to the cell width **K**, events in the same cell share an high amount of neighbor points used to compute the relative time-surfaces. In this way, it is not necessary to retrieve past events for all events inside a cell as in the case of a smaller radius. For this reason, a single memory unit $\mathbf{M}_C$ stores all the past events for events inside the relative cell C. With this configuration, when a new event is reported, events to compute time-surfaces are searched only inside $\mathbf{M}_C$, thus reducing access in memory.

Algorithm 1 HATS pseudo-code. Adapted from [5]

Input: Events $\mathcal{E} = \{e_i\}_{i=1}^{I}$
Initial Values: $\mathbf{h}_{C_l} = 0$, $|C_l| = 0$,$\mathbf{M}_{C_l} = 0$ for all l
for i=1,...,I **do**
  C_l = getCell(x_i,y_i)
  T_{e_i} = TimeSurface(e_i, $\mathbf{M}_{C_l}$)
  $\mathbf{h}_{C_l} = \mathbf{h}_{C_l} + T_{e_i}$
  $\mathbf{M}_{C_l} = \mathbf{M}_{C_l} \bigcup e_i$
  $C_l = C_l + 1$
end for
return $\mathbf{H} = [\mathbf{h}_{C_l}/C_l,...,\mathbf{h}_{C_l}/C_l]$

3.2 Classification Results

The proposed HATS method is born as a feature extractor. Hence, to asses its contribution to classification performances, in [5] outputs of HATS are used as features for a linear **Support Vector Machine** (**SVM**) classifier. Results of the SVM are then compared with the results of a **Spiking Neural Network**, with **H-First** [17] and with HOTS [4]. For the features extracted with HOTS, the same linear SVM approach as in HATS is used. For all methods, a 20% of the training data is used as a validation set. In this way it is possible to retrain the classifiers on all the training set to find the best configuration. All the cited approach are tested on five datasets:

- N-MNIST: collection of handwritten digits divided in 60000 training samples and 10000 test samples;
- N-Caltech101:images of objects divided in 100 categories plus a beckground;

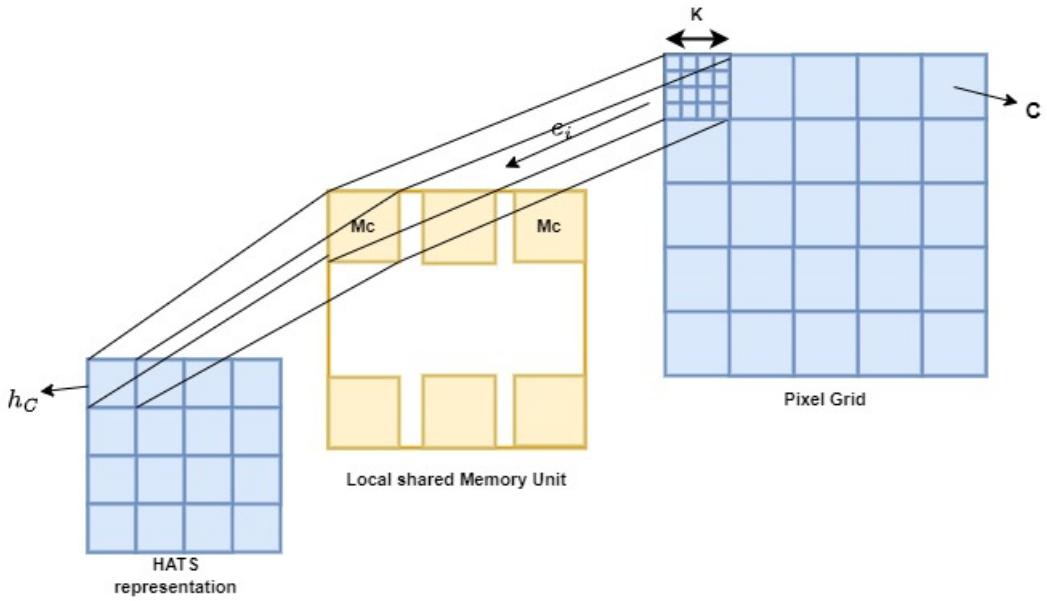


Figure 3.2: Overview of HATS architecture. Pixels are grouped in Cells C with size K x K. When an event e_i is generated, the neighbor events are searched inside the shared local memory M_C. In this way a time surface is computed and added to the correspondent histogram h_C. Image adapted from [5].

- CIFAR10-DVS: 10000 frame-based images converted in 10000 event streams;
- MNIST-DVS: 300000 DVS recording of handwritten digit;
- NCARS: collection of objects in urban environments recorded with an ATIS mounted on a car.

N-MIST and N-Caltech101 are converted from the standard MNIST and Caltech101 datasets by acquiring each image inside the datasets with a moving ATIS [see 2.1.2]. MNIST-DVS and CIFAR10-DVS are created by recording a moving image on a monitor with a non-moving DVS [5].
The conversion is necessary to transform frame-based images into a stream of events suitable for event-based classifiers.
NCARS dataset is instead a dataset created by the authors of [5] and it is a first example of data recording using DVS in a real situation. As reported in [5], HATS outperforms all the other classification models and with a large margin, as can be seen in Table 3.1. In [5], performances of the tested classifiers are compared also in terms of latency and computational time.
Average Computational time is computed as the ratio between the total time required to extract features on the whole training set and the amount of samples in the training set.

	N-MNIST	N-Caltech101	MNIST-DVS	CIFAR10-DVS	N-CARS
H-First [17]	71.2	5.4	59,5	7,7	56,1
HOTS [4]	80,8	21,0	80,3	27,1	62,3
SNN	83,7	19,6	82,4	24,5	78,9
HATS [5]	99,1	64,2	98,4	52,4	90,2

Table 3.1: Comparison of classification accuracy (%) on five datasets. Table adapted from [5].

Latency is instead defined as the time window used to accumulate information in order to classify a sample.
Results in [5] show how HATS reaches a 20x speed-up factor with respect to the HOTS and a 40x factor with respect to SNN. For what concern latency, there is

N-CARS	Average Time per Sample (ms)
HOTS	157.57
SNN	285.95
HATS	7.28

Table 3.2: Comparison of average time per sample (ms) between HOTS, HATS and SNN on N-cars dataset. Table adapted [5].

a trade-off between the length of the time window and accuracy: with a narrower time window, even if latency is reduced, the number of collected information is low and leads to incorrect classification. In [5], HATS reaches an accuracy of 90,2% on the N-CARS dataset with a latency of $l = 100ms$. For latencies lower than 20 ms, the resulting accuracy goes below 80%.

Chapter 4

Digital Design

In this section, the main contribution of this thesis will be reported. In particular, the goal of this thesis is to implement an **Hardware Accelerator** based on hyperdimensional computing for image classification, with images captured with Dynamic Vision Sensors. The objective is also to design an hardware module which is efficient and which can be implemented in different FPGA platforms. Indeed, hyperdimensional computing is based on the utilization of words with high dimensionality, around 10000 bits, and for this reason not all FPGA have the necessary resources to implement an hyperdimensional module. Hence, the hardware design is configurable, i.e. the parallelism of the words used at each iteration can be specified by the user.
The proposed classifier was developed in collaboration with Ing. Fabrizio Ottati. The design of the classifier is divided in two parts: a training phase made in software, and a test phase implemented in hardware. The software implementation is used not only to train hypervectors which represent the classes of the model, but also for accuracy comparison with the test phase implemented in hardware.
For classification, a set of features is needed to represent the characteristics of images. Given the results reported in [5], features of input images are extracted using the **HATS** model. The **HATS** feature extractor was reproduced in software by ing. Fabrizio Ottati.
The HATS module produces as output a series of **histograms** which represent the features of images to be classifies, with a single histogram associated to each image. The design is tested on the **N-MNIST** dataset, which is a set of images of handwritten digits [see 3.1.1]. The dataset is divided in 60000 samples for training and 10000 samples for testing, with digits belonging to 10 different classes.

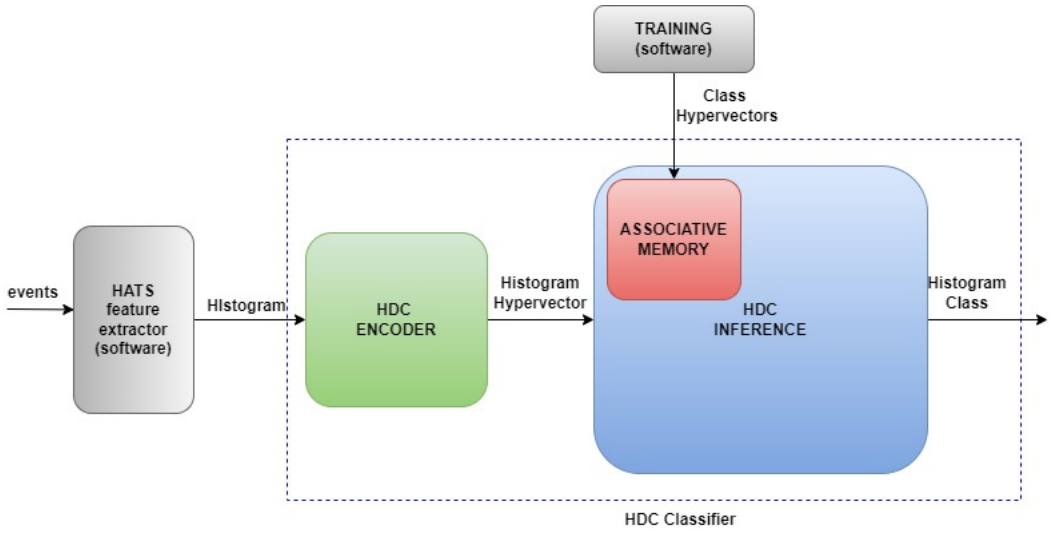


Figure 4.1: Overview on the implemented HDC classifier.

4.1 Proposed HDC Model Description

4.1.1 Encoding Phase

The HATS feature extractor presented in [5], and reproduced via software by ing. Fabrizio Ottati, represent samples of the dataset as histograms. In particular, histograms are organized as follows [Figure 4.2]:

- **2 Polarities**, representing the ON and OFF polarity that each event reported by a DVS can assume [see 3.1.1];
- **49 cells** for each polarity. Cells are organized in a matrix of 7x7 cells, referred in this work as *polarity matrix*;
- **25 subcells** for each cell. Subcells are arranged in a matrix of 5x5 subcells, referred in this work as *grid of cells*.

for a total of 2450 values in each histogram.

During encoding phase, each histogram is mapped into a single hypervector. This is done at first by encoding a value and its coordinates inside the subcell into an hypervector; the produced cell hypervector is then combined with other cell hypervectors inside the matrix representing a single polarity; finally, the two hypervectors representing negative and positive polarity are xored together, thus producing as output an hypervector which contains information about each feature value and its position inside the histogram [Algorithm 2].

Each value of the feature is mapped into a **Level Hypervector**. Level hypervectors are hypervectors which represent the possible values that an input feature can

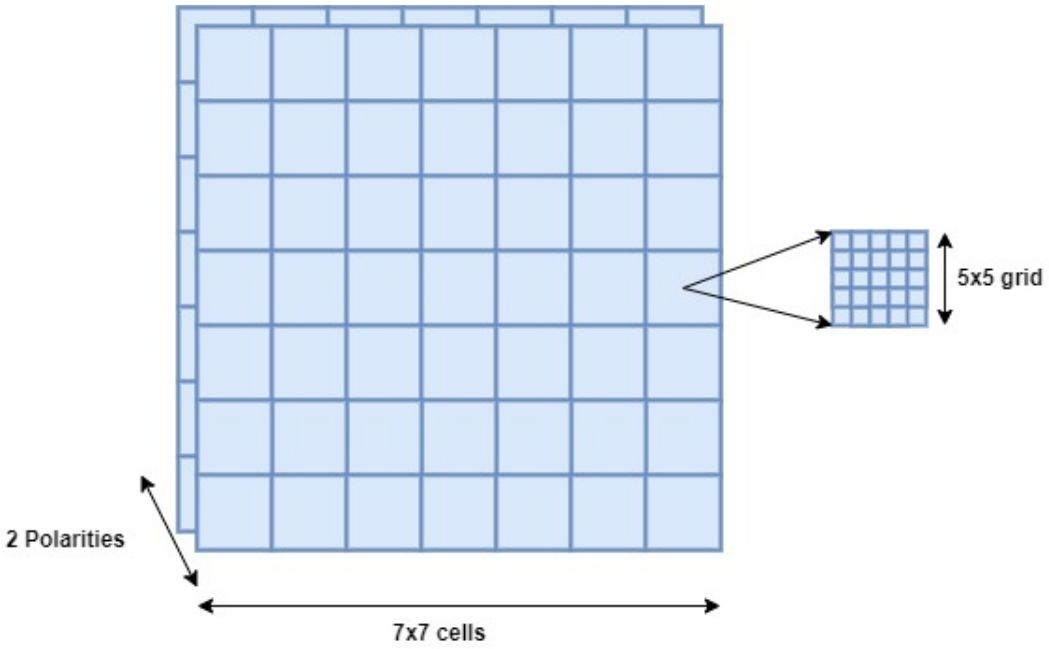


Figure 4.2: Overview on features organized as histograms with the HATS [5] method

assume [see 1.3.4]. Level hypervectors must preserve the similarity between values: two input features with similar values shall be mapped into two level hypervectors with a small percentage of different bits, whereas two different input values shall be translated into two orthogonal hypervectors. For this reason, Level Hypervectors in this project are produced by starting from a single hypervector which represents the minimum level, called **seed hypervector**, whose parts are progressively complemented in order to obtain the next levels, see 4.3.2 for further details.
Each level hypervector is then **binded** to the cell it belongs. As mentioned before, values in the histogram are organized in matrix of cells. The position inside the matrix is described by a set of two coordinates (**x**, **y**), which represent respectively the column and the row of the cell to which the input value belongs. An hypervector is associated to each possible values of x and y, for a total of 7 **column hypervectors** and 7 **row hypervectors** for the polarity matrix, and 5 **column hypervectors** and 5 **row hypervectors** for the grid of cells.
In this work, instead of storing all possible **position hypervectors** inside a memory [see 1.3.3], they are computed when needed starting from a **seed position hypervectors**. Since four coordinates are needed, x and y for the grid of cells and x and y for polarity matrix, only four orthogonal **seed position hypervectors** are stored. Those seed position hypervectors represent the first column/row of each matrix. The next rows and columns are computed by **rotating N-times** the proper seed hypervector, with **N = index of the row/column** [Figure 4.3].

An hypervector representing a single cell is obtained by accumulating hypervectors

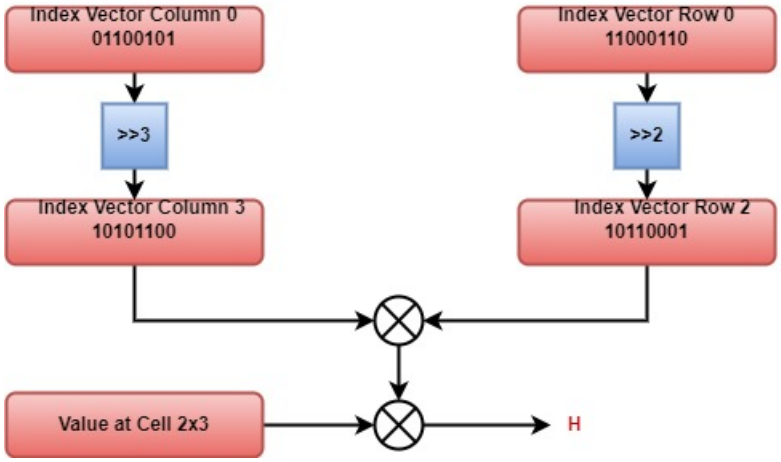


Figure 4.3: Example of a cell encoding. The seed hypervectors representing the column 0 and the row 0 are rotated respectively by three positions and two positions in order to obtain the position hypervector related to the column number 3 and the row number 2. Position hypervectors are then binded together before being binded with the level hypervector that represents the feature value inside the cell at position 2x3.

of each subcell inside the grid, and then performing a majority voting. Hypervectors inside the grid of cells are added together through a component-wise addition, producing as output a non-binary cell hypervector. The cell hypervector is henceforth binarized via thresholding: if the values accumulated inside each component are greater or equal than the number of the accumulated hypervectors, they are represented with a '1'; in the opposite case, they are represented with a '0'. The same accumulation and thresholding process is repeated to obtain the polarity hypervector.
Once both polarity hypervectors are computed, they are binded together by rotating the negative polarity hypervector and then xoring it with the positive polarity hypervector.

Algorithm 2 Encoding pseudo-code.

```
Input: histogram h (2x7x7x5x5)
Initial Values: hist_hv = "0"; pol_hv = "0"; cell_hv = "0"; input_hv = "0";
subcell_hv = "0"; grid_hv = "0".
for p=0,1 do
  for cy=0,...,6 do
    for cx=0,...,6 do
      for y=0,...,4 do
        for x=0,...,4 do
          input_hv = vectorize(h[p,cx,cy,x,y])
          subcell_hv = input_hv xor rotate(grid_col_seed,x) xor ro-
          tate(grid_row_seed,y)
          grid_hv = grid_hv + subcell_hv
        end for
      end for
      grid_hv = binarize(grid_hv,25)
      cell_hv = grid_hv xor rotate(cell_col_seed,cx) xor ro-
      tate(cell_row_seed, cy)
      pol_hv = pol_hv + cell_hv
    end for
  end for
  pol_hv = binarize(pol_hv,49)
  hist_hv = pol_hv xor totate(hist_hv,p)
end for
return hist_hv = hypervector representing the entire histogram
```

Algorithm 3 Binarization pseudo-code.

```
Input: hv = hypervector, n = number of accumulated vectors, D = length of the
hypervector.
for i = 0,....,D-1 do
  if hv[i] ≥ n then
    bin_hv[i] = '1'
  else
    bin_hv[i] = '0'
  end if
end for
return bin_hv = binarized hypervector
```

4.1.2 Training Phase

The second step for the classification model is represented by **Training**. This phase consists in creating a number of hypervectors equal to the number of possible classes in which samples from the dataset are divided. These hypervectors are referred as **Class Hypervectors**, stored in the **associative memory** (**aM**) [see 1.3.3].
A class hypervector shall represent the main characteristics of all hypervectors related to the samples belonging to that particular class. In order to do this, the properties of HDC are exploited. As illustrated in section 1.3.2, addition of hypervectors gives as output a resulting hypervector which is **similar** to all the addends. Hence it is possible to create a class hypervector which is similar to all the hypervectors inside the class it represents by adding those hypervectors. The operation of adding hypervecctors is also called **bundling**.
Addition is pointwise, thus all dimension are added independently from each other. Components inside the result hypervector are however non-binary. For this reason a binarization process is needed to transform a non-binary class hypervector into a binary hypervector. Binarization works as in the encoding phase, i.e. through thresholding.
In conclusion, training consists in encoding an input hypervector and then adding it to its class hypervector.
Once this process is repeated on the whole training set, class hypervectors are binarized according to the number of accumulated vectors [Algorithm 4].

Algorithm 4 Training pseudo-code.

```
Input: histogram h (2x7x7x5x5)
Parameters: L = length of the dataset; C = number of classes.
Initial Values: class_hv[0:C-1] = "0", accumulated[0:C-1] = 0
for i=0,...,L-1 do
    hist_hv = encode(h[i])
    class_hv[label] = class_hv[label] + hist_hv
    accumulated[label] = accumulated[label] + 1
end for
for i=0,...,C-1 do
    class_hv[i]= binarize(class_hv[i], accumulated[i])
end for
return  class_hv[0:C-1] = set of class hypervectors
```

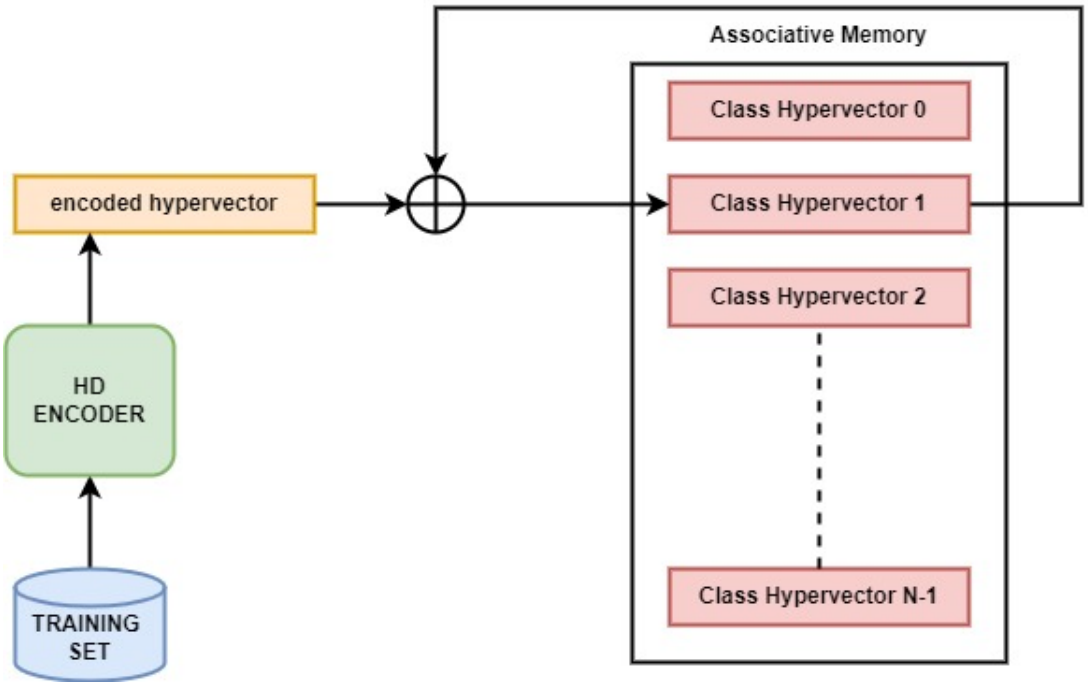


Figure 4.4: Overview of HD training process. Once a sample from the training set in encoded into an hypervector, it is added to its corresponding class inside the Associative Memory.

