

Article

A General-Purpose AXI Plug-and-Play Hyperdimensional Computing Accelerator

Rocco Martino , Marco Pisani , Marco Angioli , Marcello Barbirotta , Antonio Mastrandrea ,
Antonello Rosato  and Mauro Olivieri 

Department of Information Engineering, Electronics and Telecommunications (DIET), Sapienza University of Rome, 00184 Rome, Italy; marco.pisani@uniroma1.it (M.P.); marco.angioli@uniroma1.it (M.A.); marcello.barbirotta@uniroma1.it (M.B.); antonio.mastrandrea@uniroma1.it (A.M.); antonello.rosato@uniroma1.it (A.R.); mauro.olivieri@uniroma1.it (M.O.)

* Correspondence: rocco.martino@uniroma1.it

Abstract

Hyperdimensional Computing (HDC) offers a robust and energy-efficient paradigm for edge intelligence; however, current hardware accelerators are often proprietary, tailored to the target learning task and tightly coupled to specific CPU microarchitectures, limiting portability and adoption. To address this, and democratize the deployment of HDC hardware, we present a general-purpose, plug-and-play accelerator IP that implements the Binary Spatter Code framework as a standalone, host-agnostic module. The design is compliant with the AMBA AXI4 standard and provides an AXI4-Lite control plane and DMA-driven AXI4-Stream datapaths coupled to a banked scratchpad memory. The architecture supports synthesis-time scalability, enabling high-throughput transfers independently of the host processor, while employing microarchitectural optimizations to minimize silicon area. A multi-layer C++ software (GitHub repository commit 3ae3b46) stack running in Linux userspace provides a unified programming model, abstracting low-level hardware interactions and enabling the composition of complex HDC pipelines. Implemented on a Xilinx Zynq XC7Z020 SoC, the accelerator achieves substantial gains over an ARM Cortex-A9 baseline, with primitive-level speedups of up to $431\times$. On end-to-end classification benchmarks, the system delivers average speedups of $68.45\times$ for training and $93.34\times$ for inference. The complete RTL and software stack are released as open-source hardware to support reproducible research and rapid adoption on heterogeneous SoCs.

Keywords: hyperdimensional computing; FPGA acceleration; edge Intelligence; AXI4 interface; open source hardware; binary spatter code



Academic Editor: Alexander Barkalov

Received: 16 December 2025

Revised: 17 January 2026

Accepted: 19 January 2026

Published: 22 January 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and

conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Hyperdimensional Computing (HDC), also known as Vector Symbolic Architecture (VSA), is a family of neuro-inspired computing paradigms that originates at the intersection of symbolic Artificial Intelligence (AI) and distributed connectionist representations [1,2]. HDC/VSA encodes information using high-dimensional vectors, denoted as hypervectors (HVs), where information is distributed across all components rather than localized to specific elements [1]. At the core of this paradigm is the principle that independent concepts can be assigned random HVs that, by virtue of concentration effects in high-dimensional spaces, are (with high probability) quasi-orthogonal [3]. This property yields vast symbol spaces and enables the representation of an exponential number of distinct

concepts for a given dimensionality. HVs can then be composed using simple vector operations—bundling, binding and permutation—and queried via a similarity measure, supporting compact representations of structured information and enabling a wide range of cognitive and learning tasks [4].

This high-dimensional, holistic representation endows HDC/VSA with several favorable properties, including robustness to noise and hardware faults, few-shot learning capability, computational and energy efficiency, and massive data parallelism [3,5]. These properties make HDC/VSA particularly attractive for resource-constrained embedded systems and efficient hardware implementations, motivating substantial interest in hardware-aware HDC/VSA formulations and accelerators for edge intelligence [6,7].

Numerous HDC/VSA models have been proposed in the literature. All these models share the same basic principles but differ in the HV element data type and in how the core arithmetic operations are implemented [5]. Among them, Binary Spatter Codes (BSC) [8], which use dense binary HVs and simple Boolean arithmetic, offer an attractive trade-off between representational capacity and implementation complexity [9]. They have been shown to enable aggressive hardware optimizations while preserving competitive accuracy across many workloads [4,10].

For these reasons, many prior works have proposed hardware implementations of BSC [11–24]. However, these architectures have seen limited adoption within the broader research community. The primary barrier is not merely architectural flexibility, but accessibility and usability: many state-of-the-art accelerators are proprietary, architecturally rigid or lack standard control interfaces. Consequently, they remain effectively siloed within the hardware design domain and are inaccessible to software developers or data scientists. For instance, many designs are synthesized for fixed model structures or single tasks, requiring complete FPGA re-synthesis for minor application changes [15,25]. Similarly, instruction-set-based approaches [13,20], while programmable, often rely on tight coupling with specific processors, hindering integration into the vast ecosystem of System-on-Chips (SoCs) based on ARM or proprietary processing subsystems.

To bridge the gap between highly specialized accelerators and general-purpose software usability, this paper introduces a General-Purpose AXI Plug-and-Play HDC Accelerator. Unlike previous tightly-coupled designs, this architecture is implemented as a standalone open-source hardware module compliant with the AMBA AXI4 standard, ensuring host-agnosticism and seamless integration into diverse SoC environments. The accelerator natively supports the complete set of BSC primitives and features a configurable dedicated memory, alongside a scalable and synthesis/run-time configurable microarchitecture. The main contributions of this work are as follows:

- **Host-Agnostic Architecture Evolution:** Advancing our previous instruction-set based approach [20], we propose a modular BSC accelerator. Implemented as a standalone hardware unit compliant with the AXI4 standard, it decouples execution from specific processors, enabling seamless integration into diverse SoC environments.
- **Optimized Compute Datapaths:** We introduce targeted microarchitectural optimizations for two area-critical BSC primitives, significantly reducing resource utilization compared to our previous instruction-set based approach [20], while preserving throughput.
- **High-Level Software Abstraction:** We introduce a C++ software library that abstracts low-level hardware interactions. This layer enables developers to invoke accelerated HDC primitives and construct complex pipelines through high-level function calls, bridging the gap between hardware acceleration and software usability.
- **Extensive Validation:** We validate the design on an AMD Zynq-7000 SoC (XC7Z020, AMD Xilinx, San Jose, CA, USA) hosted on a Zybo Z7-20 board (Digilent Inc., Pullman,

WA, USA), benchmarking its performance against a software baseline executed on the embedded dual-core ARM Cortex-A9 processor (Arm Ltd., Cambridge, UK). The analysis quantifies latency and throughput improvements across both elementary HDC operations and complete classification workloads.

