KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Designing a Programmable Accelerator for Sparse Distributed Binary Hyperdimensional Computing

Tim Deboel

Thesis submitted for the degree of
Master of Science in
Electrical Engineering, option
Electronics and Chip Design

Supervisor:
Prof. dr. ir. Marian Verhelst

Assessors:
Prof. dr. ir. Georges Gielen
Prof. dr. ir. Wannes Meert
Ir. Ryan Antonio

Assistant-supervisor:
Ir. Ryan Antonio

Academic year 2023 – 2024

Preface

First of all, I would like to thank Marian Verhelst and Ryan Antonio for introducing me to this exciting topic. I really enjoyed getting to know more about the world of hyperdimensional computing and its architectures. In general, I would like to thank everybody who has helped me and kept me motivated to work hard during this project. Special thanks to my mentor, Ryan, for his guidance, insightful feedback, and willingness to always help with my problems. The meetings with him always led to very insightful discussions and kept me on the right path. I am also grateful to the faculty and staff of the electrical engineering department, for their resources and assistance in running my simulations. Lastly, I would like to thank the jury for reading this text.

Tim Deboel

Contents

Preface **i**

Abstract **iv**

List of Figures and Tables **v**

List of Abbreviations and Symbols **vii**

1 Introduction **1**

1.1 Contributions 2

1.2 Outline 2

2 Hyperdimensional Computing **3**

2.1 The Basics of HDC 3

2.2 Character Recognition 8

2.3 Language Recognition 13

2.4 Sparse HDC 15

2.5 Hardware Implementations 21

2.6 The Encoding Scheme Problem 24

2.7 Summary 25

3 Implementing the Encoding Schemes **27**

3.1 Defining the Objectives 27

3.2 Investigating the Encoding Schemes 28

3.3 Summary 35

4 Designing an Efficient Programmable SDB-HDC Accelerator **37**

4.1 Basic Architecture 37

4.2 Module Implementations 39

4.3 Accelerator Implementation 45

4.4 Improvements for the Accelerator 53

4.5 Conclusion 56

5 Performance Characteristics **57**

5.1 Measuring the Latency, Area, and Energy usage 57

5.2 Characterizing the Performance 58

5.3 Conclusion 64

6 Conclusion **65**

6.1 Conclusion 65

6.2 Recommended Studies 66

A Software code **71**

A.1 Applications in Software 71

A.2 Hardware Accelerator 72

Bibliography **77**

Abstract

Sparse Distributed Binary Hyperdimensional Computing (SDB-HDC) is a framework for executing ultra-low-power AI tasks. A programmable accelerator for SDB-HDC can make use of its properties to efficiently run IoT applications. The energy efficiency and similar performance of SDB-HDC has caused it to recently gain in popularity, however the differences in the non-generalized application-specific encoding schemes make it difficult to design an efficient programmable accelerator. State-of-the art accelerators focus on the dense distributed binary (DDB)-HDC space since it has fewer encoding operations compared to SDB-HDC. However, it underutilizes HDCs potential due to the density of its vectors, and the resulting high switching activity during its calculations. This work presents a programmable and energy-efficient SDB-HDC accelerator, that supports the commonly used encoding schemes with temporal resource reuse. Starting from an investigation of the different Sparse HDC encoding schemes, a technique to improve the accuracy was found by introducing the sorted threshold method during training. Additionally, opportunities for temporal resource reuse were identified to achieve hardware with high energy efficiency and throughput. Access to the sorted THR method and a programmable ultra-low-power SDB-HDC accelerator that can support varying IoT applications allows for a better research space for this algorithm.

List of Figures and Tables

List of Figures

2.1 How HDC vectors represent features (dog image from [22]). 5
2.2 Visualization of Hamming distance between two hypervectors. 6
2.3 Example of a bundling operation between three hypervectors. 6
2.4 Example of a binding operation between three hypervectors. 7
2.5 Example of a hypervector permutation operation. 8
2.6 Basic HDC architecture. 8
2.7 Character recognition pixel grid for the letter A. 10
2.8 Data flow for the DDB character recognition application. 11
2.9 DDB-HDC character recognition accuracy.[15] 12
2.10 Example of a sliding trigram window. 13
2.11 Architecture for DDB-HDC language recognition. (retrieved from [29]) . 14
2.12 Data flow for the SDB character recognition application. 17
2.13 SDB-HDC character recognition accuracy.[14] 18
2.14 Architecture for SDB-HDC language recognition. (retrieved from [10]) . 19
2.15 Classification accuracy for the sparse and dense HDC designs for different dimensionalities D. (retrieved from [10]) 20
2.16 The accuracy, power consumption, and execution time of the sparse and dense HDC designs for different item memory HV sparsities. (a) Accuracy. (b) Energy consumption. (c) Execution time. (retrieved from [10]) 20
2.17 Impact of the THR value on the classification accuracy, sparsity, and switching activity. (retrieved from [10]) 21
2.18 Generic HD Processor architecture. (retrieved from [5]) 21
2.19 (a) 1 HLU unit, (b) connecting HLUs to form a layer, (c) encoding steps required for a 3-gram, (d) connecting multiple HLU layers to generate terms, (e) accumulator for superposition, (f) for the generic model, the delay and permute partial terms can be separately computed and combined. (retrieved from [5]) 23
2.20 Programmable HDC encoder architecture responsible for item memory materialization. (retrieved from [6]) 24

4.1 Most common general structure for HDC algorithms. 37

4.2 High-level overview of the accelerator architecture. 38
4.3 Representation of the MAN module and an example connectivity matrix. (retrieved from [31]) 41
4.4 Example connectivity matrix for B2B bundling. (retrieved from [31]) . . 43
4.5 Bundling capacity for B2B vs. "ideal" bundling, for D=10,000. (retrieved from [31]) 44
4.6 The programmable accelerator architecture. 46
4.7 Diagram for 1 bit of the saturating accumulator and thresholding. . . . 48
4.8 Diagram for a CDT implementation with a maximum k-factor of 2. . . . 49
4.9 Visualization of the accelerator integrated with a PULP processor. . . . 51
4.10 Example of a barrel shifter implementation. (retrieved from [1]) 54
4.11 The improved programmable accelerator architecture. 56

5.1 Total area comparison based on dimensionality. 59
5.2 Breakdown of the area usage per module for D=2048. 59
5.3 Comparison between both designs of the average latency for each test. . 60
5.4 Comparison of the total average power usage based on dimensionality. . 61
5.5 Overview of the energy usage per prediction for D=2048. 61
5.6 Breakdown of the energy usage for the improved design and D=2048. . 62

List of Tables

2.1 DDB-HDC language recognition results from [29]. 14

3.1 SDB-HDC character recognition results for 0.98% IM density. 29
3.2 SDB-HDC character recognition results for 4.88% IM density. 29
3.3 SDB-HDC language recognition density disparities after training. 32
3.4 Language recognition accuracy comparison between the different implementations for varying dimensionality. 34
3.5 Language recognition accuracy comparison between the dense and the sparse sorted THR implementation. 34

4.1 Description of the CSR register map. 47
4.2 New bundling section of the CSR register map. 55

5.1 Latency breakdown for character and language recognition. 60
5.2 Performance comparison with [5]. 63
5.3 Performance comparison for language recognition with [5] and [6]. . . . 63

6.1 Language recognition accuracy comparison for varying implementation and dimensionality. 65
6.2 Performance specifications of the programmable SDB-HDC accelerator. 66

List of Abbreviations and Symbols

Abbreviations

HDC	Hyperdimensional Computing
DDB	Dense Distributed Binary
SDB	Sparse Distributed Binary
HV	Hypervector
IoT	Internet of Things
SOTA	State of the Art
THR	Threshold
CDT	Context-Dependent Thinning
IM	Item Memory
AM	Associative Memory
LUT	Lookup Table

Chapter 1

Introduction

Machine learning is used more and more in today's applications, but its downside of requiring high computational power and energy usage makes it badly suited for IoT devices. It is a subset of artificial intelligence that allows computer systems to recognize patterns and make decisions, without needing human intelligence.[32] Most machine learning applications use power-hungry floating-point operations like multiply-accumulate. These often use classical Von-Neumann architectures, which have a data transfer bottleneck between the processor and memory. These downsides make them hard to integrate with IoT devices because these often require extremely low power usage.[4]

Hyperdimensional Computing (HDC) is a brain-inspired computing paradigm, which can perform machine learning tasks with amazing power efficiency. HDC represents data based on very high-dimensional vectors and simple operations, which allows for lighter computational loads compared to using conventional scalar numbers. It encodes data based on the combination of sequences of representations through bit-wise manipulation, using much more energy-efficient operations. Its main use lies in classification algorithms, where a best match with the stored representations determines the output. It aims to provide a very low-power method of computing through a high dimensionality similar to the brain, while maintaining or even increasing the accuracy by up to 5-10%. Depending on the application, it can provide an energy-efficiency increase of over 2-10x and a throughput increase of 2-5x.[29, 12, 28, 11, 20, 15] Other notable features include fast learning, robustness, and easy parallelism on top of cheap and simple hardware implementations.

Sparse-Distributed Binary HDC is a subset of the HDC-based algorithms. Where the applications referenced above utilize binary vectors with a density of 50%, SDB-HDC is based on high-dimensional binary vectors with a density of ones less than this, usually $< 10\%$.[25] In hardware this sparsity causes less switching of transistors, resulting in less dynamic energy usage and a further increase of HDC's benefits. For non-programmable accelerators, when the result of an operation is always a zero, that part of the hardware can even be dropped as it doesn't contain useful data.

The downside of SDB-HDC is that there is a variety of non-generalized encoding schemes that are necessary to achieve good accuracy. Even though it is slightly less generalized, with the right encoding schemes there is no, or only marginal accuracy loss (1-3%) compared to DDB-HDC.[25, 30, 19, 14, 10] A major challenge is creating hardware for these different encoding schemes, as it is either very application-specific or hard to generalize because of the variety of methods.

1.1 Contributions

The goal of this work is to develop a programmable SDB-HDC accelerator. To this extent, we first investigate the SDB-HDC encoding schemes and discover opportunities for improvement. The sorted threshold method is developed, modifying the regular thresholded sum encoding scheme during training for an accuracy increase of 2-20%. Afterward, the aim is to design a programmable hardware accelerator that supports the most common encoding schemes and their various combinations. To that extent, this thesis introduces a hardware architecture that combines their operations into an efficient and flexible accelerator implementation. Optimizations are implemented to minimize the area and energy usage, by reducing registers and operations and utilizing temporal reuse.

As of the making of this thesis, the field of SDB-HDC is still emerging, and little to no research has been done regarding hardware implementations for its encoding schemes. Currently, programmable hardware accelerators for DDB-HDC exist, but none have attempted to incorporate the variety of encoding schemes for their sparse-distributed counterpart. The most common encoding schemes occurring today utilize superposition and context-dependent-thinning or accumulation and thresholding.[30, 14, 10, 19] In this thesis, we explore and discover opportunities in the different encoding schemes, and then combine their operations into an efficient programmable hardware accelerator.

1.2 Outline

Chapter 2 provides the necessary background information on the HDC and SDB-HDC algorithms and encodings, together with some applications. The challenges and concepts to create a programmable hardware accelerator for SDB-HDC are also outlined here. In chapter 3, the research objectives are described, followed by an investigation and implementation of the various encoding schemes. Chapter 4 gives an overview of the possible hardware implementations for each module and explains the resulting programmable accelerator architecture. The accelerator's simulation results and performance evaluation are discussed in Chapter 5. Finally, chapter 6 presents the conclusion and outlines possible future work.

Chapter 2

Hyperdimensional Computing

Hyperdimensional computing (HDC) aims to provide a different method of representing and encoding data, to remove the bottlenecks present in classical von Neumann architectures. The classical von Neumann computer architecture combines memory, a processor, and I/O channels to process data instruction by instruction. All architectures are based on a certain representation of the input data. In this case, components can have only two states: 0 for ’off’ and 1 for ’on’, so the representation consists of a string of bits or binary vectors. Representations can come in many forms, the only condition is that patterns for different things are unique. A human brain, for example, operates in a completely different and nonbinary way. It utilizes a vast amount of neurons and synapses to represent and compute with data, based on patterns between its activating neurons. Binary representations are simple, but can still demonstrate brainlike features because the high dimensionality is more important than what the dimensions look like. Hyperdimensional computing aims to approach a brainlike method of computing by utilizing very high-dimensional binary vectors.[13]

2.1 The Basics of HDC

Hyperdimensional computing is a framework for computing with patterns, based on ultrawide words of e.g. 10,000 bits. We call these *hypervectors* (HVs) and use them to compute in very high-dimensional spaces. Data is no longer represented by an exact sequence of ones and zeros, but rather by patterns within these hypervectors. Binary hypervectors have some interesting properties, which can drastically boost the efficiency and sometimes even the accuracy compared to more classical computing. The following three sections will talk about the "regular" dense distributed HDC, where the density of ones is 50%, and some applications. This is followed by a comparison to the sparsely distributed variant, both in its properties and for the applications.

2.1.1 Hypervector Properties

Randomness

Dense HDC utilizes hypervectors where 50% of the bits are 1 and 50% are 0. These ones are randomly distributed throughout the vector, so each vector is unique and can be used to represent or encode something different. Later, we will perform operations on them to link them and create similarity in the patterns between similar objects.

Robustness

Because the dimensionality of HVs is extremely high, there is a tolerance to some amount of bit "errors" and we have robustness. This happens because there is redundancy in the representation and multiple (slightly) different patterns can be seen as equivalent.[13] The ones are randomly distributed throughout the vector, so the encoded information ends up "equally" distributed over the entire vector. When errors occur, the information degrades solely based on the number of errors instead of also based on their position like in more regular architectures. This forms what we call a *holistic* representation and mimics how the brain spreads out information over widely dispersed neurons.[13]

Orthogonality & Capacity

Another property of randomly generated hypervectors is that they are *almost orthogonal* to each other (this is called quasi-orthogonal), this is important because quasi-orthogonal vectors are not "similar" to each other. As will be further explained in section 2.1.2, HDC classification algorithms are based on the similarity between two hypervectors. Similar hypervectors are associated or related to each other; they represent the relations between data through their similarity. We call two HVs dissimilar when they are quasi-orthogonal, and similar when they are not. The higher the dimensionality of the vectors, the higher the probability that any two random vectors are quasi-orthogonal. Each n-dimensional space has $n - 1$ exactly orthogonal vectors, but many more quasi-orthogonal vectors with an angle of almost 90 degrees to each other. The result is that two randomly generated hypervectors are extremely likely to be quasi-orthogonal.[21] For example, the probability that two 10,000-dimensional hypervectors have an angle within 90 ± 5 degrees is almost one.[30] Quasi-orthogonal hypervectors are *dissimilar* to each other; their Hamming distance is around 0.5.

Capacity

Hypervectors have an extremely high capacity for storing patterns. Linearly increasing the number of dimensions in a binary vector, leads to an exponential increase in the number of possible stored patterns. Mathematically, the capacity for n binary dimensions is 2^n.[21] This is why they work well to encode data and have plenty of unique representations.

2.1.2 Arithmetic Operations

Generalized

Hyperdimensional computing has arithmetic operations that manipulate hypervectors and transform their bit patterns to create useful output. They can for example bind two vectors together, to create a new unique vector representing a combination of features. Vectors can also be bundled, resulting in a vector similar to its inputs that contains information about them. These operations mimic the more classical sum and multiplication but utilize much more efficient hardware.[13]

Figure 2.1 shows a generalized example of how features/information can be linked together using hypervectors. A *binding* operation is used to bind two features together and create a new unique hypervector representing that combination. Here "Eyes" and "Black" are bound together to represent the dog's "black eyes" feature. Multiple of these features are then combined using a *bundling* operation. This operation creates a new hypervector that retains the information of its inputs and is *similar* to them. In this example, the main features of the dog could be described as black eyes, black fur, and pointy ears. These are bundled together to create a single hypervector, which represents a combination of the input features; this single hypervector represents the dog. Note that all the hypervectors used in these operations have the same dimensions.

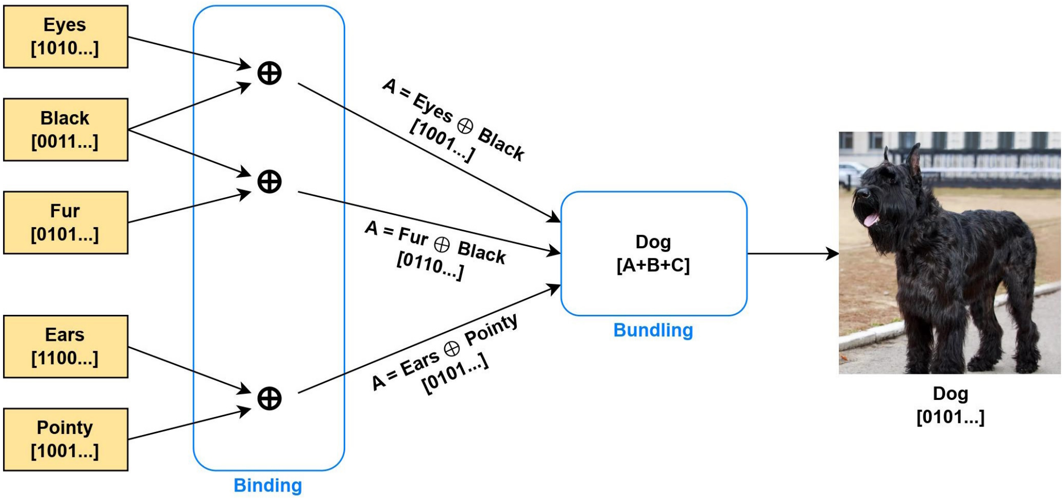


FIGURE 2.1: How HDC vectors represent features (dog image from [22]).

Similarity

One of the fundamental operations in computing is comparing vectors/data. This is necessary to enable decision-making and the sorting and filtering of data. In HDC this can be done by measuring the similarity between two vectors, using their *Hamming distance*. This operation counts the number of differences between two hypervectors, which comes down to a bitwise Exclusive-OR (XOR) operation ($\oplus$) followed by an accumulator as a counter.[30] The XOR operation outputs a 0 where the two vectors match (both 0 or both 1) and a 1 where they disagree. Because two randomly generated hypervectors are likely to be quasi-orthogonal, their mean distance will be around 50% of the size of their dimension D. For a dense hypervector with a size of 10,000 bits, we can say that $D = 10,000$ and $p = 0.5$. The expected Hamming distance would then be around 5,000.

We call hypervectors with distances 50% of their size or more apart *dissimilar* or *orthogonal*; otherwise, they can be considered *similar*. This Hamming distance can be normalized to be $\in [0, 1]$, and the complementary then gives a metric of similarity in the binary hyperdimensional space (where 0 means dissimilar and 1 means similar).[30] Similarity is an important metric for classification, and when linking certain features together. Figure 2.2 shows how the hamming distance between two hypervectors is calculated. First a bitwise XOR is taken and then the ones are summed up, resulting in a Hamming distance: $d(A, B) = \frac{26}{50} = 0.52$.

A	0	0	1	0	1	0	1	0	1	1	1	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	1	0	0	1	0	1	1	0
B	1	1	0	1	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	1	1	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	1	0	1	0	0	0	0	1	0	1	1	1	0
A ⊕ B	1	1	1	1	1	0	1	0	1	0	1	0	0	1	1	0	0	0	0	1	0	1	1	0	1	0	1	1	0	1	1	1	1	0	1	0	0	1	0	0	0	0	1	0	1	1	1	0	0	0

Figure 2.2: Visualization of Hamming distance between two hypervectors.

Bundling

Vectors can be combined by componentwise addition, through what we call a bundling operation. This operation is most commonly represented by a "+" symbol. The bundling operation superposes multiple hypervectors to create an output vector that is *similar* to its inputs while retaining dimensionality. This means that after the operation $\mathbf{H} = \mathbf{A} + \mathbf{B}$, the bundled vector $\mathbf{H}$ is similar to both $\mathbf{A}$ and $\mathbf{B}$. The bundled vector thus forms a representation for the set of its inputs.[13, 30] Usually, the operation is implemented by a simple element-wise addition, followed by a normalization step. In this case, the output should still be a binary hypervector, so a thresholding step is applied after addition.

A	1	1	1	0	1	1	1	1	0	0	0	1	0	1	1	0	0	1	0	0	0	1	0	0	1	1	1	0	1	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	1	0	1
B	0	0	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	1	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	1	1	0	1	0	0	1
C	0	1	0	1	1	1	0	0	0	1	0	1	0	0	0	1	0	1	1	1	0	1	0	0	1	1	0	0	1	1	0	0	0	1	0	1	1	0	1	0	0	0	1	0	1	0	1	1	1	0
A + B + C	1	2	1	1	3	2	1	1	0	2	1	2	0	2	1	1	0	3	1	2	1	3	1	1	3	2	1	1	2	1	1	0	0	2	0	1	3	1	1	0	1	2	1	1	3	0	2	2	1	2

Figure 2.3: Example of a bundling operation between three hypervectors.

As is visually demonstrated in Figure 2.3, bundling in DDB-HDC comes down to a bitwise majority sum. In this example, three hypervectors are added together, meaning that the bundled hypervector will have a 1 if the bitwise sum $> 3/2$ and a 0 otherwise. The thresholded sum (majority sum) can be written as:

$$G = [A + B + C]_{50\%} \tag{2.1}$$

If we calculate the Hamming distance as described above, we can see that:

$$d(G, A) = 0.2 \tag{2.2}$$

$$d(G, B) = 0.28 \tag{2.3}$$

$$d(G, C) = 0.22 \tag{2.4}$$

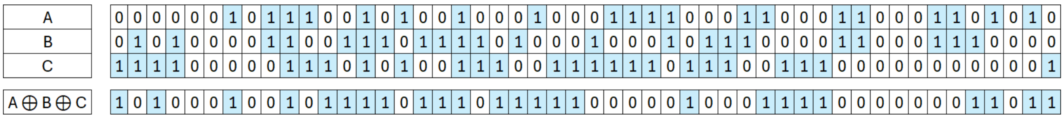
This indicates that the bundled output is indeed similar to its inputs.