4.1.3 Inference

Inference is the process in which the classifier assign a class to an input **query hypervector**. The performances of a classifier are indeed evaluated by comparing the number of query hypervector assigned to the correct class with respect to the total amount of tested hypervectors. The ratio between the number of correct predictions and the total amount of predictions represent the **accuracy** of the model.

$$accuracy = \frac{\#\text{correct predictions}}{\#\text{total amount of predictions}} \tag{4.1}$$

In the proposed design, once a histogram is mapped into an hypervector, it is compared to each class hypervector inside the associative memory. Being the proposed HDC model binary, similarity between the query hypervector and class hypervectors is computed using **Hamming distance** [see 1.3.5].
Durint the Inference phase, the Hamming distance is computed between the query hypervector and all the class hypervectors. The query hypervector is then assigned to the class with the minimum distance [Algorithm 5].

For what concern the hardware implementation of inference, the usage of Hamming distance represents a great advantage with respect to other similarity measurements, as for example **cosine similarity**, since it is realized by a bit-wise xor between the query and the class hypervector, together with a **counter of ones**. Hence, there is no need for multipliers, which are costly in terms of hardware resources.

Algorithm 5 Inference pseudo-code.

```
Input: histogram h (2x7x7x5x5)
Parameters: C = number of classes.
Initial Values: distance = 0; min_dist = MAX_VALUE.
hist_hv = encode(h)
for i=0,...,C-1 do
  distance = count_ones(hist_hv xor class_hv[i])
  if distance ≤ min_dist then
    min_dist = distance
    predicted_class = class[i]
  end if
end for
return predicted_class = class to which the input hypervector most likely belongs
```

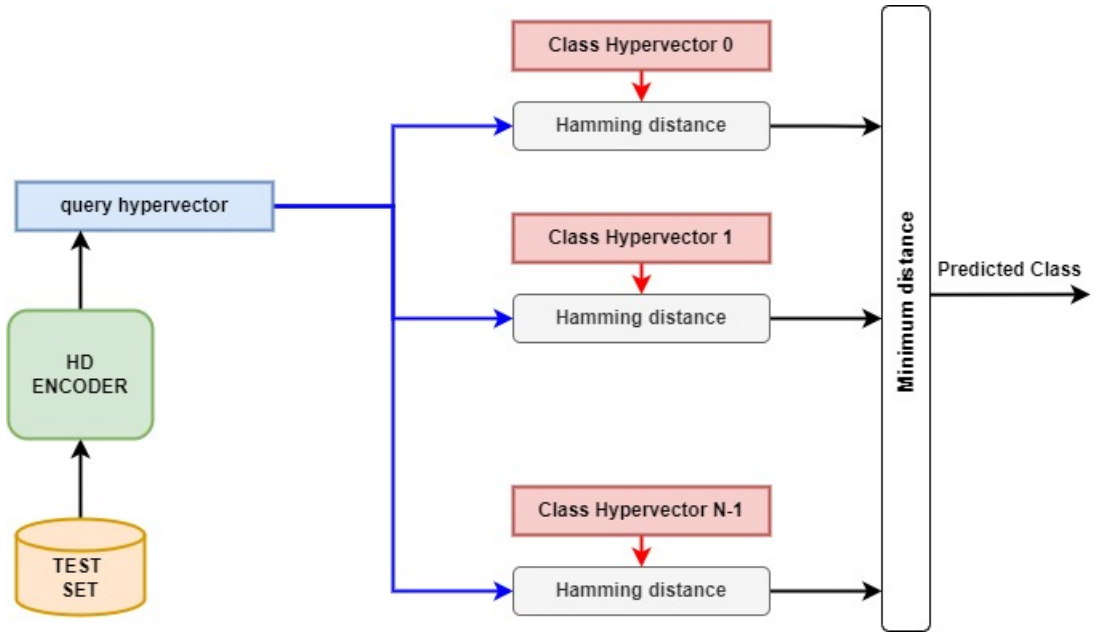


Figure 4.5: Overview of HD inference process. Once a sample from the test set in encoded into an hypervector, it is compared through Hamming distance to each class hypervector inside the Associative Memory. The query hyprvector is associated to the class with the minimum Hamming distance.

4.2 Design Choices

When talking about an HDC model, three are the main design choices to be specified: the **dimension D** of the hypervectors, the **bitwidth** of the input data, and the data representation, i.e. if the components of the hypervectors are binary or not.
The HDC model proposed in this work is a **dense binary model** with hypervectors of length **D = 8192** and with input data written on **3 bits**.
The dimensionality of hypervectors is chosen as the **closest power of 2** to the typical dimensionality of hypervectors, i.e. D = 10000, in order to simplify hardware description.
For what concern the bitwidth of the input data, input values are represented on **3 bits** to reduce the usage of unnecessary bits. In addiction, using input data with reduced bitwidth leads to a more robust system, as reported in [8] [see 1.2.1]. Hence, input data can assume all integer values between **0** and **7**, extremes included.
As mentioned before, one of the objective of this work is to design an HDC module that can be implemented on FPGAs without excessive hardware requirements. Hence, implementing a design which uses all the 8192 bits of the hypervector can be inconvenient.

For this reason, in this work the hardware design is serialized: each hypervector is not processed in a single step, but it is divided in **parts** with lower parallelism that are processed in different steps.
In this way, hardware requirements are reduced and also, by choosing the proper length of hypervector parts, it is possible to adapt the hardware design to a target FPGA [see 7].

4.2.1 Serialization

Serialization of the hardware introduces two new parameters: the number of hypervector parts **P**, and the bitwidth of hypervector parts **N**. These two parameters are correlated since $N = D/P$, i.e. the number of bits in each part of the hypervector depends on the dimension of the hypervector **D** and the number of parts **P** in which the hypervector is divided.

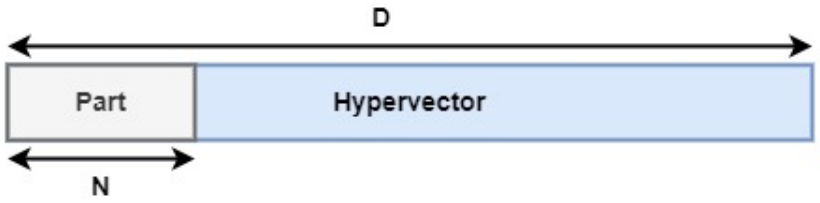


Figure 4.6: Overview of Hypervector division. The hypervector with dimension D is divided in parts with dimension N.

With serialization, each phase of the HDC model is repeated P = number of parts time. This means that encoding is not performed in a single step. Hence, if in a full parallel configuration an input sample is directly mapped into an hypervector, with a serialized configuration the sample must be read P times in order to create the P parts of the input hypervector [see 4.1.1]. This implies the necessity of an input memory which stores each value of the input histogram, since values are read at each iteration of the encoding phase.
An histogram is made by 2450 values represented on 3-bits, therefore the input memory is characterized by a **depth DM = 4096** and a **width W = 3**, in order to store the whole histogram. From now on, the input memory will be referred as **histogram memory**.
The histogram memory is organized as follows: rows from 0 to 1224 store values of the negative polarity matrix, while rows from 2048 to 3272 store values of the positive polarity matrix. Values are order according to both their coordinates inside the Matrix and their coordinates inside the grid of cells [Figure 4.7].

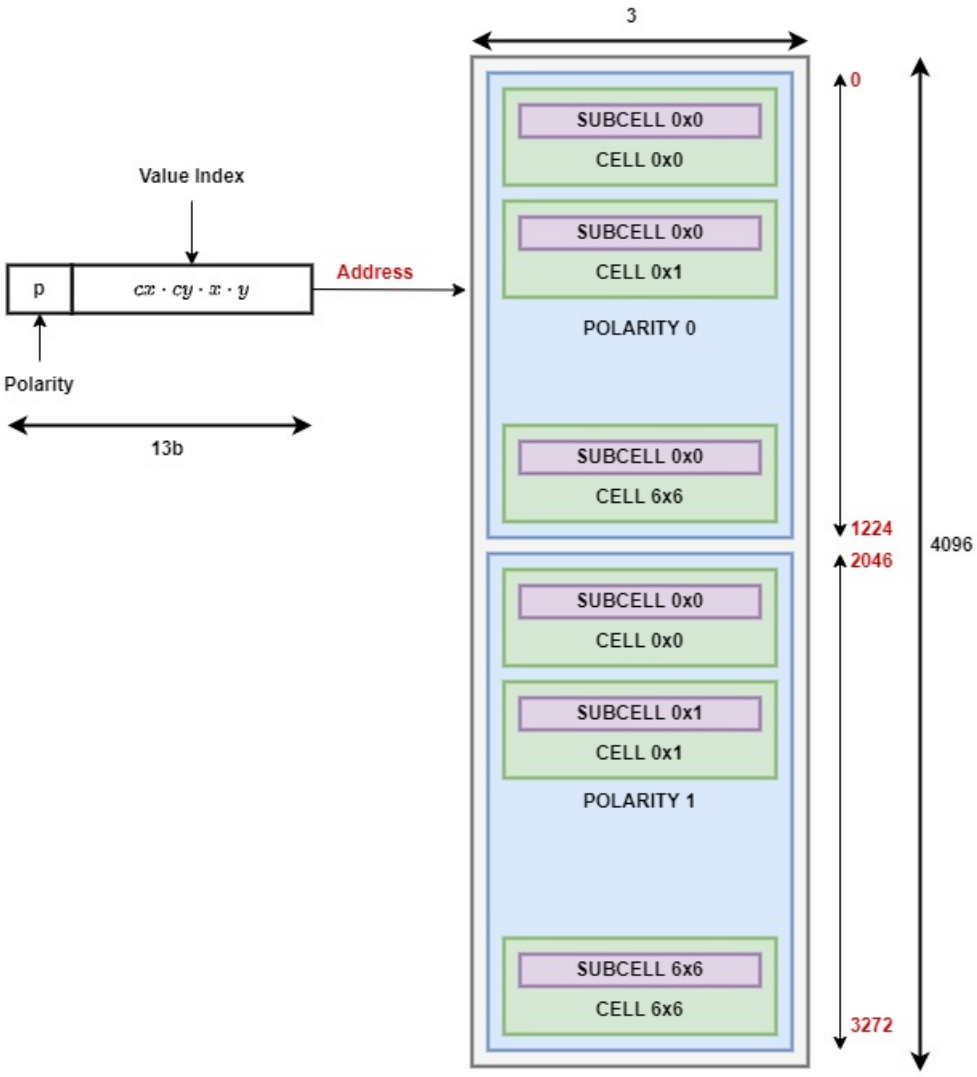


Figure 4.7: Overview on the Histogram Memory organization and addressing. Values of the input histogram are ordered according to their polarity **p**, their coordinates inside the cell (**cx**,**cy**), and their coordinates inside the subcells (**x**,**y**). The address of a value is formed by the MSB which corresponds to the polarity of the value (0,1), and by a sequence of bits representing the index of the value inside the polarity matrix.

As stated before, the implemented design is configurable by the user. This is done by describing each hardware block in RTL as a generic component. User can specify the number of parts of the hypervector by running a python code which writes the chosen parameters inside a **constants package** which is used for the hardware implementation.
The design can be configured to process from 8 parts of hypervector with 1024 bits

each, to 1024 parts of hypervector with 8 bits each. In this work, four of the possible configurations are tested: 64 parts of hypervector with 128 bits each (**64x128**); 32 parts of hypervector with 256 bits each (**32x256**); 16 parts of hypervector with 512 bits each (**16x512**) and 8 parts of hypervector with 10124 bits each (**8x10124**). A comparison in terms of resource requirements, time and power between the four implementations is reported in chapter 7.

4.2.2 Seed ROMs and Associative Memory

An important role in the encoding phase is played by **seed hypervectors**. As explained in section 4.1.1, during the encoding phase, an input value is mapped into an hypervector and then binded to the hypervectors which represent the position of the value inside the grid of cells or the polarity matrix. An hypervector is needed to represent each position. Hence, for the 5x5 grid of cells, 5 hypervectors are needed to represent rows and 5 hypervectors are needed to represent columns, while for the 7x7 polarity matrix, 7 hypervectors are needed for the rows and 7 hypervectors are needed for the columns, for a total of 1225 position hypervectors. Given the dimension D = 8192 of each hypervector, if all the position hypervectors would be stored, a total of 1225x8129 = 10035200 bits are deployed, which is a huge number of bits for just base vectors.
To avoid the usage of excessive bits, position hypervectors are computed starting from a single hypervector which represent the position 0, i.e. the **seed hypervector** [Figure 4.3]. For four coordinates, four seed hypervectors are needed. Seed hypervectors are random and mutually orthogonal, and since their value is fixed at the beginning of the algorithm, each of them is stored in a **ROM**, referred in this work as **seed ROM**. Due to serialization, position hypervectors are divided in parts; therefore each line of the seed ROMs stores a part of the seed hypervector [Figure 4.8].

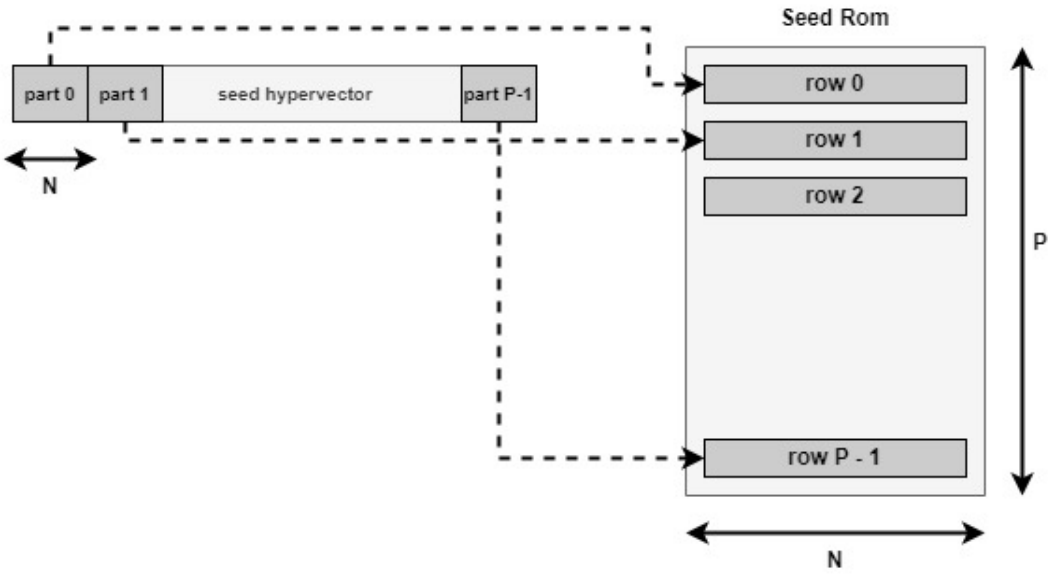


Figure 4.8: A seed hypervector stored in a Seed ROM. Each row of the Seed ROM stores a part of the seed hypervector.

Since a seed hypervector corresponds to the position hypervector of position 0, other position hypervectors are obtained by rotating the seed hypervector by a number of bits equal to the position, e.g the seed hypervector is rotated three times to obtain the position hypervector of position 3 and 6 times to obtain the position hypervector of position 6. Serialization however introduces an issue in rotation, since rotating a part of the hypervector is different from rotating the whole hypervector [Figure 4.9].

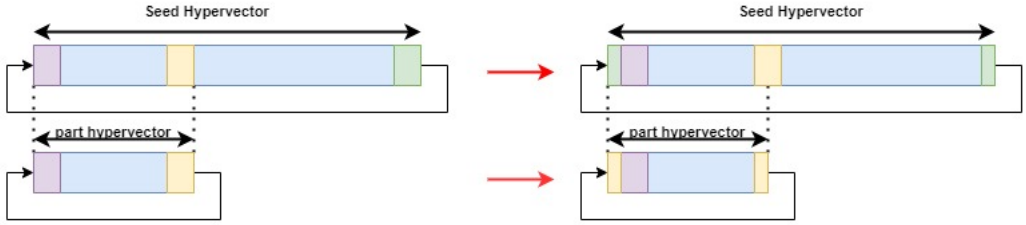


Figure 4.9: Example of seed vector rotation. Since the part hypervector does not store all bits, rotating a part hypervector gives a different result from rotating the whole hypervector.

The solution to this problem is to store additional bits in each row of the seed ROM. Those additional bits represent the input bits in case of a full hypervector rotation. Since the row and column hypervectors for the grid of subcells are rotated for a maximum of 4 positions, 4 additional bits are inserted at the beginning of

each row of their respective seed ROMS [Figure 4.10]
For the same reason, since the row and column hypervectors for the polarity matrix are rotated for a maximum of 6 positions, 6 additional bits are inserted at the beginning of each row of their ROMS. Given P = number of parts of the hypervector, 4xP additive bits are needed for the seed ROMs related to the grid of subcells, whereas 6xP additive bits are needed in the case of the polarity matrix.

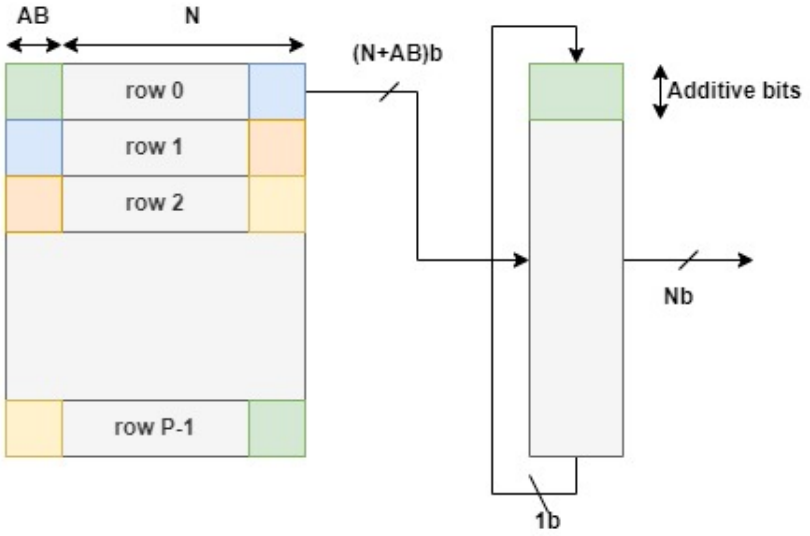


Figure 4.10: Seed ROMs with additive bits. Starting from row 1, each row store the last AB bits of the previous hypervector. At the contrary, the row 0 stores the last AB bits of the last hypervector.

In addition to the four seed ROM used to store each position hypervector, a fifth ROM is used to store the **CiM seed vector**. As reported in section 1.3.3, the Continuous Item Memory (CiM), stores **level hypervectors**, i.e. hypervectors which maps the possible values that an input variable can assume. In the design implemented in this work, data are represented on 3 bits, therefore they can assume 8 possible values. Hence, the CiM should store 8 level hypervectors. As in the case of position hypervectors, it is however possible to obtain all other level hypervectors by starting from the level hypervector representing the value 0, referred **CiM seed hypervector**. Therefore, there is no need for a memory which stores all the level hypervectors, but just a seed ROM storing the level hypervector 0, referred as **CiM seed ROM**. Details about the process used to obtain a generic level hypevector from the level hypervector 0 are reported in 4.3.2. Since the CiM seed hypervector is not rotated, no additional bits are inserted in the rows of the CiM seed ROM.
An other fundamental memory block for an HDC model is represented by the **Associative Memory** (**aM**), i.e. the memory which stores all class hypervectors. In this work, the training phase is executed in software, hence the associative memory

is updated offline. Therefore, since the hypervectors inside the associative memory are fixed and known before the start of the hardware operations, the aM is implemented in hardware as a ROM storing all classes hypervectors. Given the serialization, each class hypervector is divided in parts, hence the associative memory ROM is divided in **C = number of classes** block, with P = number of parts hypervectors.
The address for the associative memory is then divided in two parts: a **tag** which points to the class block, and an **offset** which points to the part hypervector inside the block. Since the number of classes for the dataset used in this work is C = 10 classes, the tag of the address has a length of 4bits. The number of bits for the offset depends on the number of parts.

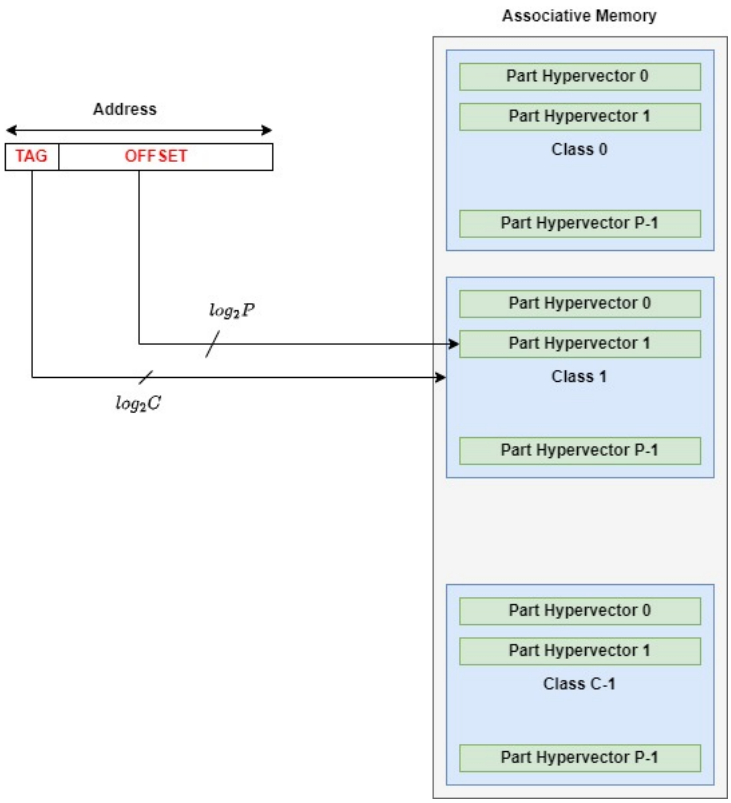


Figure 4.11: Overview on the implemented associative memory. The aM is divided in C = number of classes blocks, with P = number of parts hypervectors inside each block. The address for the associative memory is composed by a tag, which points to the blocks of the aM, and an offset which points to the part hypervector inside the block.

