# Configurable Hardware Acceleration for Hyperdimensional Computing Extension on RISC-V

Rocco Martino[1], Marco Angioli[1], Antonello Rosato[1], Marcello Barbirotta[1], Abdallah Cheikh[1], and Mauro Olivieri[1]

[1]Dept. of Information Engineering, Electronics and Telecommunications, Sapienza University of Rome

December 05, 2024



**Abstract**

Hyperdimensional Computing (HDC) is a brain-inspired computing paradigm that models information using high-dimensional distributed representations called hypervectors (HVs). HDC leverages parallel and simple vector arithmetic operations to combine and compare different concepts, emerging as a lightweight alternative for performing AI learning tasks on resource-constrained devices and as an ideal candidate for hardware implementations. In this work, we present a highly flexible hardware acceleration unit designed to optimize the execution time of HDC learning tasks. Integrated into the execution stage of the Klessydra T03 RISC-V core, the unit accelerates the core arithmetic operations on binary HVs and can be configured at synthesis time in terms of hardware parallelism, supported operations and size of the local memories, trading off execution time with hardware resources to meet the demand of different applications. A custom RISC-V Instruction Set Extension is designed to efficiently control the accelerator, with instructions fully integrated into the GCC compiler chain and exposed to the programmer as intrinsic function calls. Dedicated Control Status Registers allow users to specify the characteristics of the high-dimensional space and the target learning tasks at runtime, controlling the hardware loops of the accelerator and enabling the same hardware architecture to be used for various tasks. The dual flexibility coming from hardware configuration and software programmability sets this work apart from application-specific solutions in the literature, offering a unique, versatile accelerator adaptable to a wide range of applications and learning tasks.

# Configurable Hardware Acceleration for Hyperdimensional Computing Extension on RISC-V

Rocco Martino, Marco Angioli, Antonello Rosato, Marcello Barbirotta, Abdallah Cheikh, Mauro Olivieri
Dept. of Information Engineering, Electronics and Telecommunications,
Sapienza University of Rome, Italy,
Email: {name.surname}@uniroma1.it

***Abstract*—Hyperdimensional Computing (HDC) is a brain-inspired computing paradigm that models information using high-dimensional distributed representations called hypervectors (HVs). HDC leverages parallel and simple vector arithmetic operations to combine and compare different concepts, emerging as a lightweight alternative for performing AI learning tasks on resource-constrained devices and as an ideal candidate for hardware implementations. In this work, we present a highly flexible hardware acceleration unit designed to optimize the execution time of HDC learning tasks. Integrated into the execution stage of the Klessydra T03 RISC-V core, the unit accelerates the core arithmetic operations on binary HVs and can be configured at synthesis time in terms of hardware parallelism, supported operations and size of the local memories, trading off execution time with hardware resources to meet the demand of different applications. A custom RISC-V Instruction Set Extension is designed to efficiently control the accelerator, with instructions fully integrated into the GCC compiler chain and exposed to the programmer as intrinsic function calls. Dedicated Control Status Registers allow users to specify the characteristics of the high-dimensional space and the target learning tasks at runtime, controlling the hardware loops of the accelerator and enabling the same hardware architecture to be used for various tasks. The dual flexibility coming from hardware configuration and software programmability sets this work apart from application-specific solutions in the literature, offering a unique, versatile accelerator adaptable to a wide range of applications and learning tasks.**

***Index Terms*—Hyperdimensional Computing, Hardware Acceleration, RISC-V, Embedded AI, FPGA**

## I. Introduction

Hyperdimensional Computing (HDC), also known as Vector Symbolic Architectures (VSA), is a lightweight computing paradigm especially promising for learning tasks on resource-constrained edge-devices. HDC relies on the mathematical properties of high-dimensional vector spaces and models information using high-dimensional distributed representations, denoted as Hypervectors (HVs), with independent and identically distributed components [1]–[3]. HVs are manipulated and combined using a set of well-defined simple vector arithmetic operations, and are compared through a similarity measure. These operations form the core computational structure of a VSA. In recent years, HDC has been applied on a variety of contexts, such as robotics [4], recommendation systems [5], healthcare [6]–[9], natural language processing [10], [11], DNA sequencing [12], and speech recognition [13], reservoir computing [14] for modelling and explaining traditional Machine Learning models [15] and solving classification [16], clustering [17], [18], and regression [19] problems. In all these domains, HDC has demonstrated comparable or better performance than traditional Artificial Intelligence (AI) approaches while having distinct and concrete advantages in terms of computational complexity and energy efficiency [1], [20]. Furthermore, in diverse applications, HDC-based solutions have proven to be scalable and extremely parallelizable [2], [3]. As a whole, such characteristics can make HDC an efficient and light-weight alternative for performing learning tasks in IoT applications on the edge [20], [21] and align well with the capabilities of Field-Programmable Gate Arrays (FPGAs) [22], offering significant opportunities for hardware acceleration [1].

Many works in the literature have proposed hardware solutions for HDC on both FPGA [7], [12], [23] and Application Specific Integrated Circuit (ASIC) [24], [25]. However, these architectures often rely on fixed design choices tailored to the target learning task, dataset, size of the HVs and constraints on hardware resources and energy consumption, lacking the flexibility needed to adapt to different contexts. As a result, the literature currently does not provide general-purpose hardware solutions that can be adapted by the user to efficiently support the execution of a wide range of HDC tasks and satisfy the requirements of the problem at hand.

This work aims to fill this gap by introducing the Hyperdimensional Coprocessor Unit (HDCU), an innovative highly flexible hardware accelerator that optimizes the core arithmetic operations on binary HVs of any size while meeting the specific constraints of different target applications through hardware configuration and software programmability. Integrated into the execution stage of a RISC-V core, the proposed unit acts like a coprocessor including highly optimized functional units for each basic HV operation along with dedicated local memories. The unit is configurable at synthesis time in amount of hardware parallelism, supported operations, and local memories' sizes, and it allows setting the size of the HVs at runtime. The unit is controlled by a custom RISC-V Instruction Set Extension (ISE), covering the essential HDC primitive operations, which has been designed, fully integrated in the GCC compiler toolchain, and made accessible to the software programmer via simple intrinsic function calls.

The main contributions of this work are:

- Introducing a flexible hardware accelerator for HDC learning tasks, integrated into a RISC-V core and configurable at synthesis time;

- Proposing a RISC-V ISE that covers the fundamental operations of HDC, enabling easy and intuitive acceleration of various learning tasks within a unified architecture;
- Characterizing the hardware requirements of the accelerator by synthesis on FPGA with varying data parallelism;
- Evaluating the speedup provided by the proposed design across both the basic operations in HDC and on two different real-world learning tasks;
- Demonstrating the effectiveness of a flexible, general-purpose solution in optimizing diverse learning problems with respect to application-specific hardware architectures;
- Allowing an efficient trade-off between execution time and hardware resource usage to meet the constraints of diverse applications ranging from embedded systems to more complex computational environments.

The rest of the work is organized as follows: Section II presents the basics of HDC, introduces the Klessydra core and provides an overview of the related works in the literature. Section III presents the proposed accelerator, detailing its integration in the Klessydra core, the hardware architecture, its reconfigurability features and the proposed RISC-V ISE. Section V analyzes the hardware requirements and performance by synthesizing the proposed design on FPGA and evaluating the speedup achieved in the basic operations of HDC and in two learning tasks, while varying the hardware parallelism and the HV size. Finally, Section VII summarizes the main outcomes of this work and discusses the recommended next steps.

## II. Background and Related Works

### A. Hyperdimensional Computing Basics

The HDC paradigm combines the strengths of symbolic and connectionist approaches in AI, allowing for interpretable symbolic reasoning while maintaining the efficiency and scalability found in distributed connectionist models [3]. In HDC, logical concepts are encoded into HVs, high-dimensional vectors typically in the order of thousands of elements, in a distributed form akin to kernel expansion methods used in machine learning. Concepts or symbols are mapped to random HVs that, thanks to the hyperspace mathematical properties, result orthogonal, i.e. linearly independent. HVs are manipulated using a set of three primitive arithmetic operations and one comparison operations - namely bundling, binding, permutation, and similarity - that define the mathematical space of VSA models and enable composing complex information without increasing dimensionality.