Binding

The binding operator corresponds to a multiplication of hypervectors. Its key properties can be defined as follows[13, 30]:

- Binding is invertible (called unbinding).
- The output is *dissimilar* to the inputs: $H = A \oplus B$ is dissimilar to both A and B.
- Structured similarity is preserved: say A' is similar to A and B' is similar to B, then $A \oplus B$ is similar to $A' \oplus B'$.

For DDB-HDC, this operation is implemented by a component-/bitwise XOR ($\oplus$) This is exactly equivalent to element-wise multiplication in binary and provides a quasi-orthogonal (dissimilar) output. The XOR operation provides an exact inverse; it can be used for binding and for unbinding by simply reapplying the XOR.[30] A major advantage is that a single XOR gate per bit makes for very efficient and parallelizable hardware implementations. Figure 2.4 visualizes what an XOR binding operation between three randomly generated hypervectors looks like.



A	0	0	0	0	0	0	1	0	1	1	1	0	0	1	0	1	0	0	1	0	0	0	1	0	0	0	1	1	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	1	0	1	0
B	0	1	0	1	0	0	0	0	1	1	0	0	1	1	1	0	1	1	1	1	0	1	0	0	0	1	0	0	0	1	0	1	1	1	0	0	0	0	1	1	0	0	0	1	1	1	0	0	0	0
C	1	1	1	1	0	0	0	0	0	1	1	1	0	1	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	0	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1
A⊕B⊕C	1	0	1	0	0	0	1	0	0	1	0	1	1	1	1	0	1	1	1	0	1	1	1	1	1	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	1	1	0	1	1

FIGURE 2.4: Example of a binding operation between three hypervectors.

Again, we can calculate the Hamming distances for each of these:

$$J = A \oplus B \oplus C \tag{2.5}$$

$$d(J, A) = 0.58 \tag{2.6}$$

$$d(J, B) = 0.58 \tag{2.7}$$

$$d(J, C) = 0.44 \tag{2.8}$$

As expected, the inputs are dissimilar to the output after binding. (Note that for high dimensionality the mean goes to 50%.)

Permutation

Next to bundling, binding, and the similarity measurement, the permutation operator is essential for encoding information as will be discussed more in the applications (see 2.2.1 for example). Permutations resemble vector multiplication as a mapping operation because they are invertible and their result is *dissimilar* to the input vector. A permutation can also just be thought of as a multiplication of the vector with a special permutation matrix. More simply put, it is an operation that shuffles the components of the vector, creating a new one, dissimilar to the original. An example of a permutation by 1 and 2 can be seen in Figure 2.5. The permutation operator is denoted by ρ here. In hardware, the permutation operation comes down to a simple circular shift.

A	0	1	1	1	1	1	0	1	1	1	1	1	0	1	0	1	0	1	1	1	1	0	1	0	0	1	1	1	1	0	0	1	1	1	1	0	0	1	0	1	0	1	0	0	0	1	0	0	0	1
ρA	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1	0	1	0	1	1	1	1	0	1	0	0	1	1	1	1	0	0	1	1	1	1	0	0	1	0	1	0	1	0	0	0	1	0	0	0
ρρA	0	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1	0	1	0	1	1	1	1	0	1	0	0	1	1	1	1	0	0	1	1	1	1	0	0	1	0	1	0	1	0	0	0	1	0	0

Figure 2.5: Example of a hypervector permutation operation.

After calculating the Hamming distances, the dissimilarity is shown:

$$d(\rho A, A) = 0.52 \tag{2.9}$$

$$d(\rho\rho A, A) = 0.4 \tag{2.10}$$

2.2 Character Recognition

One of the main uses of hyperdimensional computing is classification tasks; many of these have been worked out in literature already.[21, 29, 12, 28, 11, 20, 15] These algorithms are all based on a basic architecture, shown in Figure 2.6. This architecture exists of an item memory that stores a bunch of randomly generated hypervectors, to be used during the encoding step. The encoder assigns these hypervectors to certain features or data and applies arithmetic operations to transform the entire input into one single hypervector.

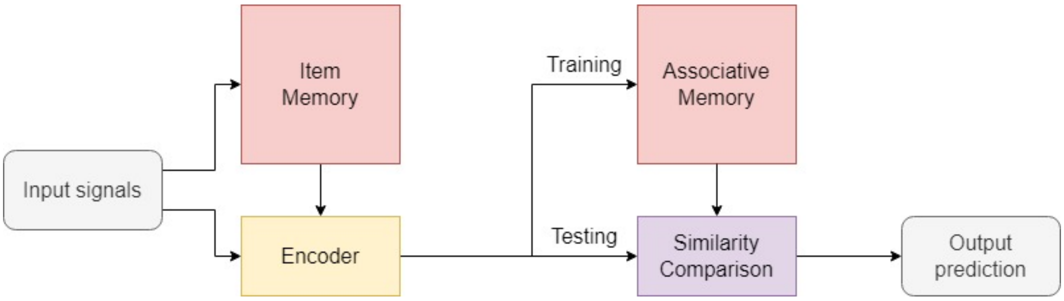


Figure 2.6: Basic HDC architecture.

During training, these hypervectors are linked to a finite number of different inputs, and stored in the associative memory. The associative memory thus serves as a collection of the different encoded inputs. Afterward, in the testing step, a new input is given, encoded, and then compared against all of the hypervectors stored during training. Through a similarity comparison, the closest match is found and then outputted as the answer to the classification task. This architecture description serves as a basis for nearly all DDB-HDC classification algorithms, including all applications referenced above. The next sections will describe each module in depth, and demonstrate how they work by guiding the reader through the example application of character recognition.

2.2.1 Encoding Module

The encoding module encompasses everything needed to encode the raw inputs into a single *object hypervector* that represents a "full" input or object. It consists of an input mapper, to map the inputs to the functions that manipulate them in the encoder. The inputs are always mapped to an item memory, which provides the starting hypervectors for transformation during encoding. Then the actual encoding of the HVs happens through the arithmetic operations described in section 2.1.2. Finally, a single encoded HV is the output that serves as a representation of the applied input.

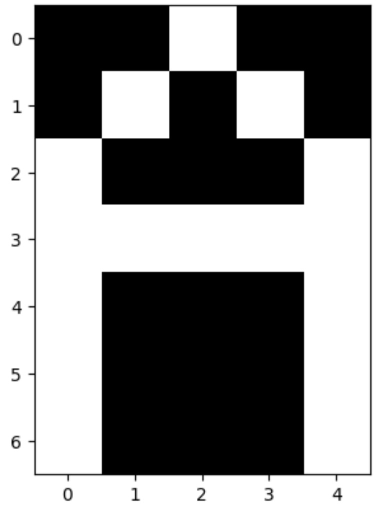
Character Input

To illustrate how the DDB-HDC framework and aforementioned architecture can be used in a real-world application, it can be applied to a simple character recognition task.[15] The input for this task consists of a 5x7 pixel grid, where each pixel is either 0 (black) or 1 (white). All 26 letters of the alphabet can be represented this way and converted to a simple string of 35 bits by sequentially looping over the pixels from left to right and top to bottom. As an example, the letter A and the binary string it converts to are shown in Figure 2.7.

Item Memory

The item memory (IM) acts as a storage place for the hypervectors required during the encoding step. Here, each HV becomes meaningful by representing some data, entity, or feature.[13] In the case of this character recognition algorithm, the item memory consists of 35 randomly generated hypervectors with a density of 50%. These are sequentially linked to each of the 35 pixels in the grid/positions in the string from Figure 2.7. This means that we can instantiate an item memory, where hypervector (HV) 0 will describe the pixel on the coordinate (0,0), HV 1 will describe coordinate (1,0), HV 2 coordinate (2,0), and so on.

These "random" binary hypervectors can be generated using a multitude of ways in both software and hardware. One of these is simply starting from a single random hypervector and permuting/shifting it.[15] As mentioned in section 2.1.2, permutation shuffles the input into a dissimilar hypervector. Another useful property is that the



[0 0 1 0 0 0 1 0 1 0 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1]

Figure 2.7: Character recognition pixel grid for the letter A.

operation comes down to a cyclic shift, so the density of the original and permuted vectors will be the same. However, this could lead to some inconsistencies in accuracy when also using permutation of the IM vectors during the encoding step, as is the case for character recognition.

A secondary method is generating them in software through "random" functions. A way to do this is by randomly permuting numbers counting from 0 to D, where D is the dimensionality of the wanted hypervector. Then, the bits on all indices where the number is $>= M$ are set to 0 and all other bits are set to 1. M is a definable threshold that represents the number of ones in the hypervector. In the case of DDB-HDC, we take $M = D/2$ as we want a density of 50%. The code for this algorithm can be found in appendix A.1.1.

Besides fully generating the item memory in software and then hardcoding it in hardware or loading it in, there are also other hardware implementations possible. A number of these will be described in chapter 4.

Encoder

The methods of encoding for DDB-HDC algorithms are heavily application-specific, but all use the arithmetic operations defined in section 2.1.2. All of the inputs follow the same procedure, which can be seen in Figure 2.8. For character recognition, the hypervector corresponding to the current "input pixel" is loaded from the item memory. The color of the pixel then decides the next step. When the input is a 1–the pixel is white–the loaded HV is permuted by 1 bit to the right, while for dark pixels no permutation happens. This way each pixel has a "unique" hypervector corresponding with it, based on whether it is black or white.

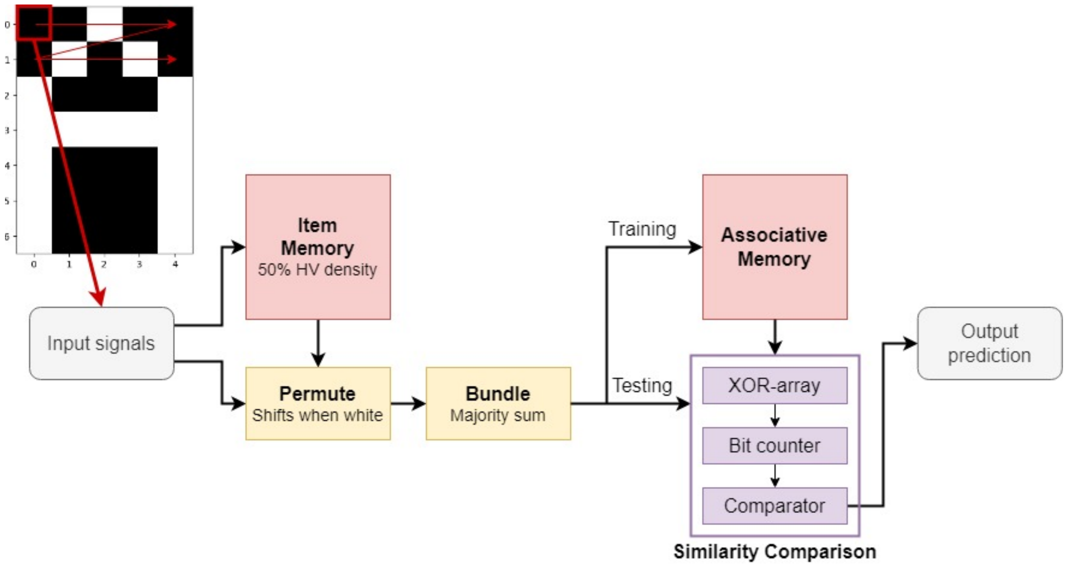


FIGURE 2.8: Data flow for the DDB character recognition application.

After the permutation step, all 35 hypervectors are bundled together using a majority sum. That is, each hypervector is added to a sum, and the majority threshold is taken after all 35 HVs are summed. This then produces a single encoded hypervector, representing the entire character.[15] In general, the encoding step always aims to encode the entire input (in this case the 35 pixels) into a single vector.

2.2.2 Associative Memory Module

The associative memory module consists of the associative memory itself, and all of the logic required to do the similarity comparison and determine the best matching vector. This module takes the encoded hypervector as input and converts it to the predicted output during testing.

Associative Memory

The Associative Memory (AM) stores the encoded hypervectors; it stores the hypervectors representing a full input or an object. For every possible output, the algorithm should be able to predict, the corresponding input will be encoded into a representing object HV and stored in the AM during training. After the training phase, the AM is a database of encoded inputs that serve as a comparison for the new inputs applied during testing. In the testing phase, unseen and potentially slightly distorted inputs are encoded in the exact same way as the training inputs. These are then compared to those stored in the AM to find the best matching output.

In this example of character recognition, the training data consists of the 26 letters of the alphabet, so the AM stores an object HV for each of these. These are loaded into the AM during the training phase. Afterward, each testing input is again converted to an object HV and compared to see which letter it has the most similarity to. This then forms the predicted output of the algorithm.

Similarity Comparison

As mentioned above, inputs have to be compared against the HVs already present in the AM to determine the output prediction. This comparison is done by the similarity comparison module. The module computes the similarity between the object HV and each of the HVs stored in the AM. This is done by computing the Hamming distance to all stored HVs (as described in section 2.1.2).[15] The letter corresponding to the HV in the AM with the lowest Hamming distance (highest similarity) to the object HV forms the predicted output.

2.2.3 Performance Evaluation

To test the accuracy in various circumstances, distortions can be introduced in the character images. A "distortion" in this case is the flipping of a pixel from black to white or vice versa[15]. So 2 distortions mean that 2 random pixels out of the 35 have their color flipped. This distorted pixel grid of a certain character then serves as input for the algorithm. In Figure 2.9, "Holographic GN" shows the average accuracy vs. the number of distortions for the DDB-HDC character recognition algorithm with an HV dimensionality $D = 10,000$. For each of the 26 letters, the average accuracy over 1000 repetitions with N random distorted bits is taken. Do keep in mind that these accuracies can vary slightly (±1%) from run to run because of the random placement of the distortions.

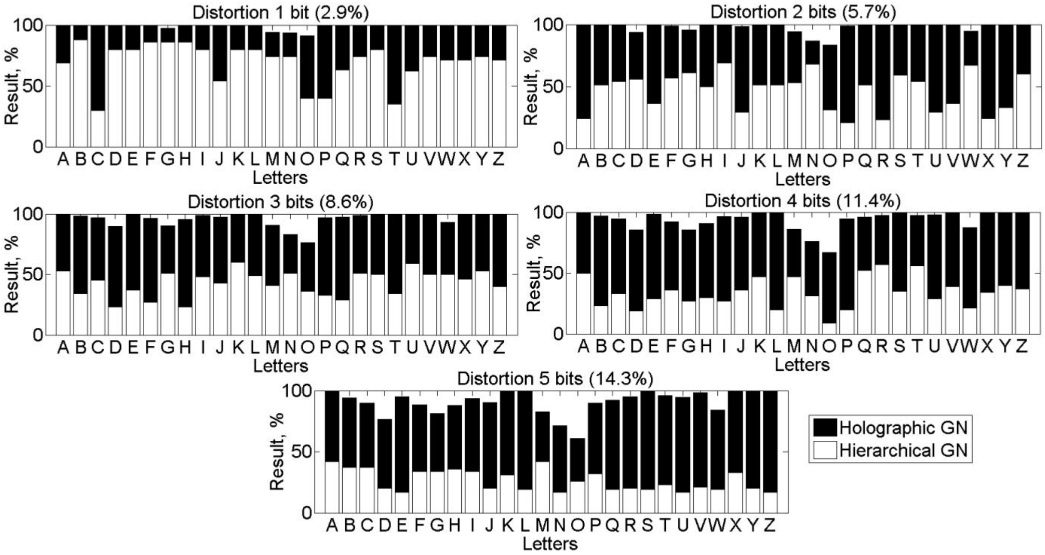


Figure 2.9: DDB-HDC character recognition accuracy.[15]

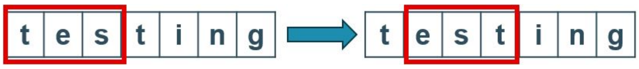


FIGURE 2.10: Example of a sliding trigram window.

2.3 Language Recognition

This section will explain the working of a more complex DDB-HDC algorithm, that performs language recognition as described in [29]. The full architecture can be seen in Figure 2.11. This algorithm takes a stream of letters from a text in a certain language as input and produces a predicted language as output. It does this by encoding the entire text into a single HV that represents the text. The closest match to the precomputed "language hypervectors" stored in the associative memory is then found.[29] The amount of languages the algorithm can differentiate between depends on which ones are stored in its associative memory.

Once again, it starts with an item memory: each letter is assigned a different random hypervector contained in the IM. As this is a DDB representation, the vectors in the IM have a density of 50%. There are 27 quasi-orthogonal HVs stored in the IM: one for each letter of the alphabet and one to represent the "space" character. The encoder works based on a sliding window; a window of size N slides over the input text and gives N letters at a time as input to the encoder. Figure 2.10 depicts how a trigram window (size 3) would slide over the text for each new input. As the window slides by one letter for each input, a new HV is loaded each time, while the other N-1 HVs in the window all undergo a cyclic shift to the right by 1 position. After the shifting, all N vectors making up the sliding window are XOR'ed together into a single N-gram hypervector. Consider a trigram with the letters "ABC" as input: A is permuted twice ($\rho\rho A$), B is permuted once (ρB) and C undergoes no permutation. The resulting trigram hypervector then equals to $\rho\rho A * \rho B * C$, where the multiplication can be implemented by an XOR array in binary.[29]

This N-gram hypervector is fed into an accumulation stage, where each bit is added to the total accumulated value. After the window has passed over the entire input text, a thresholding (THR) operation is applied to the accumulated values. In DDB, the threshold is set at $\frac{1}{2}$ of the amount of accumulated inputs (the entire operation comes down to a majority sum). Similar to character recognition, the encoding module is used for both training and testing.[29]

After encoding, the associative memory module ("search module" in the figure) performs a similarity comparison based on the HVs stored in its memory. The similarity is determined here by a *cosine similarity* metric. "It measures distance$_{\mathrm{cos}}$ between a language hypervector (LV$_{\mathrm{i}}$) and an unknown query hypervector (QV) as follows:

$$distance_{cos} = \frac{LV \cdot QV}{|LV_i||QV|} \tag{2.11}$$

where LV$_{\mathrm{i}}$ · QV is the dot product between the two hypervectors, |LV$_{\mathrm{i}}$| and |QV| are the magnitudes of LV$_{\mathrm{i}}$ and QV, respectively."[29] A distance$_{\mathrm{cos}}$ close to 1 means a high similarity and likelihood that the text is written in the same language, while

Table 2.1: DDB-HDC language recognition results from [29].

	Accuracy		Memory (Kb)	
Window size	HD	Baseline	HD	Baseline
Bigrams (N=2)	93.2%	90.9%	670	39
Trigrams (N=3)	96.7%	97.9%	680	532
Tetragrams (N=4)	97.1%	99.2%	690	13837
Pentagrams (N=5)	95.0%	99.8%	700	373092

one close to 0 means the vectors are dissimilar. The similarity is measured for all languages stored in the AM, and the highest similarity then corresponds to the predicted language.

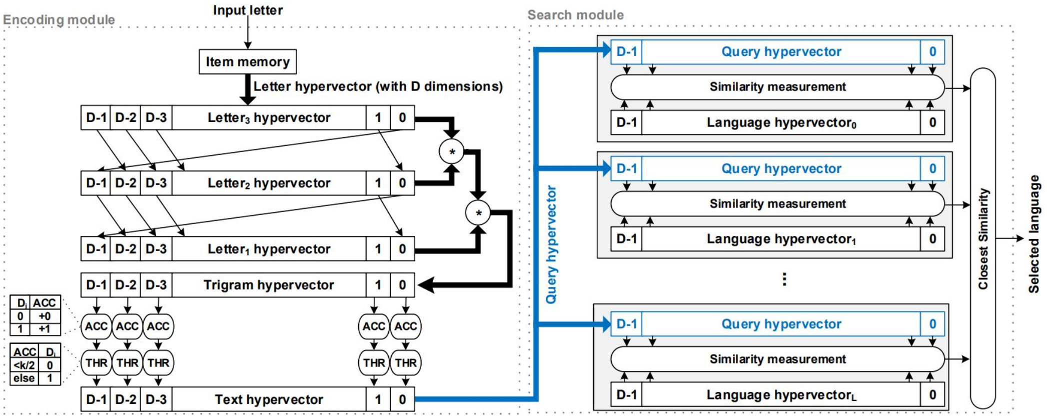


Figure 2.11: Architecture for DDB-HDC language recognition. (retrieved from [29])

In [29], 21 European languages are considered, thus 21 *language hypervectors* are stored in the AM after training. For training about a million bytes of text for each language are used, taken from the Wortschatz Corpora. [24] The test set consists of 1,000 samples per language, each consisting of a single sentence, retrieved from the Europarl Parallel Corpus. [18] The accuracy of the algorithm is then determined by checking the percentage of correct predictions over these 1,000 samples. The total average accuracy can be calculated by averaging these percentages over all 21 languages and is shown in Table 2.1 for HVs with dimensionality $D = 10,000$. "Baseline" refers to: "a nearest neighbor classifier that uses histograms of N-grams."[29] The histograms have integer components, which have been converted to binary using their mean value as a threshold to reduce the memory footprint.[29] Table 2.1 shows that there is a slight accuracy drop of <3% compared to the baseline, but an immense decrease in required memory size.

2.4 Sparse HDC

Sparse distributed binary hyperdimensional computing (SDB-HDC) is a modification of the DDB-HDC space, utilizing sparse instead of dense binary HVs. The exact density of the hypervectors is dependent on the algorithm/application, but the HVs in the item memory usually contain around 1-10% ones.[25] SDB-HDC uses the same four operations as DDB, which are described in section 2.1.2. However, because these hypervectors are sparse, their properties are slightly different. On top of that, some level of sparsity must be retained throughout the encoding to get good accuracy.[17, 10] This means the operations should be modified to work with sparse HVs and control the sparsity. There are two main encoding schemes for SDB-HDC: context-dependent thinning and thresholded sum.[30]

2.4.1 Properties

Randomness

Like their dense counterpart, SDB hypervectors are "random"; the 1 bits they contain should be in random positions. Because, of their large size ($>\sim$1,000 dimensions) they will still be able to encode a lot of information (albeit less than dense HVs). Instead of the encoded information being solely dependent on the pattern, however, SDB encodings are more locality-based. This means that the exact positions of the ones are more important.