4.3 Encoder Module

4.3.1 Encoder Building Blocks

The firs block of the **Encoder Module** is represented by the **histogram memory** [Figure 4.7]. The histogram memory is composed by 4096 rows with 3 bits each. As illustrated in section 4.2.1, only 2450 values of the histogram memory comes from the input histogram. The other rows are initialized to 0, but they are not used for encoding.
The address of the histogram memory is divided in two segments: the first segment (MSBs) represent the polarity of the addressed value, while the remaining bits indicate the index of the value inside its polarity matrix.
Both the components of the histogram memory address are generated by counters. The MSBs of the address is generated by a counter referred as **polarity counter** which compute how many polarities are processed. The parallelism of the polarity counter is $\mathbf{BPL} = \lceil log_2 NP \rceil$ with NP = number of polarities. In this project, only two polarities are used, hence NP = 2, meaning that BPL = 1 bit.
The second counter, called **memory address counter**, generates the remaining bits of the histogram memory address. The parallelism of the memory address counter is $\mathbf{DM} = \lceil log_2 NV \rceil$, with NV = number of values for each polarity. Since for each polarity matrix 1225 values are present, DM = 11 bits. The total number of bits for the histogram memory address is then BPL + DM = 12 bits.
After the **start** signal, which indicates that the encoding process has began, input values are stored inside the histogram memory at each cycle.
When both terminal counts of the polarity counter and the memory address counter are asserted, the encoder module stops memorizing values and starts the encoding phase.
Due to serialization, the encoding phase is repeated P times, with P = number of parts. In order to compute the number of parts already processed, a counter is employed. This counter, referred as **part counter**, has parallelism of $\mathbf{Pb} = \lceil log_2 P \rceil$. Once the part counter asserts its terminal count, the encoding phase is concluded, meaning that all parts of the **histogram hypervector** have been sent to the inference module.
The encoder module contains also five **seed ROMs** [see 4.2.2].
One of the seed ROMs is used as the input of the **CiM generator** [see 4.3.2]. This ROMs is called **CiM seed ROM** and has depth DPM = P = number of parts, and width WM = N = lengths of each part hypervector. Outputs of the CiM seed ROM represent the parts of the level hypervector which maps the 0 value and which is transformed into the other level hypervectors by the CiM generator.
Two of the seed ROMs, called **grid row seed ROM** and **grid col seed ROM**, store parts of the hypervectors which represent respectively the **row 0** and the **column 0** of the grid of subcells.

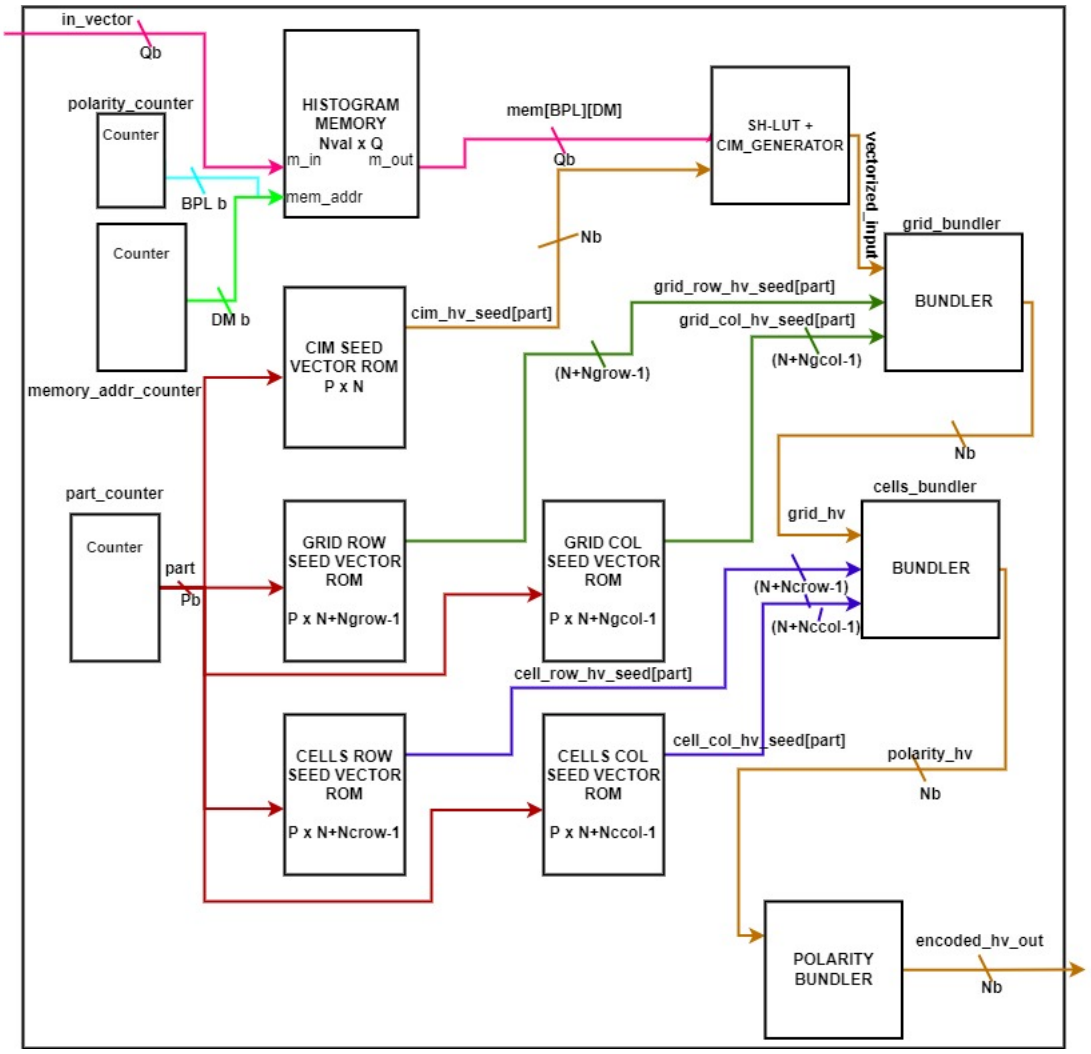


Figure 4.12: Overview of the encoder module.Each building block is reported with its name and parallelism.Different parallelisms are highlighted with different colours.

The grid row seed ROM has depth DM = P and width WM = N + Ngrow - 1, with Ngrow = number of rows in the grid of subcells.
The grid col seed ROM has depth DM = P and width WM = N + Ngcol - 1, with Ngcol = number of columns in the grid of subcells.
The width of these two memory is different from N because of the presence of the additive bits [see 4.2.2]. In this project, since the grid of subcells is a square matrix with 5 columns and 5 rows, Ngcol = Ngrow = 5, i.e. the grid row seed ROM and the grid col seed ROM have the same size. Outputs of the two grid seed ROMs are used as inputs for the **grid bundler** [see 4.3.3].

The remaining two seed ROMs, called **cells row seed ROM** and **cells col seed ROM** store parts of the seed hypervectors which represent the row 0 and column 0 inside the polarity matrix.
The cells row seed ROM has depth DM = P and width WM = N + Ncrow - 1, with Ncrow = number of rows in the polarity matrix.
The cells col seed ROM has depth DM = P and width WM = N + Nccol - 1, with Nccol = number of columns in the polarity matrix.
Also in these two seed ROM, the width is different from N because of the additive bits. The polarity matrix has 7 columns and 7 rows, hence Nccol = Ncrow = 7, meaning that the cells row seed ROM and the cells col seed ROM have the same size. Outputs of the two cells seed ROMs are used as inputs for the **cell bundler** [see 4.3.3].
All five seed ROMs are addressed using the output of the part counter. This in order to use the proper part of seed hypervectors at each iteration.

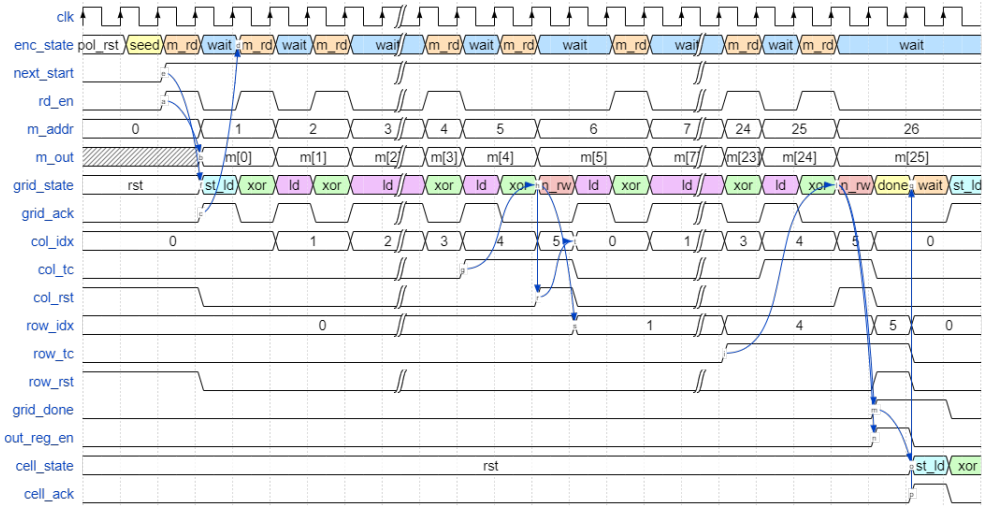


Figure 4.13: Portion of the encoder timing diagram.State transitions of the grid bundler and cell bundler are also reported.

4.3.2 CiM Generator

The CiM generator is a block used to map an input value into an hypervector. In baseline approaches, the vectorization of the input is achieved by mapping the value to a level hypervector, stored in the **Continuous Item Memory** (**CiM**). In particular, the input value is used as an address for the CiM, so that the output of the CiM is the level hypervector which represents the input value.
This method is however inconvenient, since it requires a large CiM for an higher number of possible levels. For this reason, in this project level hypervectors are not

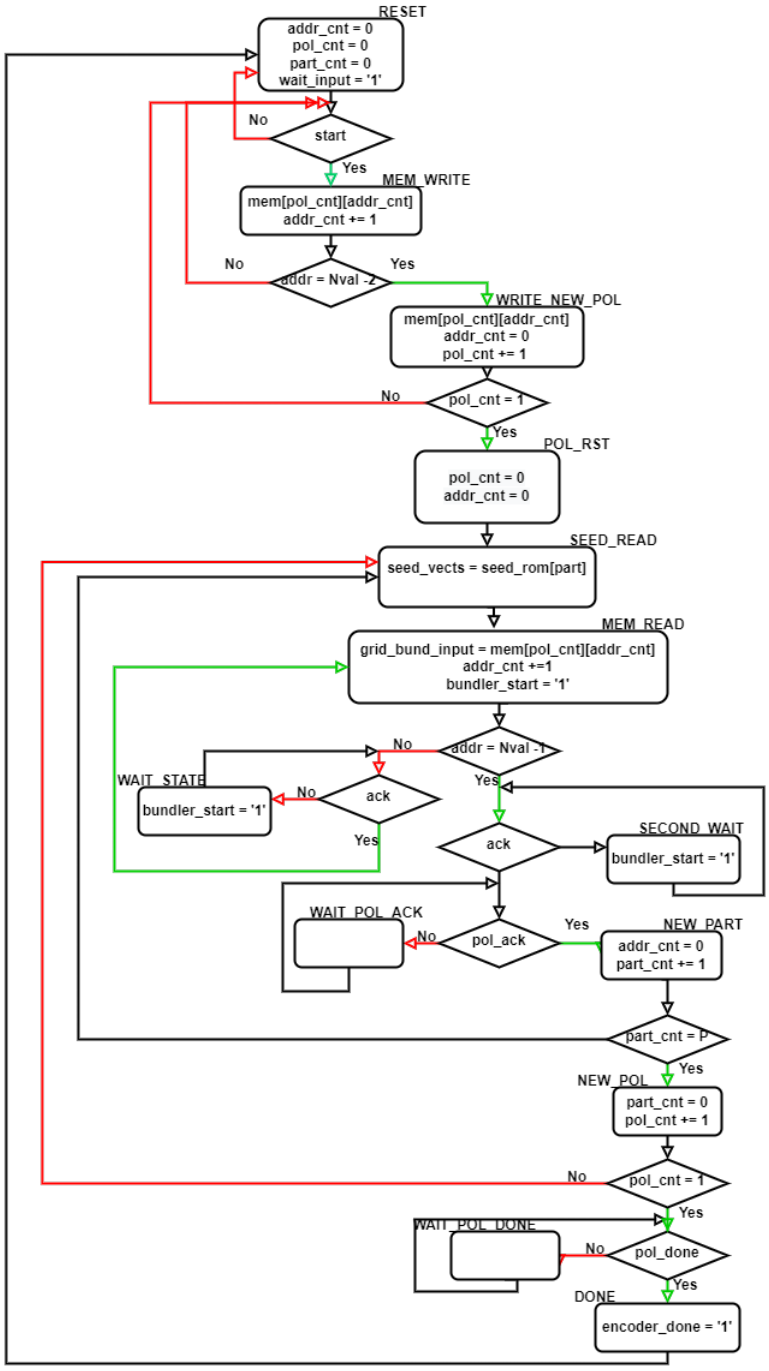


Figure 4.14: Sequence of encoder operations.

stored inside the CiM, instead they are created when needed by operating on a seed hypervector. This seed hypervector represent the zero level of the input values. The seed hypervector is at first divided in **NL = number of possible levels**

parts. Then, each other level hypervector can be computed by complementing a number of parts of the seed hypervector equal to the input value. For example, the level hypervector 3 is obtained by complementing three parts of the seed hypervector.
This approach grants a fundamental property of level hypervectors, i.e. that level hypervectors representing close values are similar while level hypervector representing opposite values are orthogonal. Indeed, with the proposed approach the difference between two input values is mapped as the number of different parts between the corresponding level hypervectors [Figure 4.15].
In hardware, the CiM generator is implemented by xoring the proper part of the

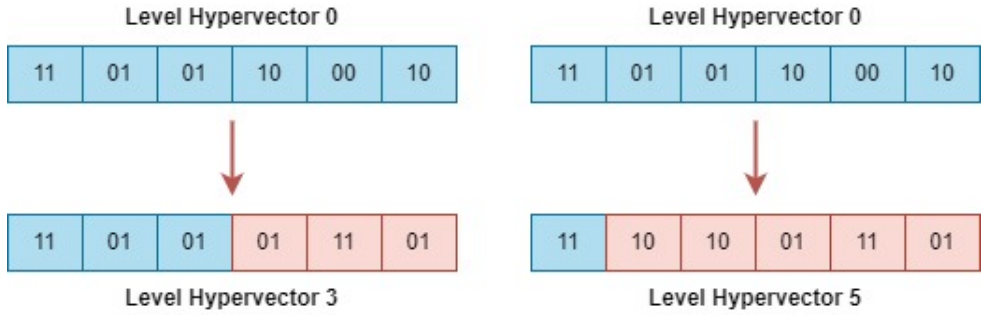


Figure 4.15: Example of two level hypervectors generated by complementing parts of a seed hypervector, with 6 possible levels.

CiM seed hypervector with a single bit coming from the **s-hot LUT**.
The s-hot LUT [2] is a LUT which takes an input of Q bits, and gives as output a stream of 2^Q bits where the number of bit set to '1' is equal to the input value. If the input value is equal to **s**, s consecutive bits, starting from the LSB, of the s-hot output are equal to '1', while the others are set to '0'. In this work, inputs of the s-hot LUT are the input data value. Since the input data value are written on $Q = 3$ bits, the output of the s-hot LUT is 8 bit wide. A multuplexer with 8 inputs of 1 bit length, a selection signal of three bits and a single bit output is used to select the proper bit from the output of the s-hot LUT to be sent to the CiM generator, in order to be xored with the target part of the CiM seed vector [Figure 4.16]. The selection signal of the multiplexer is obtained by dividing the output of the part counter by the length of the s-hot LUT output. Indeed, due to serialization, each bit of the s-hot LUT output is used $N = L/P$ times, with L = length of the s-hot LUT output, i.e. the number of possible level hypervectors, and P = number of parts in which the hypervector is divided. The sequence of operations done in the CiM generator are reported in Algorithm 6.

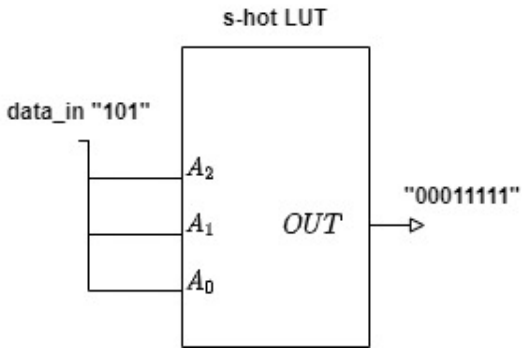


Figure 4.16: Overview on the s-hot LUT. For an input data written on 3 bits, the output is written on 8 bits. The number of bits set to '1' in the output is equal to the value of the input data.

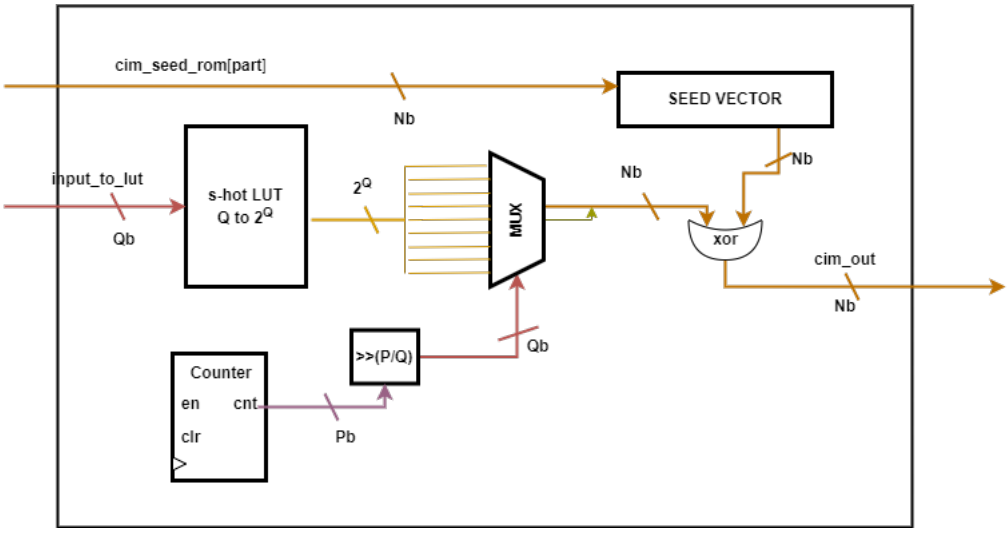


Figure 4.17: Overview on the datapath of the s-hot LUT together with the CiM generator. Signals with different parallelism are highlighted with different colours.

4.3.3 Cell and Grid Bundler

The function of the **Grid Bundler** is to bind an hypervector to the hypervectors representing its coordinates inside its cell in order to create a cell hypervector. Then, cell hypervectors are accumulated to create an hypervector which represent the whole matrix of cells.
This operation is repeated two times: at first to create an hypervector representing the gird of subcells, (**grid hypervector**), starting from level hypervectors which are obtained from input values with the CiM generator; then, all the grid hypervectors are binded with their coordinates inside the polarity matrix and added together

Algorithm 6 CiM generator pseudo-code.

Input: value[2:0]
Parameters: Q = bits of the input; L = 2^Q = length of the LUT output; P = number of parts; N = length of a part of the hypervector.
for i=0,...,P-1 **do**
  CiM_seed_vector = CiM_seed_ROM[i]
  out_lut= getSHLUTOut(value)
  bit_index = i / L
  CiM_out = CiM_seed_vector xor out_lut[bit_index]
end for
return CiM_out[N-1:0]= part of the level hypervector associated to the input value.

to form the **polarity hypervector** [see Algorithm 2].
Since the operation to create grid hypervectors and polarities hypervectors are the same, their are implemented with the same block. To avoid confusion, the first bundler which takes inputs from the CiM generator and produces a **grid hypervector** is referred as **grid bundler**; the second bundler which takes inputs from the first bundler and produces a **polarity hypervector** is referred as **cell bundler**.

The first block of the grid bundler is an input register which stores the output of the previous block. In particular, in the case of the grid bundler used to obtain the grid hypervector, the input comes from the CiM generator, whilst in the case of the grid bundler used for the cell hypervector, the input came from the first grid bundler. The parallelism of this register is N = D/P, where D = 8192 bits, i.e. the dimension of the hypervector, and P = number of parts introduced by serialization. Also two **shifter registers** are present at the input. They store the output of the seed ROMs which contain the position hypervectors related to the column index, in case of the **col seed ROM**, and the row index in case of the **row seed ROM** [see 4.2.2].
Shift registers are used to rotate the seed vector in order to obtain the proper position hyervector [see 4.1.1]. The parallelism of the shift register is NS = N + AB, where AB are the additive bit needed to obtain a rotation of the part of the seed hypervector consistent to the rotation of the whole seed hypervector [see 4.2.2]. However, only the first N bits of the output of the shift registers are used. This in order to perform a bitwsise operation between the output of the two shift registers and the output of the input register, thus producing the **subcell hypervector**, for grid bundler, or the **cell hypervector** for the cell bundler.