To support adoption and lower the barrier to entry for hardware-accelerated HDC research, we release the complete Register Transfer Level (RTL) code, testbenches and software stack as an open-source repository [26].

2. Background and Related Works

2.1. Binary Spatter Code

In BSC [8], each HV is a dense binary vector of dimensionality D , denoted as $\mathbf{h} \in \{0,1\}^D$, where D is typically in the order of thousands, so that randomly generated HVs are quasi-orthogonal with high probability. Computation relies on a small set of elementary vector operations—bundling, binding, and permutation—together with a similarity measure that quantifies how close two HVs are in high-dimensional space and is used for retrieval and recognition. In the following, $\mathbf{h}$ and $\mathbf{g}$ denote two generic hypervectors in $\{0,1\}^D$.

Bundling ($\oplus$), or superposition, combines multiple HVs into a single composite representation that remains similar to its inputs. Given a set of K HVs $\{\mathbf{h}_1, \dots, \mathbf{h}_K\}$, BSC bundling is performed by element-wise accumulation:

$$\mathbf{s} = \sum_{k=1}^K \mathbf{h}_k. \quad (1)$$

Since the resulting vector is no longer binary, a clipping operation is applied to project the representation back into its original domain. In BSC, clipping is implemented via an element-wise majority vote, where ties are broken randomly:

$$[\mathbf{h}_{\text{bun}}]_i = \begin{cases} 1, & \text{if } s_i > \frac{K}{2}, \\ 0, & \text{if } s_i < \frac{K}{2}, \\ \text{rand}\{0,1\}, & \text{if } s_i = \frac{K}{2}, \end{cases} \quad i = 1, \dots, D. \quad (2)$$

Binding ($\otimes$) associates two (or more) HVs into a single HV that is dissimilar to its inputs, enabling the representation of key–value pairs and other structured relations. In BSC, binding is implemented as bit-wise XOR and is invertible:

$$\mathbf{h}_{\text{bind}} = \mathbf{h} \otimes \mathbf{g}, \quad \mathbf{h} = \mathbf{h}_{\text{bind}} \otimes \mathbf{g}. \quad (3)$$

Permutation (ρ) reorders the components of an HV, producing a dissimilar HV, and is commonly used to encode order or positional information. In BSC, permutation is typically implemented as a cyclic right shift.

Finally, similarity (δ) captures how closely two HVs match in the high-dimensional space and supports retrieval, matching, and classification. In BSC, it is commonly evaluated through the Hamming distance:

$$\delta(\mathbf{h}, \mathbf{g}) = d_H(\mathbf{h}, \mathbf{g}) = \sum_{i=1}^D (h_i \text{ xor } g_i), \quad (4)$$

often normalized as $\hat{\delta} = d_H / D \in [0, 1]$.

With these operators, a wide range of objects and their relationships can be encoded and manipulated in the high-dimensional space without increasing dimensionality, supporting both logical reasoning and learning tasks [4,27].

2.2. Related Works

The hardware-acceleration landscape for HDC/VSA spans a wide spectrum of solutions, ranging from highly optimized, task-tailored datapaths to programmable engines that expose HDC/VSA primitives to software. The existing literature can be broadly categorized into: (i) *application-specific accelerators* (fixed-function or domain-tuned datapaths), (ii) *HLS-generated accelerators* (high-level synthesis designs mapped from C/C++/OpenCL) and (iii) *programmable HDC/VSA substrates* (Instruction Set Architecture (ISA) extensions, processor-coupled co-processors or reconfigurable overlays). Across these categories, two recurring limitations emerge: (i) most solutions remain *application- or configuration-specific* (e.g., a fixed learning pipeline, dimensionality D or memory organization) and (ii) only a small subset of designs are released as *open artifacts*, which hinders reuse and reproducibility in custom SoC contexts. Recent surveys further highlight both the rapid progress of HDC for edge intelligence and the fragmentation of available hardware/software artifacts across platforms and toolflows [6].

This limited availability of public artifacts is also observed in adjacent HDC application domains. For instance, a recent review [28] on biomedical HDC studies reports that only 12 out of 62 papers (19.35%) provide publicly available software code, whereas 50 out of 62 (80.65%) do not, and that 35 out of 62 articles are not freely accessible. Although the survey targets biomedical applications, these figures reinforce the broader concern that the lack of openly available artifacts hinders reproducibility and slows technology transfer to real systems.