Robustness

Intuitively, a lower number of ones in the HVs means that less information can be encoded in them. This, together with the fact that SDB is more locality-based removes some of the redundancy and robustness. For most applications this isn't a problem though, as due to the large dimensionality of HVs, SDB HVs can still hold enough information to provide similar accuracy to DDB-HDC. There is no, or only a marginal application-specific accuracy loss, often around 0-3%.[25, 30, 19, 14, 10]

Orthogonality

SDB hypervectors have the same quasi-orthogonality properties as DDB. As an effect of their changed structure, similarity comparison by Hamming distance is no longer optimal. Instead, the similarity is measured by the amount of overlap between the two HVs. This comes down to a simple AND operation (Instead of XOR), followed by a count of the ones. A greater overlap of ones between two HVs means there is a greater similarity / better match between them.[30]

Capacity

"For n dimensions and density d (the rate of ones in the vector), the capacity is $\binom{n}{\lfloor d \cdot n \rfloor}$. Even if there are only 5% non-zero entries, a 1000-dimensional vector can store more patterns than the supposed number of atoms in the universe (presumably about 10^{80})."[21] This means that the capacity of hypervectors is so great that even sparse HVs can store more than enough information.

2.4.2 Context-Dependent Thinning

The context-dependent thinning (CDT) operation acts like a hash function of Z that performs binding of $\mathrm{x_i}$ while controlling the sparsity and preserving similarity.[17] In short, it is a method to control the sparsity after binding multiple vectors together. The CDT-based encoding scheme uses a non-orthogonal binding, through disjunction followed by a context-dependent thinning operation. It uses overlap for similarity measurement, disjunction for bundling, and CDT for binding.[30]

Operations

The use of an additive operator (disjunction) greatly increases the density of the encoded hypervector. A disjunction (bitwise OR) keeps all the 1's of its input HVs in its output, so because of the randomness of their locations, the density will increase with the number of disjuncted HVs. A method to reduce this again is through applying a "context-dependent thinning" step, proposed by Rachkovskij in [25] This comes down to the following procedure, where X_i are the hypervectors to be bound, $\vee$ represents disjunction (OR) and $\wedge$ represents conjunction (AND):

$$Z = \bigvee_{i} X_i \tag{2.12}$$

CDT thins vector Z as follows:

$$\langle Z \rangle = \bigvee_{k=1}^{K} (Z \wedge Z\tilde{}(k)) = Z \wedge \vee_{k=1}^{K} Z\tilde{}(k) \tag{2.13}$$

$\langle Z \rangle$ is the thinned vector, while $\langle Z\tilde{}(k) \rangle$ represents the k-th permutation of Z (k cyclic shifts). The density after superposition is directly linked to the number of vectors disjuncted, thus the same goes for the density after CDT. The value of K in the operation can control the density of the thinned vector. The lowest density is achieved when K=1, and the greater K, the higher the resulting density. It can be convenient to let the resulting density equal the density of the components. [14, 25, 27] This means that to keep the density low after this operation, the number of superposed vectors should remain low.

Applied to Character Recognition

To show how the SDB variant based on context-dependent thinning can be used in practice, it is applied to a sparse variant of the character recognition example from section 2.2, according to [14]. Firstly, the 35 HVs stored in the item memory are replaced by randomly generated sparse HVs with a density of ∼1%. The encoding through permutation based on the pixel color remains the same (i.e., white pixels

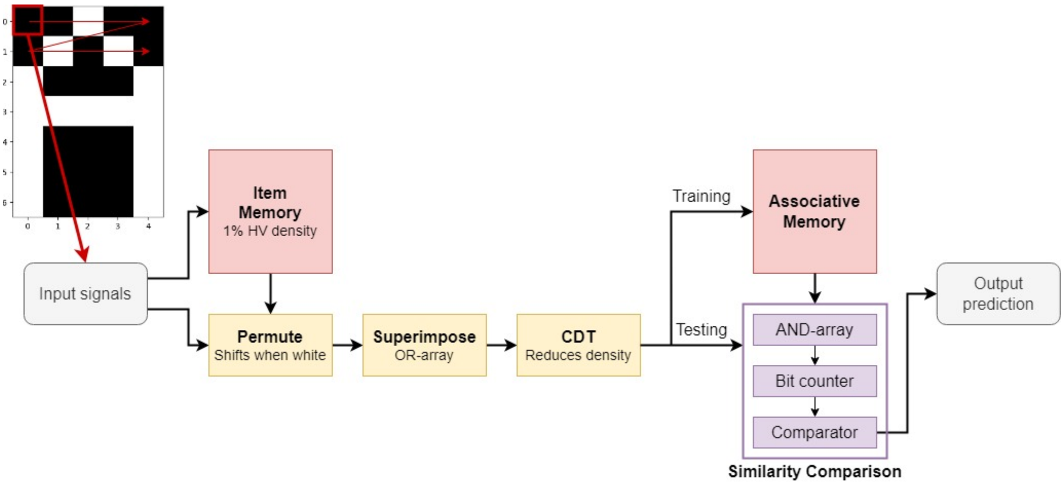


FIGURE 2.12: Data flow for the SDB character recognition application.

are permuted once). The next step where the vectors are added and a majority sum is taken, is replaced by the disjunction + CDT procedure as shown in Figure 2.12.

This means that instead of each bit being summed up over the 35 input HVs, a disjunction between the input HV and the resulting superposed HV is taken. The disjunction operation increases in density with respect to the number of input HVs.[14] This means that the density is too high after the 35 inputs, and we want to bring it back down for better classification. Thinning the hypervectors before classification improves the accuracy (given that the density is high enough before CDT). The CDT operation is a perfect fit for this, as it retains the similarity and we can control the resulting density using its K-factor.

After the thinning, the encoded "object HV" undergoes the same steps through the associative memory module. The only difference here is that the similarity is computed by measuring the overlap (using an AND operation) instead of through an XOR operation. Finally, the letter in the associative memory corresponding to the highest overlap is given as the predicted output.

Figure 2.13 shows the average accuracy per letter for N distortions and different dimensionalities. The accuracy is calculated in the exact same way as the DDB character recognition experiment above (see 2.2.3). Do keep in mind that these accuracies can vary slightly (±1%) from run to run because of the random placement of the distortions. As can be seen, the accuracies are very similar to those for the DDB version and there is no significant loss in performance. The higher dimensionalities only give a very small accuracy improvement for the higher distortion counts.

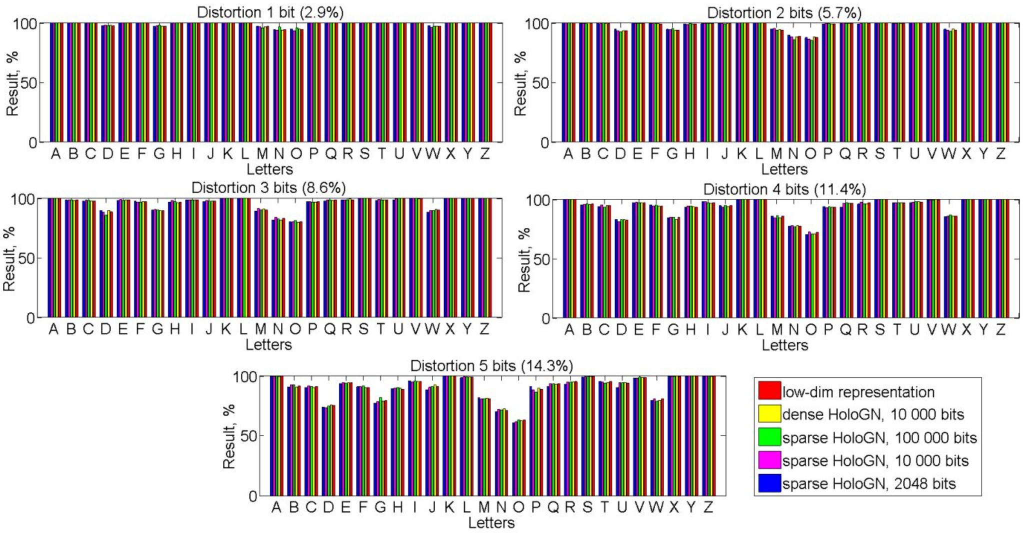


Figure 2.13: SDB-HDC character recognition accuracy.[14]

2.4.3 Thresholded Sum

Operations

A second SDB-HDC encoding scheme uses accumulation and thresholding to perform its bundling operation. This functions like the majority sum described in section 2.1.2 but with a controllable threshold instead of 50%. It uses shifting for binding and overlap for similarity measurement.[30] This scheme is based on the multiplication, addition, and permutation (MAP) coding from [7]. Its main advantage lies in the fact that the density of the resulting vectors can easily be controlled by changing the threshold. The bundling comes down to a bitwise accumulation, followed by a thresholding step (like the majority sum for DDB-HDC).[10] This concept will be demonstrated by applying it to the *language recognition* algorithm described in section 2.3.

Applied to Language Recognition

The item memory for SDB language recognition remains similar to the DDB version, where a random sparse orthogonal hypervector is assigned to each letter and the "space" character (27 total). During encoding, the DDB language recognition algorithm uses XOR operations to bind the hypervectors in the N-gram. When this is applied to sparse HVs however, it does not bind them effectively. The XOR operation used in the DDB-HDC language recognition encoding has a behavior similar to that of an OR operation for sparse HVs, so its result is similar to its constituents. For SDB language recognition, a binding using XOR would produce N-grams that are too similar to differentiate properly, because languages have a bias toward similar N-letter permutations. Additionally, the N-gram hypervectors would not be sparse when pushing toward the desired accuracy under this encoding scheme.[10]

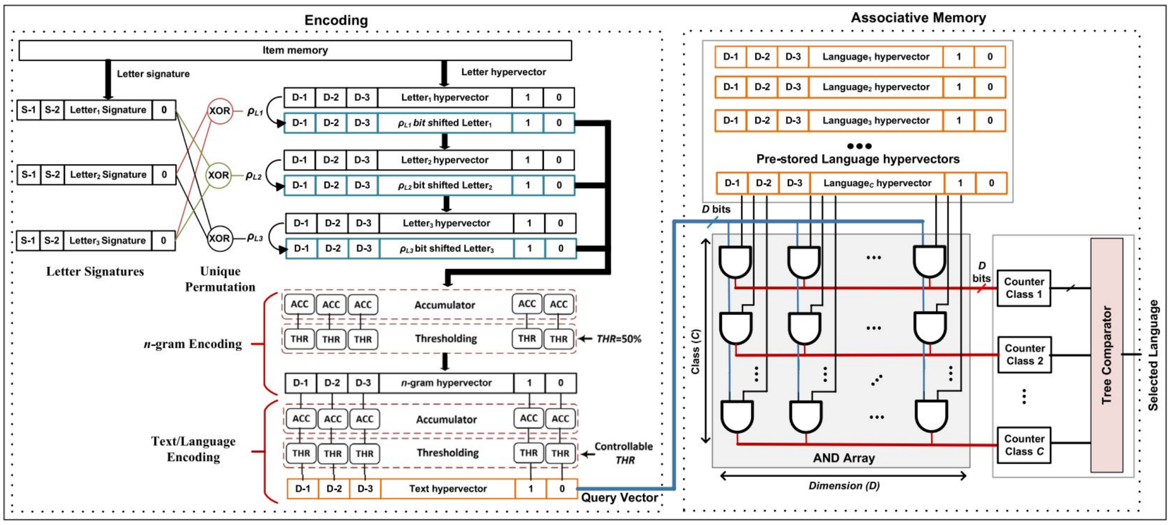

Encoding
Associative Memory
Item memory
Letter signature
Letter hypervector
S-1 S-2 Letter₁ Signature 0
S-1 S-2 Letter₂ Signature 0
S-1 S-2 Letter₃ Signature 0
XOR
ρL1
ρL2
ρL3
D-1 D-2 D-3 Letter₁ hypervector 1 0
D-1 D-2 D-3 ρL1 bit shifted Letter₁ 1 0
D-1 D-2 D-3 Letter₂ hypervector 1 0
D-1 D-2 D-3 ρL2 bit shifted Letter₂ 1 0
D-1 D-2 D-3 Letter₃ hypervector 1 0
D-1 D-2 D-3 ρL3 bit shifted Letter₃ 1 0
Letter Signatures
Unique Permutation
n-gram Encoding
ACC Accumulator
THR Thresholding
THR=50%
D-1 D-2 D-3 n-gram hypervector 1 0
Text/Language Encoding
Controllable THR
D-1 D-2 D-3 Text hypervector 1 0
Query Vector
D-1 D-2 D-3 Language₁ hypervector 1 0
D-1 D-2 D-3 Language₂ hypervector 1 0
D-1 D-2 D-3 Language₃ hypervector 1 0
Pre-stored Language hypervectors
D-1 D-2 D-3 Language_C hypervector 1 0
D bits
Class (C)
AND Array
Dimension (D)
Counter Class 1
Counter Class 2
Counter Class C
Tree Comparator
Selected Language

FIGURE 2.14: Architecture for SDB-HDC language recognition. (retrieved from [10])

An alternative approach to binding sparse HVs is to apply a unique shift to each letter in the sliding window. A random m-bit signature S is assigned to each sparse letter hypervector, with the goal of creating a unique permutation for each of them. The XOR of the m-signature bits of the other letters in the N-gram sliding window will determine the amount that each HV is (circularly) shifted during the binding step.[10] This thus applies a unique shift to each letter hypervector (leading to better results when compared to just shifting relative to the letter's position in the window), which can be calculated in the following way:

$$\rho_K = K - 1 + (S_N \oplus S_{N-1}... \oplus S_{K+1} \oplus S_{K-1}... \oplus S_1) \tag{2.14}$$

Where ρ_K is the number of shifts applied to the K$^{\text{th}}$ vector in the N-gram (with N $\geq$ K).[10] The permutation applied to the letter HVs is thus not solely dependent on position in the N-gram (like for the DDB version), but also on the random signatures. These unique m-bit signatures should have a size of a maximum of 5 bits, as this gives $2^5 = 32$ different values, which is enough to represent each letter in the EU languages.

After the permutations, the letter HVs are not XOR'ed together, instead, they are summed and thresholded. Bitwise accumulation over the N vectors takes place, followed by thresholding with THR=50% (= majority sum). This means that all positions with an accumulated value $\geq$ 0.5*N (the cutoff value) are represented by "1", and all others by "0". Mathematically this can be written as:

$$Z_{M_i} = [\rho_N(L_N) + \rho_N - 1(L_{N-1}) + ... + \rho_1(L_1)]|_{THR=50\%} \tag{2.15}$$

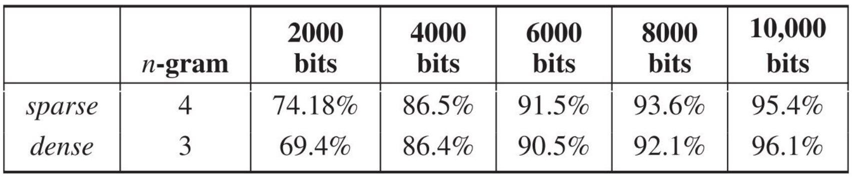
where $(L_1, L_2, L_3, ..., L_N)$ are the sparse HVs representing the letters and $[+]\ |_{\text{THR}}$ is the thresholding function, with THR determining the cutoff value.

This is followed by another accumulation + thresholding step, where the N-gram HVs are accumulated and thresholded to form the text/language hypervector. Mathematically it comes down to:

$$T_{M_C} = [Z_{M_1} + Z_{M_2} + ... + Z_{M_H}]|_{THR} \tag{2.16}$$

where T_M is the fully encoded text hypervector. The threshholding value for this step is controllable, to be able to thin the encoded HV as necessary and control the sparsity.[10] It can be tweaked during testing for optimal accuracy or energy efficiency.

During testing, the text HV is then again compared to the language HVs stored in the associative memory by a similarity comparison. Like SDB-HDC character recognition, the similarity is measured through overlap using an AND array instead of through Hamming distance or cosine similarity, as the overlap is more suited for sparse HV. The language HV with the highest overlap then determines the predicted output language. The training and testing happen in the exact same manner and on the exact same datasets as those for DDB language recognition mentioned in section 2.3.[10]



	n-gram	2000 bits	4000 bits	6000 bits	8000 bits	10,000 bits
sparse	4	74.18%	86.5%	91.5%	93.6%	95.4%
dense	3	69.4%	86.4%	90.5%	92.1%	96.1%

Figure 2.15: Classification accuracy for the sparse and dense HDC designs for different dimensionalities D. (retrieved from [10])

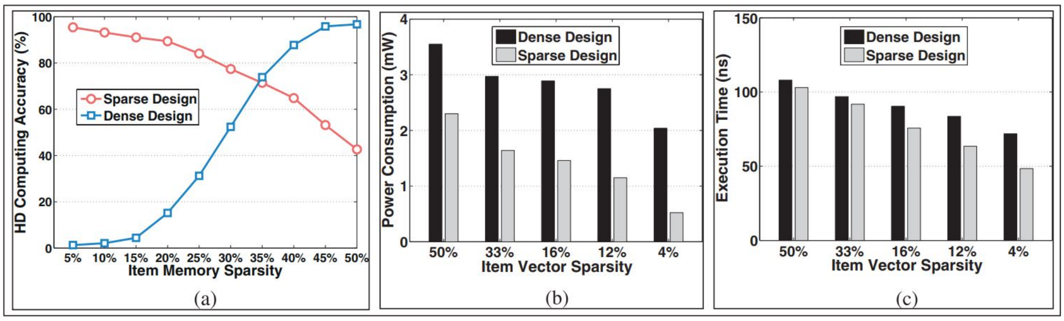
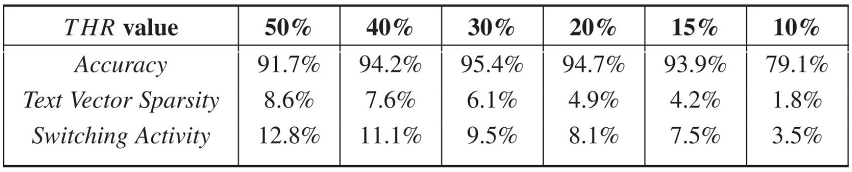


Figure 2.16: The accuracy, power consumption, and execution time of the sparse and dense HDC designs for different item memory HV sparsities. (a) Accuracy. (b) Energy consumption. (c) Execution time. (retrieved from [10])

Figure 2.15 shows the accuracy compared to the aforementioned DDB-HDC design for different HV dimensionalities. In Figure 2.16 the real benefits of using a sparse design can be seen: the power consumption is drastically reduced compared to its



THR value	50%	40%	30%	20%	15%	10%
Accuracy	91.7%	94.2%	95.4%	94.7%	93.9%	79.1%
Text Vector Sparsity	8.6%	7.6%	6.1%	4.9%	4.2%	1.8%
Switching Activity	12.8%	11.1%	9.5%	8.1%	7.5%	3.5%

FIGURE 2.17: Impact of the THR value on the classification accuracy, sparsity, and switching activity. (retrieved from [10])

dense counterpart. Note that in the power consumption graph, parameters like sliding window size and threshold value are chosen for maximal energy improvement.[10] Figure 2.17 shows how controlling the THR value changes the performance metrics of the system. When comparing the SDB and DDB implementation for this application, about 1-2% accuracy is lost, but traded for a large decrease in power consumption. We can conclude that accumulation & thresholding is more suited for combining a high number of vectors than disjunction & CDT, because of its controllable threshold value. The method of encoding that gives optimal results is heavily application-dependent, which is one of the drawbacks of HDC in general.

2.5 Hardware Implementations

Unlike for SDB-HDC, programmable DDB-HDC accelerators do exist and their structure can be used as a basis for a programmable SDB accelerator. They closely follow the main structure of item memory, encoder, and associative memory with similarity computation described in section 2.2. The encoder implementation is where the main differences lie.

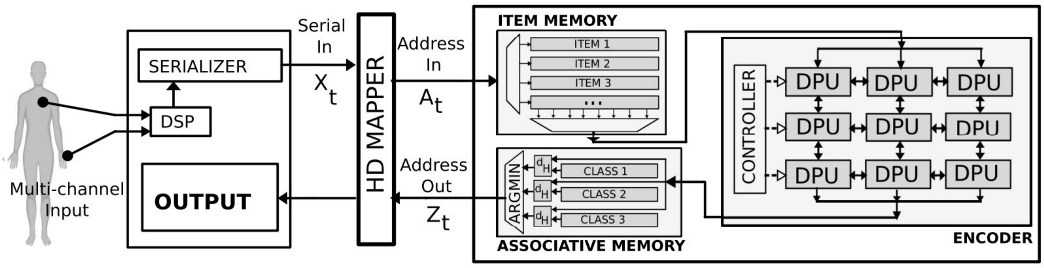


FIGURE 2.18: Generic HD Processor architecture. (retrieved from [5])

For example, take the architecture from [5], which can be seen in Figure 2.18. The main blocks from the previously mentioned general HDC architecture are visible, with the similarity comparison being integrated into the associative memory module. A dataflow architecture is the most suitable candidate for processing HDC computations, as all data follows a similar flow and the only changes to this are given concurrent with the input.

As DDB-HDC doesn't have different encoding schemes like SDB, a Data Processing Unit (DPU)-based structure can be used here. This DPU block then only needs to implement the multiply and permute functions. Figure 2.19 shows the basic Hyper-Dimensional Logic Unit (HLU) building block, and how they can be combined. Each DPU block is either a HLU layer or an accumulator, and each DPU's output can serve as the next DPU's input. The basic 1-bit HLU block can be combined in a layer and programmed to have the following functions: multiply, permute, delay, or permute-and-multiply. These layers can then be stacked to perform the permuting and binding before an accumulator block performs bundling. For example, 3-grams would require at least 3 HLU layers for storing the 3 operand hypervectors, so the n-gram determines the encoder depth. An example setup could have 3 HLU layers followed by an accumulator, followed by 5 more HLU layers, and lastly a second accumulator. This setup would be able to compute with an n-gram where $n \leq 3$.[5]

The other logic besides the encoder doesn't have special implementations in this architecture. The item memory for this encoder is implemented using a read-only memory (ROM), filled through offline hypervector generation. Meanwhile, the associative memory is a simple register bank containing the HVs encoded during training. HDC algorithms increase in accuracy as their dimensionality increases but so do the area and power usage. This is why the implementation only has a dimensionality D = 2,048.