To produce the grid hypervector, or the polarity hypervector, all the subcell hypervectors, or the cell hypervectors, are added together and then binarized by thresholding. In hardware, this is implemented using **Up/Down Counters** (**U/D counter**). In particular, each bit of a subcell hypervector, or cell hypervector, is

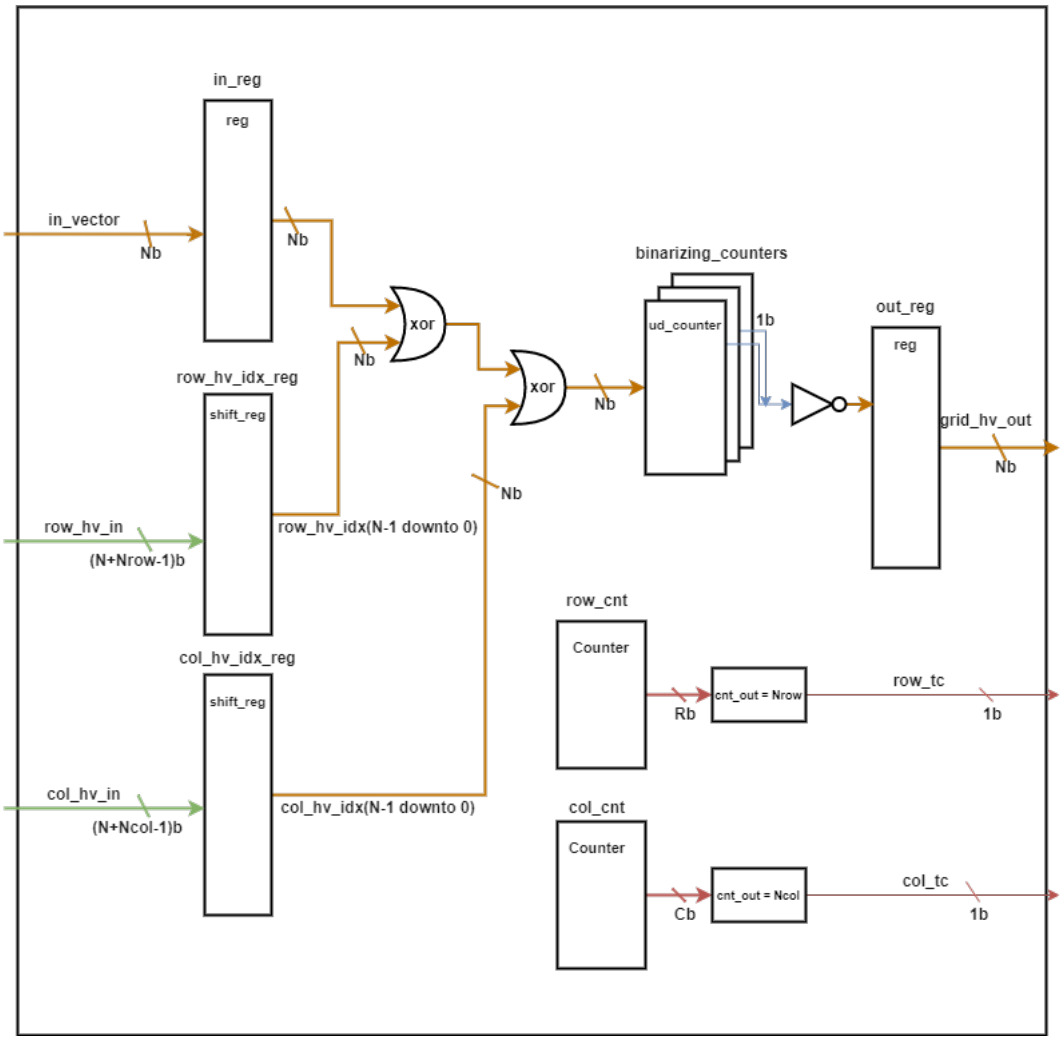


Figure 4.18: Overview on the datapath of the Grid/Cell Bundler. Signals with different parallelism are highlighted with different colours.

used as the up/down signal of the U/D counter. If the bit of the hypervector is '0', the value inside the counter is reduced by a factor -1, whilst if the bit is '1', the value of the counter is increased by 1. For an hypervector with N dimensions, N U/D counters are needed, one associated to each dimension. Using the U/D counter it is possible to know for each dimension if the number of '1'-bits in the accumulated hypervectors is higher than the number of '0'-bits. Indeed, if after accumulating all hypervectors the value inside the U/D counter is positive, i.e. the MSB is '0', there is a majority of '1' in that dimension, otherwise if the value is negative, MSB equal to '1', there is a majority of '0'. Since data inside the U/D counters can be positive or negative, they are represented in complement two. Hence, the parallelism of the U/D counters is $U = \lceil log_2(N_{row} \cdot N_{col}) \rceil + 1$, where N_{row} and N_{col} are respectively

Algorithm 7 Grid/Cell bundler pseudo-code.

```
Input: input_hv[N-1:0]
Parameters: N = bits of the part hypervector; N_c = number of column; N_r =
number of row; P = number of parts.
for i=0,...,P-1 do
  row_hv = row_seed_ROM[P]; col_hv = col_seed_ROM[P].
  for y=0,...,N_r do
    col_hv = col_seed_ROM[P]
    row_hv = rotate(row_hv,y)
    for x=0,...,N_c do
      col_hv = rotate(col_hv,x)
      subcell_hv = col_hv xor row_hv xor input_hv
      grid_hv = grid_hv + subcell_hv
    end for
  end for
  grid_hv_out = binarize(grid_hv)
end for
return grid_hv_out = part of the hypervector which encode the whole grid/-
matrix.
```

the number of rows and columns of the matrix, therefore the result of their product corresponds to the number of hypervectors inside the matrix, i.e. the number of accumulated hypervectors. For the grid bundler, $N_{row} = N_{col} = 5$, hence $U = 6$, while for the cell bundler $N_{row} = N_{col} = 7$, hence $U = 7$.
Once all hypervectors are accumulated with the U/D counter, the output hypervector is obtained by complementing the signal bit of the U/D counters for each dimension.
The output hypervector is then stored in the output register.
Two additional counters are used to indicate the column and the row of the input vector. Once the terminal counts of both counters are asserted, all hypervectors inside the grid of subcells or polarity matrix have been accumulated.

4.3.4 Polarity Bundler

The **polarity bundler** is used to bind together the two polarity hypervectors coming from the cell bunlder. In this way, the final hypervector which encode the whole histogram is obtained. This hypervector is referred as **histogram hypervector**.
The operations of the polarity bundler are divided in two phases. In the first phase, all the part hypervector of the first polarity vector are stored in a set of shifter registers with parallel inputs. In the second phase, the previously stored parts of polarity hypervector are shifted by one position and then progressively xored with

the part of the second polarity hypervector. A **part counter**, is used to count how many parts were processed for each polarity. A second counter, called **polarity counter**, indicates how many polarity were encoded. The parallelism of the part counter is $Pb = \lceil log_2 P \rceil$, with P = number of parts, while the parallelism of the polarity counter is $BPL = \lceil log_2 NP \rceil$, with NP = number of polarities. Since two polarities are encoded, NP = 2 and BPL = 1.

As shown in Figure 4.19, the first block of the polarity bundler is an input register

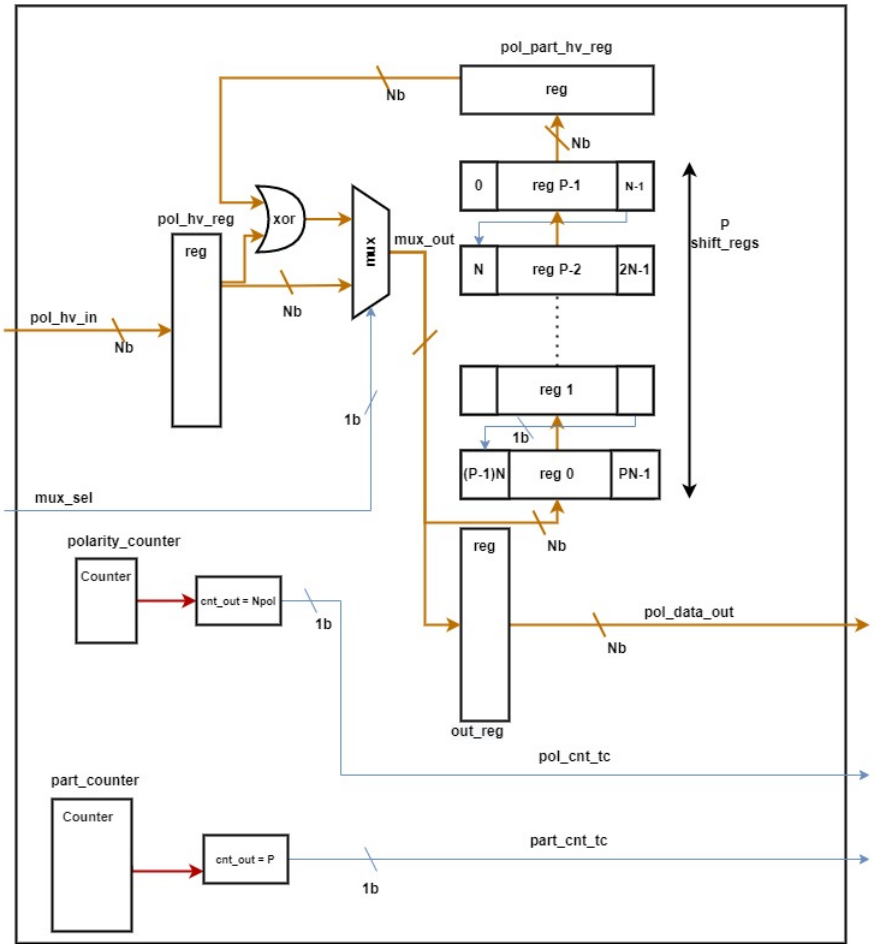


Figure 4.19: Overview on the datapath of the polarity bundler. Signals with different parallelism are highlighted with different colours

which stores the part of the polarity hypervector coming from the cell bundler. A multiplexer is used to select which is the input of the set of shifter register between the input hypervector, or the product of the xor between the input hypervector and the respective hypervector already stored.

The output register is used to store the part of the final histogram hypervector presented at the input of the inference module.

Algorithm 8 Polarity bundler pseudo-code.

```
Input: input_hv[N-1:0]
Parameters: N = bits of the part hypervector; P = number of parts; NP =
number of polarities.
for i=0,...,NP-1 do
  if i > 0 then
    shift_reg_hvs[P-1:0] = rotate(shift_reg_hvs[P-1:0],1)
  end if
  for p=0,...,P-1 do
    pol_part_hv = shift_reg_hvs[P-1]
    shift_reg_hvs[P-1:1] ← shift_reg_hvs[P-2:0]
    if i=0 then
      shift_reg_hvs[0] = input_hv
    else
      shift_reg_hvs[0] = input_hv xor pol_part_hv
    end if
    if i=NP-1 then
      pol_data_out = shift_reg_hvs[0]
    end if
  end for
end for
return pol_data_out = part of the hypervector which encode the whole his-
togram.
```

4.4 Inference Module

4.4.1 Inference Module Design

In standard conditions where all dimensions of an hypervector are processed together, inference is performed in a single phase, by comparing the **query hypervector** to all the **class hypervectors** inside the **associative memory**. Similarity between query and class hypervectors is computed using Hamming distance.
In this project, however, the whole HDC model is serialized, hence hypervectors are divided in parts. This means that it is not possible to assign a class to an hypervector by just comparing one of its part to one part of the class hypervectors. Hence, operations of the inference module are divided in two phases: in a first phase, the distances between each part of the query hypervector and the correspondent part of the classes hypervector are accumulated; then in the last phase, all the accumulated distances are compared in order to find the minimum distance, and therefore the class to which assign the input hypervector. In the first step of the inference execution, differences between the query hypervector and the class hypervectors are evaluated. Since hypervectors are divided in parts, due to serialization, before obtaining the real difference between two hypervectors, the difference between each part processed at each iteration of the inference must be accumulated. For this reason, a second memory, called **difference memory** is introduced, alongside the associative memory, to store the differences computed at each iteration of the serialized inference. Indeed, at each iteration of the serialized inference a part of the query hypervector is sent by the encoder to the inference block. This part of the query hypervector is then compared with the respective parts of the class hypervectors, by evaluating the distance. The distance counts for the number of different bits, and it is implemented in hardware by using a counter of ones which input is the xor between the two compared vectors.
After being computed, each distance is accumulated in the proper row of the distance memory. After each parts of the input query hypervector are processed, the accumulated distances are sent to the comparator block, which select the minimum distance and the related class.
In hardware, the distance memory is address by the **tag counter**, i.e. a counter which output represents the tag of a class in the associative memory. Since the dataset in this project is composed by 10 classes, the parallelism of the tag counter is $TB = \lceil log_2 10 \rceil = 4$ bits.
The tag counter is also used to address the associative memory together with the **part counter**. Indeed, the tag counter points to the class blocks inside thee associative memory, while the part counter works like an offset that points to the part of the class hypervector inside the class block [Figure 4.11].
Once a part of the class hypervector is read, it is xored with the input part of the query hypervector. The output of the xor is sent to a counter of ones, which gives

as a result the partial distance between the two vectors. In the first iteration of the inference, i.e. for the first part, the partial distance is directly stored in the difference memory location pointed by the tag counter. For the following iterations, the partial distance is added to the difference previously stored in the same memory location.

The input of the distance memory is selected using a multiplexer, which output

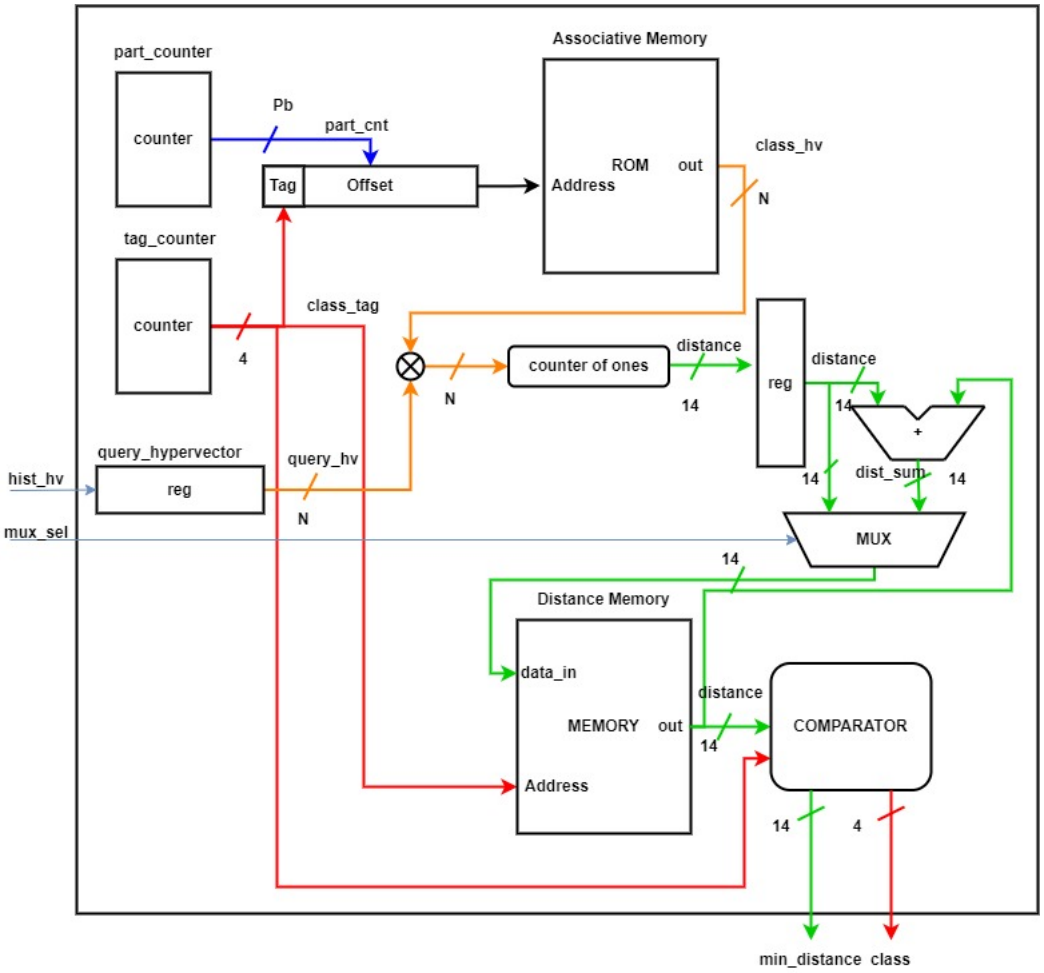


Figure 4.20: Overview on the Inference Module datapath. Signals with different parallelism are highlighted with different colours.

corresponds to the output of the counter of ones for the first iteration, and to the result of the addition between the new partial distance and the previous partial distance for the other iterations.

Since the total dimension of the hypervectors is D = 8192, two hypervector can differ for 8192 bits. Hence the total difference, and also the partial distance for computations, are represented in hardware with $DMB = \lceil log_2 8192 \rceil + 1 = 14$ bits.

After all differences between all parts of the query hypervector and all parts of all classes are computed, the comparison process starts.
The module responsible for comaprison is called **comparator**. It takes as inputs the output of the tag counter, which represents the tag associated to a class, and the distance stored in the distance memory location pointed by the tag counter output. At each iteration, the input distance is compared to the value stored in the internal register (**result register**) of the comparator; if the input distance is lower than the previously stored difference it is stored in the result register, together with the tag output which is stored in the **tag register**. Once the tag counter asserts the terminal count, the comparison process is terminated, and the values stored in the tag register and the result register are presented in output respectively as the **inferred class** and its relative **minimum distance**.

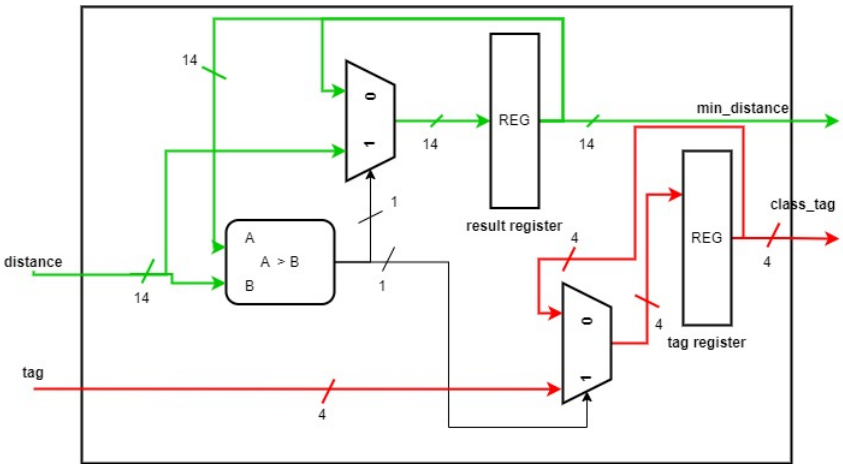


Figure 4.21: Overview on the datapath of the comparator componet of the inference module. Signals with different parallelism are highlighted with different colours.

Algorithm 9 Inference pseudo-code.

```
Input: query_hv[N-1:0]
Initial conditions: min_dist = MAX_VALUE
Parameters: N = bits of the part hypervector; C = number of classes; P =
number of parts.
for i=0,...,P-1 do
  for c=0,...,C-1 do
    bind_hv = query_hv xor class_hv[c][i]
    distance = countOnes(bind_hv)
    if i=0 then
      dist_mem[c] = distance
    else
      dist_mem[c] = dist_mem[c] + distance
    end if
  end for
end for
for c=0,...,C-1 do
  if dist_mem[c] < min_dist  then
    min_dist = dist_mem
    class_tag = c
  end if
end for
return  class_tag = inferred class; min_dist = distance between the query
hypervector and the associated class
```

Operations of the encoder module and the inference module are pipelined. Indeed, once the encoder sends the encoded part of the histogram vector to the inference module, it restarts its operations. In this way, while the inference module is computing and accumulating the distances between the received hypervector and the stored class hypervectors, the encoder processes the next part of the hypervector. Once the encoder have processed all parts of the histogram hypervector, it waits until the inference module concludes the distance comparison and gives the output results. After inference results are are given, the encoder can restart its operations.

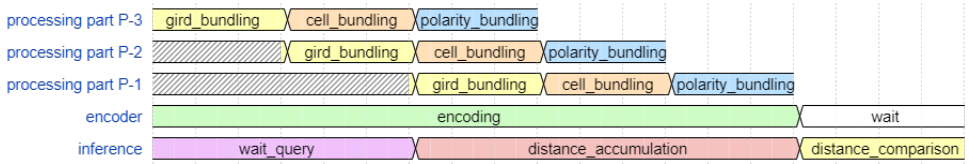


Figure 4.22: Scheduling of operations of the encoder and the inference modules. While the inference module computes and accumulates the distances related to a part hypervector, the encoder is operating on the next part hypervector.

4.4.2 Inference Results

In this project, four possible configurations of the design are tested: the first configuration divides hypervectors in 64 parts with 128 bits each(64x128); the second and the third configurations divide hypervectors respectively in 32 parts with 256 bits each (32x256) and 16 parts with 512 bits (16x512); the fourth and last configuration uses 8 parts of the hypervector with 1024 bits each (8x1024).
The firs test on the four design is done using the ModelSim Simulator. The test used for the simulation comes from the **N-MNIST** dataset [see 3.2]. All the four implemented design achieve on the 10000 samples of the test set an accuracy of **83%**, which is coherent with the result of the software implementation without retraining.

Chapter 5

The proFPGA system

The **proFPGA quad V7** system is a scalable and modular prototyping solution for multiple FPGAs [19].
The system architecture is modular, i.e. it is implemented using a set of building blocks of different typologies. A block can be a motherboard, a FPGA, a memory or a cable [20].
Remote system configuration, automatic board detection, automatic I/O voltage

Figure 5.1: Top view of the proFPGA system. Image taken from [6].

settings and safety mechanism are some examples of the features provided by the proFPGA prototyping system [6]. Communication between the implemented design inside proFPGA and the workstation is done using the **Module Message**

Interface 64 [see 6].
The first part of this chapter illustrates the used device among all the possible module that proFPGA can support.
Then, a brief explanation on how to create and how to use the configuration file is reported.
Finally, in the last part the work-flow followed to set-up and run the proposed design on the FPGA is presented.

5.1 ProFPGA Devices

5.1.1 Motherboard

The motherboard represents the infrastructure of the proFPGA system.
Features provided by the Motherboard are [7]:

Figure 5.2: ProFPGA Motherboard.

- Mechanical support: the Motherboard is used as the base structure on which FPGA modules are mounted;
- Clock generation and distribution: up to 8 clocks can be generated;
- Power Management and Protection: the system sustain up to 1.3 kW;
- JTAG chain: for Xilinx and Altera Modules debugging;
- MMI-64 communication access: for data transmission between workstation and proFPGA;

- others.

Modules mounted on the motherboard are organized following a coordinate system [Figure 5.3]. Coordinates are indicated with two letters and a number. The first letter of the coordinate code is a **T**, if the module is in the top side, or a **B** if the module is in the bottom side.
The second letter goes from **A** to **D** and indicates the column of the module.
The number, from **1** to **4**, represents the rows of the coordinate system.