2.2.1. Task-Specific and Optimization-Driven Architectures

A large body of work maximizes throughput and energy efficiency by specializing the hardware around a fixed HDC pipeline and a narrow configuration space (often a fixed D and encoding scheme). Hardware optimizations for dense binary HDC, including rematerialization of hypervectors, binarized bundling and in-memory computing, are systematically studied in [10] and represent a canonical example of efficiency gains enabled by BSC representations. HD-Core [21], for example, proposes an FPGA accelerator that improves the efficiency of both encoding and associative search by exploiting *computational reuse* in the HDC flow, reducing redundant operations in similarity evaluation. Tiny-HD [11] and the standard-cell memory-based always-on accelerator in [12] similarly pursue aggressive area–power–delay (APD) optimizations for embedded sensing, at the cost of a constrained and largely fixed execution flow. Locality-based encoding and model quantization techniques are explored in [9] to reduce memory/compute pressure while maintaining accuracy, and GENERIC [19] further pushes end-to-end efficiency for edge learning engines by tailoring the pipeline to the target workload.

QuantHD [16] focuses on *model/representation quantization* to reduce arithmetic and storage costs while maintaining competitive accuracy, enabling more resource-efficient implementations. In edge settings where on-chip memory pressure dominates, E3HDC [23] targets the encoder front-end and regenerates item-memory patterns on-the-fly (e.g., via lightweight pseudo-random generators such as linear feedback shift registers), substantially reducing the need for large stored seed memories. Beyond classification, task-tailored accelerators also emerge in other domains, such as genomic pattern matching, where GenieHD [18] demonstrates the suitability of HDC primitives to high-throughput, data-intensive search pipelines.

A complementary direction optimizes the *numeric representation* and storage precision of HDC models. AdaptBit-HD [22] explores *adaptive model bitwidth* to trade accuracy for area/energy, while FATE [24] proposes flexible data types to improve performance/efficiency across workloads. These approaches can yield excellent APD results, but typically preserve the core constraint of fixed datapath and memory organization determined at synthesis/compile time, limiting portability across diverse HDC pipelines and multi-tenant scenarios.

2.2.2. High-Level Synthesis and Hardware-Generation Frameworks

To reduce the engineering effort of handcrafted register-transfer level (RTL) design, several works provide automation frameworks that translate high-level HDC specifications into FPGA implementations. F5-HD [15] automatically generates designs using high-level synthesis by taking as input application characteristics and hardware constraints, while HD2FPGA [25] offers a graphical user interface-driven flow that can produce accelerators for both HDC classification and clustering. AeneasHDC [29] further promotes end-to-end deployment by coupling software and hardware generation in an open-source environment. Despite their practical value, these approaches predominantly deliver *compile-time flexibility*: once a bitstream is synthesized, the resulting accelerator remains largely static, and adapting to a new HV size, encoder or learning task typically requires regeneration and re-synthesis.

2.2.3. Programmable Accelerators and ISA Extensions

A more general-purpose direction is to expose HDC operations through a programmable interface, shifting part of the control and algorithmic flexibility to software while keeping the dominant HDC primitives in hardware. GP-HDCA [30] follows this trend with a general-purpose accelerator for edge computing.

A representative example of a *CPU-accelerator co-processing* approach is EcoFlex-HDP [31], which couples a minimal hyperdimensional processing unit (HPU) to an ARM Cortex-A9 host on Zynq-7000. EcoFlex-HDP is explicitly designed to be *programmable* at the algorithm level by composing a small instruction set covering the three core BSC primitives (Bind/XOR, Permutation/circular shifts and Bound/majority via counters) and by providing a dedicated software stack (library + assembler) that abstracts HPU control from the CPU side. Communication and bulk transfers are handled through memory-mapped control and direct memory access (DMA), using AXI4-Lite/AXI4-Stream connectivity, enabling tight integration with existing software while keeping HDC-heavy kernels offloaded to the HPU. Notably, EcoFlex-HDP releases its platform and experimental code publicly, which is still uncommon in this research space.

FLHDC [17] instead couples a learnable HDC pipeline with an ultra-tiny accelerator to target edge-side workloads, prioritizing compactness and end-to-end deployment.

The tightest coupling is achieved via ISA extensions. RISC-HD [13] integrates selected HDC operations into a RISC-V core microarchitecture to accelerate inference, but remains constrained by fixed architectural assumptions (notably around the supported flow and dimensionality). Our previous Hyperdimensional Coprocessor Unit (HDCU) [20] extends this concept by exposing a richer set of HDC primitives to software (covering training and inference operations), but it also inherits the fundamental drawback of ISA-coupled accelerators: portability is limited by the dependency on a specific core interface and toolchain assumptions.

Table 1 highlights the main gap addressed in this work: although prior HDC accelerators achieve impressive efficiency, only a small subset simultaneously offers (i) *runtime programmability at the primitive level*, (ii) *host-agnostic integration as a standard SoC peripheral* and (iii) *openly released hardware/software artifacts* that enable reuse and reproducibility.

Our design targets this intersection by providing a plug-and-play AXI accelerator that preserves fine-grained control over HDC primitives while decoupling the accelerator from any specific processor microarchitecture or toolchain.

Table 1. Representative HDC hardware acceleration approaches, emphasizing (i) the abstraction exposed to software, (ii) retargeting effort across workloads and (iii) openness of hardware/software artifacts. Retargeting: **Fixed** (redesign typically needed), **Gen.** (tool/HLS-generated per workload at synthesis time), **Runtime** (reprogrammed via instructions/registers). Open artifacts: **Yes** indicates that both HW and SW artifacts are publicly available. “n/s” = not stated/unclear in public artifacts at the time of writing.