The specific implementation of the basic operations depends on the nature of the hyperspace in which the hypervectors (HVs) reside [26]. The elements of the HVs, in fact, can belong to various *data types*, including binary {0,1}, bipolar {-1,1}, ternary {-1,0,1}, quantized, integer, real, or complex [21], each requiring a tailored implementation of the operators. Among the various approaches, Binary Spatter Code (BSC), which uses binary elements, is particularly well-suited for resource-constrained systems and for direct hardware implementations [5], [7], [27], [28]. Binary HVs offer significant advantages in terms of reducing computational complexity, memory footprint, and energy consumption, making them an ideal choice for embedded systems, wearable devices, and other low-power environments. The study in [29] demonstrated that using binary HV elements instead of 32-bit integers can decrease energy consumption by $150\times$, with only a minimal impact on accuracy. These reasons lead us to adopt BSC in this work to maximize efficiency and enable accelerator deployment even in resource-limited environments. The definition of the basic operations on HVs follows:

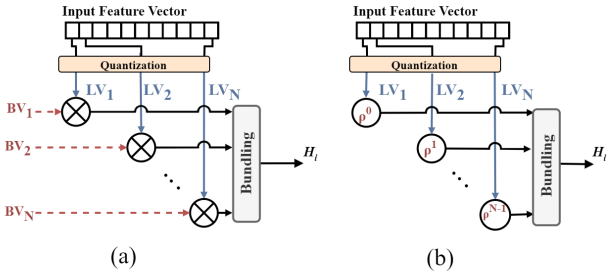



Fig. 1: Visualization of encoding techniques for feature vectors: (a) Record-based (b) N-gram based

The *Bundling* operation ($\oplus$) combines two HVs into a single one that encapsulates the essential aspects of both inputs resulting in a vector that is similar to the original ones. Bundling is associative and commutative, simulating the simultaneous activation of multiple neural patterns. In BSC, bundling is performed via element-wise addition; since the result is non-binary, clipping or binarization is achieved through a majority vote function, with randomly broken ties.

The *Binding* operation ($\otimes$) combines multiple HVs into a single one that represents the association between the original information while appearing dissimilar to the vectors that compose it. This operation is associative, commutative, self-invertible, and can be distributed over bundling. An inverse operation called unbinding is used to recover the original HVs. In BSC, binding is performed via bit-wise XOR.

The *Permutation* operation ($\rho$) reorders the elements of an HV, creating a dissimilar one. The operation is invertible and can be distributed over binding and bundling. In BSC, permutation is performed via cyclic right shift.

Lastly, the *similarity* measurement ($\delta$) quantifies the angular distance between HVs in high-dimensional space, allowing for efficient comparison of encoded information and enabling symbolic reasoning. HVs representing related concepts will be closer in the space and have a small angular distance, i.e. high similarity. In BSC, similarity measurement is performed via Hamming distance.

### B. Learning Tasks in HDC

Equipped with HDC's arithmetic operators, a diverse array of objects and their inherent relationships can be mapped in the high-dimensional space, enabling the execution of various learning tasks [1], [17], [19].

The first step in this process is encoding, where an N-dimensional input feature vector $\mathbf{x} = [x_1, x_2, \ldots, x_N]$ must be projected into a D-dimensional HV, preserving the semantic properties. Multiple encoding strategies have been proposed in the literature according to the adopted HV datatype and the application needs. For binary elements, the two main approaches are the record-based and the n-gram-based methods [30], summarized in Figure 1. In these techniques, feature values are discretized into $m$ levels and mapped into level hypervectors (LVs) that ensure close scalars in the original space are projected into similar HVs. In *record-based encoding*, each feature-ID or position $i$ of $\mathbf{x}$ is mapped to a randomly generated base vector ($BV_i$), which is then binded with the corresponding $LV_i$ to encode the id-value pair. Conversely, in *n-gram-based* encoding, feature positions are represented through permutations, leveraging this operation's properties to generate orthogonal HVs. The $i^{th}$ id-value pair is obtained by permuting the corresponding $LV_i$ by $i$ positions. Regardless of the chosen technique, the HVs representing each feature's value-ID pair are aggregated through bundling, producing the final encoded HV. These encoded HVs can then be combined to create sequences. When dealing with sequences—such as temporal data, biological sequences (e.g., DNA strands), or structured sequences found in natural language processing and robotic control tasks—position information is represented in the high-dimensional space using permutations to preserve the order of elements. The resulting permuted HVs are finally aggregated through bundling, resulting in a comprehensive representation of the entire sequence.

The steps that follow the encoding stage depend on the specific learning task. For instance, in a classification problem, during the *training phase* all the encoded HVs belonging to the same class are aggregated through bundling to build a representative prototype HV $CV_j$ for each class $j$. *Inference* is then performed by encoding the input pattern and searching for the most similar class prototype in memory, a procedure often called *associative search*.

### C. Related Works

Since HDC is well-suited for hardware acceleration on FPGAs and ASICs due to the highly parallel nature of the operations on HVs, various studies have proposed hardware solutions to accelerate HDC tasks. For example, the works in [7], [12], [23] present efficient FPGA-based accelerators for healthcare, recommender systems, and speech recognition. All these architectures are highly specialized, with design choices tailored to the specific tasks, datasets, HV sizes, and hardware constraints, limiting their applicability to any different scenario.

Initial steps towards more flexible HDC hardware designs can be represented by the automatic generation of HDC hardware accelerators. F5-HD [31] introduces a framework for deploying HDC accelerators on FPGAs, allowing users to specify parameters such as dataset properties, target FPGA, power constraints, and accuracy trade-offs. The framework then configures the architecture using predefined processing elements. Similarly, HD2FPGA [32] offers a tool for generating hardware accelerators connected to a host CPU. This tool adds flexibility by supporting clustering tasks, variable HV sizes, and automatic calibration of hardware parallelism to meet user-specific requirements. However, these frameworks are still limited by fixed design choices, preventing the customization of the underlying HDC model. The encoding techniques are rigid, making them unsuitable for handling temporal sequences or diverse input types, for example. Furthermore, the hardware lacks the flexibility to adjust parallelism throughout different stages of the learning process. The work in [30] seeks to overcome these limitations, introducing AeneasHDC, an automatic framework for generating highly flexible hardware accelerators that can be configured across various parameters, including model configuration, encoding techniques and hardware parallelism, and can be generated to perform different learning tasks. The accelerators generated by this framework are synthesized using High-Level Synthesis (HLS) languages and are primarily intended to explore the impact of various design choices on the hardware-accuracy trade-off. Consequently, they are not suitable for applications in resource-constrained systems or in applications with stringent real-time requirements, where optimizing each individual Functional Unit (FU) is crucial. Moreover, like in the previous works, the accelerator is specifically synthesized for a target HDC model, making it non-adaptable to other learning tasks.

Another move toward flexibility is represented by RISC-HD [28], a RISC-V core for resource-constrained applications with built-in support for the inference of HDC classification tasks with binary HVs. This core, based on the low-power RISCY architecture, features dedicated hardware units and three 4.5 kB read-only BRAMs containing the HVs. Two custom instructions are introduced to the RISC-V instruction set, one for encoding and one for inference, making it suitable for classification problems, independent of the dataset. While being a very efficient design, this architecture still exhibits flexibility limitations, as it does not support training and is restricted to processing 1000-dimensional binary HVs with a fixed parallelism of 100 bits per clock cycle. The designed hardware units and the two RISC-V instructions are restricted to the record-based encoding and to classification tasks, preventing their use in other learning problems or with different input data, such as temporal sequences, significantly limiting the flexibility and applicability of this core.

Overall, the field still lacks a general-purpose HDC hardware acceleration design.

In this work, we aim to fill this gap by presenting a highly flexible configurable hardware accelerator microarchitecture targeting the basic HDC operation set, along with a RISC-V ISE constituting an HDC Application Programming Interface and thus integrating the HDC acceleration subsytem into the full RISC-V programming environment. The resulting framework offers an adaptable and general-purpose solution for modelling any HDC problem and learning task. The designed accelerator is configurable in terms of hardware parallelism, size of the local memories and supported operation, allowing users to trade off execution time with hardware resources according to the specific resource constraints at hand.

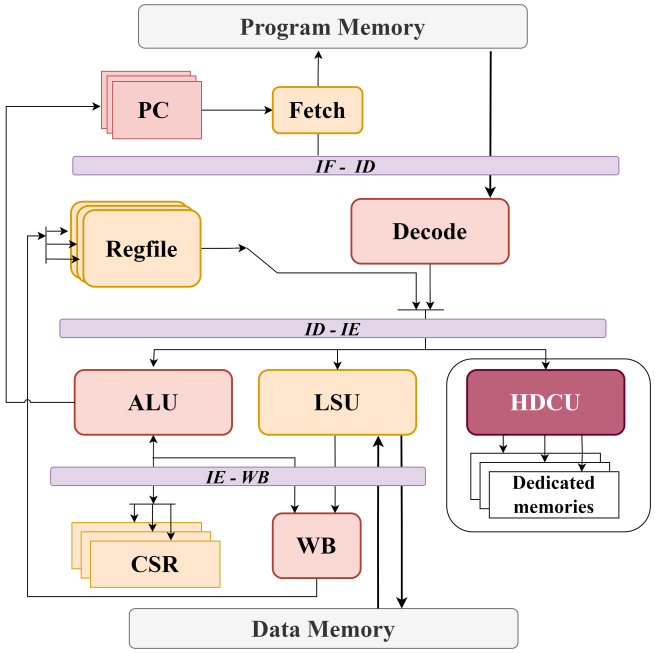



Fig. 2: Integration of the HDCU in the Klessydra T03 core. The accelerator works in parallel with the ALU and the LSU during the execution stage, acting like a coprocessor.