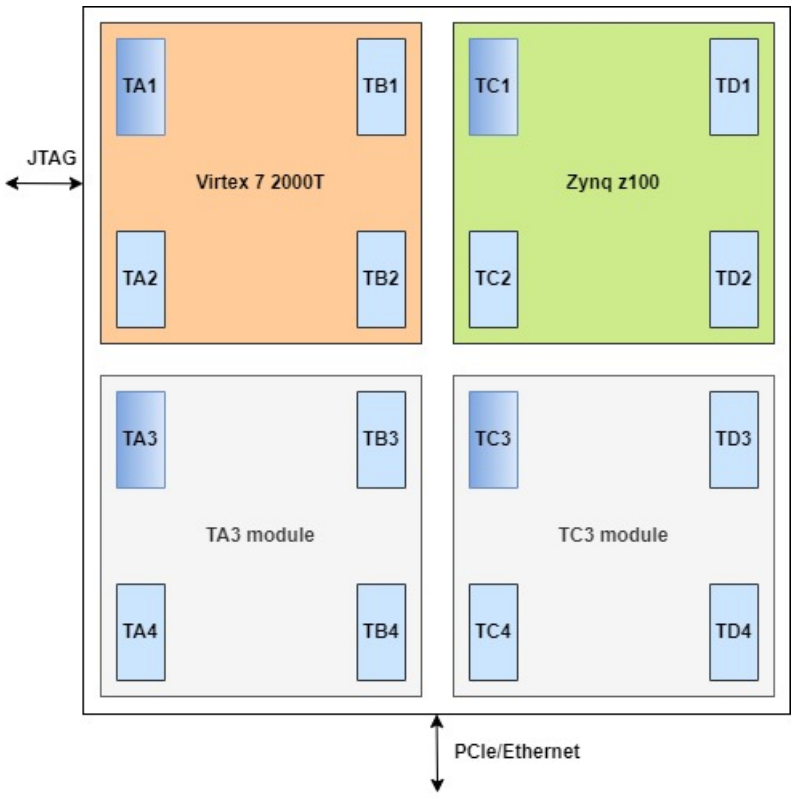


Figure 5.3: Motherboard coordinate system.

The proFPGA system used in this project has two available FPGA modules: the **Virtex 7 2000 T** FPGA and the **Zynq z100** FPGA. For tests, only the Virtex-7 module will be used.
The Virtex-7 requires four connectors, in this case: **TA1**, **TA2**, **TB1**, **TB2**. According to proFPGA notation, a module is named after the top left connector. Hence, the Virtex 7 will be referred as **fpga_module_ta1**. For further information about the motherboard see the relative chapter in [19].

5.1.2 Xilinx Virtex 7 2000T

The design proposed in this project is tested on the Xilinx Virtex 7 2000T FPGA. The full part name to be inserted in a Vivado project is "xc7v2000tflg1925-2".
In Virtex series 7, Configurable Logic Blocks are made by two SLICEs, which contains four 6-inputs LUT and eight flip-flop each [Figure 5.4].
Available resources on Xilinx Virtex 7 2000T are reported in Table 7.1.

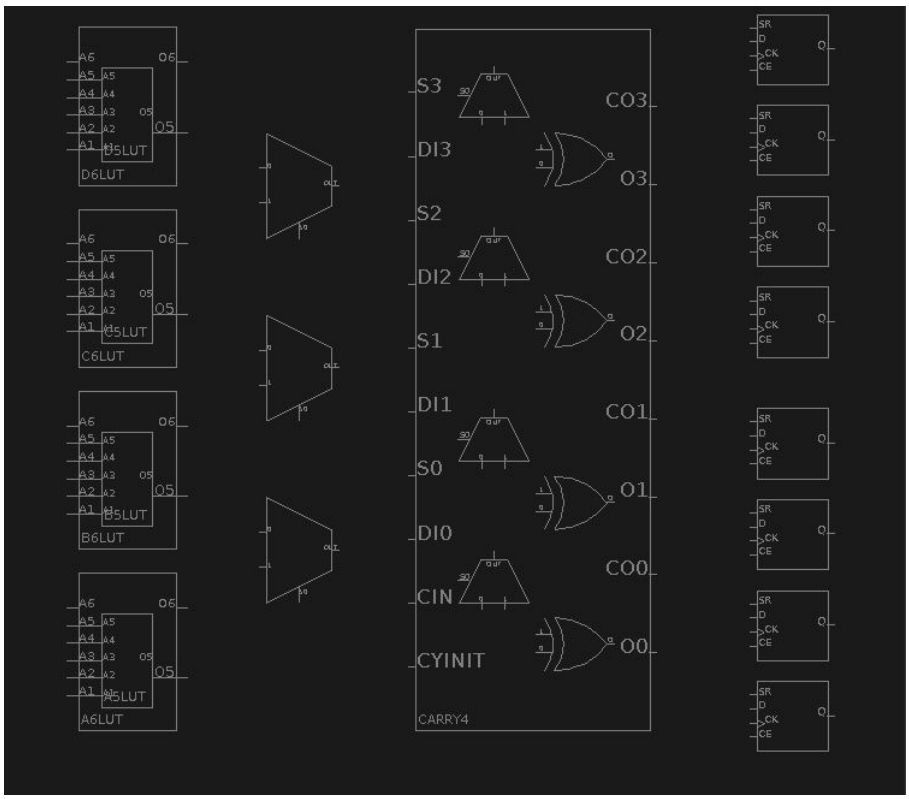


Figure 5.4: Configurable logic block inside the Virtex 7 FPGA. Image taken from [6].

5.2 ProFPGA Builder and Configuration File

The **proFPGA Builder** is one of the software tools provived by proFPGA alongside the hardware.
The builder is only used to create the configuration file which is needed to set up the FPGA and load the bitstream. To launch the builder, open a terminal and type the command:

```
profpga_builder
```

A complete description on the required steps to create a project and a configuration file are reported in [21]. In general, the steps to follow with the proFPGA builder are:

- Specify the location of the project;
- Specify the ip address of the system to connect, in this case http://172.16.0.230;
- Specify the location of the board description files;
- Specify the FPGA image file for each used FPGA.

Once all steps are executed correctly, the builder produces the **configuration file**. The configuration file used in this project is reported below:

```
name = "profpga";
profpga_debug = 0;
debug = 0;
backend = "tcp";
backends :
{
  tcp :
  {
    ipaddr = "172.16.0.230";
    port = 0xD11D;
  };
  pcie :
  {
    device = "/dev/mmi64pcie0";
  };
};
system_configuration :
{
  sysconfig_match = "FIT";
  fpga_speedgrade_match = "FIT";
  motherboard_1 :
  {
    type = "MB-4M-R2";
    fpga_module_ta1 :
    {
      type = "FM-XC7V2000T-R2";
      speed_grade = 2;
      bitstream = "user_mmi64.bit";
      v_io_ta1 = "AUTO";
```

```
  v_io_ta2 = "AUTO";
  v_io_tb1 = "AUTO";
  v_io_tb2 = "AUTO";
  v_io_ba1 = "AUTO";
  v_io_ba2 = "AUTO";
  v_io_bb1 = "AUTO";
  v_io_bb2 = "AUTO";
};
fpga_module_tc1 :
{
  type = "FM-XC7Z100-R1";
  speed_grade = 1;
  v_io_ta1 = "AUTO";
  v_io_ta2 = "AUTO";
  v_io_ba1 = "AUTO";
  v_io_ba2 = "AUTO";
  boot_mode = "JTAG";
  usb_mode = "DEVICE";
  usb_id = "UNUSED";
  ps_npor = "SWITCH";
  ps_nsrst = "SWITCH";
  geth_config2 = "GND";
  geth_config3 = "LED1";
};
clock_configuration :
{
  clk_0 :
  {
    source = "LOCAL";
  };
  clk_1 :
  {
    source = "125MHz";
    multiply = 5;
    divide = 125;
  };
  clk_2 :
  {
    source = "125MHz";
    multiply = 5;
    divide = 125;
  };
```

```
      clk_3 :
      {
        source = "125MHz";
        multiply = 5;
        divide = 125;
      };
      clk_4 :
      {
        source = "125MHz";
        multiply = 5;
        divide = 125;
      };
    };
    sync_configuration :
    {
      sync_0 :
      {
        source = "GENERATOR";
      };
      sync_1 :
      {
        source = "GENERATOR";
      };
      sync_2 :
      {
        source = "GENERATOR";
      };
      sync_3 :
      {
        source = "GENERATOR";
      };
      sync_4 :
      {
        source = "GENERATOR";
      };
    };
  };
  x_board_list = ( );
};
```

In the reported configuration file, it is possible to specify which type of connection is used. In particular, line:

```
backend = "tcp";
```

indicates that the Ethernet connection is selected. It is possible to switch to the PCIe connection by writing:

```
backend = "pcie";
```

As can be seen in the file, five clocks are generated. The first clock, i.e **clk_0**, is the clock assigned to the HDL module of the proFPGA system. The frequency of clk_0 is fixed at 100 MHz. For what concern the remaining clocks, from **clk_1** to **clk_4**, their frequency is user defined. Indeed, in the configuration file three parameters for each clock are reported:

- source : the staring frequency;
- multiply : multiplication factor;
- divide : value which divides the product between the source and the multiplication factor.

Taken clk_1 as example, its frequency can be computed starting from the values in the configuration file as follows:

$$f_{clk_1} = \frac{source \cdot multiply}{divide} = \frac{125MHz \cdot 5}{125} = 5MHz \tag{5.1}$$

In this work, only clk_0 will be used as a clock, hence there is no interest in the frequency of the other clocks.
In order to load the bitstream on the FPGA, the configuration file is launched with the command:

```
profpga_run <configuration_file_name> --up
```

This command will turn ON the FPGA. To power down the FPGA run the command:

```
profpga_run <configuration_file_name> --down
```

.

5.3 Work-Flow

5.3.1 Directory Organization

Files of the project are organized as in Figure 5.5. The directory organization is inspired by the **demo_designs** [22] reported as examples from ProDesign [6]. Defined as $WORK the path of the project directory, what follows is a description of the content of the directory.

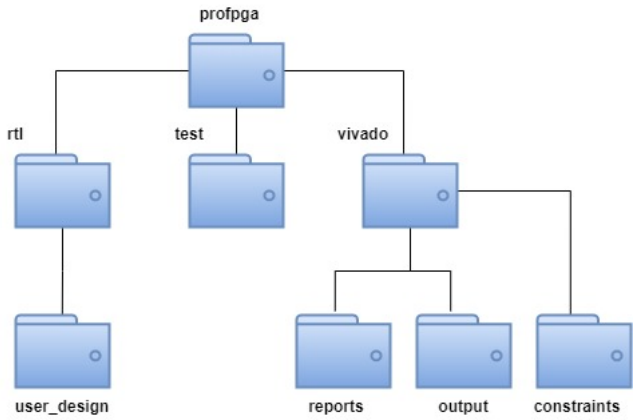


Figure 5.5: Work Directory Organization.

- **$WORK/rtl/user_design/***: Includes all the VHDL (or Verilog) files of the user design. The presence of testbenches in this directory should be avoided, since testbenches are not synthesized.

- **$WORK/rtl/user_mmi64.vhd** : HDL of the module from which the bitstream is generated. It contains the components of the MMI64 interface and the user design **top entity**.User can modify this file only in the blocks delimited with the **-START USER SPACE-** and **-END USER SPACE-** keywords.

- **$WORK/test/main.c**: C program used to the define the operations of the MMI64 interface.

- **$WORK/test/compile_me.sh**: script that compiles the **main.c** program and creates the **usertest** executable. The script must be launched after changing the main.c file and before turning up the FPGA.

- **$WORK/test/profpga.cfg**: Configuration file which sets up the FPGA and loads the bitstream. It is described in 5.2.

- **$WORK/test/user_mmi64.bit**: Bitstream generated from the **user_mmi64.vhd** file.

- **$WORK/test/usertest**: Executable of the **main.c** file, created with the **compile_me.sh** script.

- **$WORK/vivado/vivado.tcl**: Script used to generate the synthesized netlist, i.e the **user_mmi64_synthesized.dcp** inside the **$WORK/vivado/output** folder

- **$WORK/vivado/output/***: Folder that contains the outputs generated by running the **vivado.tcl** script.
- **$WORK/vivado/report/***: Folder that contains the reports generated by running the **vivado.tcl** script.
- **$WORK/vivado/synthesize_me.sh**: script which launches the **vivado.tcl** script to perform synthesis.
- **$WORK/vivado/user_design.tcl**: contains the name of the user defined components. It is used by **vivado.tcl** script to perform synthesis. User must specify the name of each component of the implemented design inside this file.
- **$WORK/vivado/constraints/user_mmi64.xdc** File with clock constaints and PIN mappings. It must not be modified.

5.3.2 Steps for Project Implementation

In this section, all the steps required to synthesize, load and run the project on the proFPGA system will be reported.

1. Copy the VHDL (or Verilog) files of all components of the design in the **$WORK/rtl/user_design** folder.
2. Modify the **$WORK/rtl/user_mmi64.vhd** file by inserting the top entity of the design and all the needed signals.
3. Move in the **$WORK/vivado** directory and write the names of all HDL components in the **user_design.tcl** script.
4. Launch the **synthesize_me.sh** script by typing:

   ```
   ./synthesize_me.sh vhdl
   ```

   If vhdl is the target language. If the target language is Verilog, run the same command by changing **vhdl** with **verilog**. If the synthesis is correctly executed, the **user_mmi64_synthesized.dcp** file will be generated in the **reports** folder. Otherwise, errors will be reported.
5. copy **user_mmi64_synthesized.dcp** in the **$WORK/test** folder.
6. Move in **$WORK/test** folder.
7. Edit the **main.c** file and compile it using the **compile_me.sh** script, which is launched by typing:

```
./compile_me.sh
```

If compilation succeeded, the **usertest** executable is created.

8. Turn ON the FPGA and load the bitstream by typing:

```
profpga_run profpga.cfg --up
```

This command can be also used to reboot the FPGA.

9. Start the emulation by running:

```
./usertest profpga.cfg
```

10. Once the emulation is done, switch off the motherboard by typing:

```
profpga_run profpga.cfg --down
```

Chapter 6

Interface with proFPGA

6.1 Module Message Interface 64

6.1.1 Introduction on MMI64

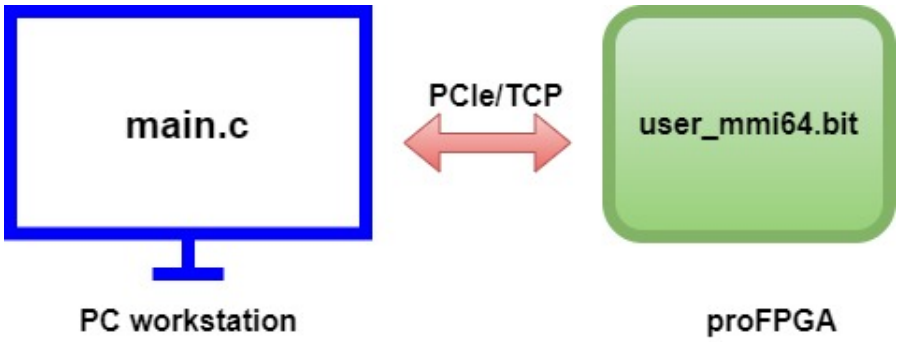


Figure 6.1: Overview of the MMI64 communication system

The **Module Message Interface 64** (**MMI64**) is a module provided by proFPGA system, which allows the communication between the user workstation and the design inside the FPGA.
Communication can happen through two types of connectors: the **PCIe** and the **Ethernet**.
The PCIe is the fastest connector, since it reaches a data exchange rate of 3.2 Gbps, however it is less reliable.
The Ethernet connectors is slower, with a maximum data exchange rate of 100 Mbps, but it is more reliable than the PCIe.
For this reason, in this project the Ethernet will be used as the communication standard.
User can select which connector too use by editing the configuration file [see 5.2].

The MMI64 module is implemented in hardware as a register file. The user can define the number of registers and their parallelism by editing the **user_mmi64.vhd** file [see 5.3.1].
Data are read or written in the register file using the proper functions in the C program.

6.1.2 C Program

The **main.c** file [see 5.3.1], is the file used to exchange data between the workstation and the FPGA.
In order to read or write data, the proper MMI64 functions should be used. Since datawidth can be chosen between 8, 16, 32 and 64 bits, there are four functions for data reading and four functions for data writing, one for each possible datawidth. Those functions are:

```
mmi64_regif_write_8_ack();
mmi64_regif_write_16_ack();
mmi64_regif_write_32_ack();
mmi64_regif_write_64_ack();
```

to write data in registers, and

```
mmi64_regif_read_8();
mmi64_regif_read_16();
mmi64_regif_read_32();
mmi64_regif_read_64();
```

to read data from registers. Each function requires four parameters: the name of the module, the initial address of the register file, the number of written (read) registers and the pointer to the variable to be written (read).
An example on how these function are used is reported below:

```
// Array of 8 uint32_t data , corresponding to the
   instantiated
// register file
uint32_t wdata[8];
// Reading variable
uint32_t rdata32;
// You can write one to all the words of the register
   file
wdata[0] = 10;
wdata[1] = 5;
wdata[2] = 9;
// Write 3 words (10 ,5 and 9), from address 0 till third
```

```
// register
status = mmi64_regif_write_32_ack(user_module,0,3,wdata32
   );
CHECK(status);
wdata[0] = 666;
// Write the value 666 in the address 5 (the sixth
   register)
status = mmi64_regif_write_32_ack(user_module,5,1,wdata32
   );
CHECK(status);
// Read 1 word at address 3
status = mmi64_regif_read_32(user_module,3,1,&rdata32);
CHECK(status);
```

See [23] for more information about the MMI64 C program.

6.1.3 HDL Module

As can be seen in FIgure 6.2, the MMI64 module is made of 5 main blocks.
The **profpga_clocksync** module works as a PLL. It takes a clock input and mul-

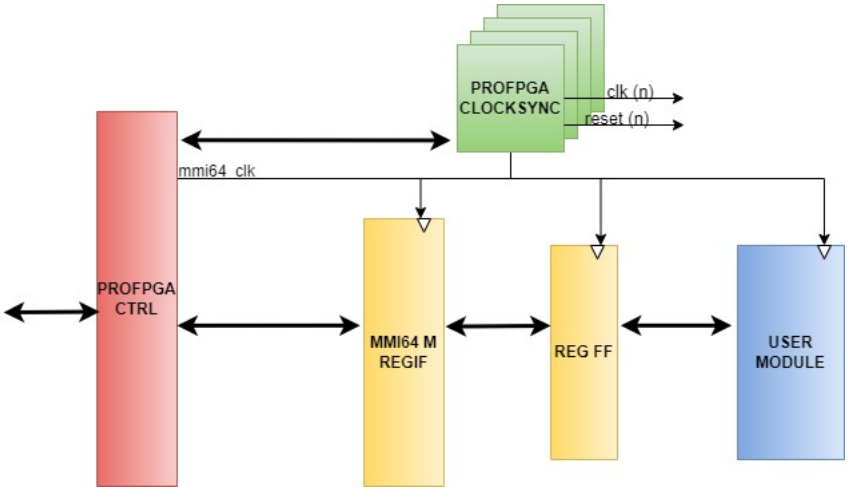


Figure 6.2: Overview on MMI64 datapath. Image adapted from [7].

tiplies and divides its frequency according to the values written in the configuration file [see 5.2]. Since the system can provide up to 4 clocks, four profpga_clocksync modules are present.
Profpga_ctrl allows the communication between the C program and the user module.
The **mmi64_m_regif** module represents the interface with the MMI64 register file, i.e **reg_FF**. Through **mmi64_m_regif** it is possible to deine the number of

register and the data width of the register file.
A portion of the **user_mmi64.vhd** file used in this project is reported below.

```
----------------------USER SPACE----------------------
reg_accept <= '1';
REG_FF : process(mmi64_clk)
begin
if rising_edge(mmi64_clk) then
-- handle register transfers
if reg_en='1' and reg_accept='1' then
    if reg_we='1' then -- write to registers
      reg_rvalid <= '0';
      reg_rdata <= (others=>'0');
      case reg_addr is
        when "0001" => start <= std_logic(reg_wdata(0));
             input_write_tx <= std_logic(reg_wdata(1));
             ack_rec_tx <= std_logic(reg_wdata(2));
             data_in <= unsigned(reg_wdata(Q+2 downto3));
             output_ack_tx <=std_logic(reg_wdata(Q+3));
        when others => start <= '0';
      end case;
    else -- read from registers
     reg_rvalid <= '1';
      case reg_addr is
       when "0000" =>
            reg_rdata(0) <= std_ulogic(wait_start_rx);
            reg_rdata(1) <= std_ulogic(wait_input_rx);
            reg_rdata(2) <= std_ulogic(input_ack_rx);
       when others => reg_rdata(CTB-1 downto 0) <= (
   others => '0');
      end case;
    end if;
else -- no transfer or not accepted
        reg_rvalid <= '0';
        reg_rdata <= (others=>'0');
end if;
-- reset values
    if mmi64_reset='1' then
     reg_rvalid <= '0';
     reg_rdata  <= (others=>'0');
     data_in  <= (others=>'0');
     start  <= '0';
    end if;
```

```
end if;
end process REG_FF;
---------------------END USER SPACE--------------------
```

6.2 The Handshake Protocol

An important role in data transmission is played by **transmission protocols**. They are needed to ensure that the wanted data arrive at the wanted time.
Protocols become crucial when the communicating entities do not share the same clock, hence they work at different frequencies.
This last case happens when connecting the workstation to the design inside the FPGA using Ethernet connector. The implemented design uses a clock with a frequency $f = 100MHz$, while the Ethernet connectors works with a variable frequency which depends on the environment.
For this reason, the implementation of an handshake protocol is needed.
In particular, the implemented protocol is a 4-bit handshake, used both for inputs and outputs, Two additive bits are used to make the algorithm start.
At the input, the handshake protocol is used to write data from the PC to input **histogram memory** [see 4.2.1] of the implemented design. The handshake is repeated for each input value [Figure 6.3].