Work	Substrate	Exposed Abstraction	Retargeting	Open Artifacts
HD-Core [21]	FPGA	Fixed-function datapath	Fixed	n/s
F5-HD [15]	FPGA	Template/HLS accelerator generator	Gen.	n/s
HD2FPGA [25]	FPGA	Automated generator for CPU-attached accelerators	Gen.	n/s
AeneasHDC [29]	FPGA	Open automated deployment framework	Gen.	Yes
E3HDC [23]	FPGA	End-to-end open toolflow + pipeline	Gen.	Yes
LAHDC [14]	ASIC/FPGA	Task-specific query accelerator + generator	Gen.	n/s
GP-HDCA [30]	FPGA SoC	Instruction-driven accelerator (MMIO + streams)	Runtime	n/s
EcoFlex-HDP [31]	FPGA SoC	Programmable platform (HDC primitive set)	Runtime	Yes
RISC-HD [13]	RISC-V	Lightweight processor/ISA support for inference	Runtime	n/s
HDCU [20]	RISC-V	ISA extension + coprocessor	Runtime	Yes
This Work	FPGA SoC	Host-agnostic AXI Unit (MMIO + SPM offload)	Runtime	Yes

2.2.4. The HDCU Microarchitecture

Before presenting the proposed standalone accelerator, we briefly revisit HDCU [20] on which our work builds. HDCU represents a tightly-coupled approach to HDC acceleration based on custom instructions integrated into a RISC-V core. Summarizing its organization and programming model is useful for two reasons: (i) several design choices (e.g., scratchpad-based execution and runtime-controlled hardware loops) directly inspire parts of the new IP and (ii) the integration constraints of ISA-coupled coprocessors motivate the architectural shift toward a fully standalone, plug-and-play AXI module.

RISC-V Core Integration

HDCU is embedded into the execution stage of the Klessydra-T03 32-bit RISC-V core [32], operating alongside the scalar pipeline resources (e.g., ALU/LSU). Integration is achieved through core-level RTL modifications: HDC instructions are decoded by the core and dispatched to the coprocessor when the accelerator is available, while results are committed according to the instruction semantics. This tight coupling minimizes dispatch latency and avoids bus transactions for intermediate results, which is particularly beneficial for iterative HDC kernels.

Control is provided by a custom RISC-V instruction set extension (ISE) mapping the primitive HDC operators for BSC HVs, including memory transfers (hvmemld/hvmemstr), binding (hvbind), bundling (hvbundle) and clipping (hvclip), permutation (hvperm), similarity (hvsim) and an inference-oriented associative search primitive (hvsearch). Importantly, this form of customization is enabled by the open and modular nature of the RISC-V ISA standard, which explicitly allows designers to introduce domain-specific extensions while preserving compatibility with the base ISA and standard toolchains. The ISE follows a standard R-type encoding: the opcode selects the HDC extension space, funct3 encodes the datatype (binary in the reference implementation, with forward-looking support for additional types) and funct7 selects the operation. Instruction encodings are integrated into the RISC-V GNU toolchain and exposed via C intrinsics, enabling compiler-level optimizations while keeping the programming interface accessible.

To reduce pressure on the load/store subsystem during iterative kernels, HDCU operates on HVs stored in dedicated multi-bank scratchpad memories (SPMs). The number of banks and the per-bank capacity are configurable at synthesis time, enabling a trade-off between on-chip memory footprint and area. HVs are moved between main memory and SPMs through the scratchpad memory interface (SPMI) using dedicated custom instructions; in the reference design, the SPMI datapath is 32-bit wide, so transfers incur a latency proportional to $HV_SIZE/32$ cycles. SPMs can be initialized at synthesis time with commonly used Base and Level vectors, reducing runtime setup overhead in typical HDC pipelines.

The execution engine comprises dedicated functional units (FUs) specialized for each primitive (e.g., XOR for binding, counter/comparator arrays for bundling/clipping, rotation datapath for permutation, XOR + popcount + accumulation for similarity). Hardware parallelism is controlled at synthesis time through a SIMD parameter, and individual FUs can be enabled/disabled to match resource budgets. Importantly, HDCU decouples the logical hypervector dimension from the physical datapath width: a dedicated control/status register (HVSIZE) sets HV_SIZE at runtime and drives internal hardware loops, allowing the same synthesized microarchitecture to process different HV sizes without re-synthesis. Additional registers (e.g., prototype count for hvsearch) reduce software loop overhead in common kernels.

A key aspect of the HDCU is its synthesis-time parametrization of both compute and memory, enabling design-space exploration without modifying RTL. The primary knob is the parallelism factor SIMD, which defines the number of 32-bit lanes processed per cycle and therefore the physical datapath width:

$$W_{\text{data}} = \text{SIMD} \times 32 \text{ bits.} \quad (5)$$

On the memory side, each SPM bank is organized as a 32-bit word array whose depth is governed by ADDR_WIDTH. The total scratch pad capacity is:

$$M_{\text{total}} = \left(\frac{2^{\text{ADDR_WIDTH}} \times 32}{8 \times 1024} \right) \text{KB} \times \text{SPM_NUM}, \quad (6)$$

where SPM_NUM is the number of independent banks. Finally, COUNTER_BITS configures the precision of the saturating counters used for bundling, while an optional PERF_COUNTER flag enables profiling counters to expose per-operation cycle counts to software.

2.3. Moving Beyond ISA Coupling

Despite its fine-grained programmability and low dispatch overhead, ISA-coupled acceleration imposes practical integration constraints when targeting heterogeneous embedded ecosystems. First, the accelerator is not a drop-in peripheral: integrating HDCU requires modifying (or selecting) a compatible RISC-V core microarchitecture to add the coprocessor datapath, local SPMs and decode/dispatch support; this limits portability across mainstream SoCs dominated by ARM-based processing subsystems. Second, even when the ISE is integrated into a GNU-based toolchain, deploying custom instructions typically entails a non-standard software flow (toolchain patches, intrinsics and low-level memory management), which is at odds with Linux-centric development and rapid prototyping in conventional OS environments. Third, system-level integration features that are central in FPGA/SoC deployments are more naturally supported by bus-attached, memory-mapped accelerators than by in-pipeline coprocessors. Finally, ISA-coupled designs implicitly raise the entry barrier for end users: evaluating or reusing the accelerator often requires familiar-

ity with hardware integration (RTL, SoC build flows and toolchain customization), which may be outside the skill set of researchers focused primarily on HDC algorithms.