TABLE I: ISE for HDC Operations

| Instruction | Description |
|---|---|
| hvbundle(void* rd, void* rs1, void* rs2) | Bundle the $N$-bit precision HV in `rs1` with the binary HV in `rs2` to create a new HV in `rd`. |
| hvbind(void* rd, void* rs1, void* rs2) | Binds the HVs in `rs1` and `rs2`, resulting in a new HV in `rd`. |
| hvperm(void* rd, void* rs1, int rs2) | Permute the HV in `rs1` by `rs2` positions storing the result in `rd`. |
| hvsim(void* rd, void* rs1, void* rs2) | Hamming distance between the HVs in `rs1` and `rs2`. The similarity is stored in `rd`. |
| hvclip(void* rd, void* rs1, int rs2) | Binarize the HV in `rs1` using the threshold in `rs2`. Result in `rd`. |
| hvsearch(void* rd, void* rs1, void* rs2) | Compare the HV in `rs1` with CSR_HVCLASS HVs in `rs2`. Stores the closest match in `rd`. |
| hvmemld(void* rd, void* rs1, int size) | Loads an HV from memory location `rs1` into the SPM memory location `rd`. The `size` specifies the number of bytes to load. |
| hvmemstr(void* rd, void* rs1, int size) | Stores an HV from the SPM memory location `rs1` into data memory location `rd`. The `size` specifies the number of bytes to store. |

### D. Klessydra-T03 core

Klessydra [33], [34] is a processing core family compliant to the RISC-V instruction set architecture (ISA) and compatible with the PULPino open-source System-on-Chip platform, targeting low-power embedded systems. In particular, the Kessydra-T03 is a parametric design that supports the execution of multiple hardware threads in an Interleaved Multi-threading (IMT) scheme. The core supports the 32-bit integer instruction set RV32IMA in Machine-mode execution. The RISC-V open ISA is ideal for intergrating domain-specific hardware acceleration, thanks the possibility of extending the standard instruction set with custom instruction for specific computational tasks. Several works in the literature have designed custom RISC-V extensions for neural networks and machine learning applications [35], [36]. In this work, we leverage RISC-V extensibility to develop a custom ISE covering the primitive arithmetic operations of HDC supported by the proposed accelerator.

## III. Hyperdimensional Coprocessor Unit

The designed HDCU is integrated into the execution stage of the Klessydra-T03 core, as depicted in Figure 2, and may operate in parallel with the Load-Store Unit (LSU) and the Arithmetic Logic Unit (ALU). It may be synthesized to be replicated for each thread running in the core, or shared among the threads.

### A. Architecture overview

The HDCU operates as a coprocessor in charge of executing HDC operations. A custom RISC-V ISE was defined corresponding to the basic HDC operation set on binary HVs. The instructions composing the custom extension are exposed to the programmer as simple C-language intrinsic functions fully integrated into the RISC-V GCC compiler toolchain. Table I lists the instructions expressed via the intrinsic function syntax. The instructions do not cover specific encoding techniques, as done in [28], but rather cover the basic operations at the core of a VSA: bundling, clipping, binding, permutation, and similarity. While not tailored to a specific task, this approach implements a versatile general-purpose accelerator for HDC, allowing the user to employ the same architecture to perform any cognitive or reasoning task available in the literature by combining the operations into an HDC algorithm.

In the decode phase of the instruction processing pipeline of the Klessydra T03 core, if the fetched instruction belongs to the HDC ISE and the corresponding acceleration unit is available, the HDCU intervention is requested to handle the operation. The instructions executed by the HDCU operate on HVs contained in local Scratchpad Memories (SPMs), which are designed to be synthesized in Block-RAMs on FPGA and allow adequate bandwidth for the parallel operations on HVs. The size of each SPM can be configured at synthesis time. HVs in the Data Memory are loaded into and from the SPMs through the Scratchpad Memory Interface (SPMI) using the two dedicated instructions (Table I) *hvmemld* and *hvmemstr*, where the *size* parameter specifies the number of bytes to be transferred.

Figure 3 shows a detailed view of the microarchitecture of the HDCU. The design includes highly optimized functional units for each of the basic HDC arithmetic operations. The hardware parallelism inside each functional unit can be configured at synthesis time using a parameter called *SIMD* degree (Single Instruction Multiple Data). For instance, setting SIMD equal to 256 replicates the hardware to process 256 binary HV elements in one clock cycle. This flexibility enables trading off execution time with energy consumption and hardware

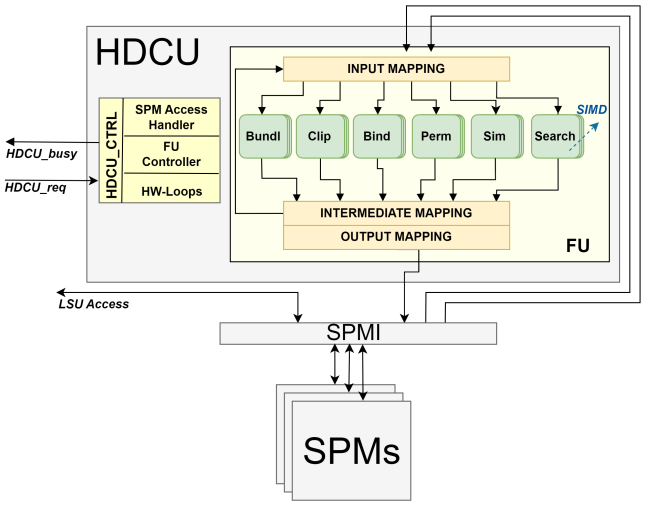



Fig. 3: Architecture of the HDCU

requirements, allowing the accelerator to adapt to the specific needs of various application scenarios, ranging from embedded systems to high-performance computing. Additionally, each functional unit can be optionally disabled to reduce hardware requirements when necessary, leaving the software execution of the corresponding HDC operation on the core.

Differently from existing solutions in the literature, the *HV size* in HDCU is not fixed but can be set at *runtime* by writing into a dedicated Control and Status Register (CSR), denoted as *HVSIZE* and visible to the software program through the CSR_HVSIZE instruction. The value of HVSIZE controls the hardware loops in the functional units. The hardware loops keep the required functional unit busy until *HVSIZE* elements are processed, avoiding the need for repetitive software loops in which the same instructions are repeatedly fetched and decoded. If the HVSIZE is bigger than the SIMD, the functional units process chunks of SIMD bits per clock cycle. For example, with $SIMD = 32$ and $HVSIZE = 1024$, the functional unit will be busy for $1024/32$ clock cycles, accessing HV data in the local memories.

In the following, we detail the design of each specialized functional unit integrated into the HDCU, focusing on how they have been optimized for performance, resource efficiency, scalability and flexibility.

### B. Binding Unit

As described in Section II, *binding* can be implemented with a bit-wise logic XOR when dealing with binary HVs. The SIMD parameter sets the unit hardware parallelism, that is the number of XOR gates in this case. The binding operation between HVs is launched via the following function/instruction:

```
hvbind(void* rd, void* rs1, void* rs2)
```

where:

- `rd` is the destination pointer in the local SPM for storing the result of the binding operation;
- `rs1` is a pointer in the SPM address space to the first binary HV;
- `rs2` is a pointer in the SPM address space to the second binary HV.