An important note is about how a larger dimensionality than 2,048 could be handled for higher accuracy. Both the item and associative memory must be full-width to avoid expensive accesses to off-chip memory, so their size must increase with increasing dimensionality. This only leaves a reduction of the encoder's size as a possibility, using sub-word encoders. Sub-word encoding means that the full HV is split into multiple sub-words that are passed through the (smaller) encoder sequentially. However, sub-word encoders are not energy efficient for the following reasons[5]:

- Permutation requires redundancy, as it is reliant on bits from neighboring sub-words. E.g. an n-gram uses an extra $n(n-1)/2$ bits/cycle per sub-word.
- Sub-word encoding requires multiple passes through the dataset, increasing latency and requiring large memories to store this dataset. Meanwhile, a full-width encoder doesn't need to store the dataset as a single pass is enough.
- The memories (IM and AM) remain the same (large) size, including the bit counter to measure the similarity score. This bit couter determines the critical path, so the clock frequency remains the same while the amount of encoding cycles increases. The result is an increase in both latency and the total energy/prediction.

A better alternative would be a multi-processor configuration, where e.g. 3 processors can be combined to increase the effective dimensionality to D = 6,144. Each processor should have a different item memory map, where the stored HVs are unique and different from those in the other processors. The mapping of the labels

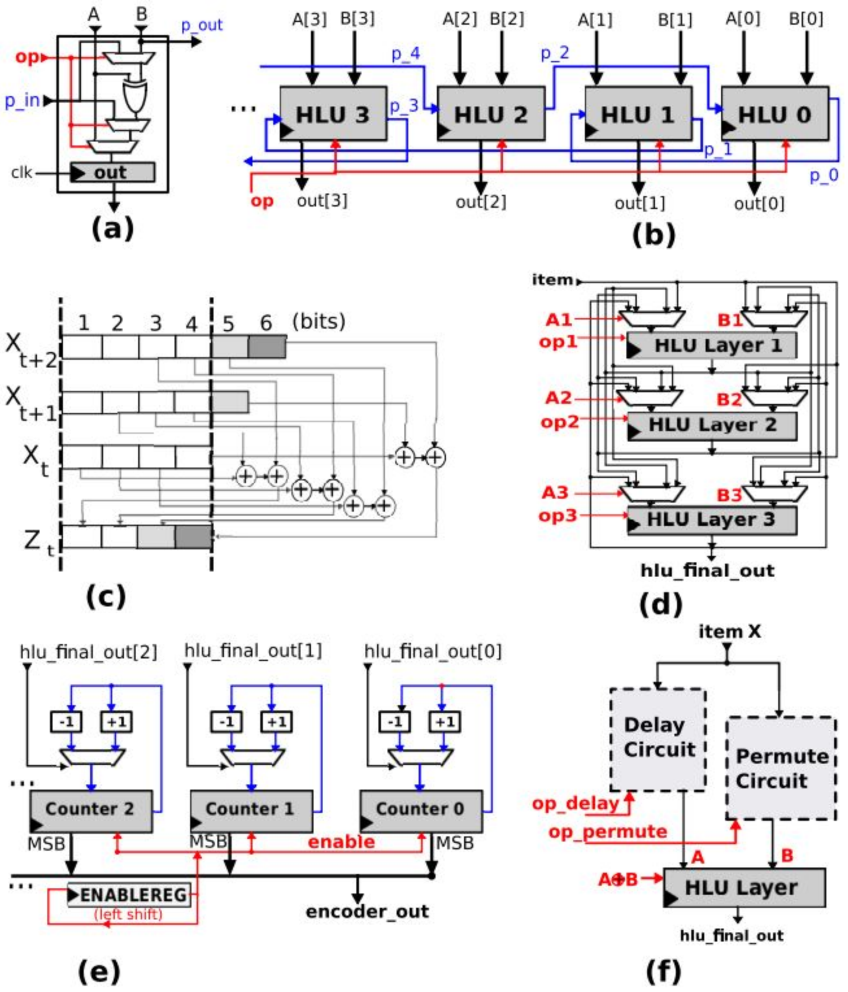


FIGURE 2.19: (a) 1 HLU unit, (b) connecting HLUs to form a layer, (c) encoding steps required for a 3-gram, (d) connecting multiple HLU layers to generate terms, (e) accumulator for superposition, (f) for the generic model, the delay and permute partial terms can be separately computed and combined. (retrieved from [5])

to the AM addresses must be the same for all processors, however. The input is then given to all three processors in parallel, and they each compute an encoded HV and a corresponding distance to the trained HV for each AM address. The distances from all three processors can then be added to compute the total distance, which provides a more accurate metric for the similarity to that AM label. For example, a Serial Peripheral Interface (SPI) module could be used to transmit this distance data.

An alternative implementation for the item memory is based on procedural HV generation through a mixing stage. Figure 2.20 shows an implementation from [6] to generate quasi-orthogonal pseudo-random HVs on the fly. This is an area-efficient method based on the iterative application of hardwired permutations. Such an implementation works well in scenarios where the IM labels are iterated over chronologically, as is the case in applications using channels like EMG gesture recognition.[20] For these applications, the access time complexity is $\mathcal{O}(1)$, but for random access it is a lot worse at $\mathcal{O}(\log_2 |\mathbb{D}|)$, where $\mathbb{D}$ is the input domain.

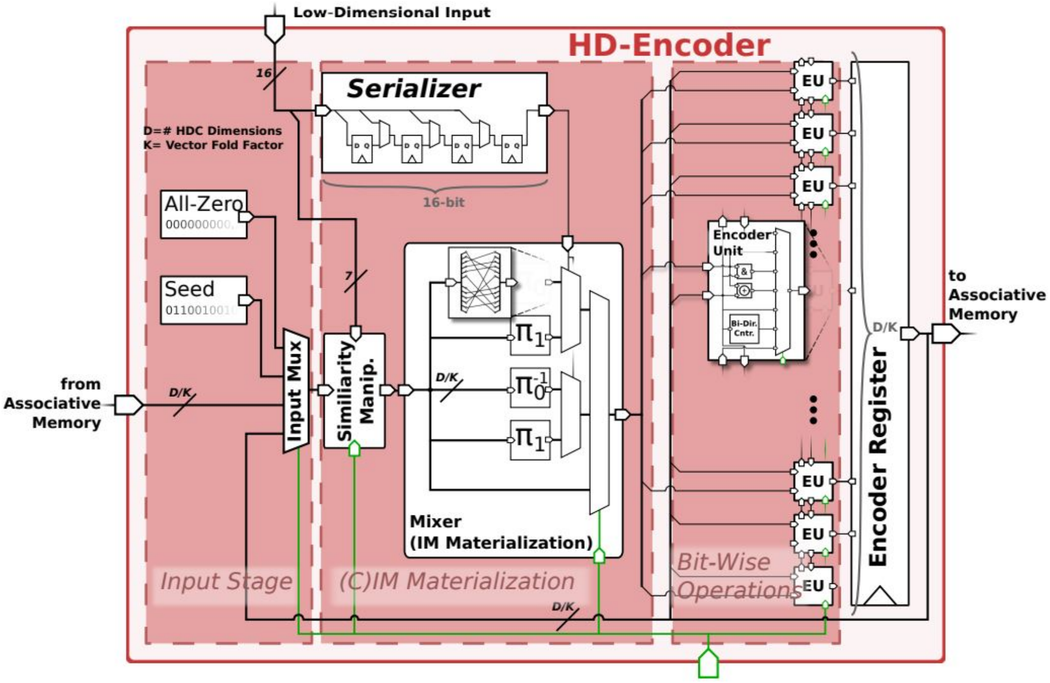


Figure 2.20: Programmable HDC encoder architecture responsible for item memory materialization. (retrieved from [6])

Instead of hardwiring the programming actions, this implementation makes use of a 26-bit instruction set architecture. They make the accelerator programmable using both No-instruction-set computing (NISC) and Complex-instruction-set computing (CISC) instructions. The NISC instructions are low-level and directly control the select signals of the multiplexers in the encoder. On the other hand, the CISC instructions have dedicated high-level multicycle instructions for multiple HDC transformations to reduce the algorithm's code size. This allows for a more "clean" dynamic control of the accelerator and its programming.

2.6 The Encoding Scheme Problem

As the number of IoT and AI applications keeps growing, so does the need for energy-efficient algorithms and hardware. Currently, lots of research has been done toward DDB-HDC algorithms and hardware for their extremely low energy usage. This has led to a slow increase of attention for the hyperdimensional computing space, and more recently for SDB-HDC as well. Yet, still very little research has been aimed at converting these DDB algorithms to an SDB-HDC representation for even higher power savings. A major downside of SDB-HDC is that multiple application-specific encoding schemes exist, and no generalized hardware has been designed for them yet. If it is not shown that SDB can get very similar accuracy to DDB and that these encoding schemes can be elegantly combined, this might slow down further research.

Currently, all papers aim at integrating SDB-HDC into one very specific application and use only one selected encoding scheme for their hardware application. To solve this, the goal of this thesis is to investigate opportunities for improvement in these encoding schemes and efficiently combine them into a single programmable accelerator. Therefore, these encoding schemes will have to be more generalized, combined together, and shown to achieve high accuracy. To achieve this, they should be integrated using as much hardware reuse as possible. This will show that SDB-HDC's low-power properties can be combined with efficient area usage, high accuracy, and low latency.

2.7 Summary

The basics and properties of both DDB- and SDB-HDC were outlined in this chapter, followed by the integration of their encoding schemes into the applications of character and language recognition. Although executing HDC algorithms is not very power- or time-efficient on regular CPUs, their operations can be massively parallelized on specific hardware.[21] This parallelization in combination with the cheap hardware operations makes HDC a great fit for low-power, low-latency IoT applications. SDB-HDC aims to remove some of the redundancy present in DDB-HDC and even further increase energy efficiency, so it is worth investigating further. The next chapter takes a deeper dive into the different SDB encoding schemes and tries to find opportunities for improvements.

Chapter 3

Implementing the Encoding Schemes

3.1 Defining the Objectives

Building a programmable hardware accelerator that efficiently combines the SDB-HDC encoding schemes, requires a deep understanding of how these algorithms function and what influences their accuracy. This means they should first be studied in software, before studying possible hardware implementations and creating the new design. Lastly, the design should be improved as much as possible and then compared against similar state-of-the-art accelerators. This is a list of all research objectives and sub-objectives for this thesis, which are worked out in the following sections:

1. Explore SDB-HDC algorithms in software
    a) Research DDB- and SDB-HDC algorithms and their different encoding schemes
    b) Implement Character Recognition
    c) Implement Language Recognition
2. Construct an efficient programmable SDB-HDC accelerator
    a) Research existing DDB-HDC (programmable) accelerators and hardware optimizations
    b) Implement the base programmable hardware accelerator
    c) Improve the hardware of the worst-performing modules
3. Profile and evaluate the energy efficiency, throughput, and area usage of the accelerator
    a) Analyze the performance before and after hardware improvements.
    b) Compare the performance against other DDB-HDC and SDB-HDC accelerators

3.2 Investigating the Encoding Schemes

To gain a thorough understanding of the impact using sparse hypervectors has on existing DDB-HDC algorithms, we implement some of them in software and try to optimize the accuracy. After studying various SOTA implementations, the most commonly used and generalizable schemes turned out to be context-dependent thinning (CDT) and thresholded sum. Simple algorithms that implement these are character recognition and language recognition, as described in section 2.4.2 and 2.4.3. By (re-)implementing these, we can learn about their functionality, observe accuracy changes based on parameter tweaks, and look for further accuracy improvements and optimizations.

3.2.1 Context-Dependent Thinning (CDT)

Additive operators like a sum or disjunction are often used for binding or bundling in SDB-HDC, but the density of the resulting HV is generally a lot higher than those of its constituents. As explained in detail in section 2.4.2, the context-dependent thinning procedure is a similarity-preserving method to control the sparsity after binding or bundling multiple HVs together. This makes it a perfect fit to reduce the density to an optimal level for maximum accuracy during classification in the associative memory module (see section 2.2.2). CDT is a well-studied procedure [26, 17] that is present in multiple applications.[14, 9, 16] The resulting density after encoding using this procedure is determined by two factors: the starting density of the IM HVs, and the K-factor of the CDT procedure as seen in its formula:

$$Z = \bigvee_{i} X_i \tag{3.1}$$

$$\langle Z \rangle = \bigvee_{k=1}^{K} (Z \wedge Z\tilde{}(k)) = Z \wedge \vee_{k=1}^{K} Z\tilde{}(k) \tag{3.2}$$

Here X_i are the input HVs, $\vee$ represents disjunction (OR) and $\wedge$ represents conjunction (AND). $\langle Z \rangle$ is the thinned vector, and $\langle Z\tilde{}(k) \rangle$ represents the k-th rotation of Z (k cyclic shifts). K=1 achieves the lowest density, as a higher K means more superposed vectors, increasing the density.

Character Recognition

To study the CDT encoding scheme, we implement the simple application of SDB-HDC character recognition from [14], as described in section 2.4.2. We recreate this in Python, starting with an HV density of ∼1% in the item memory (IM). 35 of these (rotated) HVs are then disjuncted into an encoded HV, which has a density of around 30%. To make maximal use of the possible sparsity and related energy savings, they undergo a context-dependent thinning with K=1. This brings the density of the final encoded HV down to around 9%. Table 3.1 shows the achieved results for D=1,024, and D=2,048, averaging out the accuracy over 100 repetitions of random distortions. Keep in mind that these can vary by ±0.8% per run, and that the variance is a lot higher for lower HV dimensionality (lower D). This is most likely due to the increased robustness and redundancy provided by a higher number of dimensions. We are most interested in a hypervector size up until around 2,048, as hardware designs using these would be a lot smaller and consume a lot less power than for D=10,000 while still achieving decent accuracy.[5] Additionally, research shows that the accuracy gain by going to higher dimensionality is marginal (∼0.5-3%) depending on the application.[14, 10, 11, 12]

TABLE 3.1: SDB-HDC character recognition results for 0.98% IM density.

Distortions	Accuracy (%)	
	D=1,024	D=2,048
0	100	100
1	98.62	99.11
2	96.58	97.69
3	94.15	95.96
4	89.54	92.38

In the paper about SDB language recognition, an item memory density of around 4% was found to be optimal.[10] That's why we opt to test this character recognition algorithm with CDT for higher IM densities as well. By increasing the density of the inputs for the disjunction and CDT, the output density is inadvertently also increased. The question is if the classification will still work well with highly dense encoded HVs.

Let's pick a density of ∼5% for the vectors stored in the IM. 35 of these (rotated) HVs are then disjuncted into an encoded HV, which has a density of around ∼82%. These are too dense for use in the associative memory and would lead to slightly worse prediction results (e.g. ∼97.19% acc for 2 distortions and D=1,024) in combination with high switching activity and energy usage. By applying the CDT operation once with K=1, it is then reduced to $67 \pm 3\%$. This is still quite high but achieves better accuracy and lower energy usage during the associative memory stage.

TABLE 3.2: SDB-HDC character recognition results for 4.88% IM density.

Distortions	Accuracy (%)	
	D=1,024	D=2,048
0	100	100
1	99.23	99.34
2	97.50	97.92
3	95.27	96.11
4	92.04	93.11

The results can be seen in Table 3.2; a slight accuracy increase is the result of more information being able to be encoded in the HVs due to their increased density.

Intuitively, the CDT encoding scheme is most useful when the number of superposed vectors is limited, to keep the density low enough, or when density control is wanted for a different reason. For character recognition, the accuracy starts to decrease when the encoded HV density exceeds 90%. E.g. for an encoded HV density of 97% and D=1,024, the accuracy is only around 93% for 2 distortions and inconsistent from run to run. A way to seemingly get around the density restriction would be by applying multiple CDT operations in series. However, because CDT "prunes" the HVs to reduce the density, this turns out to lose too much information and similarity, leading to poor accuracy results. In conclusion, disjunction with CDT is usable so long as the resulting HV density is not too high, and each encoded vector still contains enough information.

3.2.2 Thresholded Sum

The second main SDB-HDC encoding scheme uses thresholded sums and is explained in detail in section 2.4.3. The majority sum, which is its DDB counterpart, is the most commonly used encoding scheme in DDB-HDC applications.[29, 12, 28, 11, 20] Not many applications have been converted to SDB yet, but these can likely be converted similarly to the case studied here. Once again, we aim to improve our understanding and find optimizations by implementing the thresholded sum encoding scheme in an application. First off, DDB language recognition [29] is recreated in Python, using the publically available Matlab implementation from [3] as a guide. Afterward, the goal is to make it work using sparse HVs and a thresholded sum operation for the bundling steps.

Language Recognition

The DDB-HDC language recognition algorithm is implemented exactly as described in section 2.3. This work's findings and results match those described in the source paper [29] and shown in section 2.3. This architecture uses XOR binding for the sliding window HVs and a majority sum for the bundling. As described in more detail in section 2.4.3, the thresholded sum is implemented by accumulation and thresholding. It acts like a majority sum but with a controllable threshold instead. The thresholding part of the function can be implemented in Python as shown in listing 3.1, where text_hv contains the accumulated HVs and window_cnt equals the number of input characters:

Listing 3.1: Python implementation of the controllable threshold

```
def binarize_hv_thr(v, threshold):
    for i in range(len(v)):
        if v[i] >= threshold:
            v[i] = 1
        else:
            v[i] = 0
    return v
```

```
text_hv = binarize_hv_thr(text_hv, 0.005*window_cnt)
```

When trying to use the DDB-HDC architecture for SDB language recognition you receive poor accuracy results, even with a controllable threshold on the bundling step. For example, an IM density of 10% with D=10,000 and sliding window size N=3 leads to <90% accuracy depending on the threshold value. (see section 2.3 for an explanation of the testing procedure) This happens because the XOR doesn't bind sparse HVs well, as it acts like an OR operation for sparse hypervectors. This means that the result is similar to its constituents and the resulting encoded hypervectors for language recognition are too similar to each other for accurate classification.

As described in section 2.4.3, we can also use a majority sum for combining the sliding window HVs and an accumulation with a controllable threshold for the main bundling step. However, languages have a tendency toward the same N-letter permutations, leading to window HVs which are not easy to distinguish, and poor accuracy results for SDB due to its locality-based encoding.

An alternative method applies a unique permutation to each letter in the sliding window, by XORing the signatures randomly given to every other letter in the window and summing them with the HV's position to determine the shift amount. The resulting permutation can be expressed in the following way:

$$\rho_K = K - 1 + (S_N \oplus S_{N-1} ... \oplus S_{K+1} \oplus S_{K-1} ... \oplus S_1) \tag{3.3}$$

Where ρ_K is the number of shifts applied to the K$^{\text{th}}$ vector in the N-gram (with N $\geq$ K).[10] For a more in-depth explanation of signature encoding, see section 2.4.3.

For this work's implementation of this algorithm, an initial IM density of 100/2048 = 4.88% was chosen, because in [10] and through testing a density of 5% proved optimal. In this paper, they achieved a maximum accuracy of 95.4% for dimensionality D=10,000 and only 74.18% for D=2,000. The simulation was able to recreate these but noticeably, the average accuracy was a lot lower for some languages like Czech, Polish, and Slovak than most others. When investigating the trained HV, the resulting density after training would vary heavily based on the language. See Table 3.3 for an example of the density differences between the languages stored in the associative memory after training.

The controllable threshold for the second (main) bundling step is a fraction of the number of input letters here, as just a static threshold would give inconsistent/bad results if the input length varies. So this is 50% for the majority function, and 30% is optimal for this algorithm. However, each language tends to favor certain combinations of letters, and for some this greatly increases or decreases the resulting encoded density. This appears detrimental to the classification accuracy, as higher-density HVs naturally lead to higher overlap scores, thus favoring those languages during predictions. Most languages in Table 3.3 hover around 14-15% density, but for the languages with lower density like Czech (ces) and Polish (pol), the overlap will automatically be smaller with the encoded HVs during testing. This decreases the chance that they have the greatest overlap, and causes them to have a far worse average accuracy score during testing than most others, dragging down the total average accuracy.

Table 3.3: SDB-HDC language recognition density disparities after training.

Language	Density
bul	0.1479
ces	0.1255
dan	0.1474
nld	0.1527
deu	0.1413
eng	0.15
est	0.136
fin	0.145
fra	0.1523
ell	0.1553
hun	0.1295
ita	0.1451
lav	0.1373
lit	0.136
pol	0.1242
por	0.1573
ron	0.152
slk	0.126
slv	0.1352
spa	0.156
swe	0.1449

Sorted THR

To combat this, we implement a different method of thresholding the accumulated HVs during training, which results in nearly the same density for each language HV. Let's call this method *Sorted THR*, as it makes use of a sorting step. The training (loading the associative memory) only happens once, so the time and energy it takes to do this are less important. We modify the second bundling step of SDB language recognition, but only during training. It performs an accumulation as normal and creates a vector of length D containing the accumulated values for each dimension, but the thresholding step is replaced by the new method. Sorted THR sorts the accumulated vector from large to small values, and a cutoff index is defined as follows:

$$\mathrm{idx} = \mathrm{round}(\mathrm{thrval} * D) - 1 \tag{3.4}$$

Here *thrval* is a value between 0 and 1, representing the desired encoded HV density. The value of the sorted array at this index is then taken as the threshold value.

This new threshold value is then applied in a regular thresholding step to the (unsorted) accumulated vector. Like the regular thresholded sum, all values greater than or equal to the threshold are set to 1 and all others are kept 0 in the resulting encoded HV. Listing 3.2 shows a Python implementation of this function. Here v is the hypervector containing the accumulated values after training and *final_density* is a value between 0 and 1 that determines the resulting encoded HV density.