For these reasons, this work moves from an ISA-coupled coprocessor to a fully standalone and *plug-and-play* AXI-based hardware module. The proposed accelerator preserves primitive-level programmability, but exposes it through standard AXI4-Lite/AXI4-Stream interfaces to support DMA-driven transfers and direct control from Linux user space via a standard Application Programming Interface (API) stack. This choice also lowers the adoption barrier: the accelerator can be distributed with a ready-to-run reference platform (bitstream and Linux boot files, plus API examples), enabling algorithm developers to deploy and test HDC kernels by simply booting the target board and calling the provided software interface, without requiring RTL modifications or hardware design expertise.

3. Architecture Implementation

The proposed accelerator is implemented as a standalone, highly parameterized unit that offloads HDC/VSA primitives from the host processor. The design follows a plug-and-play integration model based on standard AMBA AXI4 interfaces: the accelerator is exposed as a memory-mapped peripheral for control and as streaming endpoints for bulk data movement, making it portable across heterogeneous FPGA SoCs (e.g., Xilinx Zynq-7000, UltraScale+) and across different host processors.

This section describes the architecture with a top-down approach. We first present the system-level integration and dataflow (Figure 1), then zoom into the internal microarchitecture (Figure 2), highlighting the key architectural changes introduced with respect to our previous ISA-coupled HDCU design. Finally, we detail the control logic enabling runtime-scalable execution and the Linux-oriented software stack (GitHub repository commit 3ae3b46) used to orchestrate composite HDC kernels.

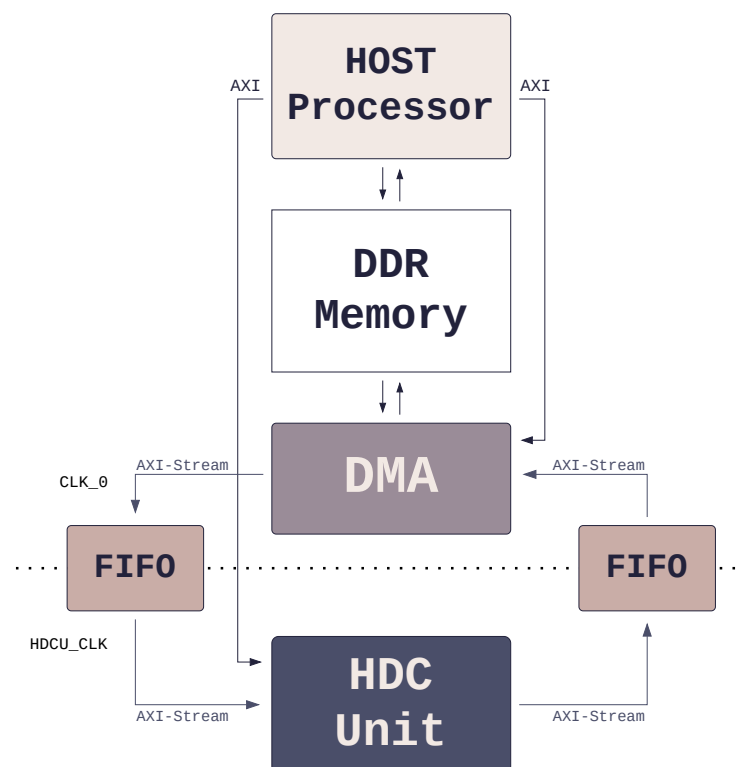


Figure 1. System-level integration of the HDC Accelerator IP within an SoC environment. The host processor configures the accelerator via AXI4-Lite and controls a DMA engine that performs bulk transfers over high-bandwidth AXI4-Stream links. Optional FIFO buffers are shown to decouple clock domains when CDC is required.

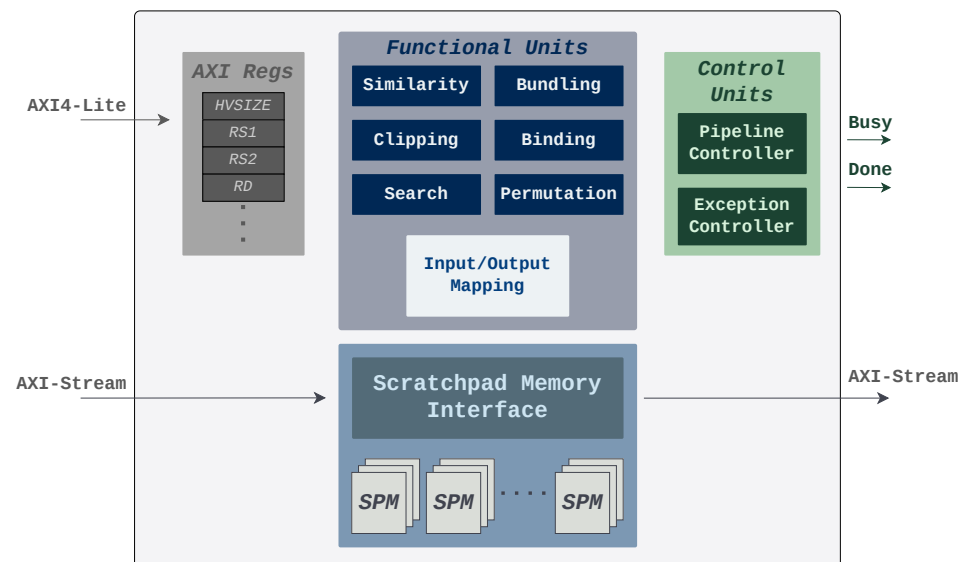


Figure 2. Internal microarchitecture of the HDC Accelerator. The design features a parameterizable SIMD datapath with dedicated Functional Units for BSC operations (Binding, Bundling, Permutation, Similarity, clipping, search), tightly coupled to a banked Scratchpad Memory via a dual-port interconnect.