### C. Bundling Unit

As described in Section II, bundling — or superposition — is a crucial operation in HDC, allowing the accumulation of information coming from two HVs into a single HV. In BSC, bundling is performed as an element-wise sum between hypervectors, resulting in an integer HV that is later binarized using a clipping operator. However, binarization is typically applied only after a certain number of bundling operations during processes like encoding or training. For example, when encoding an input sequence with $N$ features, the HV of each feature (representing index-value pairs) is accumulated through successive bundling. Binarization occurs only after all $N$ HVs have been bundled, requiring the intermediate HVs to maintain integer precision until the final clipping step [37]. We support this feature by two separate instructions for bundling and clipping operations, respectively. By decoupling these operations, the number of HVs to be bundled before clipping is defined by the software program, in the view of flexibility and scalability, with only an upper limit referred to as the Bundle Capacity ($BC$), representing the maximum number of binary HVs that can be accumulated through bundling. The parameter $BC$ is a power of 2 and is set at synthesis time as the number $M$ of bits needed to represent it, i.e. $M = log_2 BC$. For instance, if encoding and inference are performed on a dataset with 5 features, we can set $M = 3$; whereas for optimizing the training process on a large FPGA, we can set $M = 32$.

The bundling operation is is launched via the following function/instruction:

```
hvbundle(void* rd, void* rs1, void* rs2)
```

where:

- `rd` is the destination pointer in SPM for storing the result of the bundling operation, with a precision of $M$ bits per element;
- `rs1` is a pointer in SPM to the first HV with an element precision of $M$ bits;
- `rs2` is a pointer in SPM to the second binary HV.

The bundling hardware unit is composed of an array of $M$-bit counters. In order to maintain $M$-bit integer precision in cumulative bundling, the unit assumes the first HV operand, pointed to by `rs1`, has a *widened* format - i.e. each element

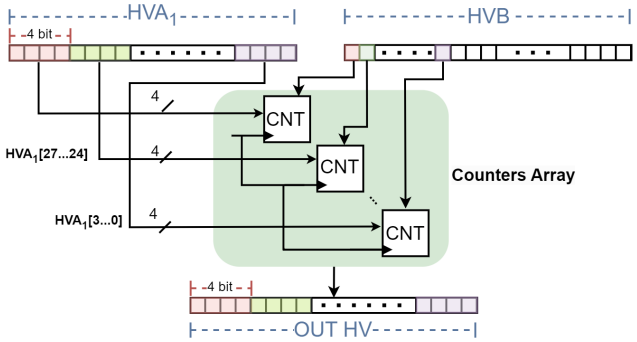



Fig. 4: Bundling Unit with $M$ set to 4

being $M$ bits wide - and it produces a similarly widened result. The value of each counter is initialized to the corresponding $M$-bit element of the HV pointed to by `rs1`, and it increments the count value if a '1' occurs in the corresponding position of the second operand, `rs2`. The output of each counter represents one element of the output HV of the operation.

This approach enables iterative HV bundling with $M$-bit element precision, allowing the output of a bundling operation to be used as the first operand in subsequent bundling operations. In order for the software program to operate correctly, the first of a cumulative bundling sequence should always have a zeroed HV as its destination.

The number of counters in the functional unit is

$$\text{Counters} = \frac{SIMD}{M} \tag{1}$$

while the number of HV elements processed per clock cycle by the bundling unit is expressed by

$$\text{Bundling latency} = \frac{HVSIZE * M}{SIMD} \tag{2}$$

### D. Clipping Unit

Binary normalization is performed by the Clipping Unit using a majority rule: each element of the output binary HV is set to '1' if the arithmetic value of the corresponding $M$-bit element in the widened input HV exceeds a threshold, which is usually half the number of bundled hypervectors, with ties broken at random. To ensure flexibility and not constrain the programmer to any HV encoding technique or number of features in the dataset, we decided to make the value threshold explicit in the Clipping operation invoked by the program.

The hardware structure of the clipping unit is similar to that of the bundling unit but replaces the array of counters with an array of comparators, as depicted in Figure 5. In this case, the unit takes as input one widened HV with $M$-bit elements and returns a binary HV. The latency for clipping completion is given by

$$\text{Clipping latency} = \frac{HVSIZE * M}{SIMD} \tag{3}$$

The clipping operation is launched via the following function/instruction:

```
hvclip(void* rd, void* rs, int s)
```

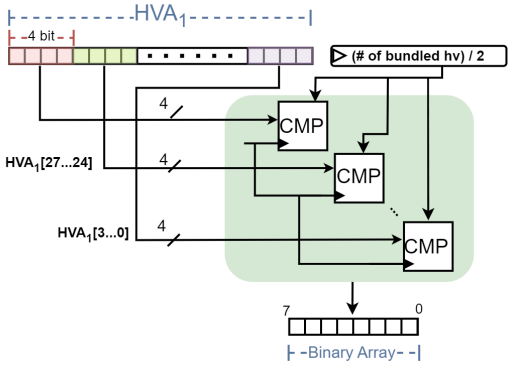



Fig. 5: Clipping Unit with $M$ set to 4

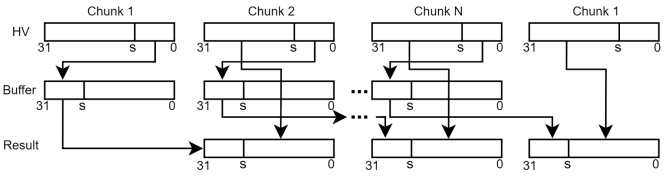



Fig. 6: Permutation Principle Scheme applied to a 128-bit HV processed using a SIMD of 32

where:

- `rd` is the destination pointer in SPM for storing the binary HV resulting from the clipping operation;
- `rs` is a pointer in SPM to the widened HV with $M$-bit elements subjected to clipping;
- `s` is a scalar representing the threshold value used for comparisons.

The threshold `s` is half the number of HVs previously bundled in the vector pointed by `rs`.

### E. Permutation Unit

As explained in Section II, the HDC permutation operation consists of a cyclic right shift (rotation), where the least significant bits that fall out of the HV replace the most significant ones.

The permutation operation is launched via the following function/instruction:

```
hvperm(void* rd, void* rs1, int s),
```

where:

- `rd` is the destination pointer in SPM for storing the permuted HV;
- `rs1` is the pointer to HV to be permuted;
- `s` represent the shift amount expressed in bits;

Like the other FUs, the permutation unit's structure is also HV length agnostic and supports configurable hardware parallelism. To rotate arbitrarily long vectors, as illustrated in Figure 6, a temporary register (buffer) stores the least significant bits of the HV chunk processed in the current clock cycle. In the first clock cycle, the most significant chunk of the HV pointed by `rs1` is read from the local memory, and a number of bits equal to the shift amount `s` are shifted out and stored in the buffer. The output of the first cycle is not written back to memory. In the second clock cycle, the buffer content is moved to the most significant `s` bits of the subsequent HV chunk, while the least significant `s` bits of this chunk are saved in the buffer, replacing its previous content. This process iterates until the entire HV has been read from memory and permuted. Once the last chunk has been processed, the first chunk is reloaded, shifted to the right, and the least significant `s` bits of the last processed chunk are inserted as its most significant bits. Figure 6 demonstrates this principle applied to a 128-bit HV processed using a SIMD width of 32.

### F. Similarity Unit

The similarity measurement between two binary HVs can be computed using the Hamming distance as described in

Section II. As depicted in Figure 7, this is implemented in the Similarity hardware unit in two steps: a bitwise XOR between the operands and a count of the number of '1's (population count or pop-count) occurring in the XOR result.

The pop-count operation can be implemented following different approaches. The work in [38] comprehensively analyzes and discusses the hardware requirements on FPGA of six different designs for the pop-count. Among these, we selected the very basic loop-based design due to its scalability in terms of hardware requirements and critical path delay. This design is well-suited for our configurable accelerator, where hardware parallelism (SIMD width parameter) is decided at synthesis time. The output of the pop-count is accumulated through a 13-bit adder. The similarity unit computes *SIMD* elements per clock cycle. After processing the entire HV, the final value in memory represents the Hamming distance. This operation can be performed inside the accelerator using the following instruction:

`hvsim(void* rd, void* rs1, void *rs2)`,

where:

- `rd` is the pointer to the memory location where Hamming Distance value will be stored;
- `rs1` is the pointer in SPM to the first HV;
- `rs2` is the pointer in SPM to the second HV;

### G. Associative Search Unit

The last operation implemented in the HDCU is the *associative search*. While this is not a standard HDC operator, it is crucial during the inference phase of many learning tasks, as described in Section II. Performing this procedure in software would require calling the designed `hvsim` instruction $k$ times in a k-class problem, which is inefficient and limits the advantages of hardware acceleration. Therefore, having a specific hardware implementation for this task is highly beneficial. While in-memory computing techniques are often used in the literature to implement this operation, such approaches are not feasible for FPGA implementations or soft-processors like Klessydra. Consequently, we decided to reuse the similarity unit and introduce a dedicated hardware instruction:

`hvsearch(void* rd, void* rs1, void* rs2)`

where:

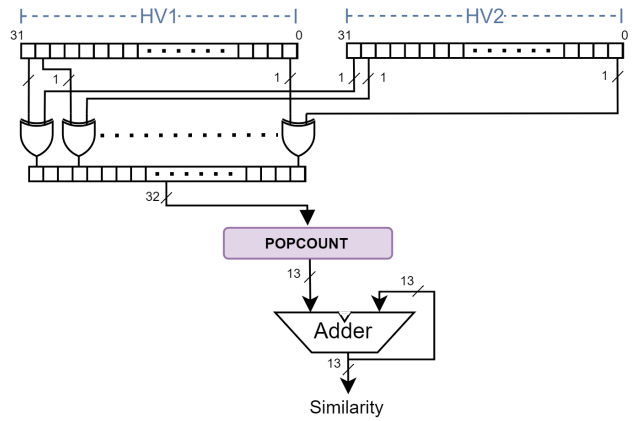



Fig. 7: Hardware Scheme of the Similarity Unit

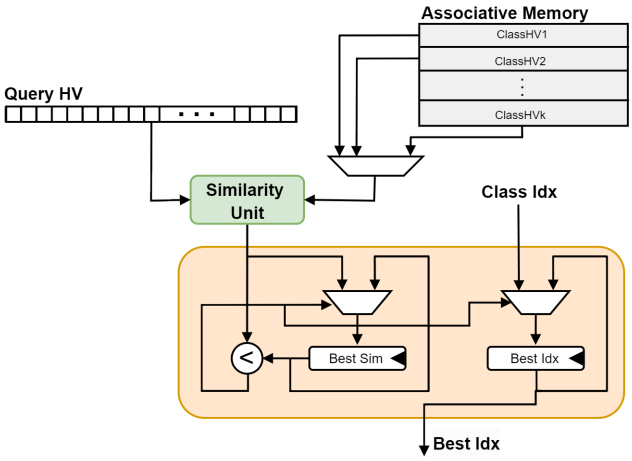



Fig. 8: Associative Search Unit: Operating principle for finding the most similar HV in the associative memory.

- `rd` is the destination register's address, containing the class index that returned the best similarity with the query vector.
- `rs1` is the address of the register containing the query vector.
- `rs2` is the address pointing to the first HV present in the associative memory.

Unlike all previous operations, the associative search needs one additional information: the number of classes (i.e., the number of HVs in the associative memory). This is communicated to the HDCU at *runtime* through the dedicated CSR function `CSR_HVCLASS()`.

This CSR controls the hardware loop of the HDCU to compare the input HV in `rs1` with `HVCLASS` CVs in `rs2`, each constituted by `HVSIZE` bits. When associative search is required, both the similarity unit and the associative search unit are enabled. As depicted in Figure 8, the similarity unit uses specific operand address management to hold the same HV (the query) on one input while the other input iterates through each HV in the associative memory. Each time a similarity is computed, it is compared with the best similarity stored in a 13-bit register that is eventually updated. When the associative search completes, the functional unit writes the index of the class that returned the best similarity to the scratchpad memory.

## IV. HDCU Configuration and Programming

Overall, the presented hardware accelerator exemplifies significant flexibility thorugh configurability and software programmability via the RISC-V ISE, making it versatile for a wide range of learning tasks and of application contexts.

### A. Hardware configuration

Table II summarizes the parameters that can be tuned at synthesis time and at runtime. At synthesis time, the size and number of Scratchpad Memories (SPMs), the hardware parallelism (SIMD), the Bundle Capacity, and the enable/disable

TABLE II: HDCU Configuration Parameters

| Parameter | Configuration Time |
|---|---|
| SPM Size | Synthesis |
| SPM Number | Synthesis |
| Hardware Parallelism ($SIMD$) | Synthesis |
| Functional Unit Enable/Disable | Synthesis |
| Bundle Capacity ($M$) | Synthesis |
| HVSIZE | Runtime |
| HVCLASS | Runtime |

status of each functional unit can be configured. These parameters allow the designer to balance the trade-offs between execution speed, energy consumption, and hardware resource utilization according to the specific requirements of the target application.

At runtime, the HV size and the number of HVs in the associative memory can be set through dedicated CSRs that control the hardware loops. This capability enhances the flexibility of the HDCU, enabling dynamic adaptation to varying workloads and characteristics of the HD space, using the same accelerator.

### B. Software programming

The set of intrinsic functions available to the programmer introduced in Section III, directly mapped on the RISC-V ISE and fully integrated into the GCC compiler toolchain, constitute a simple yet effective Application Programming Interface (API) to implement HDC tasks. Pseudocode 1 demonstrates an example of C-code that performs record-based encoding with 1024-element HVs using the HDCU accelerator. Initially, the BVs and LVs required by this process are loaded into the dedicated SPMs using the `hvmemld` instruction. The number of bytes to be loaded is determined by the number of BVs, which equals the number of features, and the number of LVs, which depends on the quantization levels used.

Next, the size of the HVs is set at runtime through the `CSR_HVSIZE` function, which controls the hardware loops of the following arithmetic instructions. These initial steps are performed only once before starting the encoding process and should not be repeated for each input feature vector.

The record-based encoding then begins. For each feature $i$, the BV[$i$] is bound to the corresponding LV (indexed by the quantized level assigned to the scalar value of feature $i$) using the `hvbind` instruction, pointing to the respective addresses in the dedicated memories. The output of this process is bundled using the `hvbundle` instruction to create the encoded HV with an element precision of $M$-bits. After processing all features, the resulting HV is binarized using the `hvclip` instruction. This completes the encoding operation, with the resulting HV stored in the SPM. If the accelerator is no longer required, the output can be saved to the data memory via the `hvmemstr` instruction. Otherwise, the training and inference processes can proceed, continuing to point to the encoded HV in the dedicated memory.

### C. HDCU Software Library

On top of the API constituted by the HDCU intrinsic functions available to the programmer, in order to facilitate using of the accelerator and testing its performance, we developed a custom software library in *C++* language for HDC programming. The library provides custom types and predefined functions to easily perform common tasks in HDC, such as the generation of BV and LV, the n-gram based and record based encodings, the temporal encoding and the training and inference phases of classification and clustering models. Importantly, each of these functions can be executed either as purely software routine on the RISC-V scalar core (Klessydra T03) or leveraging hardware acceleration on the HDCU, thus giving a mean to validate the HDCU results and evaluate its performance in terms of execution time. The library, including extensive utilization for its use, is available on GitHub [39], encouraging community collaboration.

Listing 1: Example of a record-based encoding performed using the HDCU's ISE

```
// ...Logic to generate the BV and LV

// Load the BV and LV into the SPMs and set the
// HVSIZE using the CSR. Perform these operations
// just once, before the first encoding
int HV_SIZE = 1024;
hvmemld((void*)((int*)spmA), &BV[0], FEATURE_NUM);
hvmemld((void*)((int*)spmB), &LV[0], LV_NUM);
CSR_HVSIZE(HV_SIZE);

// Iterative Record Based encoding
for (int i = 0; i < FEATURE_NUM; i++)
{
    // BIND the BV[i] with LV[quant_feature[i]]
    // quant_feature[i] is denoted as q[i]
    hvbind((void*)((int*)spmC +  i   * HV_SIZE),
           (void*)((int*)spmB + q[i] * HV_SIZE),
           (void*)((int*)spmA +  i   * HV_SIZE));

    // Bundle the feature-HVs
    hvbundle((void*)((int*)spmD),
             (void*)((int*)spmD),
             (void*)((int*)spmC + i * HV_SIZE));
}

// Normalize the final encoded HV
hvclip((void*)((int*)spmD),
       (void*)((int*)spmD),
       (void*)(FEATURE_NUM));

// Optional: if you finished using the HDCU,
// you can store the HV in the main memory
hvmemstr(&out,(void*)((int*)spmC), sizeof(out));
```

## V. Hardware Implementation

To assess the hardware requirements of the proposed acceleration units, we synthesized the Klessydra T03 core equipped with the novel HDCU on an FPGA platform. The design was synthesized on the Xilinx Zynq UltraScale+ ZCU106 (EK-U1-ZCU106-G) device, using the Vivado 2023.2 tool suite.