Listing 3.2: Python implementation of the sorted THR function

```
def binarize_hv_sort(v, final_density):
    sort_array = np.copy(v)
    sort_array[::-1].sort()
    cutoff = round(len(v)*final_density)
    threshold = sort_array[cutoff-1]

    for i in range(len(v)):
        if v[i] >= threshold:
            v[i] = 1
        else:
            v[i] = 0
    return v
```

Because Sorted THR finds the accumulated value at exactly the fraction of the number of values we want to keep, the resulting density can be precisely controlled. For example, if *thrval* is set to 0.30 and D=2,048, the cutoff index would be 613. This means that the values at and below this index in the sorted array will be "kept", meaning these values are put to 1 in the (unsorted) accumulated HV after thresholding. As a result, 614 values will be set to 1 while the rest is set to 0, thus the resulting density will be exactly 614/2048 = 0.2998.

Using this method, training and testing are separated and their methods of encoding differ slightly. The training is done in software and uses the Sorted THR method described above in its second bundling step. However, the testing should also be able to run on hardware, where sorting arrays would be a costly task. Thus during testing, the previously described method of computing the threshold value based on the number of inputs is used.

Performance Evaluation

The accuracy performance is evaluated as described in section 2.3. The language HVs stored in the associative memory can classify 21 European languages and are each based on a million bytes of training data from [24]. During testing, the classification accuracy is determined over 1,000 samples per language (from [18]) and averaged over all 21 languages.

Tests were conducted using the SDB language recognition with the signature encoding scheme and the sorted THR method during training. First, smaller tests using only 7 languages of the dataset (bul, ces, dan, deu, eng, ell, spa) are conducted to determine the optimal parameters, as these take less time to run. From these, an item memory with an HV density of 2% in combination with a sliding window size of 3 (a 3-gram) gives the best accuracy results. This has the bonus that a lower-density IM means less switching and thus less energy usage. A threshold value of 0.40 for the sorted THR and 0.001*window_cnt for the regular THR (window_cnt equals the number of input characters) then maximizes the accuracy.

Table 3.4: Language recognition accuracy comparison between the different implementations for varying dimensionality.

	N-gram	2,000 bits	4,000 bits	6,000 bits	8,000 bits	10,000 bits
Dense	3	69.4%	86.4%	90.5%	92.1%	96.1%
Sparse	4	74.2%	86.5%	91.5%	93.6%	95.4%
Sorted THR	3	95.1%	97.6%	98.0%	98.5%	98.7%

Table 3.4 shows the accuracy for the DDB implementation from [29] and SDB from [10] compared to the SDB implementation using the sorted THR method in training. The parameters that deliver the highest accuracy for the given dimensionality and N-gram are used for each test. The results show that the sorted THR method achieves an enormous accuracy improvement, especially for the lower dimensionalities. Clearly, the SDB signature encoding in combination with sorted THR has a high robustness to dimension reduction. The uniformity of the densities in the trained HVs created by this method allows for accurate classification, as bias toward any one HV due to higher densities is removed. This proves an important key in assuring maximal accuracy when using SDB with an overlap calculation to measure similarity instead of Hamming distance.

Table 3.5: Language recognition accuracy comparison between the dense and the sparse sorted THR implementation.

Window size	Dense (D=10,000)	Sparse (D=2,000)
Bigrams (N=2)	93.2%	79.9%
Trigrams (N=3)	96.7%	95.1%
Tetragrams (N=4)	97.1%	90.0%
Pentagrams (N=5)	95.0%	77.9%

In Table 3.5 the accuracy for different window sizes can be seen. Notice how the sparse design with the sorted THR method can get accuracy close to the dense design with a much lower HV size, but the optimal window size is different. A smaller optimal window size is only advantageous though, as it reduces the latency and consequently the energy usage. This makes the sparse design with sorted THR even more energy-efficient than the DDB-HDC design, on top of the reduced switching due to its sparsity. In summary, the SDB design with signature encoding and the sorted THR method beats the DDB design in all metrics.

This method manages to solve one of the key problems that SDB-HDC has; it removes the slight accuracy drop and even manages to outperform DDB-HDC in language recognition. Further research is necessary to test how it performs in other thresholded sum-based applications like [11, 20, 12], as the results for SDB language recognition are very promising. The accuracy improvement is heavily correlated with how consistent the encoded HV density after training is without sorted THR, so this will determine which applications benefit most from it.

3.3 Summary

By studying and exploring the SDB-HDC algorithms in software, a deeper understanding of their workings was created and the sorted THR optimization was found. The insights acquired during this investigation will be applied when designing the programmable hardware accelerator in the next chapter. Further, the sorted THR method for summed thresholding manages to drastically increase the accuracy of sparse language recognition and should be looked into more. It could boost the performance of all SDB-HDC algorithms based on summed thresholding where the density of the trained hypervectors is inconsistent. This innovation has the potential to enhance the viability and adoption of the SDB-HDC computing space by improving its downside of sometimes having lower accuracy than more conventional machine learning techniques.

Chapter 4

Designing an Efficient Programmable SDB-HDC Accelerator

This section will describe how the HDC research listed in chapter 2 and the investigation of the encoding schemes in chapter 3 can be used to create an ASIC implementation of a programmable SDB-HDC accelerator. First, possible implementations for each module are discussed and DDB optimizations for these modules are converted to SDB. Then, a programmable accelerator design is created and implemented, and tests are worked out to verify its functionality. Afterward, the latency is reduced and the design is further improved through hardware reuse and trimming. Lastly, metrics to compare the performance with state-of-the-art applications and HDC accelerators are defined.

4.1 Basic Architecture

The basic architecture for the programmable SDB-HDC accelerator defines the main building blocks that make up the design. The accelerator is programmable and should thus support all operations from the common encoding schemes described

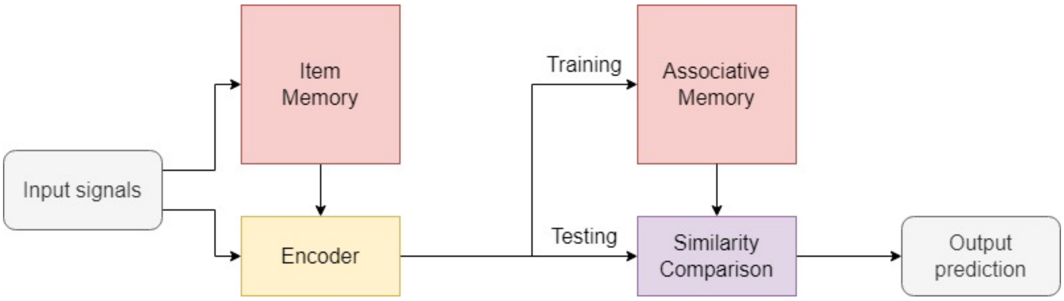


FIGURE 4.1: Most common general structure for HDC algorithms.

in the previous chapter, and combinations of their functions. This can be done by separating the multiple stages of the encoding schemes into building blocks like the mapper, item memory, binding, and the different bundling stages. Each stage must have the functionality of the different encoding schemes, and be programmable to select the desired functions.

These stages are implemented in the encoder section from Figure 4.1. This figure represents the general implementation structure of nearly all SDB- or DDB-HDC applications and is explained in detail in section 2.2. That's why this is also chosen as a basis for the architecture of our programmable accelerator.

Figure 4.2 shows the main building blocks in the architecture designed according to the principles listed above. The communication of the accelerator to other cores/processors happens through a Control and Status Register (CSR) and the associative memory is accessed through a memory interface. This CSR is a register bank that functions as the main I/O for the accelerator. By writing to these registers, a processor can program the accelerator to control its functionality, apply start and reset signals, and feed input data. Meanwhile, by reading it can retrieve the accelerator's status and output. This provides a semi-standard but customizable interface for the accelerator. The mapper connects the CSR values to the control signals for every module, to activate the desired functionality. Besides this, it also relays the input signals to their proper item memory address and provides the signature-encoding functionality for sliding windows as described in section 2.4.3.

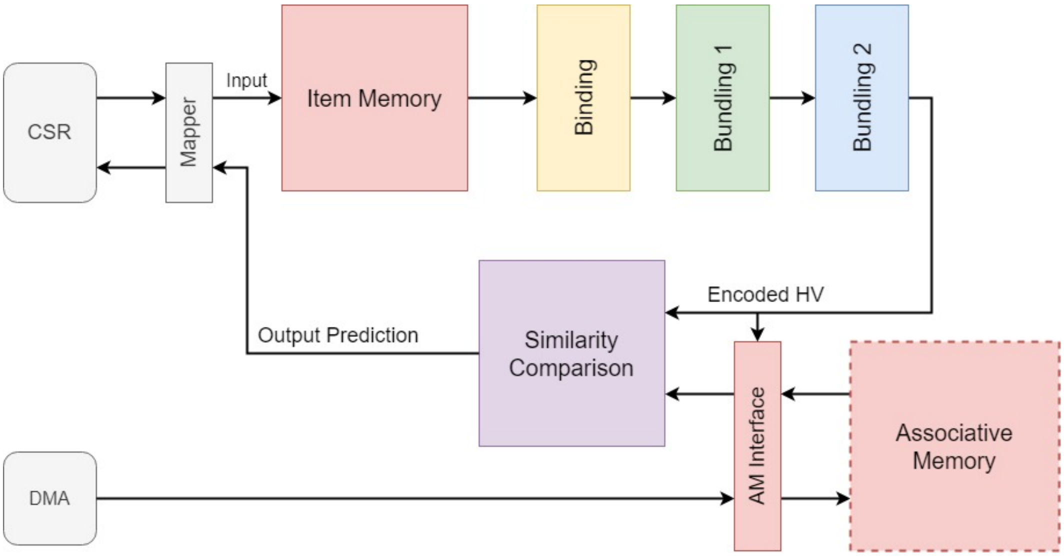


Figure 4.2: High-level overview of the accelerator architecture.

The blocks for encoding combine the functionalities of the character and language recognition algorithm; they implement the functions used by the CDT and the thresholded sum encoding schemes. The item memory is a storage that can produce a set of quasi-orthogonal hypervectors linked to the inputs, to be used during encoding. Next up, the binding module implements the permutation function to encode additional input information into the HVs. This is followed by some bundling stages to combine a series of inputs into one encoded HV. A design with 2 bundling stages is chosen here, to be able to support sliding window inputs, as they are very common in the HDC space. More than 2 stages would be necessary if we would want to process more input data in parallel, such as multi-sensor inputs, but this is not that common. The result of the encoding modules is then a single encoded HV that represents a collection of input data.

This encoded HV can either be stored in the associative memory during training or compared to the trained HVs already present in the AM. The similarity comparison module checks for the highest overlap with the trained HV and determines the best matching output. Lastly, the output is directed back to the CSR, to be read by the interacting processor or core.

The associative memory is accessed through the AM interface. It has the capability to read or write D bits at a time, where D is the dimensionality of the HVs, and send/receive data to the rest of the accelerator or the processor. The associative memory is not part of the accelerator itself and can be seen as part of the larger memory of a processor. When the training takes place in software, the trained HVs can be loaded into the AM through for example direct memory access (DMA). This allows the connected processor to preload and modify the AM before the accelerator starts its computations.

4.2 Module Implementations

When implementing custom hardware tailored to DDB- or SDB-HDC algorithms, a variety of implementations and optimizations are possible. There are existing module optimization strategies for DDB-HDC.[31] These can be slightly modified or sometimes even directly used to work for the SDB counterpart as well. This section explores hardware implementations for each of the accelerator's modules by investigating existing DDB-HDC optimizations and arguing if they are fit for SDB-HDC.

4.2.1 Control and Status Register

The control and status register (CSR) is an assembly of registers that can usually be described by a register map. It exists of a register bank of e.g. 10 rows of 32 bits wide, where each register (row) holds in- or output data whose signals are connected to the accelerator. The CSR inputs consist of the register address (the row), a write enable, the write data, a request valid, and a response ready input. Its outputs are a request-ready signal, the read data, and whether or not the response is valid. On a read request, it returns the data at the specified address in the same clock cycle, and on a write request, it writes the data in the next cycle. This provides a clean interface for a processor to program the accelerator and transfer data.

4.2.2 Mapper

The mapper module's main function is determining when a new input can be requested and implementing the signature encoding scheme. It determines when inputs can be requested through logic based on the accelerator's settings/programming, as they determine the time it takes for the first bundling step to complete. The signature encoding scheme is a special case because it only takes a new external input when the N-gram sliding window moves, and outputs N "internal" inputs for each window (N inputs to the IM). (See section 2.4.3.) We can implement the signatures by letting them equal the character's item memory address, as this simple implementation doesn't result in a noticeable accuracy drop. This means that e.g. for language recognition: "a" has the signature 0, "b" has the signature 1, etc. The sliding action of the window is implemented using a FIFO queue, and the XOR values and permutations are determined through combinational logic. An XOR of *all* signatures in the sliding window is taken and XOR'ed with the signature of the currently selected letter, to cancel out that XOR. This comes down to an XOR of the signatures for every letter in the window that is not the currently selected one, which determines the number of permutations undergone by that letter's HV.

4.2.3 Item Memory

The item memory (IM) typically stores a limited number of hypervectors, used as a starting point for the encoding step. A straightforward implementation would be hardcoding the HVs in a register bank or using a ROM, as its contents don't change. Another simple but effective option is using a lookup table to link the outputs (1s in the selected HV) to the given input (IM address). However, some other more optimized implementations for a DDB item memory exist.[31] For example, we have the Cellular automata (CA) and the Hypervector Manipulator (MAN).

Cellular Automata (CA)

When DDB IM inputs are applied in the same chronological order each time, like for an application using channels such as EMG gesture detection, cellular automata can be used to generate the HVs. A one-dimensional CA can produce chaotic behavior suitable for a sequence of quasi-random HVs, by using a *neighborhood* of 3 and applying *rule 30*.[31] A CA with D cells can be initialized with a random hypervector, and will then produce quasi-orthogonal hypervectors every cycle. The same sequence of HVs can be produced each time by resetting it, so long as the initial HV is the same.[31] The initial state of the CA can be defined as a random seed HV and it can generate all other quasi-orthogonal HVs through its algorithm. This forms a suitable replacement for the IM, providing a low-cost implementation at the expense of flexibility. The downside here is that it's only useful if the IM vectors are accessed chronologically one after the other every time, otherwise, the latency would be very great due to the CA having to switch until it reaches the desired HV.

This is not suitable for our programmable accelerator, as we want it to be able to handle random IM accesses and have low latency while doing so. Additionally, it is not explored if this method can be transformed to create sparse HVs instead of dense ones.

Hypervector Manipulator (MAN)

A flexible IM can be implemented by a lookup table, but its footprint can be reduced by utilizing a hypervector manipulator (MAN) module instead. The MAN module manipulates HVs in a controlled manner, by starting from a seed vector and switching multiple bits at a time from the previous vector. It uses a fixed (hardcoded) connectivity matrix to determine which bits should be flipped for each input. A visualization of its implementation with a possible connectivity matrix can be seen in Figure 4.3.

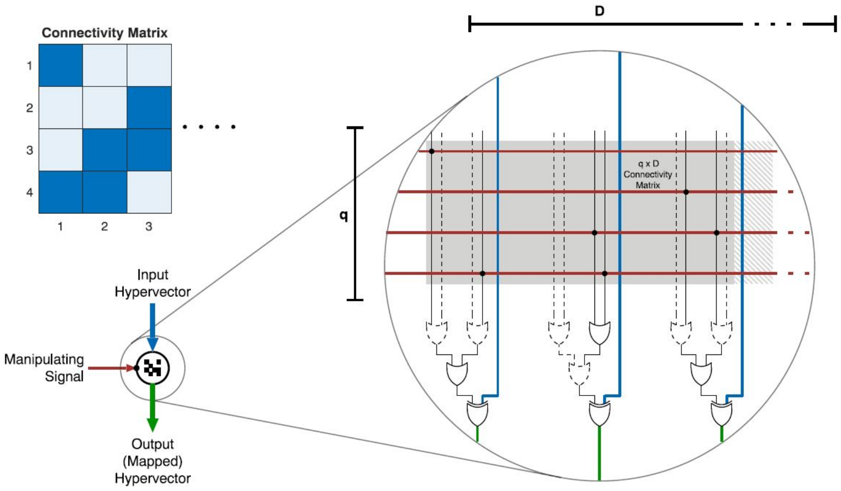


FIGURE 4.3: Representation of the MAN module and an example connectivity matrix. (retrieved from [31])

The MAN is able to replace the IM, by incorporating the HV patterns in the connectivity matrix. It starts from a hardwired seed HV, and each row of the connectivity matrix flips some bits. The seed HV can be created by simply hardwiring its bit connections to source and ground.[31] The entire module consists of a connectivity matrix, OR- and XOR-gates to combine the outputs into a single output HV, and an s-hot lookup table (LUT) to control which input is triggered.

If we want to convert this to an SDB implementation, we would have to be careful to maintain the same density/sparsity across all output HVs for the IM. This means that the connections that flip the bits have to be selected carefully, so the resulting HV has the same density as the input HV. This could for example be generated by first flipping the input bits of the seed HV, and then randomly flipping p*D bits afterward, where p is the requested density. Because of the sparsity of the HVs in the IM for SDB however, using a MAN won't nearly give as good of an area improvement compared to a LUT implementation as it does for the DDB representation. In SDB, if the IM doesn't contain that many HVs, there will be a lot of outputs that never get switched to 1 because each HV contains a 0 in that position. This means they don't have to be included in the LUT logic, so the LUT for an SDB IM will be significantly smaller than for a DDB one. Further testing is necessary to determine whether or not a MAN will have an area improvement on a LUT in SDB representation.

4.2.4 Binding

The binding module only implements the permutation operation. It consists of a cyclic shifter and a register for this purpose. The variable cyclic shift can be implemented in several ways, the easiest of which are a variable shifter and a fully combinational shifter. A variable shifter is the most area-efficient implementation and executes one permutation per cycle, thus low area is traded for high latency. The latency can be decreased at the expense of higher area, by implementing a partly or fully combinational shifter. For example, a fully combinational barrel shifter can execute the permutation within one cycle.

4.2.5 Bundling

A *Majority Function* or *Thresholded sum* can be implemented in hardware by counters/accumulators and a thresholding function. Accumulators have the clear downside of requiring a large register array to store the sum over D dimensions. Thus, we seek to decrease their size where possible or look for a different implementation.

Bidirectional Saturating Counters

A possible DDB hardware implementation is building *bidirectional saturating counters*. Each 1 increments the count and each 0 decrements the count by one. By letting the counter count both up and down, the output value is expected to be around the mean (zero) for DDB-HDC. The memory footprint can then be further reduced by letting the counters saturate in both directions because a long sequence of 1s or 0s has a very small probability of occurring. By tuning the saturation values according to the application, the maximum accuracy is still reachable.[31]

This principle of saturating counters can be applied to SDB as well. Because of the sparsity, the average distribution of 1s is way lower 50%, and using a bidirectional counter wouldn't make much sense as the value would go far into the negatives for most bits due to the great majority of 0s. Thus, we use a regular counter where we increment by one if the bit value is 1, and don't increment otherwise. We can however make these counters saturating to save on area.

The sparsity reduces the chances of a 1 getting added to the counter for each vector, thus they will not increase as fast as for DDB, which results in an inherent area saving by being able to reduce the size of the accumulator registers. Further, the useful threshold is often $\leq 50\%$, which means that we can stop counting after this threshold is reached. Due to this, we can saturate the accumulators after they've reached a certain value and subsequently reduce the size of their registers. Additionally, the amount of inputs that can be bundled into one encoded HV (the input length) is also a determining factor for the accumulator register size. The saturating accumulator size is thus dependent on the number of inputs we want to be able to handle and the maximum threshold value we want to support. These are choices we will have to make for our programmable accelerator to determine the final accumulator size.

Binarized Back-to-Back Bundling (B2B)

Binarized Back-to-Back Bundling (B2B) is another DDB implementation aimed at optimizing bundling.[31] This approximates the bundling operation but decreases the footprint by staying in the binary space. Instead of storing the current majority vote, it bundles the HVs iteratively, by assigning a "weight" to each vote. Each vote is assigned a chance to turn the majority around, but this only happens if the current vote differs from the majority. The first vote has a probability of 1, the second 1/2, the i-th 1/i, etc. The probability turns into a weight when looking at all of the HV's dimensions. It can be implemented by a MAN module, where the connectivity matrix is randomly generated with each row decreasing the number of active bits relative to its probability. An example connectivity matrix to bundle 10 HVs with D=64 is shown in Figure 4.4. For larger dimensions, row n will have around D/n connections, where each connection determines whether that vote has the ability to turn the majority around if it differs from the majority. This means that for the given example, all bits in the HV in row 1 can turn the majority around, while in row 2, 50% of the bits have this ability, etc. B2B bundling thus only needs a memory of D bits without using any adders or saturation logic, making it very efficient.

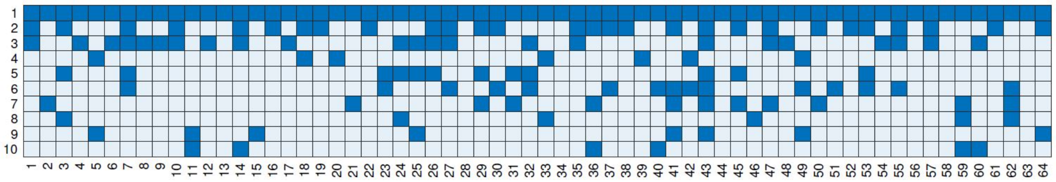


FIGURE 4.4: Example connectivity matrix for B2B bundling. (retrieved from [31])

It is unclear if this method could be applied or transformed to work for SDB-HDC, as no research has been done toward this. Because the density is not 50%, the probability of the result actually getting flipped from votes with a low weight/probability is small. This means that similarly to the MAN module, it could cause density irregularities in the resulting encoded vector, and the density of this resulting vector cannot really be controlled. Based on the experiments with language recognition described in section 3.2.2 we can see that at least for this application, consistency in the encoded HV density boosts the accuracy. This could be a major issue when using SDB B2B bundling and would make it unsuitable for use in the programmable accelerator.