3.1. System Overview

Figure 1 illustrates the reference system-on-chip integration scheme adopted for this work. While the proposed design is implemented as a standalone hardware accelerator adaptable to various interconnect topologies, this specific block design represents the recommended configuration employed for the experimental validation presented in Section 5. The architecture partitions the workload between the Host Processor and the rest of the FPGA fabric, utilizing the AMBA AXI4 standard to establish two distinct operational planes:

- **Control Plane:** The accelerator unit can be configured via write operations on memory-mapped registers. Each operation requires different parameters such as the address of the operands, the size of the hypervectors and the destination address. Read-only status registers allow the host processor to monitor the execution of each operation.
- **Data Plane:** A Direct Memory Access (DMA) engine manages bulk data movement between the off-chip DDR memory and the accelerator via AXI4-Stream interfaces. It is worth noting that the software driver (detailed in Section 4) is tightly coupled with this topology, providing abstract primitives specifically designed to coordinate accelerator and Xilinx DMA transactions.

From a timing perspective, the system integration inserts asynchronous First-In-First-Out (FIFO) buffers at the streaming boundaries of the accelerator. While a fully synchronous design sharing a common clock tree could theoretically allow direct AXI-Stream connections to the internal memories, the deployment of asynchronous FIFOs provides robust Clock Domain Crossing (CDC). This isolation ensures that the accelerator logic can be clocked at its maximum achievable frequency (HDCU_CLK), independent of the system bus frequency (CLK_0).

3.2. Architectural Optimizations

The proposed design introduces targeted microarchitectural enhancements compared to the baseline HDCU architecture summarized in Section 2.2.4. To address performance bottlenecks and optimize area utilization, we implemented specific modifications at both the arithmetic and memory levels. In particular, the *Permutation* and *Clipping* functional units have been redesigned to reduce hardware resource utilization while maintaining execution efficiency. Table 2 details the reduction in hardware resources achieved by the proposed

designs. Notably, these optimizations preserve the original execution latency. As neither unit lies on the critical path of the architecture, these area savings do not compromise the operating frequency of the system. Complementing these compute-centric optimizations, the SPMI has been re-engineered to seamlessly support the streaming-based dataflow.

Table 2. FPGA Resource Utilization Comparison: Original vs. Optimized Functional Units.

Version	Permutation Unit		Clipping Unit	
	#LUTs	#FFs	#LUTs	#FFs
Original	14,384	64	335	64
Optimized	REMOVED		16	34

3.2.1. Functional Units

Permutation

In the baseline HDCU architecture, the permutation logic (*hvxperm*) implements the BSC cyclic right rotation with bit-level granularity. While this design maximizes flexibility by supporting arbitrary shift amounts s , it necessitates a temporary buffer register to carry the s least-significant bits across consecutive SIMD chunks during streaming. This mechanism introduces non-trivial hardware complexity, as wide rotations require resource-intensive shuffle networks and multiplexing logic, representing a well-known area bottleneck compared to purely bitwise primitives.

In the proposed standalone architecture, we eliminate the dedicated permutation FU entirely, opting instead to virtualize the permutation operation within the memory addressing logic. This architectural choice draws upon the optimization strategy analyzed in MCR-HDCU [33], which demonstrates that structured permutations—such as block-cyclic shifts—are sufficient to maintain the quasi-orthogonality required by HDC/VSA models, thus avoiding the overhead of arbitrary bit-level rotations. Consequently, permutation is performed at the granularity of whole SIMD blocks: rather than physically shifting bits, the controller reorders the data stream by manipulating the SPM read pointers.

Implementation-wise, let W_{data} be the physical datapath width and let a hypervector of size D be stored as $N = D/W_{\text{data}}$ consecutive blocks. A permutation by k blocks is realized simply by offsetting the read address so that the i -th output block is fetched from the $(i + k) \pmod{N}$ input block. This approach reduces the permutation logic to basic address arithmetic (offset and wraparound), effectively removing the latency penalty associated with data shuffling logic.

The primary trade-off of this optimization is the coarser granularity: the shift amount is constrained to multiples of W_{data} (i.e., $s = k \cdot W_{\text{data}}$). However, since the permutation operator in HDC primarily serves as a bijective transformation for position or role encoding, exact bit-level granularity is rarely critical. As evidenced by prior work [33], block-cyclic schemes retain sufficient decorrelation properties while yielding significant savings in silicon area and routing resources. To verify this, we performed an empirical comparison between the proposed block-granular permutation and standard bit-wise rotation. The results confirmed that the block-level approach preserves the bijective and decorrelating properties required by the BSC algebra, yielding no degradation in classification accuracy across the tested benchmarks while eliminating the hardware cost of a fine-grained shifter.

Clipping Unit

In BSC, the bundling operation aggregates multiple binary vectors into integer components, effectively widening the dynamic range of the hypervector. The *hvcclip* primitive is responsible for projecting these integer counters back into the binary domain by applying an element-wise threshold.

In the baseline HDCU architecture, clipping is implemented using a comparator array that evaluates all SIMD lanes in parallel. Since the integer counters are packed within 32-bit words (specifically, `COUNTERS_NUMBER` counters of `COUNTER_BITS` bits each), the baseline unit reconstructs the final binary output progressively over `COUNTER_BITS` cycles. However, it relies on variable bit-indexing to place the comparison results into the destination register, a mechanism that requires wide dynamic multiplexers and complex routing logic to address individual bit positions.