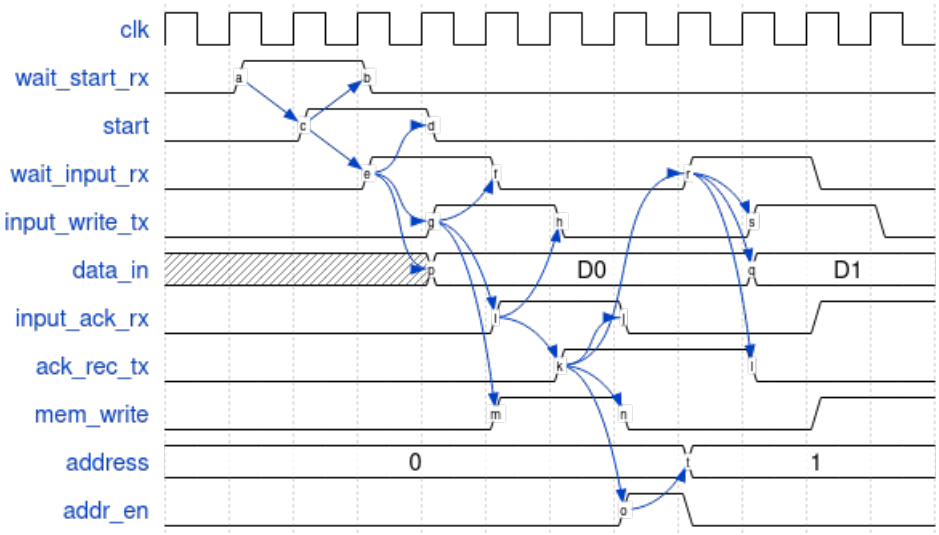


Figure 6.3: Input handshake timing diagram.

At the output, the handshake protocol is used to write the inferred class and its relative distance on the PC. In this case, the handshake is repeated a single time,

since the class and the distance are sent together [Figure 6.4].

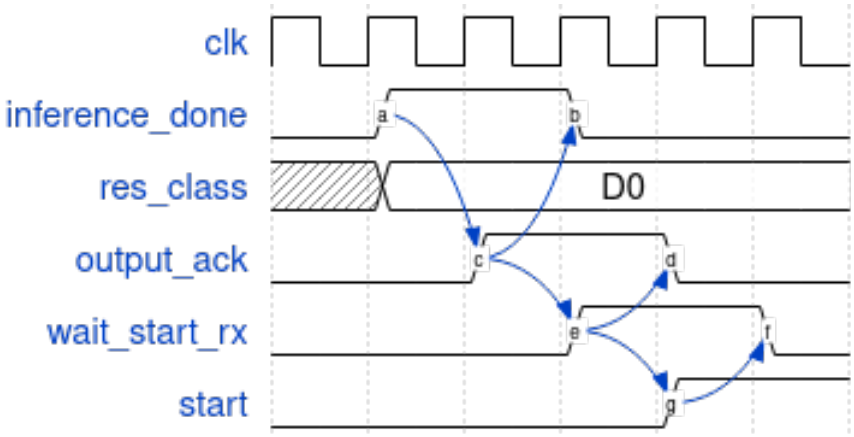


Figure 6.4: Output handshake timing diagram.

In the ideal case, i.e. if the handshake is performed with the minimum number of clock cycles, 6 clock cycles are needed to write a single value. Hence, the handshake introduces an overhead of 5 clock cycles with respect to the implementation without protocol, in which a data is written in a single cycle. However, the last case is an optimistic implementation. Indeed, when connecting two different blocks is always good practice to introduce a protocol for data exchange.
Figure 6.5 reports the flow of the handshake. In the handshake, a total of 8 signals are involved:

- **wait_start_rx**: transmitted from the FPGA. Signal which indicates that the design is ready to start the algorithm. It is asserted at the beginning of the algorithm and it is deasserted when the *start* signal is sent from the workstation.
- **start**: sent from the workstation. Signal which starts the algorithm. It is asserted when the *wait_start_rx* signal is received, and deasserted when the *wait_input_rx* signal is received.
- **wait_input_rx** transmitted from the FPGA. Signal which indicates that the design is waiting for an input to be written in the memory. It is asserted when the *start* signal is received, and deasserted when the *input_write_tx* signal is received.
- **input_write_tx** transmitted from the workstation. Signal which indicates that a valid data is sent to the memory of the design. It is asserted when the *wait_input_rx* is received, and deasserted when the *input_ack_rx* signal is received.

- **input_ack_rx** transmitted from the FPGA. Signal which indicates that the data transmitted by the workstation was received. It is asserted when the *input_write_tx* signal is received, and deasserted when the *ack_rec_tx* signal is received.

- **ack_rec_tx** transmitted from the workstation. Signal which inidicates that the workstation received the acknowledges signal. It is asserted when the *input_ack_rx* is received, and deasserted when the *inference_done* signal is received.

- **inference_done** transmitted from the FPGA. It is asserted when the output of the design is valid. It is deasserted when the *output_ack_tx* signal is received.

- **ouptut_ack_tx** transmitted from the workstation. Signal which indicates that the workstation received the output of the FPGA. It is asserted when the *inference_done* signal is received, and deasserted when the *wait_start_rx* signal is received

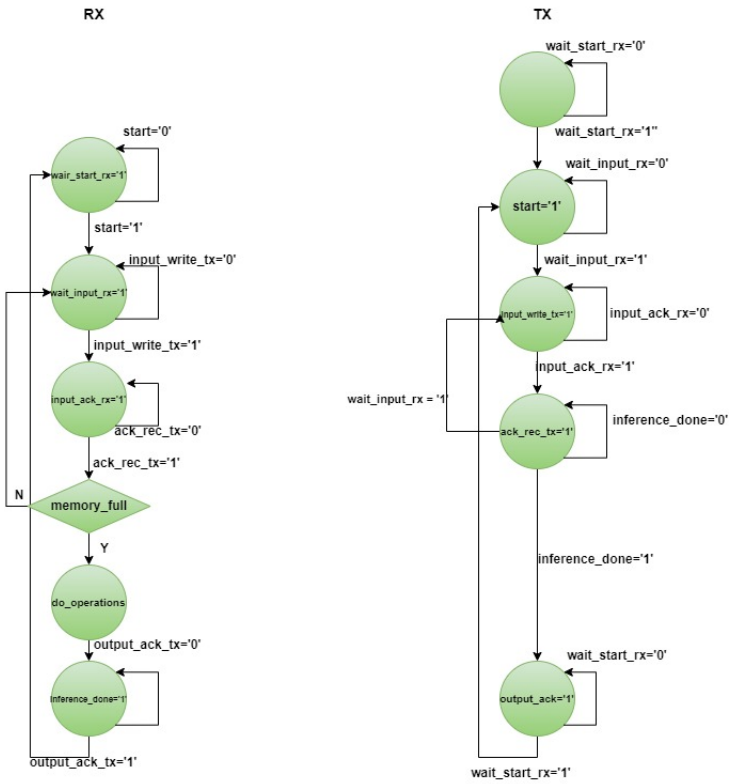


Figure 6.5: Flow of the handshake.

6.3 Results from FPGA Implementation

All the four proposed configuration for the HDC classifier were tested on the Virtex 7 2000T FPGA mounted on the proFPGA system. Thanks to the handshake protocol, data were correctly transmitted to the FPGA. Inference results are coherent with the simulation on ModelSim and with the software emulation. In particular, the sequence of predicted classes and their relative minimum distances are the same for all the three types of tests. The design held an accuracy of **acc = 83%** for all the configurations on the N-MNIST dataset.
An issue related to the implementation on FPGA is the time required to perform the test. Data transmission using the MMI64 interface adds a large time overhead. Indeed, for the 64x128 implementation, around **7.6s** for sample are needed to transmit all data and receiving results. For this reason, it was necessary to understand how long it takes for each design to give an output without considering the time overhead introduced by communication.
This measurement is done by implementing a counter on each design which counts how many cycles are required from the moment when all the input data were transmitted and the moment when the output is transmitted from the FPGA. Hence, cycles required for the input and the output handshake are not counted.
The number of required cycles, and the relative elapsed time computed with a clock period of 10 ns, are reported in Table 6.1.

From what can be seen in Table 6.1, the effective required time for the 64x128

Configuration	Required Cycles	Elapsed Time (ms)
64x128	358474	3.6
32x256	179274	1.8
16x512	89674	0.9
8x1024	44874	0.45

Table 6.1: Required cycles and elapsed time for a single sample.

configuration is 2000x lower with respect to the total required time considering data transmission by Ethernet connector. More on timing results is reported in 7.3.

Chapter 7

Results

7.1 Setup

In this chapter, measurements in term of resources utilization, timing and power will be reported.
Measurements are obtained for all the four proposed configurations through the **Xilinx Vivado** software. The target FPGA used for the vivado project is the **Virtex 7 2000T FPGA**. The complete FPGA name used in Vivado software is **xc7v2000tflg1925-2**.
To obtain more accurate results during implementation using Vivado software, a clock constraint should be assigned to the project. In order to have all the four design using the same clock frequency, a period of $\mathbf{T = 10\ ns}$ was chosen for the constrained clock, for a frequency of 100 MHz.
In order to test the possibility of implementing the proposed design also on smaller FPGAs, all the four configurations were tested on FPGA from four different families. In particular, two FPGAs per family were tesetd: one with the minimum number of resources and one with the maximum number of resources. The tested FPGA families are, in order of capacity: Spartan 7, Artix 7, Kintex 7 and Virtex 7.
A full parallel version of the hardware accelerator is implemented as a reference for utilization and timing measurements.
Power results are extracted using back annotation. The design are simulated with Vivado Simulator, and from the simulation a SAIF file is generated. The **Switching Activity Interchange Format** (**SAIF**) is a file which annotates the switching activities of the implemented netlist. Using the SAIF file, it is possible to obtain a more accurate power estimation.
The script to run the simulation, write the SAIF file and extract power results was provided by ing. Fabrizio Ottati.

7.2 Utilization

7.2.1 Virtex 7 2000T

The utilization results taken as reference come from the implementation of the four configurations of the proposed design on the Virtex 7 2000T FPGA.
The total amount of resources in the target FPGA are reported in Table 7.1. Table 7.2 reports the number of used resources for all four configurations.

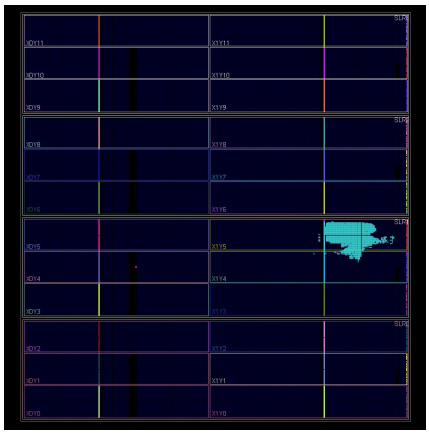
LUT	REGs	BRAM	DSP	I/O
1221600	2443200	1292	2160	1200

Table 7.1: Virtex 7 2000T resources

Configuration	LUT		REGs		BRAM		DSP		I/O	
	n°	%	n°	%	n°	%	n°	%	n°	%
64x128	8061	0.66	11590	0.47	14.5	1.12	0	0	47	3.9
32x256	13793	1.13	15088	0.62	20.5	1.59	0	0	47	3.9
16x512	22529	1.84	24421	1	0.5	0.04	0	0	47	3.9
8x1024	41417	3.39	40589	1.66	0.5	0.04	0	0	47	3.9

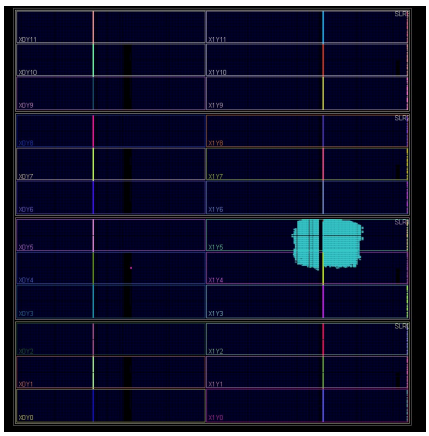
Table 7.2: Number of used resources for the implementation of the four configurations on the Virtex 7 2000T FPGA.

As can be seen, the number of used resources is very limited, implying that the proposed HDC accelerator does not require a large FPGA to be implemented [Figure 7.1]. The number of I/O pin used is the same for each configuration, since the parallelism of input and output signals does not change. In addition, the percentage of used DSPs is 0, since the accelerator does not perform complex arithmetic operations.

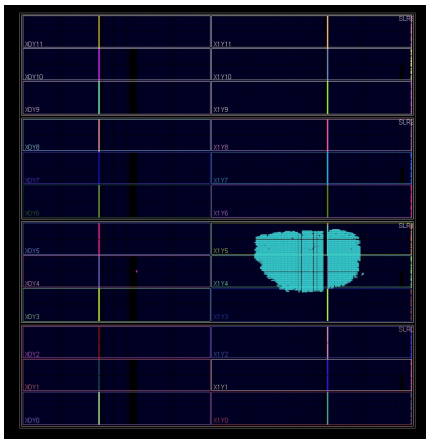


(a) 64x128 configuration

(b) 32x256 configuration



(c) 16x512 configuration



(d) 8x1024 configuration

Figure 7.1: Area of the HDC module wwith respect to the total area of the Virtex 7 2000T FPGA.

For what concern BRAM utilization, in the 16x512 and the 8x1024 configuration the percentage is close to 0 because of timing constraints. Indeed, in order to satisfy timing requirements, the Vivado software maps the memory used in the 16x512 and 8x1024 implementation not with BRAM but with sparse registers, hence the growth in the number of used registers.

7.2.2 Spartan 7 25C

The next FPGA where the design is implemented is the Spartan 7 xc7a25C. It is chosen because it is one of the FPGA with the lowest number of resources [Table

7.3].

LUT	REGs	BRAM
14600	29200	45

Table 7.3: Spartan 7 25C resources

Given the limited amount of resources, only the implementation of the 64x128 and 32x256 configurations are correctly executed, meaning that the other configurations cannot be implemented on this FPGA.

Configuration	LUT		REGs		BRAM	
	n°	%	n°	%	n°	%
64x128	8060	55.21	11590	39.69	14.5	32.22
32x256	13632	93.37	15123	51.79	20.5	45.56

Table 7.4: Number of used resources for the implementation of the 64x128 and 32x256 configurations on the Spartan 7 25C FPGA.

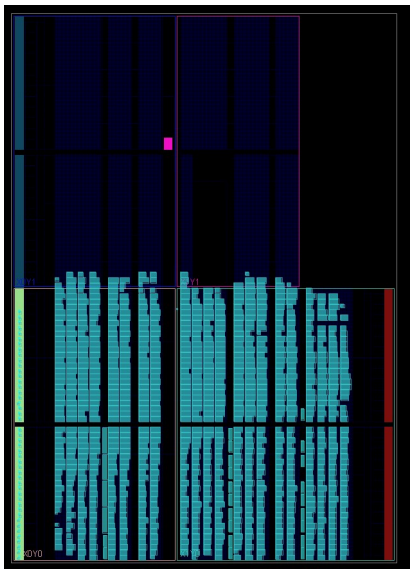


Figure 7.2: Area of the 64x128 configuration module with respect to the total area of the Spartan 7 25C FPGA.

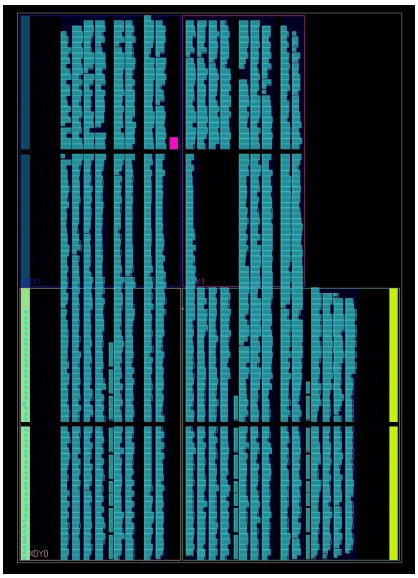


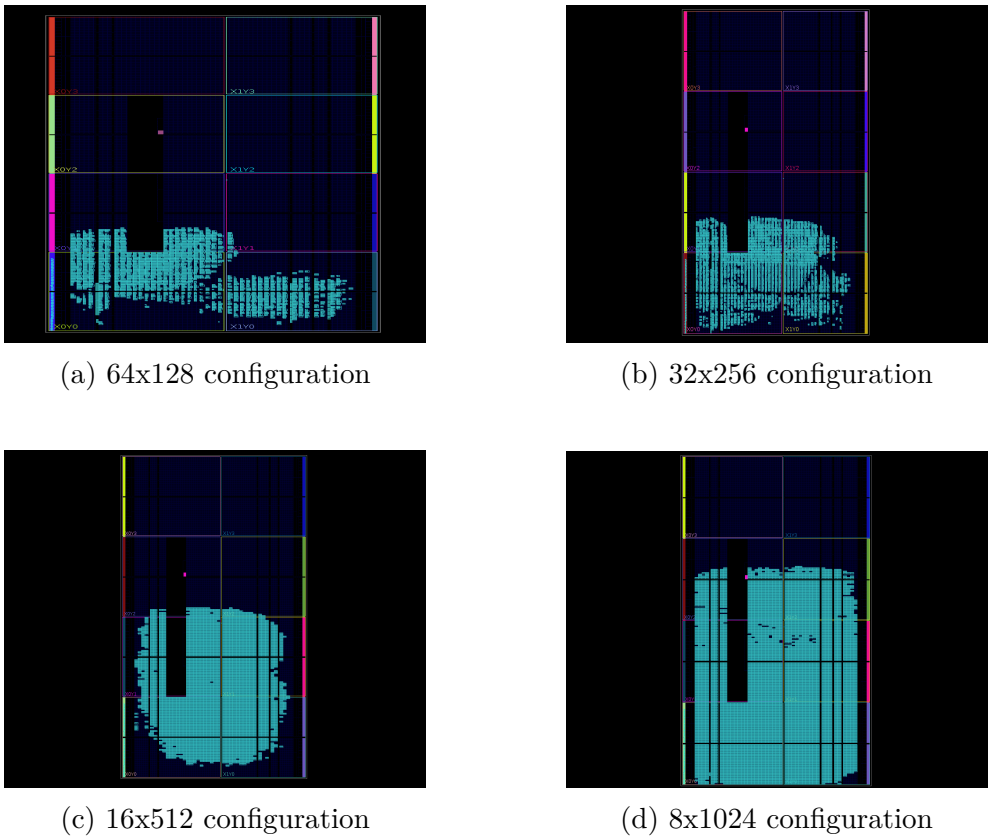
Figure 7.3: Area of the 32x256 configuration module with respect to the total area of the Spartan 7 25C FPGA.

7.2.3 Spartan 7 100F

The second tested FPGA of the Spartan 7 family is the model xc7s100f. In this case, the implementation of all four configuration is possible. However, the 16x512 and the 8x1024 implementations failed to meet the timing requirements, i.e. they cannot be correctly executed on this FPGA with the selected clock frequency of 100 MHz.

LUT	REGs	BRAM
64000	128000	120

Table 7.5: Spartan 7 100F resources



(a) 64x128 configuration

(b) 32x256 configuration

(c) 16x512 configuration

(d) 8x1024 configuration

Figure 7.4: Area of the HDC module with respect to the total area of the Spartan 7 100F FPGA.

Configuration	LUT		REGs		BRAM	
	n°	%	n°	%	n°	%
64x128	8060	12.59	11590	9.05	14.5	12.08
32x256	13606	21.26	15090	11.79	20.5	17.08
16x512	22534	35	24421	19.00	0.5	0.42
8x1024	41417	64.7	40589	31.71	0.5	0.42

Table 7.6: Number of used resources for the Spartan 7 100F FPGA.

7.2.4 Artix 7 15T

The first tested FPGA of the Artix 7 family is the Artix 7 xc7a15t model. In this case, only the 64x128 configuration is correctly implemented.

LUT	REGs	BRAM
10400	20800	25

Table 7.7: Artix 7 15T resources

Configuration	LUT		REGs		BRAM	
	n°	%	n°	%	n°	%
64x128	8060	77.50	11590	55.72	14.5	58

Table 7.8: Number of used resources for the implementation of the 64x128 configuration on the Artix 7 15T FPGA.

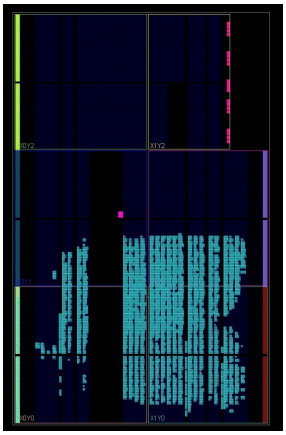


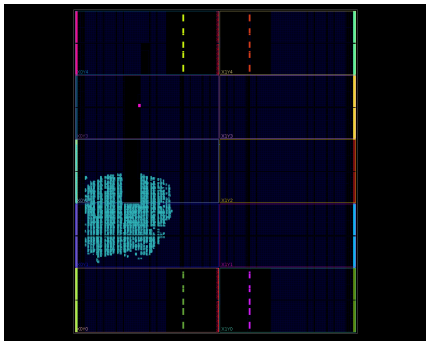
Figure 7.5: Area of the HDC module with respect to the total area of the Artix 7 15T FPGA.

7.2.5 Artix 7 200T

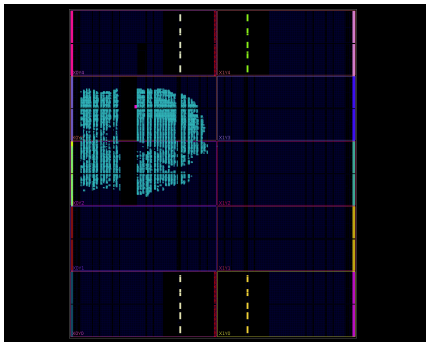
The second FPGA from the Artix 7 families is the model xc7a200t. In this case all the configurations are correctly implemented.

LUT	REGs	BRAM
133800	267600	365

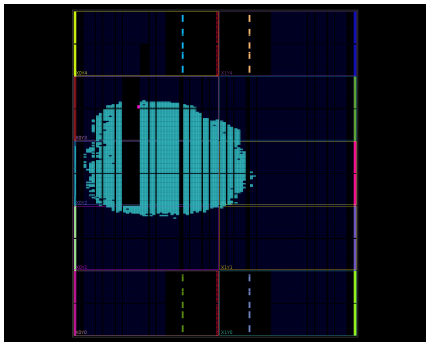
Table 7.9: Artix 7 200T resources



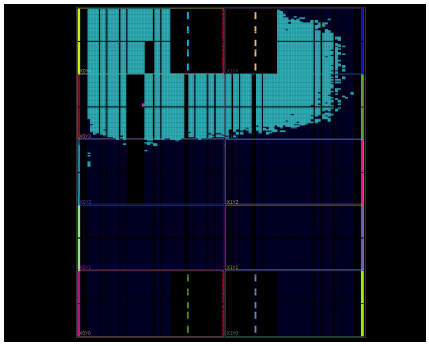
(a) 64x128 configuration



(b) 32x256 configuration



(c) 16x512 configuration



(d) 8x1024 configuration

Figure 7.6: Area of the HDC module with respect to the total area of the Artix 7 200T FPGA.