Additionally, as can be seen in Figure 4.5, B2B bundling has a limited capacity. When bundling orthogonal hypervectors, the bundled vector should be similar to all of its constituents. This means that when it is orthogonal to one of the input

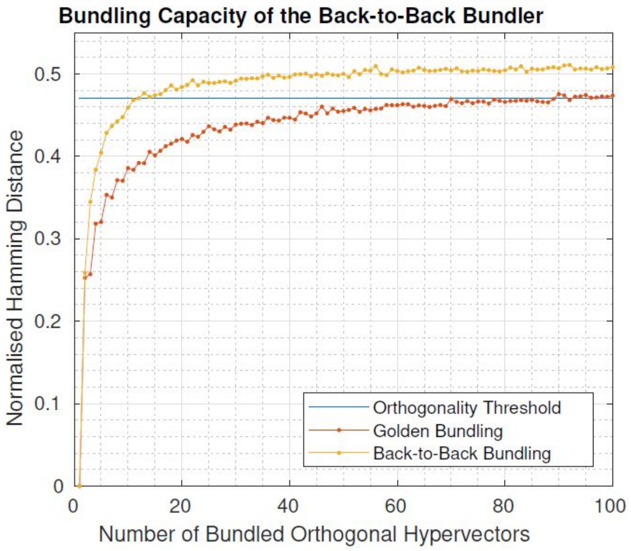


Figure 4.5: Bundling capacity for B2B vs. "ideal" bundling, for D=10,000. (retrieved from [31])

vectors (Hamming Distance > 0.5), information is lost and its maximum bundling capacity is reached. The figure compares B2B bundling to a majority vote, which is referred to as "golden bundling" here. It can be seen that the capacity starts to break down when bundling more than 10 vectors together, thus this at least cannot be used for the 2nd bundling step of our accelerator.

4.2.6 Associative Memory

The associative memory (AM) can be implemented by a simple register file or SRAM. Other possibilities might be possible, but this falls outside the scope of this thesis. As we want our accelerator to be flexible and generalized, we choose not to include the memory part for the AM in the accelerator's design. Instead, we implement an interface for the accelerator to read and write to/from this memory and use a small memory implementation solely for testing purposes. This way, the associative memory data can be stored in a larger memory that is potentially also used by other cores or processors.

As we want to make the accelerator compatible with RISC-V cores, we make it compatible with the PULP platform.[23] This is done by using a byte-addressable memory that exists of multiple 64 by 32 (or higher) banks. For example, for D=2,048, 32 banks of 64x32 bits provide the storage capability to store 32 encoded hypervectors in the AM. This means that each "row" of the memory consists of 256 bytes, that can be accessed in parallel.

A "base AM address" and "max AM address" are programmed in the CSR, to define where in the memory the HVs are loaded. These define the first byte of the first loaded HV, and the first byte of the last loaded HV. During the similarity comparison step, the D bits for each stored HV can then be read sequentially, starting from the base AM address.

4.2.7 Similarity Comparison

The last step in the HDC algorithms is predicting the output value, through a similarity comparison with the HVs stored in the AM. The overlap between the encoded HV and trained HV (from the AM) is taken through a D-wide AND array.

For the similarity calculation, a count of all active bits in the resulting vector has to be taken. This can be done through a simple iterative counter or different adder tree structures. A tradeoff has to be made between area usage, power drain, and latency/throughput here. Thus, the choice will be heavily dependent on the design of the encoder section of the accelerator and the slack in this section when synthesizing.

Deep adder tree structures will allow the AM to compute the bit count in only a single cycle, at the expense of long logic delay and high area. The depth or number of adder stages for a full adder tree is $\log_2(D) = n$, where the i-th stage has $D/2^i$ adders with a width of i. The total number of equivalent 1-bit adders is then: $\sum_{i=1}^{n} \frac{D \cdot i}{2^i}$ which comes down to 4,083 for a dimensionality D=2,048.[31] This will require quite a bit of area, but it will be relatively low in comparison to the bundling stage.

The alternative of using counters will require a number of cycles in the order of $\mathcal{O}(D)$, drastically increasing the latency of the system. On top of this, they have a lot of area overhead because each bit from the HV has to be routed to the counter, which would require huge multiplexers or shift registers.[31] This makes them not a good choice for the accelerator design, as we aim to minimize the latency. A deep adder tree or groups of adder trees will probably be the best choice, but the best specifications have to be determined by testing multiple synthesized designs.

4.3 Accelerator Implementation

Based on the basic/high-level architecture and module implementations, an accelerator implementation can be created. For this implementation, we aim to design and implement an architecture based on the previously discussed module implementations. Its main goal is to achieve the full desired functionality of the final design and create a working hardware implementation. Later, this design can be further improved by modifying the worst-performing modules through hardware reuse and trimming unnecessary registers.

4.3.1 Design Specifications

The design broken down into its detailed building blocks can be seen in Figure 4.6. It has the same structure as Figure 4.2, but an implementation for each module is defined. Important to note is that the main synthesized and simulated design has a hypervector dimensionality D = 2,048. This value is chosen as it provides good accuracy, while the area and power usage will still be reasonable in 130nm Skywater technology. This section will describe the exact specifications and design choices for this implementation. The design is implemented and tested using SystemVerilog, while the test data is generated using Python scripts.

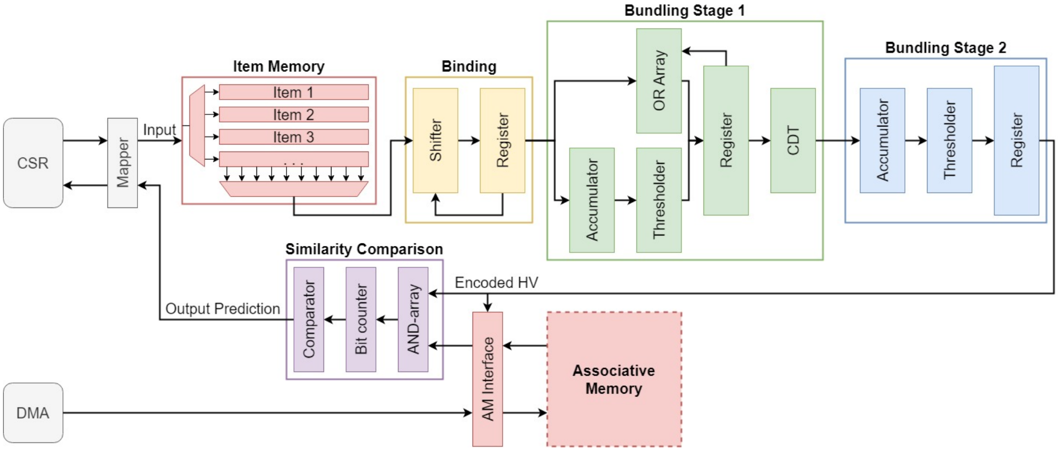


Figure 4.6: The programmable accelerator architecture.

CSR

The control and status register is implemented by a 32-bit wide register bank with 8 rows. Each row has an address and corresponds to different settings/inputs for the accelerator, which are described in Table 4.1.

All addresses except for 2 (the status register) are for an external processor/core to write in and program the accelerator. When a 1 is written to bit 0 of address 0, the accelerator is started and will start accepting inputs. The inputs can be written to address 1 and should remain constant during the "running" phase until the accelerator is ready to receive a new input. Note that *input_done* should be set high when the second stage accumulator should stop accumulating, and the threshold should be taken. The accelerator's status can be read by reading CSR address 2. Running will output 1 after the start signal has been given and until the output is valid and its value can be read. Address 3 and 4 are for programming the binding and bundling settings respectively. Note that the "enable encoder writing" signal programs the accelerator to write the encoded HV into the associative memory on the base address set by CSR_AM_BASE. Next up, addresses 5 and 6 define the address range where the HV's are loaded in the memory, for retrieval during the similarity computations. Finally, address 7 resets the accelerator (the accumulator registers, etc.) when 1 is written to it. This should be done in between a new set of inputs or reprogramming of the accelerator.

Table 4.1: Description of the CSR register map.

CSR Name	Address	Bits	Description
CSR_START	0	0	Starts the accelerator. Reads return 0
CSR_INPUT	1	0	Input valid
		1-6	Input value
		7-12	Shift amount
		13	Input done
CSR_STATUS	2	0	Running
		1	Ready to receive new input
		2	Output valid
		3-7	Output value
CSR_P_BINDING	3	0	Enable sliding window
		1	Enable signature encoding
		2	Enable permutation binding
CSR_P_BUNDLING	4	0	Enable stage 1 accumulator
		1	Enable CDT
		2	Enable stage 2 accumulator
		3-8	Stage 1 window size
		9-12	CDT k-factor
		13-15	Stage 1 threshold value
		16-22	Stage 2 threshold value
		23	Enable superposition bundling
		24	Enable encoder writing to AM.
CSR_AM_BASE	5	0-31	Base address of loaded HV in AM
CSR_AM_MAX	6	0-31	Max address of loaded HV in AM
CSR_RESET	7	0	Resets the accelerator. Reads return 0

Mapper

This module is implemented as described in section 4.2.2. It can handle a maximum sliding window size of 12, as we want the accelerator to be flexible and this size is used in [12], which is a promising application of HDC in gene sequencing. Smaller window sizes are sufficient for most other applications. SDB language recognition, for example, has an optimal sliding window size of only 3.

Item Memory

For the item memory, we want to start with a simple implementation. This is why a lookup table is chosen that can store 35 hypervectors with a dimensionality of D=2,048. There are 35 addresses/hypervectors stored in the IM, as this is enough to support most applications while keeping the area usage low. As mentioned earlier, the synthesis will automatically optimize this LUT to not include logic for bits where all HVs contain a 0, greatly reducing its size. As the item memory only contains 35

HVs, the synthesis will be able to optimize it quite a bit. This makes a LUT a good option for this SDB-HDC IM, so it doesn't necessarily have to be improved later.

Binding

The binding module implements the permutation function. For the initial implementation, a variable cyclic shifter is used that takes 1 clock cycle per 1-bit shift and stores its result in a register. This means that e.g. a 5-bit shift would take 5 cycles and that it has a low area cost, but high latency. This will later be replaced by a fully combinational implementation instead.

Bundling

The bundling module consists of 2 stages, as language recognition and other similar applications require 2 sequential accumulations. The first stage consists of an OR array to implement the superposition used for the CDT scheme, in parallel with the accumulation and thresholding used in the thresholded sum scheme. The result of both operations is stored in a register, and followed by a block that implements the context-dependent thinning function. This means the first bundling stage can be programmed to perform superposition, accumulation & thresholding, or no operation, followed by an optional CDT operation. The superposition and accumulator can process a maximum of 64 inputs, before generating an output.

To optimize the accumulator, it is implemented to saturate after a certain value has been reached. The stage 1 accumulator register has a size of D by 4 bits, where 3 bits are used to store the accumulated result and 1 is used to indicate whether that counter has saturated. This means that it can saturate after a minimum of 8 HVs are accumulated. Figure 4.7 shows the implementation for 1 bit of the saturating accumulator from stage 1, followed by a comparator for the thresholding. If the MSB (most significant bit) of the accumulator is 0, the input value is added to the accumulated value stored in the register, otherwise, no addition is performed.

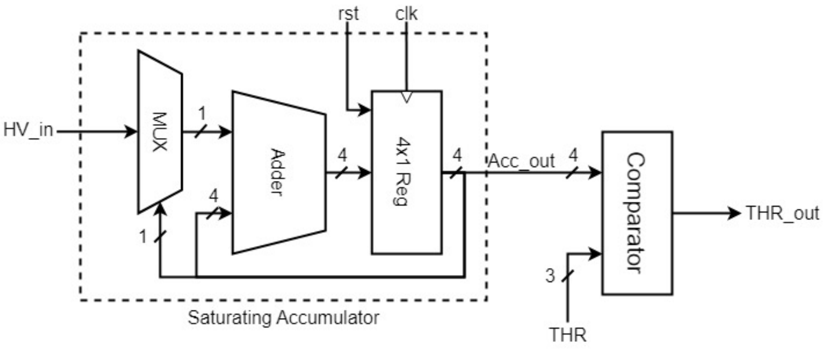


Figure 4.7: Diagram for 1 bit of the saturating accumulator and thresholding.

The accumulator can saturate after the threshold value is reached since going to higher values does not affect the thresholded result. For the first stage, a majority sum or smaller threshold is usually taken. This means that for a maximum sliding window size of 12, and a threshold value of 50%, a minimum of 6 HVs should be able to be accumulated before saturation happens to achieve maximal accuracy. A 4-bit accumulator register for each of the D dimensions only saturates after a minimum of 8 additions, so this suffices in combination with a 3-bit threshold value (max value 7).

After the first accumulation and thresholding, the data passes through the CDT module. The CDT operation is implemented in a fully combinational way and can handle a k-factor of 1-3. This limit on the k-factor is chosen, as a higher k-factor increases the resulting density. When the resulting density after CDT nears the HV density before the CDT operation, the benefit of using a CDT is removed, so a too-high k-factor wouldn't make sense. This implementation produces the result in a single cycle and stores it in a register. Its footprint is a relatively small part of the total bundling module footprint, at 5.4% for D=1,024 and 4.5% for D=2,048.

An example of how the CDT operation for a maximum k-factor of 2 would be implemented in a fully combinational way can be seen in Figure 4.8. A total of *k* conjunctions between the input and a permuted version are made, and these are disjuncted together. The mux array at the end selects the desired output, based on the k-factor. In this example, the top output would be selected for a k-factor of 2, and the bottom one for a k-factor of 1.

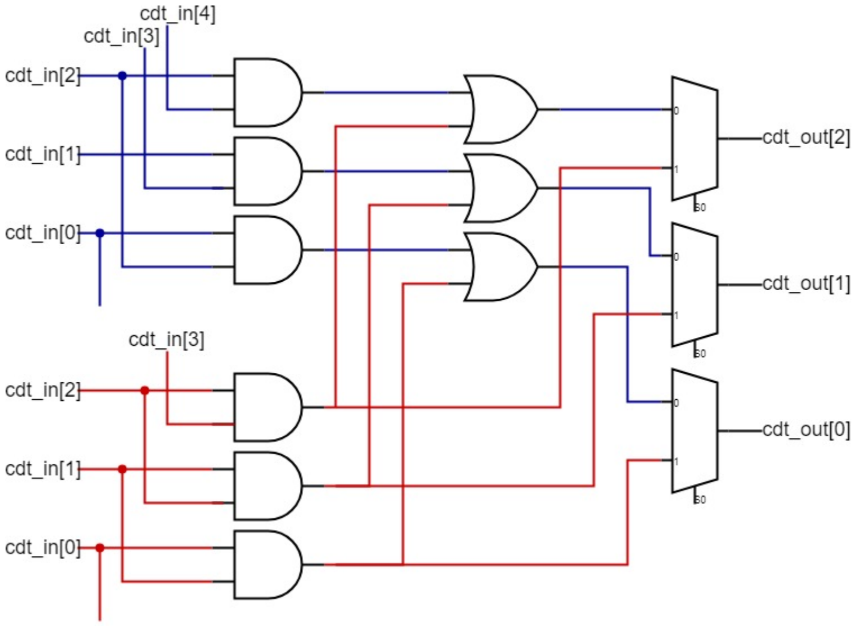


FIGURE 4.8: Diagram for a CDT implementation with a maximum k-factor of 2.

The second bundling module implements an optional accumulation & thresholding function, followed by a register to store the encoded HV. A sliding window produces the most bundling inputs per accelerator input (N bundling inputs for an N-gram).

This means we can e.g. look at language recognition to determine the maximum input sequence length we want to encode into a single HV. Language recognition uses testing files where the maximum length is 1139 characters and the second largest file contains 975, so we limit the accelerator to 2,048 inputs to save on area. Note that most testing inputs are far shorter than this. The accumulator of this stage makes use of a D-by-8-bit register, so a minimum of 128 HVs is necessary before saturation happens. This means that e.g. if THR=0.30%, 426 ones can be added before saturation has an effect, which should be plenty given that we work with sparse HVs. This saturation allows for the accumulator and counter registers to be smaller, decreasing the area usage.

Associative Memory

The associative memory is not part of the (synthesized) accelerator design, as we are focusing on the data path. For the synthesized design, we assume that the AM data is stored in the larger (shared) memory of the processor that controls this accelerator. Figure 4.9 shows how we could integrate the accelerator with the PULP platform.[23] This integration has not been tested, but the accelerator is designed to be compatible. The AM memory interface of the accelerator is connected to the processor's interconnect and interfaces with the memory this way. This means that the trained HVs read during the similarity comparison are not stored locally in the accelerator, but instead sequentially retrieved from this shared memory. The base and max AM address entered into the CSR should describe the address range where the AM hypervectors are stored in this memory.

A memory of 2,048 by 32 bits consisting of 32 64x32 bit register banks is used in the simulation for testing purposes, making it able to store 32 trained HVs. This is necessary to verify the functionality of the AM interface, but this memory is not included in the synthesized design. It has a single read/write port that can transfer 2,048 bits per cycle, so the accelerator can read a new HV each cycle during the similarity comparison.

Similarity Comparison

A 2,048-wide AND array is used to compute the overlap, followed by a bit counter to count the resulting high bits. The initial design for the bit counter consisted of a fully combinational sequential adder. When implementing this bit counter in a single cycle, the synthesized design couldn't meet the timing for a clock frequency of 100 MHz, so a register to store the AND-array result before counting the bits was implemented. Another register was implemented in the middle, to make it a 2-cycle computation that is able to meet the timing requirements. Lastly, the amount of overlap for each HV stored in the AM with the encoded HV is compared to the best overlap for that encoding by a comparator. The address where the highest overlapping HV is stored (subtracted by the base address) determines the predicted output and is fed back to the CSR.

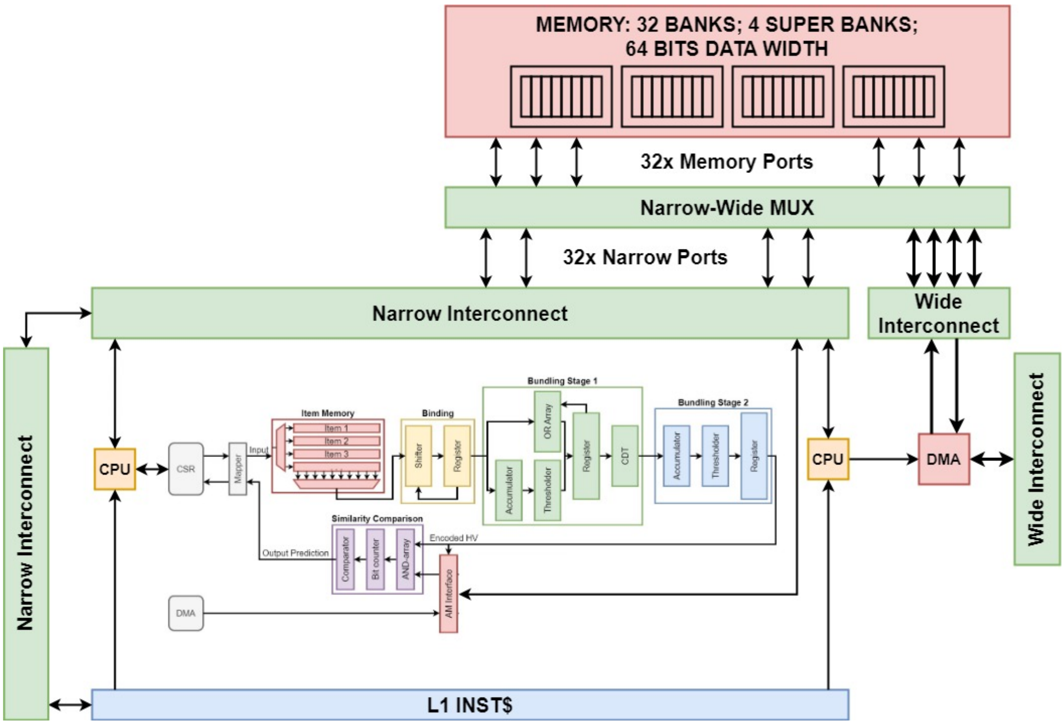


FIGURE 4.9: Visualization of the accelerator integrated with a PULP processor.

4.3.2 Testing the Functionality

The functionality of the entire encoder and similarity comparison is tested by simulating the character recognition and language recognition examples worked out in section 3.2. Both the encoded HV and the predicted output from the hardware are compared to those generated from the software implementations. If these match, the test passes; otherwise, it will log an error. This is done for all 26 letters for character recognition and for a simple 26-character sample for language recognition. Besides this, a "mixed testbench" tests the different possible combinations when combining functions from the various encoding schemes. Different cases for sliding window size, threshold values, etc. are tested, in combination with the activation/deactivation of various functions on a 20-character test sample. Additionally, loading the AM is tested using varying delay between loads, and varying base and max address.

Character Recognition

A first test aims at testing character recognition, with the CDT encoding scheme and a proper functioning associative memory over all loaded hypervectors. The training data (AM HVs) are generated in software, and only the testing happens on the hardware. The communication between the testbench or external processor and the accelerator happens through the control and status registers. Each time you want to write to these registers, the request valid and write enable signals have to be set high.

Reading these registers can happen at any time by setting the desired address and reading the “read_data_o” signal, without any requirement for other active signals. Before the test starts, the accelerator is reset by writing to the CSR_RESET field in the CSR. Afterward, the 26 encoded HVs, each representing a pixel grid with a different letter, are loaded into the AM through its interface. In practice, the loading can happen as long as the accelerator is not readying the AM, but it should normally happen before the start signal is given. Data can be loaded by applying a write enable, the load address, and the to-be-loaded data/HV simultaneously to the AM interface. Since the memory interface can handle 2048 bits at a time, each load only takes a single cycle, thus 26 clock cycles are necessary for this step. Next, the accelerator is programmed to the right settings by modifying the CSR. Shift binding mode, the OR-array, and the CDT function are turned on, the window size is set to 35, and the AM address range is set from 0 to 25*256 as 26 HVs of 256 bytes each have been loaded. Finally, the start signal is applied to the accelerator.