In this work, we retain the semantic behavior and latency model of the baseline but restructure the datapath to minimize silicon area. Instead of employing variable indexing, the optimized unit generates a packed comparison block of width $W_{\text{blk}} = \text{SIMD} \times \text{COUNTERS_NUMBER}$ in a single step using constant part-selects. This block is then concatenated into an output shift-register, which reconstructs the full $\text{SIMD} \times 32$ -bit binary word by shifting and appending partial results over `COUNTER_BITS` cycles. By replacing dynamic multiplexing with a regular shift/append structure, we significantly simplify the routing congestion. This optimization results in a highly compact footprint of just 16 LUTs and 34 FFs, while maintaining full runtime scalability with respect to `HVSIZE`.

3.2.2. Memory Subsystem

To match the throughput of the functional units, the memory subsystem is organized as a series of independent banked SPMs. The number of banks directly depends on the SIMD parameter and also matches the width of the axi-stream interfaces, in order to maximise the bandwidth of the data transfers. The use of multiple SPMs allows the unit to perform load, store and BSC operations simultaneously, as long as different SPMs are targeted. A dedicated SPMI manages read and write operations as well as access conflicts. Each memory bank is implemented as a simple dual-port memory, which is most suitable for FPGA implementations that leverage BRAM units.

A significant architectural enhancement over the original HDCU design [20] is the redesign of the SPMI to support high-bandwidth streaming without CPU intervention. We integrated native *AXI4-Stream* interfaces directly into the memory controller to enable burst transfers to and from the dedicated memories. The data width of these interfaces scales dynamically with the SIMD generic ($W_{\text{stream}} = \text{SIMD} \times 32$), ensuring that external bandwidth matches the internal datapath throughput.

Furthermore, the interface now includes dedicated hardware for address decoding and routing. In the previous HDCU implementation, this address translation task was offloaded to the Load Store Unit (LSU) of the host Klessydra RISC-V processor; incorporating this logic directly into the accelerator decouples memory management from the host architecture, enabling true standalone operation.

Finally, the internal memory controller features sophisticated arbitration logic that supports simultaneous dual-operand reads by utilizing both BRAM ports to fetch two operands (e.g., for Binding or Similarity) in a single clock cycle, effectively preventing pipeline stalls. Additionally, it handles concurrent DMA access by prioritizing the HDC pipeline while granting cycles to the DMA engine whenever the functional units are idle, thereby maximizing bandwidth efficiency.

4. Software Stack and Programming Model

To facilitate rapid prototyping and flexible orchestration, we developed a multi-layered C++ software stack that operates entirely in Linux userspace. By leveraging standard kernel interfaces, the stack abstracts low-level hardware details while exposing fine-grained control over memory and compute resources in the accelerator. As summarized in Figure 3, the architecture is organized into three abstraction layers—Application, HDC API and

Driver—built on top of standard Linux kernel interfaces for memory reservation and address-space mapping. The main API entry points and their description are summarized in Table 3, while a representative end-to-end inference flow is provided in Listing 1.

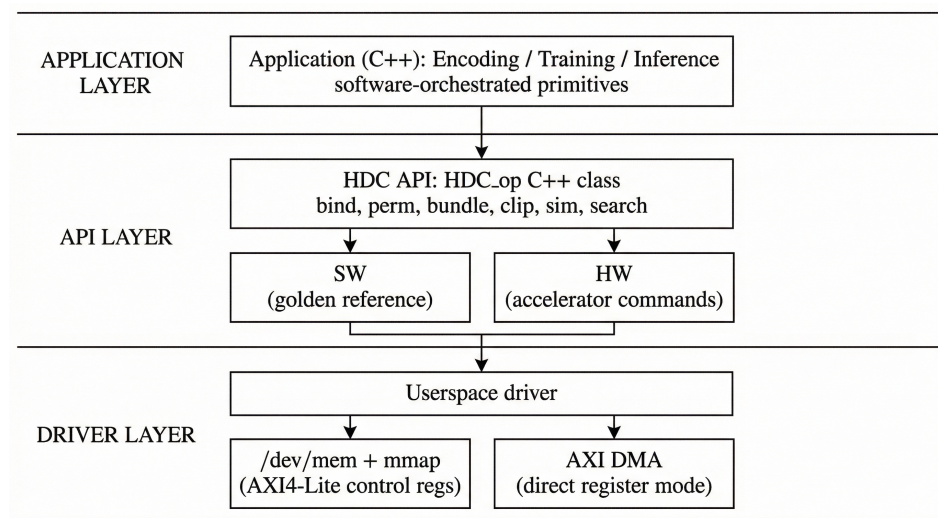


Figure 3. Overview of the proposed software stack and programming model. The C++ application orchestrates HDC primitives through the `HDC_op` API, which supports both software emulation and hardware dispatch. A userspace driver maps AXI4-Lite control registers via `/dev/mem` and manages DMA transfers through pre-reserved contiguous buffers defined in the Device Tree.

Table 3. Main High-Level C++ API for the Hardware-Accelerated HDC Pipeline. Functions operate on Scratchpad Memory (SPM) unless otherwise noted.

Category	Function	Description
Data Movement	<code>hvmemld</code>	Transfers data from DDR (Host) to SPM via DMA (MM2S).
	<code>hvmemstr</code>	Transfers data from SPM to DDR (Host) via DMA (S2MM).
Primitives	<code>hvbind</code>	Binding: Bitwise XOR of two binary HVs.
	<code>hvbundle</code>	Bundling: Accumulates a binary HV into a multi-bit vector (counters) with saturation.
	<code>hvclip</code>	Clipping: Binarizes a multi-bit vector via majority vote using a configurable threshold.
	<code>hvperm</code>	Permutation: Cyclic shift of an HV by k positions.
	<code>hvsim</code>	Similarity: Computes Hamming distance between two HVs.
	<code>hvsearch</code>	Associative Search: Queries a hypervector against a class memory to find the closest match (prediction).
Accelerated Pipelines	<code>accl_enc</code>	Full encoding pipeline: orchestrates bind, bundle and clip primitives over input features.
	<code>accl_train</code>	Executes encoding and updates the class prototype (bundling).
	<code>accl_infer</code>	Executes encoding and predicts the class label using the associative search.