Table III provides a detailed breakdown of hardware resource utilization of the baseline configuration of the HDCU integrated with the Klessydra T03 core. For this configuration, we adopted a SIMD width of 32, a bundling precision of 4 bits, and 3 SPMs of 2 KB each. This setup represents the minimal hardware parallelism supported by the HDCU, and it is suitable for resource-constrained environments, such as IoT applications on the edge. The hardware utilization is detailed

TABLE III: Hardware Requirements of the Klessydra T03 and the HDCU with SIMD = 32; f = 220 MHz; SPMs = 2kB

| Device | LUTs | FF | DSPs | BRAM |
|---|---|---|---|---|
| Klessydra T03 Core | 4281 | 1418 | 7 | 0 |
| HDCU | 1030 | 450 | 0 | 0 |
| Scratchpad Memory Interface | 384 | 151 | 0 | 2 |
| Scratchpad Memory | 268 | 0 | 0 | 2 |

as the utilized Look-Up Tables (LUTs), Flip-Flops (FFs), Digital Signal Processors (DSPs), and Block-RAM (BRAM) on the FPGA device. The baseline HDCU occupies 1030 LUTs and 450 FFs and, importantly, it does not affect the critical path of the microarchitecture, allowing the processor to maintain its original operating frequency of 220 MHz.

TABLE IV: Hardware Requirements of HDCU for Different SIMD Configurations

| Configuration | Synthesis Results | | | |
|---|---|---|---|---|
| SIMD | #LUTs | #FFS | #CARRY8 | $f_{max}$ [MHz] |
| 32 | 1030 | 450 | 87 | 234 |
| 64 | 2040 | 651 | 101 | 221 |
| 128 | 3243 | 868 | 131 | 218 |
| 256 | 4016 | 1101 | 172 | 215 |
| 512 | 13811 | 3731 | 464 | 160 |
| 1024 | 34189 | 7478 | 1424 | 140 |

In the view of exploring how the hardware resource requirements of the HDCU scale when varying the SIMD width from 32 to 1024, we synthesized other HDCU configurations, reporting the resulting hardware utilization details in Table IV. This analysis is crucial for understanding how the accelerator can be adapted to different application scenarios, trading-off performance for hardware utilization. When increasing the SIMD from 32 to 1024, the number of LUTs grows from 1030 up to 34189, evidencing the importance of selecting an appropriate SIMD configuration based on the specific needs of the target application. In the context of resource-constrained applications focused on edge inference computation, a low SIMD such as 32 is likely to be suitable. Conversely, when resource constraints are less of an issue and the goal is also to accelerate the training phase, higher parallelism with SIMD 1024 may be employed to maximize performance.

Overall, the obtained results demonstrate the flexibility and scalability of the proposed configurable design. By carefully tuning its parameters, users can effectively trade off execution time with area occupancy, pursuing the optimal balance based on the specific application requirements. These characteristics establish the HDCU as a versatile component capable of addressing the diverse needs of various contexts, from resource-constrained tasks to performance-intensive tasks.

## VI. Performance Results

We carried out two types of analysis to clearly assess the performance advantage of the HDCU:

- impact on the execution time of the HDC basic operations
- impact on the execution time of two real-world computation kernels using HDC.

We leveraged the designed software library described in Section IV-C to evaluate the speedup metric as the ratio between the clock cycle count required by the standard non-accelerated operation (i.e., compiled for the RISC-V instruction set and executed on the Klessydra-T03 processor without HDCU support) and the cycle count required by the accelerated operation (i.e., compiled on the extended instruction set exploiting the HDCU support of the Klessydra-T03 core). The speedup was analyzed across different configurations of the hardware parallelism and of HV size. The number of clock cycles has been extracted from cycle-accurate Register-Trasfer-Level simulations on QuestaSim.

### A. Speedup on the Basic HDC Arithmetic Operation

Figure 9 presents the speedup achieved by the HDCU for each core arithmetic operation configuring the SIMD in the range from 32 to $1024$ and programming the HV size from 32 to $8192$. These values were chosen to demonstrate how the advantage of the accelerator scales with both HV size and SIMD. As the HV size increases, the hardware loops within the HDCU allow an efficient processing without re-fetching instructions repeatedly, thereby maximizing the execution speed. On the other hand, increasing SIMD values enables greater parallelism, allowing for the simultaneous processing of more HV elements, but at the cost of increased hardware complexity, as demonstrated in the previous section.

In the *Binding* operation, which is the simplest operation in binary HDC, the hardware loops and the dedicated memories of the HDCU allow for a speedup ranging from $2.70\times$ to $26.64\times$ when using a SIMD $= 32$. The advantage significantly increases both with the hardware parallelism and the size of the HVs, reaching $313.92\times$ when using a SIMD $= 1024$ and an HV size of $8192$.

A similar trend can be observed for the *Permutation* operation, where despite the intricate nature of the operation, the HDCU achieves a maximum speed up of $191.48\times$.

A more significant advantage is observed in the more complex *Bundling* and *Clipping* operations. These instructions, as described in Section III, internally handle elements with a precision that can be configured at synthesis time through the $M$ parameter. When using $M = 4$, the HDCU achieves for the Bundling an impressive speedup ranging from $23.36\times$ when the SIMD and the HV size are equal to 32, to $2665.20\times$ when the $SIMD = 1024$ and HV size $= 8192$. The acceleration achieved in this operation is essential for reducing the execution time of the encoding, the most expensive step for training an HDC model for classification and clustering tasks. Similarly, the HDCU shows a speed-up ranging from $14.34\times$ up to $1554.16\times$ for the Clipping operation.

Reducing the execution time required by the computation of the similarity is essential to optimize the inference stage in learning tasks with HDC. The last two plots in Figure 9 show the speedup achieved by the HDCU on the Similarity and the Associative Search instructions, respectively. The former achieves a speedup ranging from $11.91\times$ to $1854.44\times$ when varying the SIMD from 32 to 1024 and the HV size from 32 to 8192. However, for the same values of the parameters, the *Associative Search* instruction is capable of obtaining an impressive maximum speedup of $3844.09\times$ when using

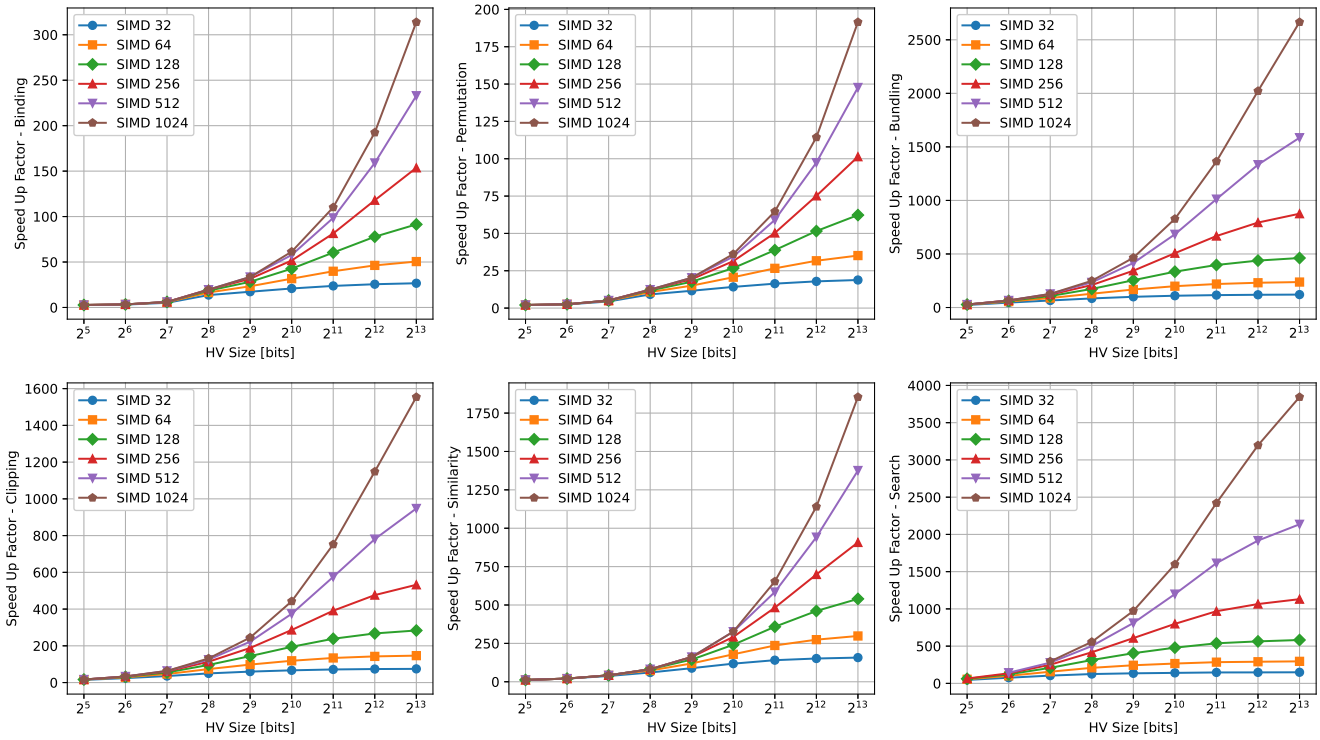



Fig. 9: Speed-Up Factor for each HDC operation. The x-axis represents the HV size, ranging from 32 to 8192 bits, while different colors represent varying SIMD widths from 32 to 1024.

three classes, clearly showing the advantage of a dedicated instruction for performing this task.