After these preparation steps, the actual character recognition test begins. For each of the 26 characters, the following steps are executed: Each pixel is iterated over; the input value is set to the pixel number, the shift amount is set to 0 or 1 depending on the pixel’s color, and the input valid signal is set high. At the moment that the CSR signals it is ready for a new request, this cycle repeats until all 35 pixels have been read. Then, the testbench continuously reads output valid in the CSR and waits until its value is high. When this happens, the encoded HV is checked to match with the software-generated version, and the generated output value (AM address) is also verified. If these both prove correct, the test is successful, otherwise, an error is logged. This is repeated for each of the 26 characters, to test the full range of the AM.

Language Recognition

A second test aims to test the language recognition functionality, with the thresholded sum encoding scheme. Similar to character recognition, a CSR reset is followed by the AM loading with the encoded HVs of 22 training languages from the Wortschatz Corpora. [24] (21 European languages and Afrikaans). To program the accelerator, sliding window mode, signature encoding mode, accumulator 1, and accumulator 2 are enabled. Then, the window size is set to 3, threshold 1 is put to 0.5, and 0.008 is written as the second threshold value. (Actually, 8 is written, but it is internally divided by 1024 as we can’t pass decimal numbers in binary.) Finally, the base AM address is assigned 256, and the max AM address is set to 21*256, as this is the range where the trained HVs got loaded. The base address is increased from 0 here, to test this functionality.

The test sample consists of 26 characters and is a sample randomly taken from the database of English test samples from [18]. After the start signal is applied, the input value is applied (0 for the letter "a", 1 for the letter "b", etc.), and input valid is set high for one clock cycle. The program then waits for the CSR signal that it is ready for the next request and applies the next input. When all 26 letters are given as input, the input done signal is written high in the CSR, and the program waits for the output to be valid. Finally, the encoded HV is compared to the software-generated encoded HV and the predicted AM address is checked for correctness.

Mixed Testing

The mixed testing section contains a number of possible scenarios, to test the functionality when combining different encoding schemes together or modifying their parameters. These tests have a different programming, but the input remains the same and consists of a 20-character sample in the English language. Different cases for sliding window size, threshold values, CDT k-factor, etc. are tested, in combination with the activation/deactivation of various functions. This includes different window sizes up to 12, different k-factors up to 3, only CDT in stage 1 bundling, only stage 2 bundling active, etc. The only metric to verify their functionality is a comparison of the encoded HV with a software implementation of the accelerator. This software implementation mimics the flow of the hardware and was proven to work by running the character and language recognition tests through it. The SystemVerilog code for the accelerator's test executions and mixed test settings from its testbench can be found in A.2.1.

In summary, these different tests in combination with additional visual verification through the simulation results are used to verify the accelerator's functionality. It is hard to create a complete testing set that covers all possible scenarios, but this implementation tests both the nominal values and edge cases.

4.4 Improvements for the Accelerator

In this section, we aim to redesign the accelerator a bit, to further improve its performance by reducing the latency, trimming unnecessary hardware, and reusing hardware where beneficial. We strive to create a low-latency design, with reasonable size and energy usage. The energy usage will already be low because we are using SDB-HDC, which is inherently very energy-efficient, so this isn't the main optimization objective. We can however lower the area and energy usage while maintaining the latency, by employing as much hardware reuse and trimming as possible.

4.4.1 Binding Improvements

The initial implementation uses a variable shifter, but the latency can be greatly reduced at the cost of an area increase by replacing this with a combinational shifter. That's why it is replaced by a fully combinational barrel shifter, capable of permuting the input HV anywhere from 0 to 63 times in a single cycle. We chose a maximum of 63 shifts, as the maximum window size for signature encoding is 12 and this can create a shift of $35 + 12 = 47$. Figure 4.10 shows the basic structure used to create this barrel shifter. This structure is modified to take 2,048 inputs and contains 6 layers of muxes, for a maximal total shift of $1 + 2 + 4 + 8 + 16 + 32 = 63$. The result of the permutation is then stored in a register, and applied as input to the bundling module.

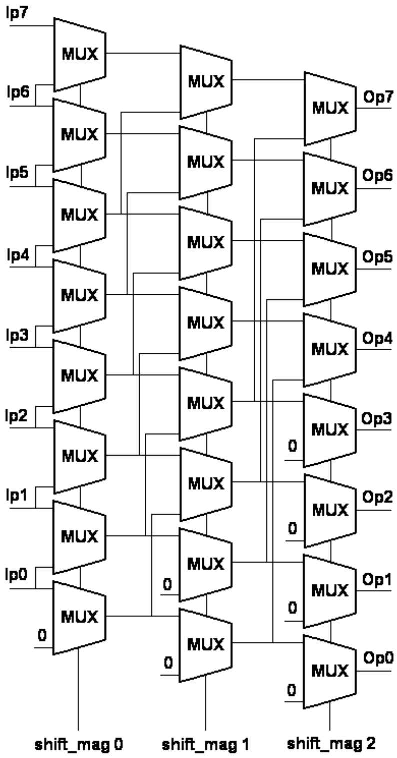


Figure 4.10: Example of a barrel shifter implementation. (retrieved from [1])

The item memory implementation is just kept as a LUT, as the synthesis already makes use of the sparsity to optimize this quite well. Additionally, the item memory only takes up 0.59% of the total accelerator's area footprint, thus optimizing this wouldn't make a significant difference.

4.4.2 Pipelining the Design

The main improvement we want to make for the accelerator is reducing the latency to be as low as possible. To this extent, we want the accelerator to accept inputs as quickly after each other as possible. We pipelined the design where necessary, so it can process 1 input per cycle for character recognition.

For language recognition with e.g. a window size of 3, the mapper produces 3 outputs before a new input enters the window. This means that every 3 cycles, a new input should be requested and processed for optimal speed. By modifying the mapper to be able to handle this, the bundling module receives a constant stream of 1 input per cycle, minimizing the total latency (without parallelizing inputs). The modifications necessary to make this work mainly consisted of tuning the control logic to make sure it was ready to receive a new input in time. This improvement greatly decreases the latency and by doing so, reduces the energy usage per execution, while barely increasing the area footprint.

Furthermore, the bit counter module was pipelined and made faster. The implementation was improved by making use of groups of adder trees. These are optimized by the synthesis tool and can compute a full bit count every cycle. The improved design not only meets timing but also improves the area cost for the bit counter module from 303 105 µm^2 to 113 923 µm^2.

4.4.3 Bundling Improvements

The bundling module can also be improved, using hardware reuse and trimming. First off, the disjunction used in the CDT encoding scheme was implemented using an OR array in parallel with the accumulator in the initial design. However, when the accumulation is done with a threshold of 1, this functions exactly like a disjunction / OR function. We can thus remove the OR array and use the accumulator with a threshold of 1 when we want to do disjunction instead. Table 4.2 shows the bundling section of the CSR after modifying it to remove the superposition bundling setting.

Table 4.2: New bundling section of the CSR register map.

CSR Name	Address	Bits	Description
CSR_P_BUNDLING	4	0	Enable stage 1 accumulator
		1	Enable CDT
		2	Enable stage 2 accumulator
		3-8	Stage 1 window size
		9-12	CDT k-factor
		13-15	Stage 1 threshold value
		16-22	Stage 2 threshold value
		23	Enable encoder writing to AM.

Further, the initial accelerator design contains some redundant registers in its bundling section. The register containing the result from the thresholding (and previously superposition) function can be removed, as the CDT implementation is fully combinational. On top of this, the slack proved high enough for the register containing the CDT result to be removed as well. Thus, the output from the stage 1 accumulator is directly fed through all thresholding and CDT logic, to either the accumulator in stage 2 bundling or the similarity comparison module. A similar principle can be applied to bundling stage 2, where we can remove the register containing the final encoded HV. This HV is only read by the similarity comparison after encoding has finished, which means that both accumulators retain their values. We can thus directly wire the values from the accumulators through the enabled logic to the similarity comparison input.

4.4.4 Summary

The improvements mentioned above have been implemented successfully, in combination with some extra changes to the control logic. These changes aim to minimize the latency as much as possible by e.g. not wasting cycles before data is read, when it is already ready. Furthermore, the switching of signals and registers is minimized wherever possible, to reduce the power draw and increase the energy efficiency. The full improved hardware architecture can be seen in Figure 4.11. After synthesizing the design and tweaking the clock frequency, the lowest clock period the improved design can support is $t_{CLK} = 8.5$ ns without running into timing errors. This gives a slight latency decrease compared to the 10 ns period the initial design uses, meaning it can run at a frequency of 117.65 MHz instead of 100 MHz. The next chapter will analyze its performance, and compare it to the naive design and other SOTA accelerators.

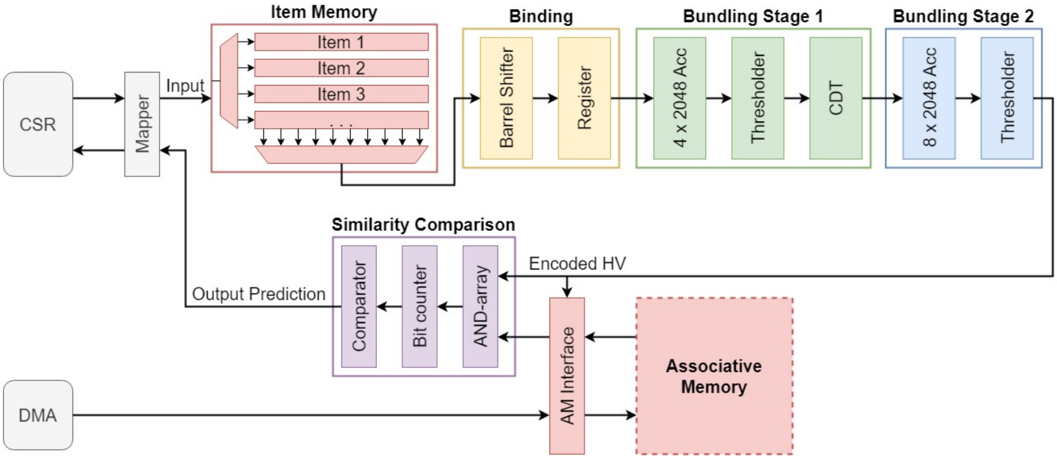


Figure 4.11: The improved programmable accelerator architecture.

4.5 Conclusion

In this chapter, we presented a comprehensive approach to designing a programmable accelerator for sparse distributed binary hyperdimensional computing (SDB-HDC). Our exploration of SDB-HDC algorithms in software provided deep insights into their inner workings and let us define improvements. The design process gave a detailed explanation of how the modules can be translated from software to hardware implementations and how DDB optimizations can be applied to this design. The different encoding schemes are neatly integrated with each other using hardware reuse and optimizations while maintaining the flexibility of matching their different functions together. Further improvements to the design, led to the creation of a flexible and efficient accelerator architecture. Afterward, an extensive testbench is used to verify the accelerator's functionality by testing its various modes in simulation. To conclude, this proposed programmable hardware accelerator design can advance the field of hyperdimensional computing by addressing the challenge of efficiently combining SDB-HDC's different encoding schemes.

Chapter 5

Performance Characteristics

This chapter will focus on the programmable accelerator's performance metrics. The design will be synthesized, and metrics like latency, area, and energy usage will be measured for both the initial architecture (which we will call the "naive design" here, and the improved architecture. These results will then be interpreted and compared to other SOTA models. We start by defining how we measure each metric, before comparing and interpreting them.

5.1 Measuring the Latency, Area, and Energy usage

The programmable accelerator's performance can be measured based on the simulation and synthesis results of the design. The simulation is done using SystemVerilog and Questasim and can give us the required number of clock cycles for each algorithm. The synthesis creates an ASIC using the Hammer flow[2], and the open source Skywater 130nm PDK[8]. We use commercial Cadence tools like Genus and Innovus, and simulation tool VCS Synopsys. Timing, area, and power reports will be created for the synthesized design. Based on the slack from the timing report, we can increase the clock frequency if we have positive slack and decrease it for negative slack, to avoid setup or hold errors. The area report can give an area footprint for each module, while the power report gives a result for the entire accelerator based on the average power used during the simulation.

The performance metrics we define for this design, to compare it to other SOTA designs are execution time per application, area usage, average power usage, and energy usage per application. We strive to create a low-latency design, with reasonable size and energy usage. This order of importance is chosen, as an SDB-HDC already makes use of simple operations that inherently have a small area footprint and low energy usage.

To retrieve the latency, we perform a simulation of the design using Questasim. The clock cycles (cc) are counted from the moment the first input is applied to when the output is valid. This value can then be multiplied by the clock period (time per clock cycle) to give the total latency per execution. The latency for character recognition is independent of the input, as the number of inputs remains constant

(35 - the number of pixels). However, for the language recognition and mixed tests, the total latency is directly proportional to the number of input characters.

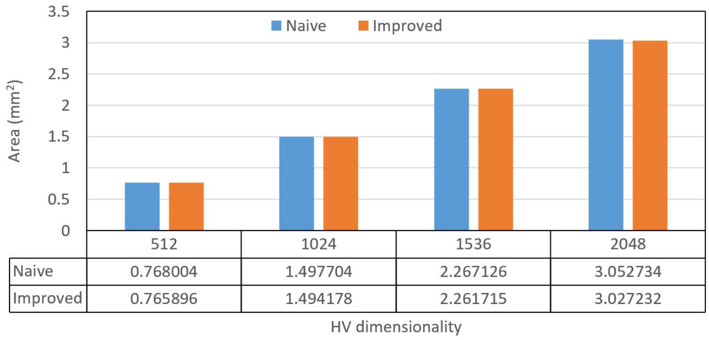
The synthesis gives the area result for each of the different modules and the total design in square micrometers. A power calculation on the synthesis result produces the total average power in milliwatt for each module separately and for the entire design, based on the switching from the testbench. We can calculate the average power used by each of the test scenarios (character, language, mixed, and all) by selectively enabling and disabling them. The required energy per algorithm can then be calculated by multiplying the average power and the latency for that application. Important to note is that the reset, loading step, and initial CSR programming from the testbench are never included in the power calculations during synthesis, as this would only have to happen once per different application. The power usage is measured from the accelerator start to the output valid signal.

As we want to focus on the accelerator data path, the top module in the testbench for synthesis only calls the main accelerator and excludes the associative memory. The memory itself is separately included in the testbench to verify the accelerator's functionality during simulation, but it's not included as input for the synthesis so it won't count toward the area and power usage metrics.

5.2 Characterizing the Performance

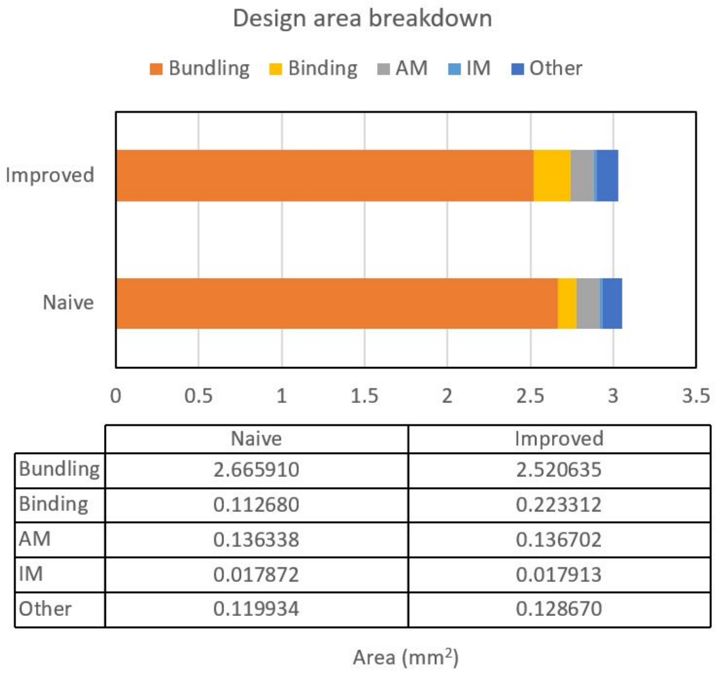
5.2.1 Area Characterization

The area for the programmable accelerator based on its dimensionality is given in Figure 5.1. This figure shows how the small improvements by removing redundant registers have more effect toward higher dimensionalities. The reason why this effect isn't larger is that a barrel shifter was introduced to replace the variable shifter, which doubles the size of the binding module. This effect can be seen in the area breakdown in Figure 5.2. The bundling module takes up nearly all of the area (83+%), due to its massive accumulators. Note that *AM* includes the similarity comparison and interface, but not the memory. These couldn't be optimized further without reducing the accelerator's flexibility/capability, as the naive design already implements saturating accumulators.



	512	1024	1536	2048
Naive	0.768004	1.497704	2.267126	3.052734
Improved	0.765896	1.494178	2.261715	3.027232

FIGURE 5.1: Total area comparison based on dimensionality.



	Naive	Improved
Bundling	2.665910	2.520635
Binding	0.112680	0.223312
AM	0.136338	0.136702
IM	0.017872	0.017913
Other	0.119934	0.128670

FIGURE 5.2: Breakdown of the area usage per module for D=2048.

5.2.2 Latency Characterization

The main optimization objective for this programmable accelerator architecture was latency. The improvements to the binding module and control algorithms in combination with a full pipelining of the design allow it to process 1 HV every clock cycle (cc) instead of taking 3 or more cycles. Because of this, in regular mode, it can take 1 input/cc and in sliding window mode with window size n, it can take 1 input per n cc (as n HVs have to be processed per input). Figure 5.3 shows the latency improvements made by this improvement in combination with a reduction of the clock period to 8.5 ns. This is especially noticeable for language recognition, where the latency is reduced by over 12x. These improvements make the final design very fast for its 130 nm technology node.

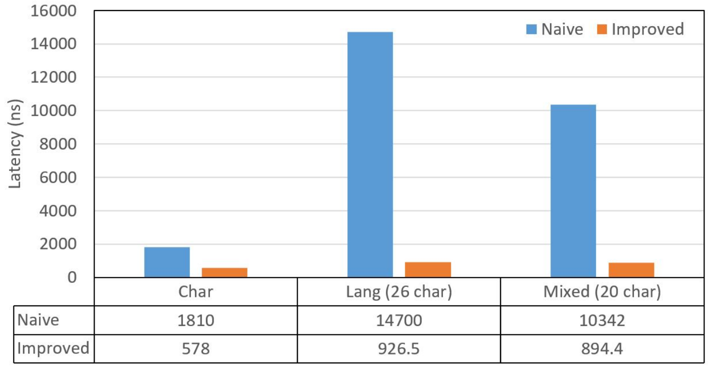


Figure 5.3: Comparison between both designs of the average latency for each test.

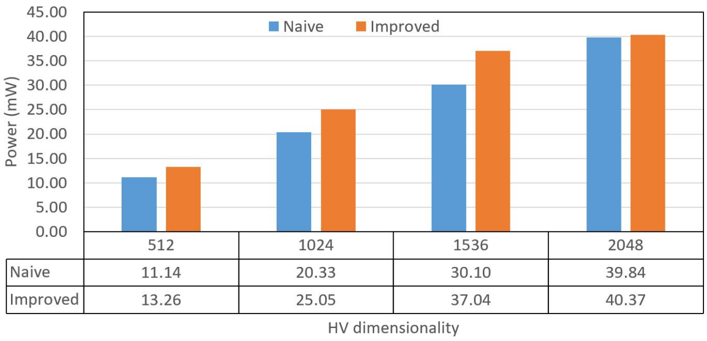
Table 5.1 shows how many clock cycles are taken by the encoding and classification steps for the character and language recognition tests. The latency numbers are computed on the right, by multiplying these with the clock period. "Input data" refers to the number of cycles during which the processor is continuously requesting data, while "classification" refers to the number of cycles it takes to read the AM data.

Table 5.1: Latency breakdown for character and language recognition.

Test	Char [cc]	Lang [cc]	Char [ns]	Lang [ns]
Input data	35	300	297.5	2550
Classification	26	22	221	187
Other	7	12	59.5	102
Total	68	334	578	2839

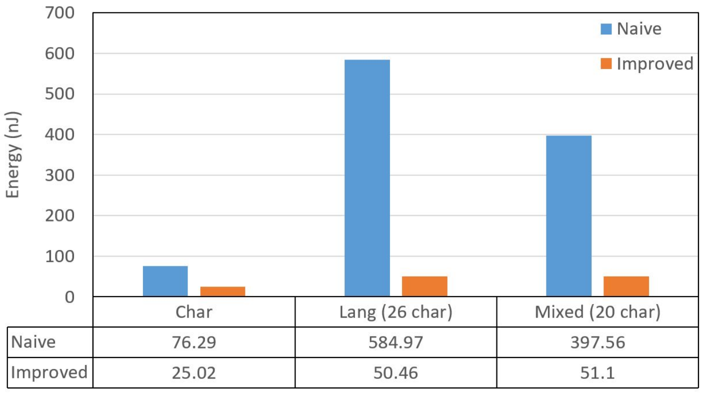
5.2.3 Power and Energy Characterization

The main benefit of using SDB-HDC over any other algorithm is its extremely low energy usage. Figure 5.4 shows the average power usage when running all tests, based on the accelerator's dimensionality. We can see that the naive design actually slightly lower power usage for a low HV size, but that the improved design beats it when going to higher sizes. This is most likely the case due to the added overhead to reduce the improved design's latency. The overhead doesn't scale with dimensionality, thus it makes up a bigger portion of the area and power usage for smaller sizes. Notably, the improved design manages to keep the power usage relatively similar to the naive design, despite greatly reducing the latency. This means that the energy usage per prediction will be significantly lower for the improved design, as can be seen in Figure 5.5.



	512	1024	1536	2048
Naive	11.14	20.33	30.10	39.84
Improved	13.26	25.05	37.04	40.37

FIGURE 5.4: Comparison of the total average power usage based on dimensionality.



	Char	Lang (26 char)	Mixed (20 char)
Naive	76.29	584.97	397.56
Improved	25.02	50.46	51.1

FIGURE 5.5: Overview of the energy usage per prediction for D=2048.

Finally Figure 5.6 shows a breakdown of the energy usage per module. This time language recognition uses a sample sentence of 100 characters, as we will use this to compare it to similar DDB accelerators.