Configuration	LUT		REGs		BRAM	
	n°	%	n°	%	n°	%
64x128	8060	6.02	11590	4.33	14.5	3.97
32x256	13794	10.31	15088	5.64	20.5	5.62
16x512	22534	16.84	24421	9.13	0.5	0.14
8x1024	41417	30.95	40589	15.17	0.5	0.14

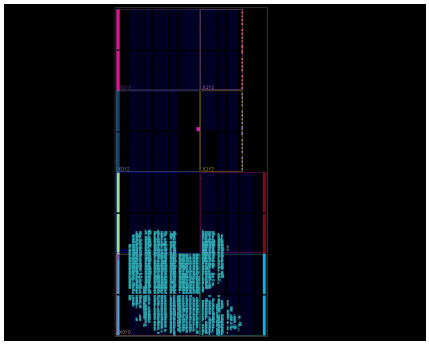
Table 7.10: Number of used resources for the Artix 7 200T FPGA.

7.2.6 Kintex 7 70T

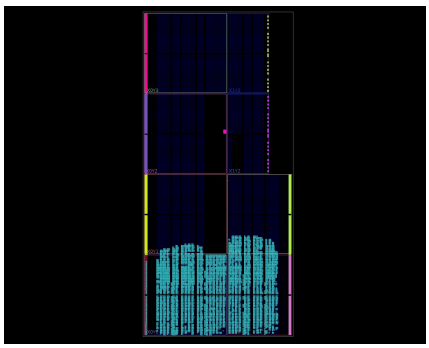
The first tested FPGA of the Kintex 7 family is the Kintex 7 xc7k70t board, which is the FPGA from the Kintex 7 family with the lowest number of resources. All four configurations are correctly implemented, except for the 8x1024 configuration.

LUT	REGs	BRAM
41000	82000	135

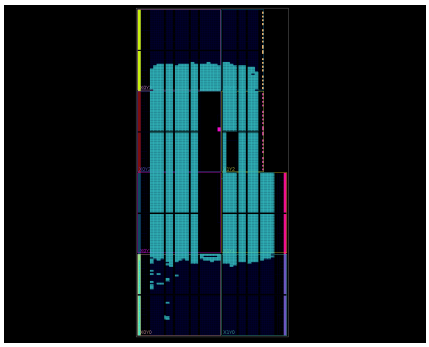
Table 7.11: Kintex 7 70T resources



(a) 64x128 configuration



(b) 32x256 configuration



(c) 16x512 configuration

Figure 7.7: Area of the HDC module with respect to the total area of the Kintex 7 70T FPGA.

Configuration	LUT		REGs		BRAM	
	n°	%	n°	%	n°	%
64x128	8060	19.97	11590	14.10	14.5	10.74
32x256	12052	29.4	14842	18.1	24.5	18.15
16x512	22534	54.96	24421	29.78	0.5	0.37

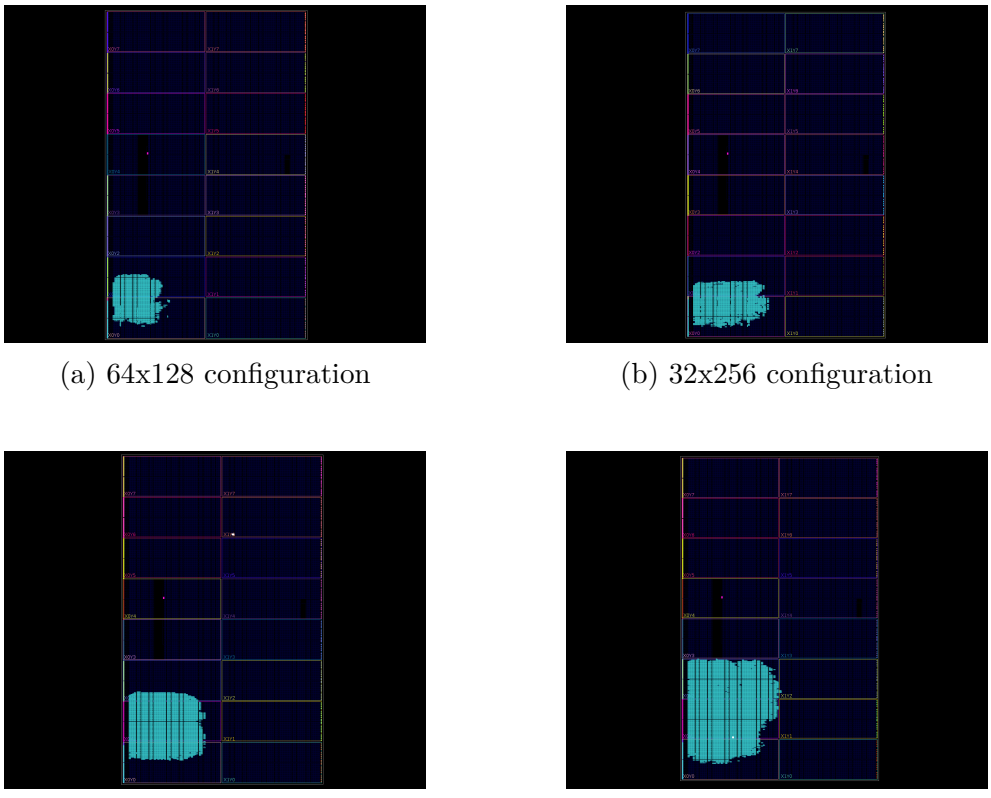
Table 7.12: Number of used resources for the Kintex 7 70T FPGA.

7.2.7 Kintex 7 480T

The second tested FPGA of the Kintex 7 family is the xc7k480t model. it is the Kintex model with the highest number of resources. All four configurations are correctly implemented.

LUT	REGs	BRAM
298600	597200	955

Table 7.13: Kintex 7 480T resources



(a) 64x128 configuration

(b) 32x256 configuration

(c) 16x512 configuration

(d) 8x1024 configuration

Figure 7.8: Area of the HDC module with respect to the total area of the Kintex 7 480T FPGA.

Configuration	LUT		REGs		BRAM	
	n°	%	n°	%	n°	%
64x128	8062	2.70	11590	1.94	14.5	1.52
32x256	13606	4.5	15090	2.5	24.5	2.57
16x512	22532	7.55	24421	4.09	0.5	0.05
8x1024	41419	13.87	40589	6.8	0.5	0.05

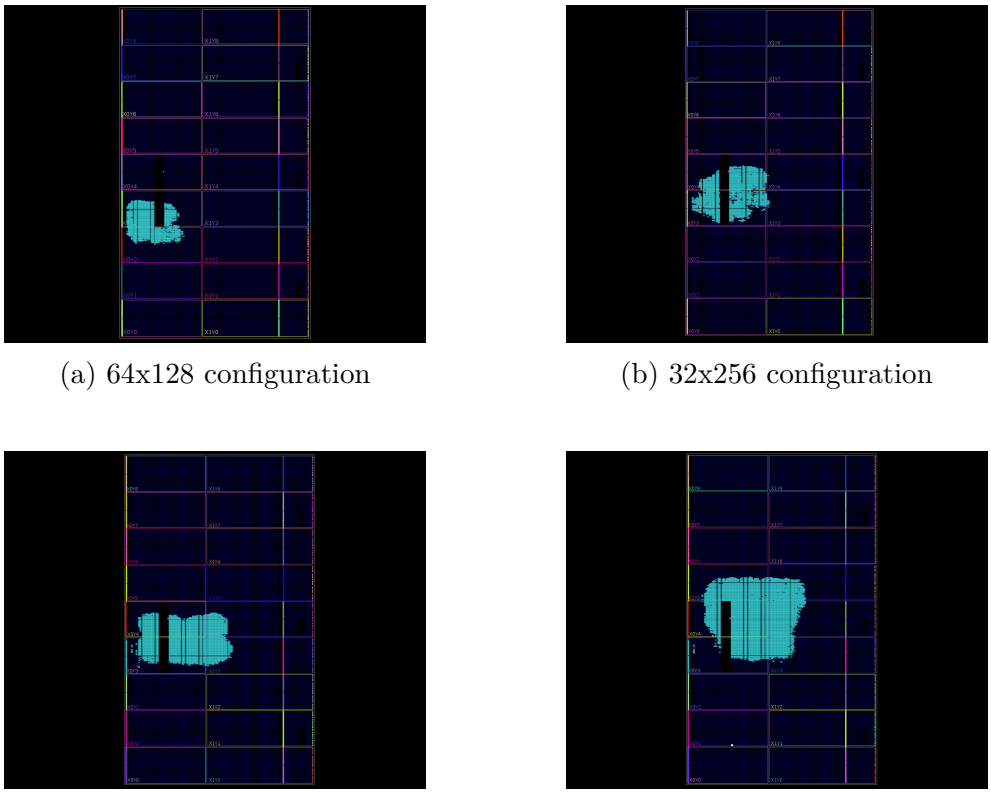
Table 7.14: Number of used resources for the kintex 7 480T FPGA.

7.2.8 Virtex 7 585T

In addition to the Virtex 7 2000T FPGA, other two FPGAs from the Virtex 7 family are tested. The first FPGA, i.e. the Virtex 7 FPGA with the lowest number of resources, is the xc7v585t model. All four configurations are correctly implemented.

LUT	REGs	BRAM
364200	728400	795

Table 7.15: Virtex 7 585T resources



(a) 64x128 configuration

(b) 32x256 configuration

(c) 16x512 configuration

(d) 8x1024 configuration

Figure 7.9: Area of the HDC module with respect to the total area of the Virtex 7 585T FPGA.

Configuration	LUT		REGs		BRAM	
	n°	%	n°	%	n°	%
64x128	8062	2.2	11590	1.59	14.5	1.82
32x256	13792	3.7	15088	2.07	20.5	3.08
16x512	22532	6.2	24421	3.35	0.5	0.06
8x1024	41418	11.3	40589	5.57	0.5	0.06

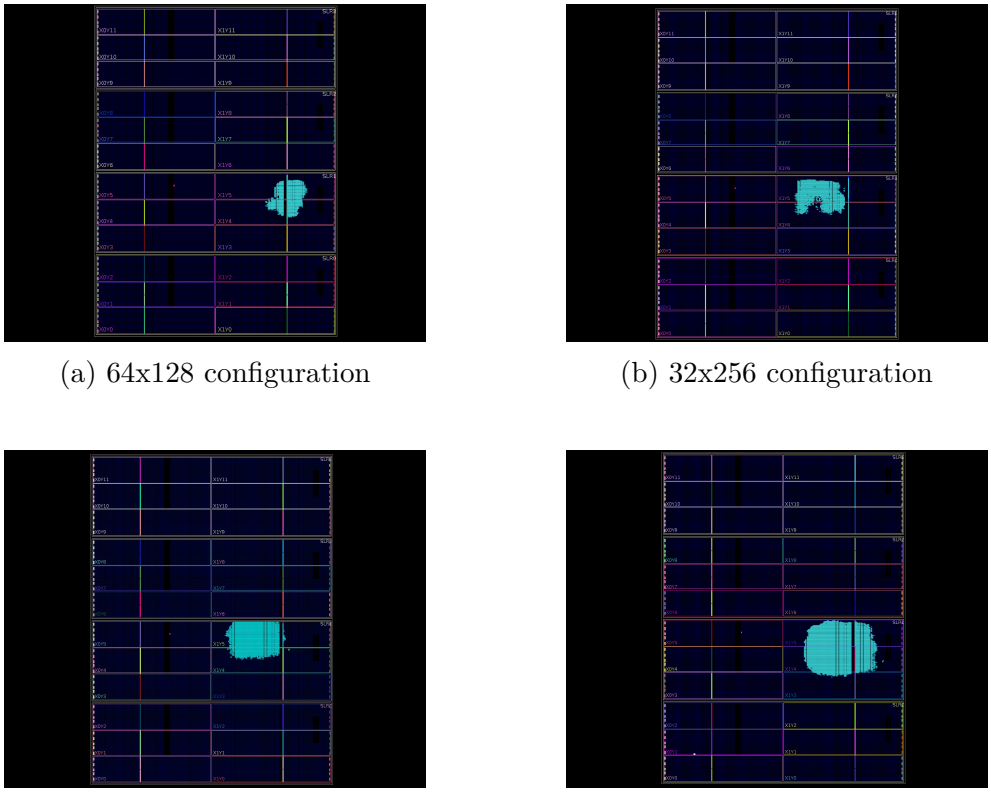
Table 7.16: Number of used resources for the Virex 7 585T FPGA.

7.2.9 Virtex 7 11400T

The third tested Virtex 7 FPGA is the xc7vx1140T model. It is the second FPGA of the Virtex 7 family in terms of number of resources, since the Virtex 7 2000T has higher capacity. All four configurations are correctly implemented.

LUT	REGs	BRAM
712000	1424000	1880

Table 7.17: Virtex 7 1140T resources



(a) 64x128 configuration

(b) 32x256 configuration

(c) 16x512 configuration

(d) 8x1024 configuration

Figure 7.10: Area of the HDC module with respect to the total area of the Virtex 7 1140T FPGA.

Configuration	LUT		REGs		BRAM	
	n°	%	n°	%	n°	%
64x128	8059	1.13	11590	0.81	14.5	0.77
32x256	13792	1.94	15088	1.06	20.5	1.09
16x512	22532	3.16	24421	1.71	0.5	0.03
8x1024	41418	5.82	40589	2.85	0.5	0.03

Table 7.18: Number of used resources for the Virtex 7 1140T FPGA.

7.2.10 Full Parallel Configuration

In order to understand the advantages in terms of area of the serialized implementation of the design, a full parallel configuration is also implemented.
In this case, hypervectors are not divided in pats. Instead, the design processes the complete hypervector, with it 8192 bits, at each iteration.
With a parallel implementation, the encoding phase is executed a single time for each input value. Hence, there is no need for the input histogram memory, since input values are only read once.
The utilization of resources for the full parallel configuration (1x8192) are taken by implementing the design on the Virtex 7 2000T FPGA.
As can be seen in Table 7.19, also for the full parallel configuration the Vivado

Configuration	LUT		REGs		BRAM	
	n°	%	n°	%	n°	%
1x8192	135837	11.12	192047	7.86	0	0

Table 7.19: Number of used resources for the implementation of the full parallel configuration on the Virtex 7 2000T FPGA.

software does not map memories with BRAM, but it uses sparse registers.
Another important result from the implementation is that the full parallel configuration has a critical path delay of 30.65 ns, hence it cannot be correctly executed using the target clock with a period of 10ns.

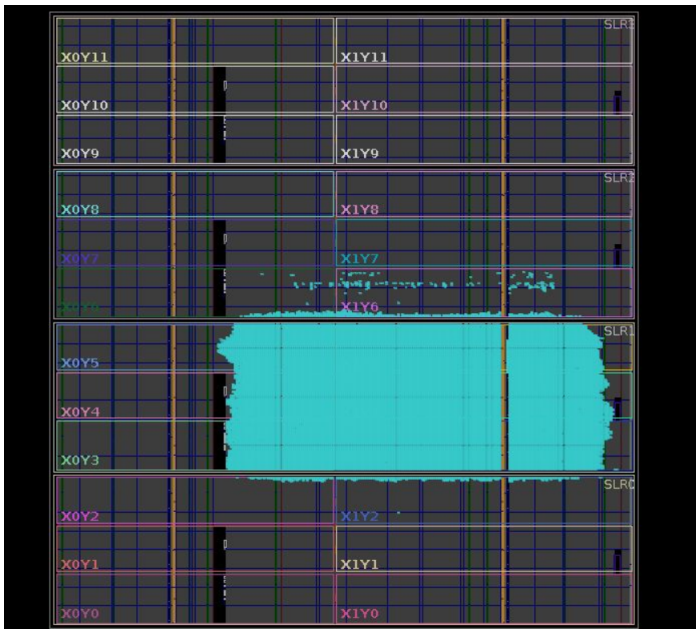


Figure 7.11: Area of the full parallel configuration for the HDC design with respect to the total area of the Virtex 7 2000T FPGA.

7.3 Time

7.3.1 Critical Path Delay

Thanks to Xilinx Vivado software, it is possible to extract information on the **critical path delay** for all the configurations and all the tested FPGAs.
Table 7.20 reports the maximum delay for each configuration and each tested FPGA. The critical path delay for the full parallel configuration is obtained by implementing the configuration on the Virtex 7 2000T. In particular, the maximum delay for the full parallel design is of 30.67ns, three times higher than the target clock frequency.

FPGA	Crititcal Path Delay (ns)			
	64x128	32x256	16x512	8x1024
Spartan 7 25c	7.20	8.33	-	-
Spartan 7 100f	7.5	9.34	10.39	11.72
Artix 7 15t	7.16	-	-	-
Artix 7 200t	8.13	8.7	8.2	9.73
Kintex 7 70t	6.19	8.4	7.12	-
Kintex 7 480t	6.4	8.61	7.52	8.38
Virtex 7 585t	6.01	7.42	7.609	7.66
Virtex 7 1140t	7.57	7.76	7.85	8.76
Virtex 7 2000t	6.3	8.21	8.28	9.9

Table 7.20: Maximum delay for all the configurations and all the tested FPGAs. Delayes highlighted in red are higher than the clock period (10 ns).

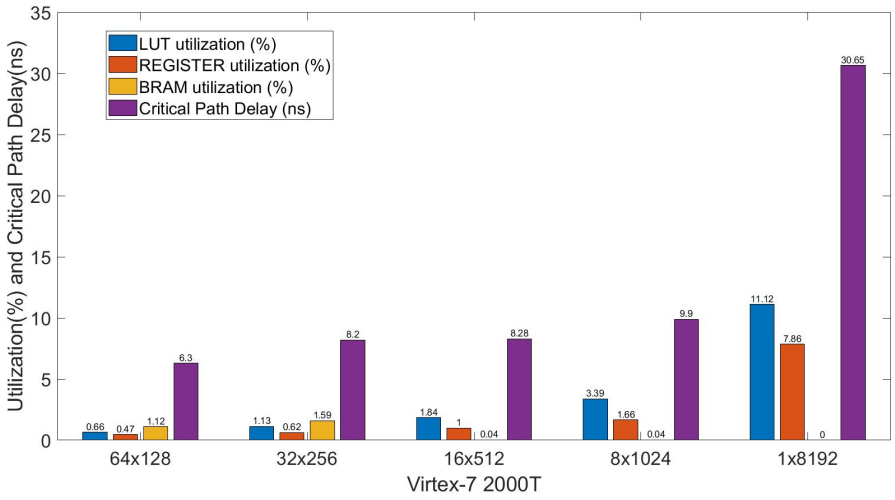


Figure 7.12: Resource Utilization (%) and Critical Path Delay (ns) for the Virtex 7 2000T with all serial and parallel configurations.

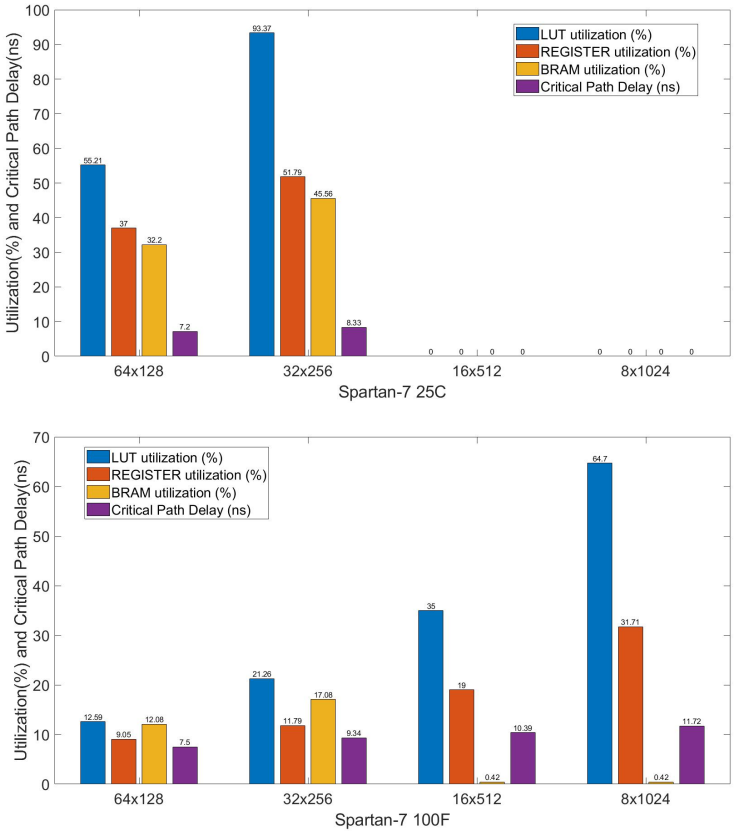


Figure 7.13: Resource Utilization (%) and Critical Path Delay (ns) for the Spartan 7 family. A value equal to 0 indicates that the configuration was not implemented.