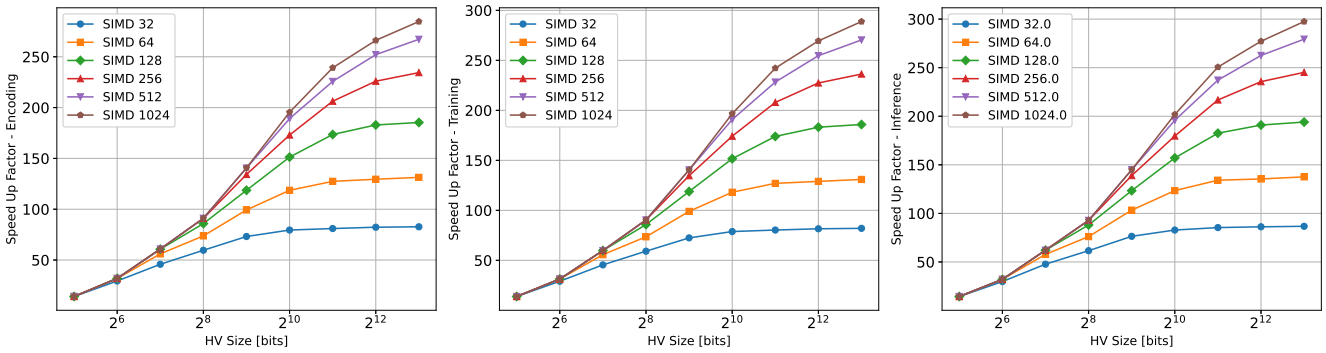
Overall, the obtained results demonstrate the effectiveness of the HDCU accelerator in significantly reducing the execution time of arithmetic operations in HDC, enabling faster and more efficient application of reasoning and learning tasks.

### *B. Speed up on real datasets*

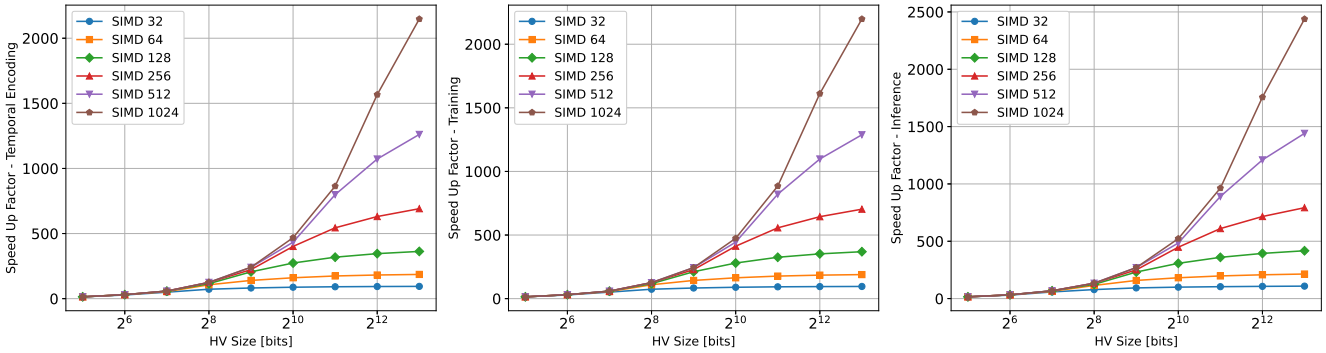
To demonstrate the flexibility of the proposed accelerator and evaluate its performance on real-world computation scenarios, we implemented and executed two classification tasks using the designed software library. In each experiment, we evaluated the speedup achieved by the HDCU across one iteration of encoding, training, and inference phases. It is important to note that the focus of this analysis is not to assess the accuracy achieved by the accelerator, since each arithmetic operation implemented in the HDCU, along with the encoding, training, and inference stages, has been rigorously tested and validated to ensure full consistency with the software model. Consequently, the accelerator delivers the exact same accuracy as an HDC software model.

The first dataset we used is *CARDIO* [40], which includes three classes and spatial sequences characterized by 21 features. Figure 10a presents the results obtained by varying the HV size and hardware parallelism. For the encoding, we used the record-based technique described in Section II and implemented as shown in Pseucode 1. In this approach, each Base Vector was binded to its corresponding Level Vector, and the results were bundled together. After processing all 21 features, a clipping operation was applied to binarize the final encoded HV. In this workload the HDCU provided a speedup ranging from $13.56\times$ to $80.91\times$ over software execution when using SIMD $=$ 32 and varying the HV size from 32 to 8192. This speedup increased to an impressive $278.37\times$ when using SIMD $=$ 1024. A similar trend was observed during the training phase, where at each iteration, the encoded HVs were bundled with the corresponding Class Vectors (CVs). The obtained speedup ranges from a minimum of $13.80\times$ with SIMD $=$ 32 and HV Size = 32, to $288.81\times$ with SIMD $=$ 1024 and HV Size = 8192. During the inference phase, the speedup is further amplified due to the associative search operation, which allows to compare the encoded HV with all the CVs using a single instruction. Here, the HDCU achieves a maximum speedup of $297.49\times$. Notably, the CARDIO dataset involves only three classes, making encoding the dominant factor in execution time. However, as the number of classes increases, the benefit of a dedicated hardware instruction for associative search becomes even more pronounced.

The second dataset we analyzed was the *EMG* [41], a temporal encoding problem with four features across five classes. For this task, we used the record-based encoding in combination with the perutation to capture information from temporal windows of three elements. Essentially, each pattern was mapped into the high-dimensional space, permuted based on its position in the time window, and then bundled with the other HVs of the same temporal window. This type of encoding, which demands significantly more complexity than spatial encoding, places a much greater emphasis on the




(a) CARDIO dataset




(b) EMG dataset

Fig. 10: Speed-Up Factors on the Encoding, Training, and Inference tasks on (a) CARDIO and (b) EMG datasets. The x-axis represents the HV size, ranging from 32 to 8192 bits, while different colours represent varying SIMD widths from 32 to 1024.

bundling operation. The HDCU's highly optimized bundling instruction, which has shown to achieve the highest speedup among the arithmetic operators in previous tests, is particularly beneficial here. As depicted in Figure 10b, the HDCU provided significant speedups in this task. During encoding, the accelerator achieves speedups ranging from $13.97\times$ to $94.24\times$ when using SIMD $= 32$ and varying the HV size from 32 to 8192. This speedup escalates to $2147.63\times$ when SIMD $= 1024$ and HV Size = 8192. Further improvements were observed during the training and inference phases, where the HDCU attained maximum speedups of $2198.26\times$ and $2438.36\times$, respectively. The analysis confirmed that as the number of classes increases, the associative search provides greater speedup, as anticipated in the CARDIO dataset analysis.

Overall, the obtained results demonstrate the flexibility and scalability of the proposed configurable HDCU design. By carefully tuning its parameters, users can effectively trade off execution time with area occupancy, enabling them to find the optimal balance based on specific application requirements.

## VII. Conclusions

In this work, we introduced the Hyperdimensional Coprocessor Unit, a highly flexible hardware accelerator specifically designed to optimize hyperdimensional computing tasks. The HDCU is configurable at synthesis time in terms of hardware parallelism, supported operations, and local memory sizes, allowing users to trade off execution speed and hardware requirements to meet the demands of different applications, ranging from embedded systems to performance-intensive contexts.

The HDCU is integrated into the execution stage of the RISC-V Klessydra-T03 core, and it is exposed to the software programmers through a custom RISC-V ISE mapping intrinsic functions, constituting a simple API for HDC. The API, including the possibility to set the HV size at runtime, further enhances the HDCU versatility, enabling the acceleration of different learning tasks in a simple and intuitive way utilizing the same hardware architecture.

To validate the design, we synthesized the Klessydra-T03 core equipped with the HDCU on the Xilinx Zynq UltraScale+ ZCU106 FPGA and analyzed its impact on the execution time of learning tasks, varying the HV size and hardware parallelism. The obtained results demonstrated the scalability and effectiveness of the design, achieving significant speedups across core arithmetic operations in HDC and on two different real-world learning tasks.

To the best of our knowledge, the HDCU represents an advancement in hardware-accelerated HDC, providing unprecedented versatility and scalability to tackle diverse learning tasks and application requirements.