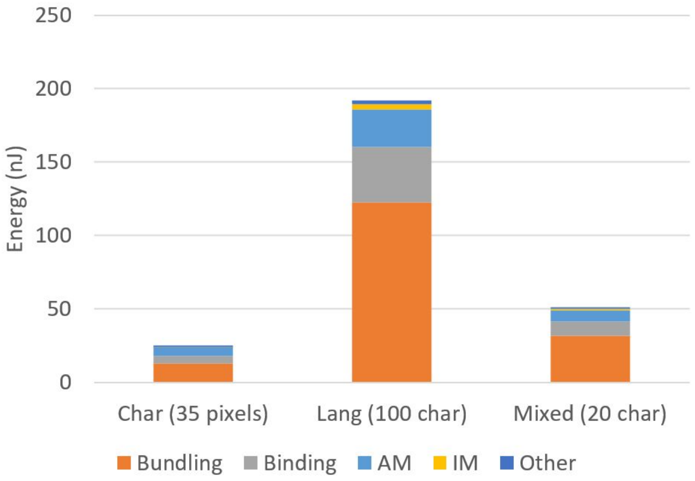


Figure 5.6: Breakdown of the energy usage for the improved design and D=2048.

5.2.4 Comparison to SOTA

It is hard to compare this work to other state-of-the-art accelerators, as currently no other programmable SDB-HDC accelerators exist. We can best compare it to the programmable DDB-HDC accelerator from Datta et al. [5], as this work also uses a dimensionality D=2,048. However, this work uses the TSMC 28nm High-K/Metal-gate (HKMG) node, which is a much smaller process than the Skywater 130nm we use. A comparison of the main specifications can be seen in Table 5.2. The total estimated power here is the average power used during the language recognition test with a 100-character input sentence. Note that the smaller process node allows for a significantly smaller cell area and clock period. Further, the item memory used in this design is significantly larger, taking up 62% of its total area, while it only takes up about 6% for our design. A larger item memory makes the design more flexible but is unnecessary for most applications.[14, 10, 12, 11] Hypervectors that are unique from those in the IM can also be created by applying a unique permutation to them before bundling. The permutation that should be applied to the HV at the input value is written to the CSR together with this input. Additionally, if a larger IM would be necessary, the design can easily be scaled up as all other hardware besides the input size doesn't need to scale.

We can see that our design is a little over twice the size and has a lower clock frequency, which is normal given the difference in process node sizes. Further, the total number of cells is far lower, mainly due to the smaller item memory used. Importantly though, our design's average estimated power usage is far lower, highlighting the main benefit of using SDB-HDC. Do note that the power used by the memory storing the AM data is not included for our design, bringing down our numbers slightly.

Another DDB-HDC programmable accelerator we can compare the design to is created by Eggimann et al. [6]. What's unique about this design, is that it uses HV

TABLE 5.2: Performance comparison with [5].

Property	DDB design	This design
Technology	TSMC 28 HPM	Skywater 130
Total Cell Area	1.27 sq. mm.	3.03 sq. mm.
t_{CLK}	2.4 ns	8.5 ns
Total estimated power	267 mW	67.7 mW
Total number of Cells	4.73×10^5	1.93×10^5
Number of Buf/Inv Cells	37501	15377

rematerialization for its item memory. This means that instead of using a ROM or LUT to store pre-generated HVs, they are generated on the fly through mixing and permutations. Doing so significantly reduces area, at the cost of latency. The area for their 65 nm implementation is 1.43 mm^2, and for their 22 nm implementation only 0.29 mm^2, which is much lower than our design, but this is also due to the smaller process node.

The DDB language recognition application described in section 2.3 and compared to SDB in 3.2.2 has also been tested on this accelerator, so we can use it to compare the performance. Their paper classifies a sentence as 100 characters long, and it takes 14 cycles to process 1 input character, thus 1400 cycles per sentence. We can test our accelerator with a language recognition sample of 100 characters long as well, to compare the energy per prediction. The results comparing this design and Datta et al.'s work[5] with our design are shown in Table 5.3. Note that the SDB version achieves the best results with a 3-gram sliding window, while the DDB maximizes accuracy for a 4-gram, which makes this version inherently have a higher latency. Eggimann et al.'s work has significantly lower average power, but this is compensated by their low clock frequency. We can see that our work has the lowest energy per prediction, highlighting the benefit of using SDB-HDC. Additionally, our design only needs 334 clock cycles (2839 ns) to perform the prediction for a single sentence (100 characters), which is a lot better than the 1400 cycles used by [6] due to their rematerialization.

TABLE 5.3: Performance comparison for language recognition with [5] and [6].

Design	Technology	Power [mW]	Clock Frequency	Energy/test [nJ]
Eggimann et al.	65 nm	86.5×10^{-3}	100 kHz	1205
Eggimann et al.	GF22	23.7×10^{-3}	100 kHz	332
Datta et al.	TSMC28	267	416.67 MHz	250
This work	**Skywater130**	**67.7**	**117.65 MHz**	**192**

5.3 Conclusion

The performance evaluation of our programmable accelerator for SDB-HDC has shown the significant improvements in latency and energy made by optimizing the design. The optimized design manages to substantially reduce the latency while maintaining the power usage and area footprint, leading to major energy savings. When comparing our design to other state-of-the-art programmable accelerators, it manages to outperform similar designs in lower technology nodes on energy per prediction and maintain a competitive area usage. These results show the viability of our SDB-HDC design and its effectiveness in integrating the encoding schemes with each other in a flexible, yet efficient accelerator.

Chapter 6

Conclusion

6.1 Conclusion

Hyperdimensional computing (HDC) is a computing platform to execute machine learning tasks with high energy efficiency for IoT applications. Sparse distributed binary HDC (SDB-HDC) is a subset of this using sparse hypervectors to achieve an even lower power usage while maintaining around the same accuracy. This thesis brings two major contributions to the SDB-HDC field.

The first contribution is the introduction of the sorted THR function into the thresholded sum encoding scheme. By applying sorted THR after accumulation in the final bundling step during training, the density of the resulting hypervectors can be controlled much better. The ability to create trained hypervectors with a consistent density for storage in the associative memory leads to remarkable accuracy gains, especially for lower dimensionalities. Training using the sorted THR encoding led to a 2-20% gain in accuracy during testing for the SDB language recognition application, as can be seen in Table 6.1. This combats the downside of SDB-HDC usually having slightly lower accuracy than its dense distributed counterpart and can lead to more widespread adoption of its algorithms.

Table 6.1: Language recognition accuracy comparison for varying implementation and dimensionality.

	N-gram	2,000 bits	4,000 bits	6,000 bits	8,000 bits	10,000 bits
Dense	3	69.4%	86.4%	90.5%	92.1%	96.1%
Sparse	4	74.2%	86.5%	91.5%	93.6%	95.4%
Sorted THR	3	**95.1%**	**97.6%**	**98.0%**	**98.5%**	**98.7%**

The second contribution was the main goal of this work: the design of a programmable hardware accelerator that can run SDB-HDC IoT applications. The creation of this programmable accelerator solves the problem of SDB-HDC having a variety of non-generalized, application-specific encoding schemes. It does this by defining the most common encoding scheme's functions and making them more generalized. The context-dependent thinning and thresholded sum encoding schemes are integrated with each other. Through hardware reuse and module optimizations, an efficient yet flexible programmable accelerator architecture is created. Its main specifications can be viewed in Table 6.2.

Table 6.2: Performance specifications of the programmable SDB-HDC accelerator.

Property	Value
Technology	Skywater 130
Total Cell Area	3.03 sq. mm.
t_{CLK}	8.5 ns
Total estimated power	67.7 mW
Total number of Cells	1.93×10^5
Number of Buf/Inv Cells	15377

This accelerator is capable of performing various combinations of the functions found in the major encoding schemes. It is able to achieve good accuracy results on par with HDC software implementations, at a fast speed comparable to other SOTA implementations. It does this while minimizing the area usage and achieving a major energy efficiency increase compared to traditional machine learning methods due to the use of SDB-HDC.

To conclude, this work presents two major contributions to overcome the difficulties faced when implementing SDB-HDC algorithms and building their hardware. The sorted THR encoding to increase accuracy and access to a programmable ultra-low-power SDB-HDC accelerator allows for a better research space for these algorithms and their integration into IoT devices. The codebase for this work can be found on GitHub: `https://github.com/TimDeboel/SDB-HDC-accelerator`.

6.2 Recommended Studies

A lot of work has yet to be done to lead to a widespread adoption of SDB-HDC. First and foremost, more applications have to be converted from DDB to SDB, and shown to achieve good accuracy while reducing energy usage even further. This work has introduced the Sorted THR method for the thresholded sum encoding scheme to aid the accuracy problem. This method is very promising and should be investigated further, to check its effectiveness when implemented in other applications. The sorted THR can potentially further boost their accuracy, and make SDB more viable, leading to a more widespread adoption. If its implementation has a similar accuracy improving effect it had to the SDB language recognition for more applications, this will be an enormous benefit to the SDB-HDC research space. An accuracy boost of this magnitude is a major increase to SDB-HDC's effectiveness in performing accurate, low-energy machine learning applications on IoT devices.

Besides this, the design of this programable SDB-HDC accelerator has opened up new possibilities and challenges. The integration of the various encoding schemes produces a flexible hardware accelerator, capable of speeding up further research. This accelerator's architecture can for example be used as a stepping stone for future hardware developments within this space. A good example is the conversion of DDB hardware optimizations to SDB, or creating new SDB hardware optimizations. Barely any research has been done toward creating or improving this kind of hardware, so there are still a lot of opportunities for improvement in this field.

Appendices

Appendix A

Software code

A.1 Applications in Software

A.1.1 HV generation

```
import numpy as np
import os

###############################################
# Set parameters
###############################################
D = 1000
M = D/2      # Number of ones
print("Density:␣" + str(M/D))

def u_gen_rand_hv(D):

    # Sanity checker
    if (D % 2):
        print("Error␣–␣D␣can't␣be␣an␣odd␣number")
        return 0

    hv = np.zeros(D, dtype = int)
    indices = np.random.permutation(D)

    hv[indices >= M] = 0
    hv[indices < M] = 1

    return hv
```

A.2 Hardware Accelerator

A.2.1 Testbench Tasks

```
// Tests
initial begin
#‘RESET_TIME
rst_ni <= 0;
#‘CLK_PERIOD;
rst_ni <= 1;
#‘CLK_PERIOD;

load_am(2'd0);
char_recognition();

reset();
load_am(2'd1);

lang_recognition();

#100ns;

h = $fopen("../tb/data/mixed_data_encoded_2k.txt", "r");
g = $fopen("../tb/tb_output_mixed.txt", "w");

reset();
$fwrite(g, "Lang␣with␣other␣input␣and␣thr\n");
mixed_testing(
        .g(g),
        .h(h),
        .sliding_window_mode_in(1'b1),
        .signature_encoding_mode_in(1'b1),
        .acc1_mode_in(1'b1),
        .acc2_mode_in(1'b1),
        .window1_size_in(6'd3),
        .or_mode_in(1'b0),
        .cdt_mode_in(1'b0),
        .cdt_k_factor_in(4'd1),
        .thr1_val_in(3'd2),
        .thr2_val_in(7'd100),
        .write_enc_am(1'b1)
);

reset();
$fwrite(g, "Max␣sliding␣window␣size\n");
```

```
mixed_testing(
        .g(g),
        .h(h),
        .sliding_window_mode_in(1'b1),
        .signature_encoding_mode_in(1'b1),
        .acc1_mode_in(1'b1),
        .acc2_mode_in(1'b1),
        .window1_size_in(6'd12),
        .or_mode_in(1'b0),
        .cdt_mode_in(1'b0),
        .cdt_k_factor_in(4'd1),
        .thr1_val_in(3'd2),
        .thr2_val_in(7'd100),
        .write_enc_am(1'b0)
);

reset();
$fwrite(g, "Sliding␣window␣without␣signature␣encoding\n");
mixed_testing(
        .g(g),
        .h(h),
        .sliding_window_mode_in(1'b1),
        .signature_encoding_mode_in(1'b0),
        .acc1_mode_in(1'b1),
        .acc2_mode_in(1'b1),
        .window1_size_in(6'd4),
        .or_mode_in(1'b0),
        .cdt_mode_in(1'b0),
        .cdt_k_factor_in(4'd1),
        .thr1_val_in(3'd2),
        .thr2_val_in(7'd20),
        .write_enc_am(1'b0)
);

reset();
$fwrite(g, "Other␣window␣and␣or_mode␣instead␣of␣acc1\n");
mixed_testing(
        .g(g),
        .h(h),
        .sliding_window_mode_in(1'b1),
        .signature_encoding_mode_in(1'b1),
        .acc1_mode_in(1'b0),
        .acc2_mode_in(1'b1),
        .window1_size_in(6'd2),
        .or_mode_in(1'b1),
        .cdt_mode_in(1'b0),
        .cdt_k_factor_in(4'd1),
        .thr1_val_in(3'd2),
        .thr2_val_in(7'd100),
        .write_enc_am(1'b0)
);

reset();
$fwrite(g, "or_mode␣and␣cdt_mode␣followed␣by␣acc2\n");
mixed_testing(
        .g(g),
        .h(h),
        .sliding_window_mode_in(1'b1),
        .signature_encoding_mode_in(1'b1),
        .acc1_mode_in(1'b0),
        .acc2_mode_in(1'b1),
        .window1_size_in(6'd2),
        .or_mode_in(1'b1),
        .cdt_mode_in(1'b1),
        .cdt_k_factor_in(4'd1),
        .thr1_val_in(3'd2),
        .thr2_val_in(7'd50),
        .write_enc_am(1'b0)
);

reset();
$fwrite(g, "Only␣cdt,␣no␣bundling␣in␣stage1\n");
mixed_testing(
        .g(g),
        .h(h),
        .sliding_window_mode_in(1'b0),
        .signature_encoding_mode_in(1'b0),
        .acc1_mode_in(1'b0),
        .acc2_mode_in(1'b1),
        .window1_size_in(6'd1),
        .or_mode_in(1'b0),
        .cdt_mode_in(1'b1),
        .cdt_k_factor_in(4'd1),
        .thr1_val_in(3'd2),
        .thr2_val_in(7'd50),
        .write_enc_am(1'b0)
);

reset();
$fwrite(g, "Only␣stage␣2␣bundling,␣no␣bundling␣in␣stage1\n");
```

```
mixed_testing(
        .g(g),
        .h(h),
        .sliding_window_mode_in(1'b0),
        .signature_encoding_mode_in(1'b0),
        .acc1_mode_in(1'b0),
        .acc2_mode_in(1'b1),
        .window1_size_in(6'd1),
        .or_mode_in(1'b0),
        .cdt_mode_in(1'b0),
        .cdt_k_factor_in(4'd1),
        .thr1_val_in(3'd2),
        .thr2_val_in(7'd100),
        .write_enc_am(1'b0)
);

reset();
$fwrite(g, "Only␣s1␣cdt,␣s2␣bundling,␣with␣cdt_k␣changed\n");
mixed_testing(
        .g(g),
        .h(h),
        .sliding_window_mode_in(1'b0),
        .signature_encoding_mode_in(1'b0),
        .acc1_mode_in(1'b0),
        .acc2_mode_in(1'b1),
        .window1_size_in(6'd1),
        .or_mode_in(1'b0),
        .cdt_mode_in(1'b1),
        .cdt_k_factor_in(4'd3),
        .thr1_val_in(3'd2),
        .thr2_val_in(7'd50),
        .write_enc_am(1'b0)
);

reset();
$fwrite(g, "or_mode␣and␣cdt_mode␣followed␣by␣acc2,␣cdt_k␣changed\n");
mixed_testing(
        .g(g),
        .h(h),
        .sliding_window_mode_in(1'b1),
        .signature_encoding_mode_in(1'b1),
        .acc1_mode_in(1'b0),
        .acc2_mode_in(1'b1),
        .window1_size_in(6'd2),
        .or_mode_in(1'b1),
          .cdt_mode_in(1'b1),
          .cdt_k_factor_in(4'd2),
          .thr1_val_in(3'd2),
          .thr2_val_in(7'd50),
          .write_enc_am(1'b0)
);


$fclose(g);
$fclose(h);


#100ns;
$stop;
$finish;

end
```

Bibliography

[1] Barrel shifter. https://esrd2014.blogspot.com/p/barrel-shifter.html.

[2] Introduction to hammer. https://hammer-vlsi.readthedocs.io/.

[3] Abbas-Rahimi. Github - abbas-rahimi/hdc-language-recognition: Hyperdimensional computing for language recognition: Matlab and rtl implementations.

[4] S. Basu, R. E. Bryant, G. De Micheli, T. Theis, and L. Whitman. Nonsilicon, non-von neumann computing-part i [scanning the issue]. *Proceedings of the IEEE*, 107(1):11–18, 2019.

[5] S. Datta, R. A. G. Antonio, A. R. S. Ison, and J. M. Rabaey. A programmable hyper-dimensional processor architecture for human-centric iot. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(3):439–452, 2019.

[6] M. Eggimann, A. Rahimi, and L. Benini. A 5 uw standard cell memory-based configurable hyperdimensional computing accelerator for always-on smart sensing. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(10):4116–4128, 2021.

[7] R. W. Gayler. Multiplicative binding, representation operators & analogy. *ResearchGate*, Jan. 1998.

[8] google. Github - google/skywater-pdk: Open source process design kit for usage with skywater technology foundry's 130nm node.

[9] M. Hersche, E. M. Rella, A. Di Mauro, L. Benini, and A. Rahimi. Integrating event-based dynamic vision sensors with sparse hyperdimensional computing: a low-power accelerator with online learning capability. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '20, page 169–174, New York, NY, USA, 2020. Association for Computing Machinery.

[10] M. Imani, J. Hwang, T. Rosing, A. Rahimi, and J. M. Rabaey. Low-power sparse hyperdimensional encoder for language recognition. *IEEE Design & Test*, 34(6):94–101, 2017.

[11] M. Imani, D. Kong, A. Rahimi, and T. Rosing. Voicehd: Hyperdimensional computing for efficient speech recognition. In *2017 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8, 2017.

[12] M. Imani, T. Nassar, A. Rahimi, and T. Rosing. Hdna: Energy-efficient dna sequencing using hyperdimensional computing. In *2018 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI)*, pages 271–274, 2018.

[13] P. Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation*, 1(2):139–159, Jan. 2009.

[14] D. Kleyko, E. Osipov, and D. A. Rachkovskij. Modification of holographic graph neuron using sparse distributed representations. *Procedia Computer Science*, 88:39–45, 2016. 7th Annual International Conference on Biologically Inspired Cognitive Architectures, BICA 2016, held July 16 to July 19, 2016 in New York City, NY, USA.

[15] D. Kleyko, E. Osipov, A. Senior, A. I. Khan, and Y. A. Şekercioğğlu. Holographic graph neuron: A bioinspired architecture for pattern processing. *IEEE Transactions on Neural Networks and Learning Systems*, 28(6):1250–1262, 2017.

[16] D. Kleyko, D. A. Rachkovskij, E. Osipov, and A. Rahimi. A survey on hyperdimensional computing aka vector symbolic architectures, part I: models and data transformations. *CoRR*, abs/2111.06077, 2021.

[17] D. Kleyko, A. Rahimi, D. A. Rachkovskij, E. Osipov, and J. M. Rabaey. Classification and recall with binary hyperdimensional computing: Tradeoffs in choice of density and mapping characteristics. *IEEE Transactions on Neural Networks and Learning Systems*, 29(12):5880–5898, 2018.

[18] P. Koehn. Europarl: A parallel corpus for statistical machine translation. https://www.statmt.org/europarl/, 2005.

[19] M. Laiho, J. H. Poikonen, P. Kanerva, and E. Lehtonen. High-dimensional computing with sparse vectors. In *2015 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 1–4, 2015.

[20] A. Moin, A. Zhou, A. Rahimi, S. Benatti, A. Menon, S. Tamakloe, J. Ting, N. Yamamoto, Y. Khan, F. Burghardt, L. Benini, A. C. Arias, and J. M. Rabaey. An emg gesture recognition system with flexible high-density sensors and brain-inspired high-dimensional classifier. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

[21] P. Neubert, S. Schubert, and P. Protzel. An introduction to hyperdimensional computing for robotics. *KI. Künstliche Intelligenz/KI - Künstliche Intelligenz*, 33(4):319–330, Sept. 2019.

[22] M. Overdeep. 35 large dog breeds that make the best pets, Mar. 2024.

[23] P. Platform. Pulp faqs. https://pulp-platform.org/.

[24] U. Quasthoff, M. Richter, and C. Biemann. Corpus portal for search in monolingual corpora. In N. Calzolari, K. Choukri, A. Gangemi, B. Maegaard, J. Mariani, J. Odijk, and D. Tapias, editors, *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, Genoa, Italy, May 2006. European Language Resources Association (ELRA).

[25] D. Rachkovskij. Representation and processing of structures with binary sparse distributed codes. *IEEE Transactions on Knowledge and Data Engineering*, 13(2):261–276, 2001.

[26] D. A. Rachkovskij and E. M. Kussul. Binding and normalization of binary sparse distributed representations by context-dependent thinning. *Neural Computation*, 13(2):411–452, Feb. 2001.

[27] D. A. Rachkovskij and S. O. Slipchenko. Similarity-based retrieval with structure-sensitive sparse binary distributed representations. *Computational Intelligence*, 28(1):106–129, Feb. 2012.

[28] A. Rahimi, S. Datta, D. Kleyko, E. P. Frady, B. Olshausen, P. Kanerva, and J. M. Rabaey. High-dimensional computing as a nanoscalable paradigm. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(9):2508–2521, 2017.

[29] A. Rahimi, P. Kanerva, and J. M. Rabaey. A robust and energy-efficient classifier using brain-inspired hyperdimensional computing. *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016.

[30] K. Schlegel, P. Neubert, and P. Protzel. A comparison of vector symbolic architectures. *Artificial Intelligence Review*, 55(6):4523–4555, Dec. 2021.

[31] M. Schmuck, L. Benini, and A. Rahimi. Hardware optimizations of dense binary hyperdimensional computing: rematerialization of hypervectors, binarized bundling, and combinational associative memory. *arXiv (Cornell University)*, Jan. 2018.

[32] C. Staff. What is artificial intelligence? definition, uses, and types, Mar. 2024.