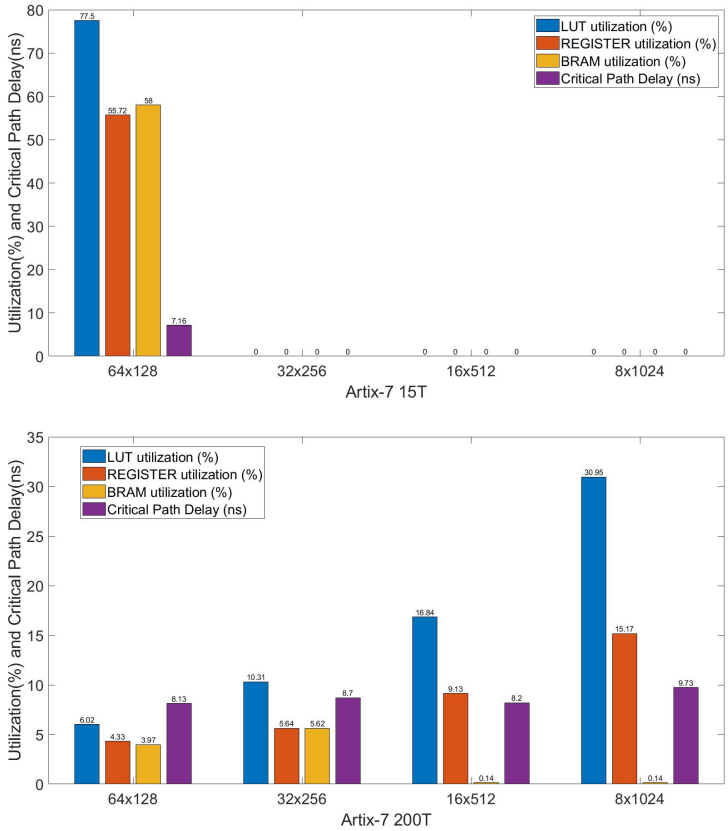


Figure 7.14: Resource Utilization (%) and Critical Path Delay (ns) for the Artix 7 family. A value equal to 0 indicates that the configuration was not implemented.

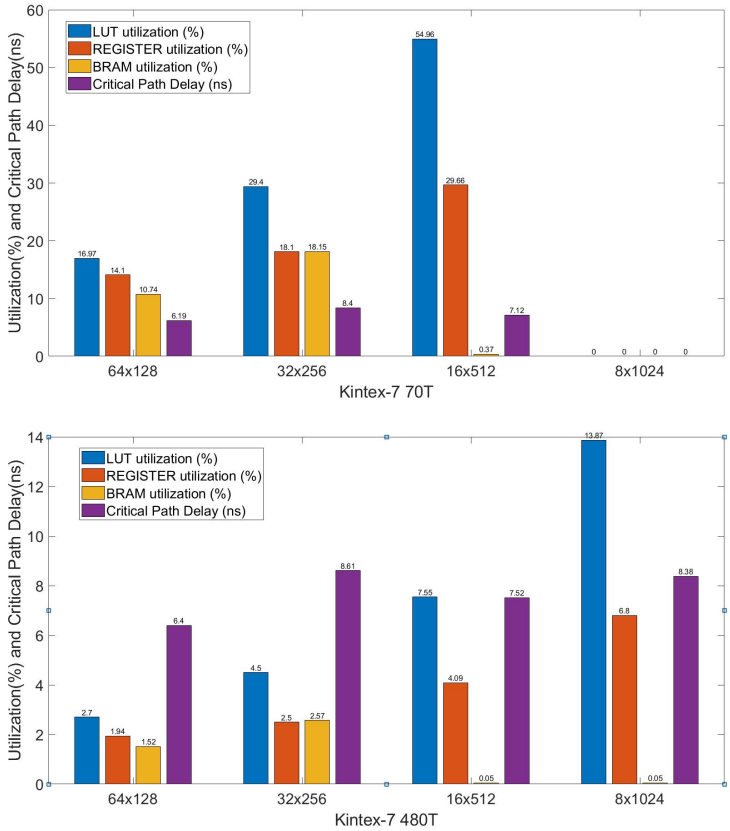


Figure 7.15: Resource Utilization (%) and Critical Path Delay (ns) for the Kintex 7 family. A value equal to 0 indicates that the configuration was not implemented.

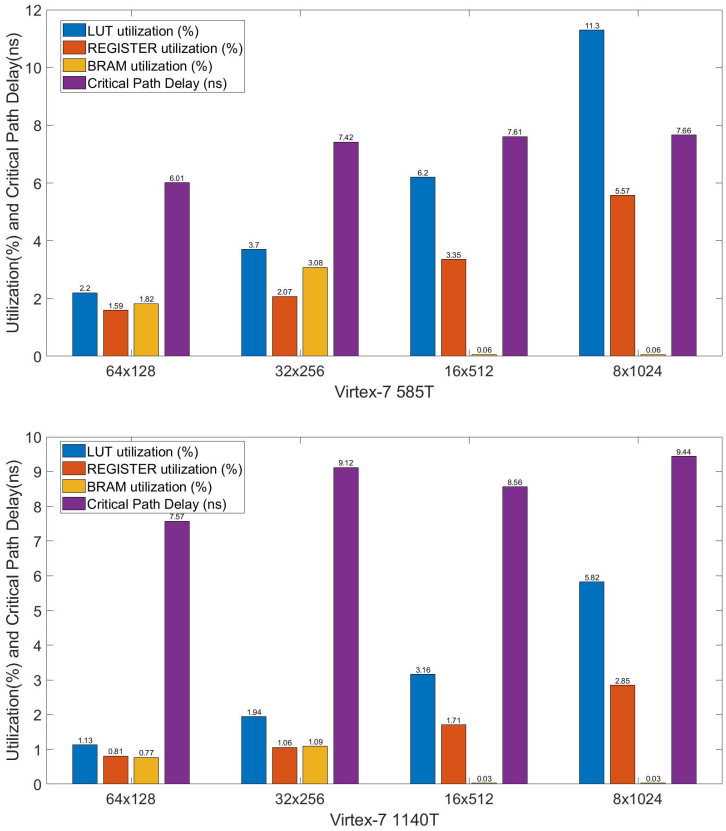


Figure 7.16: Resource Utilization (%) and Critical Path Delay (ns) for the Virtex 7 family. A value equal to 0 indicates that the configuration was not implemented.

7.3.2 Required Time For Classification

The implementation of the four configurations of the design are simulated on the Virtex 7 2000T FPGA using the Xilinx Vivado Simulator. In this way it is possible to verify the correct behavior of the implemented netlist.
Through the simulation, information about the required time to classify an input histogram are extracted. Table 7.21 reports the required time to classify a single histogram for all four configuration. These time measurements are different from the ones reported in Table 6.1 because they take into account also the time required for the handshake. In particular, for the simulation on FPGA using Vivado, the hanshake is executed in the optimal condition, i.e. with the minimum number of cycles.

Configuration	Required Time (ms)	Clock Cycles
64x128	3.7	370000
32x256	1.92	192000
16x512	1.02	102000
8x1204	0.57	57000

Table 7.21: Time and Clock cycles required to process a single histogram for all configuration. Clock cycles are computed by dividing the required time by the clock period equal to 10 ns.

As expected, the required time decreases as the number of processed parts decreases. This because in a configuration where an hypervector is divided in a lower number of parts, the number of time a phase of the classifier is repeated is lower. Through ModelSim simulator, the required time for the full parallel configuration to process a single histogram is measured. The full parallel configuration takes about $387\mu s$ to process a single histogram, with a clock period of 32ns. Hence, the full parallel configuration requires about 12094 clock cycles, which is a much lower value with respect to the clock cycles required for the serialized implementation.

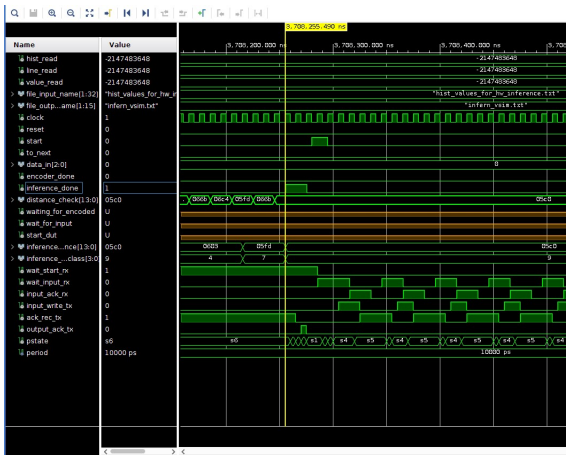


Figure 7.17: Required time to process a single sample with the 64x128 configuration. The clock period is 10ns. Image from Vivado Simulator.

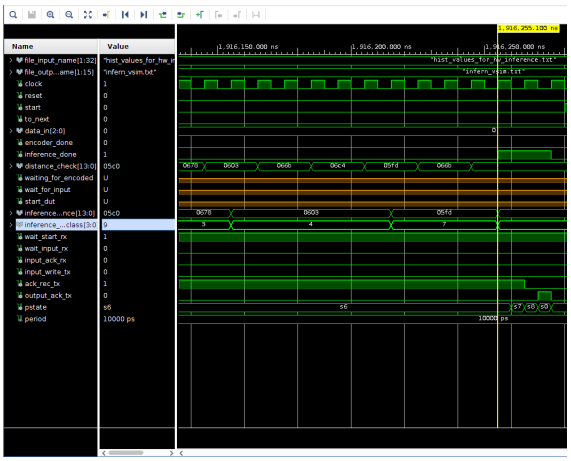


Figure 7.18: Required time to process a single sample with the 32x256 configuration. The clock period is 10ns. Image from Vivado Simulator.

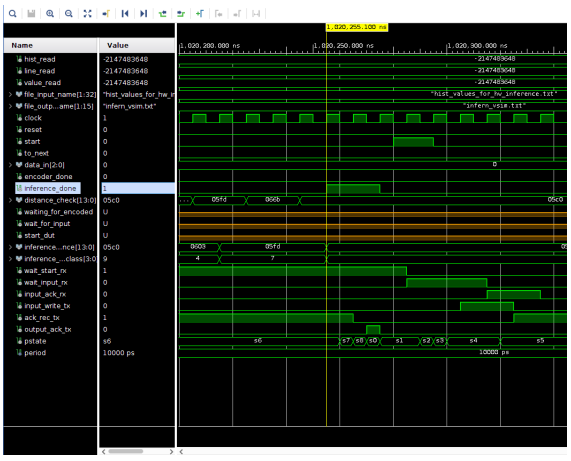


Figure 7.19: Required time to process a single sample with the 16x512 configuration. The clock period is 10ns. Image from Vivado Simulator.

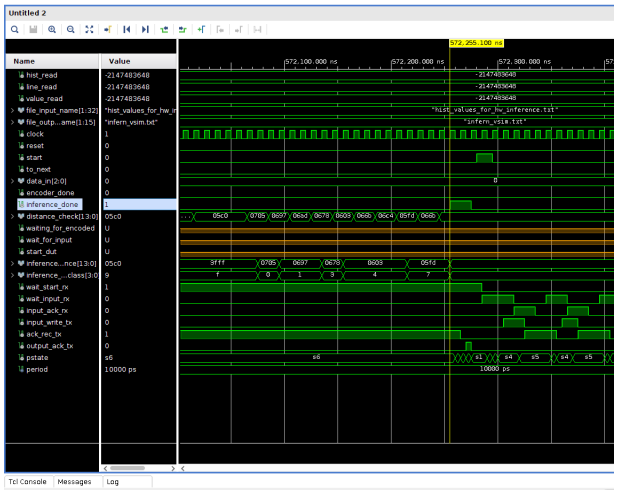


Figure 7.20: Required time to process a single sample with the 8x1024 configuration. The clock period is 10ns. Image from Vivado Simulator.

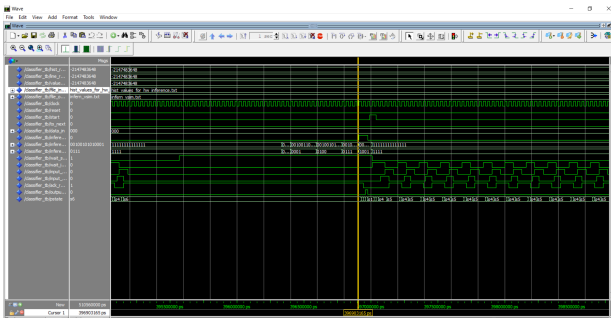


Figure 7.21: Required time to process a single sample with the full parallel configuration. The clock period is 32ns. Image from ModelSim Simulator.

From the required clock cycles reported in Table 7.21, and the critical path delays in Table 7.21, it is possible to compute the required time to process a single sample for all the configuration on all the tested FPGAs when working at the maximum frequency.

FPGA	64x128		32x256		16x512		8x1024	
	f_{max}	Time	f_{max}	Time	f_{max}	Time	f_{max}	Time
Spartan 7 15C	125MHz	2.96ms	111MHz	1.73ms	-	-	-	-
Spartan 7 100F	125MHz	2.96ms	100MHz	1.92ms	91MHz	1.12ms	83MHz	0.69ms
Artix 7 12T	125MHz	2.96ms	-	-	-	-	-	-
Artix 7 200T	111MHz	3.33ms	111MHz	1.73ms	111MHz	0.92ms	100MHz	0.57ms
Kintex 7 70T	143MHz	2.59ms	111MHz	1.73ms	125MHz	0.82ms	-	-
Kintex 7 480T	143MHz	2.59ms	111MHz	1.73ms	125MHz	0.82ms	111MHz	0.51ms
Virtex 7 585T	143MHz	2.59ms	125MHz	1.54ms	125MHz	0.82ms	125MHz	0.46ms
Virtex 7 1140T	125MHz	2.96ms	125MHz	1.54ms	125MHz	0.82ms	111MHz	0.51ms
Virtex 7 2000T	143MHz	2.59ms	111MHz	1.73ms	111MHz	0.92ms	100MHz	0.57ms

Table 7.22: Required Time for a single sample for all configurations on different FPGAs. Results estimated by using the maximum frequency for each configuration.

7.4 Power

The power measurements are obtained through the **back annotation** method. Through Xilinx Vivado software, the four designs are implemented on the target FPGA to obtain a **netlist** which is then simulated with the Vivado Simulator. From the simulation of the netlist, a **SAIF** file is reported. The SAIF file contains the switching activity for each net of the design and it is used by the Vivado power analysis tool to compute the power consumed by the implemented design.
In order to obtain a more accurate power analysis, the simulation with the Vivado Software must be set as **post-implementation timing simulation** .
The post-implementation timing simulation works only on verilog files, hence a verilog testbench must be implemented in order to simulate a design and obtain correct results.
All the steps for the simulation, saif file extraction and power measurements can be performed with a script.
In this project, the script to obtain power reports was provided by ing. Fabrizio Ottati.
More on how to run the power analysis tool on Vivado can be found in [24].
Table 7.23 reports power measurements for all the four configuration using the Virtex 7 2000T FPGA, with a clock frequency of 100MHz.

Configuration	Dynamic Power		Static Power		Total Power
	W	%	W	%	W
64x128	0.033	2.3	1.373	97.7	1.406
32x256	0.055	3.9	1.373	96.1	1.428
16x512	0.099	6.72	1.373	93.28	1.472
8x1024	0.181	11.6	1.377	88,4	1.558

Table 7.23: Power Measurements of the four configuration of the HDC design, using the Virtex 7 2000T as the target FPGA.

What follow is a series of tables which report the power measurements for all the four configuration tested on different FPGAs.

FPGA	Dynamic Power		Static Power		Total Power
	W	%	W	%	W
Spartan 7 25C	0.029	25.66	0.084	74.34	0.113
Spartan 7 100F	0.032	17.67	0.149	82.33	0.181
Artix 7 15T	0.030	24.39	0.093	75.61	0.123
Artix 7 200T	0.054	23.28	0.177	76.72	0.231
Kintex 7 70T	0.031	20.94	0.117	79.06	0.148
Kintex 7 70T	0.031	6.9	0.417	93.1	0.448
Virtex 7 585T	0.033	6.38	0.485	93.62	0.518
Virtex 7 1140T	0.033	2.63	1.223	97.37	1.256

Table 7.24: Power measurements of the 64x128 configuration on various FPGAs

FPGA	Dynamic Power		Static Power		Total Power
	W	%	W	%	W
Spartan 7 25C	0.048	36.09	0.084	63.15	0.132
Spartan 7 100F	0.053		0.150		0.203
Artix 7 200T	0.054	23.37	0.177	76.62	0.231
Kintex 7 70T	0.048	28.91	0.118	71.08	0.166
Virtex 7 1140T	0.053	4.15	1.224	95.85	1.277

Table 7.25: Power measurements of the 32x256 configuration on various FPGAs

FPGA	Dynamic Power		Static Power		Total Power
	W	%	W	%	W
Kintex 7 480T	0.100	19.31	0.418	80.69	0.518
Virtex 7 1140T	0.096	7.27	1.224	92.7	1.32

Table 7.26: Power measurements of the 16x512 configuration on various FPGAs

FPGA	Dynamic Power		Static Power		Total Power
	W	%	W	%	W
Kintex 7 480T	0.189	31.03	0.420	68.97	0.609
Virtex 7 1140T	0.187	13.22	1.227	86.78	1.414

Table 7.27: Power measurements of the 8x1024 configuration on various FPGAs

As expected, the dynamic power grows with the parallelism of the parts of hypervectors. The static power is instead releated to the complexity of the FPGA. It is lower for FPGAs with lower capacity and higher for larger FPGAs.

Chapter 8

Conclusion and Future Works

In this project, an hardware accelerator for images classification based on hyperdimensional computing was implemented. In particular, the classified images are acquired using a novel type of bio-inspired sensors called Dynamic Vision Sensors. To my knowledge, this is one of the first works which implement an hardware solution for DVS classification using Hyperdimensional Computing.
The proposed design is scalable, since the number of processed parts of hypervectors can be configured. This lead to the realization of a design that can be adapted for the implementation on smaller FPGAs.
Results report an accuracy of the proposed design on the N-MNIST dataset of 83%, without retraining.
For what concern utilization, time and power measurements, a configuration which uses parts of hypervector with lower parallelism takes longer time to provide a single output; however, it uses a lower percentage of resources and consumes less dynamic power.
Results in terms of accuracy can be improved in future works by applying a retraining process on the whole dataset, in order to better tune the model.
In particular, training in this project is performed via software. A possible task for future works is to implement training online, i.e. directly on the hardware.
To better assess the capabilities of the proposed design, the hardware module can be tested on other datasets different from the N-MNIST dataset used in this project.

Bibliography

[1] Lulu Ge, Keshab K. Parhi. *Classification using Hyperdimensional Computing: A Review.* IEEE Circuits and Systems Magazine, 2020.

[2] Manuel Schmuck, Luca Benini, Abbas Rahimi. *Hardware Optimizations of Dense Binary Hyperdimensional Computing: Rematerialization of Hypervectors, Binarized Bundling, and Combinational Associative Memory.* ACM Journal on Emerging Technologies in Computing Systems, 2019.

[3] Guillermo Gallego, Tobi Delbruck, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew J. Davison, Jorg Conradt, Kostas Daniilidis, Davide Scaramuzza. *Event-based Vision: A Survey.* IEEE, 2020.

[4] Xavier Lagorce, Garrick Orchard, Francesco Galluppi, Bertram E. Shi, and Ryad B. Benosman. *HOTS: A Hierarchy of Event-Based Time-Surfaces for Pattern Recognition.* IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, 2017.

[5] Amos Sironi, Manuele Brambilla, Nicolas Bourdis, Xavier Lagorce, Ryad Benosman. *HATS: Histograms of Averaged Time Surfaces for Robust Event-based Object Classification.* Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.

[6] Emanuele Garola. *Adaptation and implementation of a LIM architecture on a multi FPGA system.* Politecnico di Torino, 2017.

[7] Giulia Altamura. *Development of hardware accelerators on FPGA for convolutional neural networks.* Politecnico di Torino, 2018.

[8] Sizhe Zhang, Ruixuan Wang, Jeff (Jun) Zhang, Abbas Rahimi, Xun Jiao. *Assessing Robustness of Hyperdimensional Computing Against Errors in Associative Memory.* IEEE 32nd International Conference on Application-specific Systems, Architectures and processors (ASAP), 2021.

[9] Mohsen Imani, Deqian Kong, Abbas Rahimi, and Tajana Rosing. *VoiceHD: Hyperdimensional Computing for Efficient Speech Recognition.* IEEE, 2017.

[10] Mohsen Imani, John Messerly, Fan Wu, Wang Pi, and Tajana Rosing. *A Binary Learning Framework for Hyperdimensional Computing.* IEEE, 2019.

[11] Abbas Rahimi, Pentti Kanerva, Jan M. Rabaey. *A Robust and Energy-Efficient Classifier Using Brain-Inspired Hyperdimensional Computing.* Association for Computing Machinery,New York,NY,United States, 2016.

[12] Mohsen Imani, Samuel Bosch, Sohum Datta, Sharadhi Ramakrishna, Sahand Salamat, Jan M. Rabey, Tajana Rosing. *QuantHD: A Quantization Framework for Hyperdimensional Computing.* IEEE, 2019.

[13] Mohsen Imani, Chenyu Huang, Deqian Kong, Tajana Rosing. *Hierarchical Hyperdimensional Computing for Energy Efficient Classification.* IEEE, 2018.

[14] Pentti Kanerva. *Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors.* Springer Science+Business Media, 2009.

[15] Mohsen Imani, Justin Morris, John Messerly, Helen Shu, Yaobang Deng, Tajana Rosing. *BRIC: Locality-based Encoding for Energy-Efficient Brain-Inspired Hyperdimensional Computing.* IEEE, 2019.

[16] Mohsen Imani, Xunzhao Yin,John Messerly, Saransh Gupta, Michael Niemier, Xiaobo Sharon Hu, Tajana Rosing. *SearcHD: A Memory-Centric Hyperdimensional Computing With Stochastic Training.* IEEE, 2019.

[17] Garrick Orchard, Cedric Meyer, Ralph Etienne-Cummings, Christoph Posch, Nitish Thakor. *HFirst: A Temporal Approach to Object Recognition.* IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, 2015.

[18] José Antonio Pérez-Carrasco, Bo Zhao, Carmen Serrano, Begoña Acha, Teresa Serrano-Gotarredona, Shouchun Chen, Bernabé Linares-Barranco. *Mapping from Frame-Driven to Frame-Free Event-Driven Vision Systems by Low-Rate Rate Coding and Coincidence Processing—Application to Feedforward ConvNets.* IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, 2013.

[19] PRODESIGN Electronic GmbH. *profpga Hardware User Manual.* 2019.

[20] PRODESIGN Electronic GmbH. *proFPGA Extension Board Design Guide.* 2019.

[21] PRODESIGN Electronic GmbH. *profpga Builder User Manual.* 2019.

[22] PRODESIGN Electronic GmbH. *proFPGA Example Designs.* 2019.

[23] PRODESIGN Electronic GmbH. *Getting Started with MMI64*. 2019.

[24] Xilinx. *Vivado Design Suite Tutorial: Power Analysis and Optimization*. 2022.