## References

[1] D. Kleyko, M. Davies, E. P. Frady, P. Kanerva, S. J. Kent, B. A. Olshausen, E. Osipov, J. M. Rabaey, D. A. Rachkovskij, A. Rahimi, *et al.*, "Vector symbolic architectures as a computing framework for emerging

hardware," *Proceedings of the IEEE*, vol. 110, no. 10, pp. 1538–1571, 2022.

[2] D. Kleyko, D. Rachkovskij, E. Osipov, and A. Rahimi, "A survey on hyperdimensional computing aka vector symbolic architectures, part ii: Applications, cognitive models, and challenges," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–52, 2023.

[3] K. Schlegel, P. Neubert, and P. Protzel, "A comparison of vector symbolic architectures," *Artificial Intelligence Review*, vol. 55, no. 6, pp. 4523–4555, 2022.

[4] A. Mitrokhin, P. Sutor, C. Fermüller, and Y. Aloimonos, "Learning sensorimotor control with neuromorphic sensors: Toward hyperdimensional active perception," *Science Robotics*, vol. 4, no. 30, p. eaaw6736, 2019.

[5] Y. Guo, M. Imani, J. Kang, S. Salamat, J. Morris, B. Aksanli, Y. Kim, and T. Rosing, "Hyperrec: Efficient recommender systems with hyperdimensional computing," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pp. 384–389, 2021.

[6] N. Watkinson, D. Devineni, V. Joe, T. Givargis, A. Nicolau, and A. Veidenbaum, "Using hyperdimensional computing to extract features for the detection of type 2 diabetes," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 149–156, IEEE, 2023.

[7] A. Burrello, L. Cavigelli, K. Schindler, L. Benini, and A. Rahimi, "Laelaps: An energy-efficient seizure detection algorithm from long-term human ieeg recordings without false alarms," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 752–757, IEEE, 2019.

[8] A. Moin, A. Zhou, A. Rahimi, S. Benatti, A. Menon, S. Tamakloe, J. Ting, N. Yamamoto, Y. Khan, F. Burghardt, *et al.*, "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)*, pp. 1–5, IEEE, 2018.

[9] A. Burrello, K. Schindler, L. Benini, and A. Rahimi, "One-shot learning for ieeg seizure detection using end-to-end binary operations: Local binary patterns with hyperdimensional computing," in *2018 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, IEEE, 2018.

[10] F. R. Najafabadi, A. Rahimi, P. Kanerva, and J. M. Rabaey, "Hyperdimensional computing for text classification," in *Design, automation test in Europe conference exhibition (DATE), University Booth*, pp. 1–1, 2016.

[11] A. Joshi, J. T. Halseth, and P. Kanerva, "Language geometry using random indexing," in *Quantum Interaction: 10th International Conference, QI 2016, San Francisco, CA, USA, July 20-22, 2016, Revised Selected Papers 10*, pp. 265–274, Springer, 2017.

[12] Y. Kim, M. Imani, N. Moshiri, and T. Rosing, "Geniehd: Efficient dna pattern matching accelerator using hyperdimensional computing," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 115–120, IEEE, 2020.

[13] 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)*, pp. 1–8, IEEE, 2017.

[14] A. Rosato, M. Panella, E. Osipov, and D. Kleyko, "On effects of compression with hyperdimensional computing in distributed randomized neural networks," in *Advances in Computational Intelligence* (I. Rojas, G. Joya, and A. Català, eds.), (Cham), pp. 155–167, Springer International Publishing, 2021.

[15] D. Kleyko, A. Rosato, E. P. Frady, M. Panella, and F. T. Sommer, "Perceptron theory can predict the accuracy of neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 7, pp. 9885–9899, 2024.

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

[17] M. Imani, Y. Kim, T. Worley, S. Gupta, and T. Rosing, "Hdcluster: An accurate clustering using brain-inspired high-dimensional computing," in *2019 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 1591–1594, 2019.

[18] L. Ge and K. K. Parhi, "Robust clustering using hyperdimensional computing," *arXiv preprint arXiv:2312.02407*, 2023.

[19] A. Hernandez-Cano, C. Zhuo, X. Yin, and M. Imani, "Reghd: Robust and efficient regression in hyper-dimensional learning system," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 7–12, IEEE, 2021.

[20] C.-Y. Chang, Y.-C. Chuang, C.-T. Huang, and A.-Y. Wu, "Recent progress and development of hyperdimensional computing (hdc) for edge intelligence," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2023.

[21] E. Hassan, Y. Halawani, B. Mohammad, and H. Saleh, "Hyper-dimensional computing challenges and opportunities for ai applications," *IEEE Access*, vol. 10, pp. 97651–97664, 2021.

[22] S. Bertazzoni, L. Canese, G. C. Cardarilli, L. D. Nunzio, R. Fazzolari, M. Re, and S. Spanò, "Design space exploration for edge machine learning featured by mathworks fpga dl processor: A survey," *IEEE Access*, vol. 12, pp. 9418–9439, 2024.

[23] M. Schmuck, L. Benini, and A. Rahimi, "Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 4, pp. 1–25, 2019.

[24] B. Khaleghi, H. Xu, J. Morris, and T. Š. Rosing, "tiny-hd: Ultra-efficient hyperdimensional computing engine for iot applications," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 408–413, IEEE, 2021.

[25] S. Datta, R. A. Antonio, A. R. 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*, vol. 9, no. 3, pp. 439–452, 2019.

[26] K. Schlegel, P. Neubert, and P. Protzel, "A comparison of vector symbolic architectures," *Artificial Intelligence Review*, vol. 55, no. 6, pp. 4523–4555, 2022.

[27] M. Imani, S. Bosch, S. Datta, S. Ramakrishna, S. Salamat, J. M. Rabaey, and T. Rosing, "Quanthd: A quantization framework for hyperdimensional computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2268–2278, 2019.

[28] F. Taheri, S. Bayat-Sarmadi, and S. Hadayeghparast, "Risc-hd: Lightweight risc-v processor for efficient hyperdimensional computing inference," *IEEE Internet of Things Journal*, vol. 9, no. 23, pp. 24030–24037, 2022.

[29] J. Morris, R. Fernando, Y. Hao, M. Imani, B. Aksanli, and T. Rosing, "Locality-based encoder and model quantization for efficient hyperdimensional computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 4, pp. 897–907, 2021.

[30] M. Angioli, S. Jamili, M. Barbirotta, A. Cheikh, A. Mastrandrea, F. Menichelli, A. Rosato, and M. Olivieri, "Aeneashdc: An automatic framework for deploying hyperdimensional computing models on fpgas," *Authorea Preprints*, 2024.

[31] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, "F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 53–62, 2019.

[32] T. Zhang, S. Salamat, B. Khaleghi, J. Morris, B. Aksanli, and T. S. Rosing, "Hd2fpga: Automated framework for accelerating hyperdimensional computing on fpgas," in *2023 24th International Symposium on Quality Electronic Design (ISQED)*, pp. 1–9, IEEE, 2023.

[33] A. Cheikh, S. Sordillo, A. Mastrandrea, F. Menichelli, G. Scotti, and M. Olivieri, "Klessydra-t: Designing vector coprocessors for multi-threaded edge-computing cores," *IEEE Micro*, vol. 41, no. 2, pp. 64–71, 2021.

[34] M. Olivieri, A. Cheikh, G. Cerutti, A. Mastrandrea, and F. Menichelli, "Investigation on the optimal pipeline organization in risc-v multi-threaded soft processor cores," in *2017 New Generation of CAS (NGCAS)*, pp. 45–48, 2017.

[35] X. Wang, M. Magno, L. Cavigelli, and L. Benini, "Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4403–4417, 2020.

[36] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "Pulp-nn: A computing library for quantized neural network inference at the edge on risc-v based parallel ultra low power clusters," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 33–36, 2019.

[37] M. Schmuck, L. Benini, and A. Rahimi, "Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory," *J. Emerg. Technol. Comput. Syst.*, vol. 15, Oct. 2019.

[38] I. Skliarova, "Accelerating population count with a hardware co-processor for microblaze," *Journal of Low Power Electronics and Applications*, vol. 11, no. 2, p. 20, 2021.

[39] "HDCU: Hyperdimensional Coprocessor Unit," 2024. GitHub.

[40] D. Campos and J. Bernardes, "Cardiotocography." UCI Machine Learning Repository, 2010. DOI: https://doi.org/10.24432/C51S4N.

[41] A. Rahimi, "Emg dataset." https://github.com/abbas-rahimi/HDC-EMG/blob/master/dataset.mat, Jan. 2024.