The design targets portability across Zynq-class embedded Linux systems and requires only minimal platform configuration via Device Tree entries (AXI4-Lite reg ranges, reserved-memory regions for DMA buffers and appropriate memory mapping attributes).

Listing 1. Minimal inference pipeline using the proposed C++ API (illustrative pseudocode).

```

1  int main() {
2      HDC_op hdc(HV_SIZE_BIT, FEATURE_NUM, LEVEL_NUM);
3      Dataset test_dataset = load_dataset("test.csv");
4      Model model          = load_model("model.bin");
5
6      // Preload model HVs into accelerator SPMs
7      for (f=0; f<FEATURE_NUM; f++)
8          hdc.hvmemld(SPM_A_START_ADDR + f*sizeof(HV), model.base[f], sizeof(
9              HV));
10     for (l=0; l<LEVEL_NUM; l++)
11         hdc.hvmemld(SPM_B_START_ADDR + l*sizeof(HV), model.level[l], sizeof(
12             HV));
13     for (c=0; c<CLASS_NUM; c++)
14         hdc.hvmemld(SPM_D_START_ADDR + sizeof(BundledHV) + c*sizeof(HV),
15             model.cls[c], sizeof(HV));
16
17     float      qlevels[LEVEL_NUM + 2] = make_quant_levels(0.0, 1.0);
18     uint32_t qfeat[FEATURE_NUM];
19
20     // Inference on all test samples
21     for (i=0; i<test_dataset.num_samples; i++) {
22         sample = get_sample(&test_dataset, i);
23
24         for (f=0; f<FEATURE_NUM; f++){
25             qfeat[f] = get_quantized_level(sample->features[f], qlevels);
26
27             hdc.accl_inference(qfeat, SPM_A_START_ADDR, SPM_B_START_ADDR,
28                 SPM_D_START_ADDR + sizeof(BundledHV));
29
30             uint32_t y_hat = 0;
31             hdc.hvmemstr(&y_hat, SPM_C_START_ADDR, sizeof(uint32_t)*SIMD);
32         }
33     }
34 }

```

4.1. Driver Layer

The foundational layer manages physical resources and communication protocols. To avoid the complexity of custom kernel-module development during prototyping, we adopt a fully userspace driver approach. In particular, the stack uses `mmap` on `/dev/mem` to map both AXI4-Lite control registers and pre-allocated DMA buffers into the application's virtual address space, enabling low-latency configuration and data transfers in controlled environments.

High-throughput data movement relies on contiguous physical memory reserved at boot time via Device Tree reserved-memory nodes. Two dedicated regions are allocated: a source buffer at physical address `0x30000000` and a destination buffer at `0x34000000`, each sized at 64 MiB. These buffers are mapped with appropriate cache attributes (coherent or non-cacheable, depending on the platform configuration) to avoid explicit software-managed cache maintenance operations. The DMA engine operates in Direct Register Mode without scatter-gather support, transferring data between these userspace-accessible buffers and the on-chip SPMs via AXI4-Stream interfaces. Synchronization is currently implemented via busy-wait polling of both DMA status registers (checking the `IOC_IRQ` and `IDLE` flags) and HDC control registers, which simplifies the control flow and provides deterministic latency for short transfers.

4.2. API Layer

The core functionality is encapsulated in the `HDC_op` C++ class, which translates high-level algorithmic intent into hardware commands. The core API functions and their semantics are summarized in Table 3. Unlike cache-based architectures, the API exposes the internal scratchpad hierarchy as an explicitly software-managed resource. Data placement is controlled via the `hvmemld/hvmemstr` primitives, where `spm_addr` encodes both the target SPM bank and the byte offset within that bank. Transfer sizes must be specified explicitly in bytes; while the current implementation does not enforce strict alignment constraints at the API layer, optimal performance requires data sizes that are multiples of the hardware stream width $W_{\text{stream}} = \text{SIMD} \times 32$ bits. The API validates memory accesses through boundary checks against the reserved DMA buffer regions and provides error reporting via return codes and status-register inspection.

The class further provides a unified dual-mode interface that supports both pure software emulation and hardware acceleration. Each HDC primitive (`similarity`, `bind`, `permutation`, `bundle`, `clip`, `search`) is implemented in both software (for golden-reference verification) and hardware (via accelerator instruction dispatch). This design enables algorithm verification and functional testing entirely in software before FPGA deployment, facilitating rapid development and debugging cycles.

4.3. Application Layer

At the highest level, the stack enables the construction of composite HDC pipelines through software orchestration rather than fixed hardware state machines. The accelerator exposes only atomic primitives (`hvbind`, `hvbundle`, `hvperm`, `hvsim`, `hvsearch`, `hvclip`), which are composed programmatically at the application level to implement complete encoding, training and inference workflows. A minimal end-to-end inference pipeline illustrating a typical usage pattern of the `HDC_op` API (model preload to SPMs, feature quantization, accelerator invocation and result readback) is reported in Listing 1. For example, the `acc1_encoding` method sequences `hvbind` (to combine level and base vectors in record-based encoding [27]), `hvbundle` (to accumulate feature representations) and `hvclip` (to perform majority-based binarization) within a C++ loop that iterates over all dataset features. Similarly, *n*-gram language models [29] can be constructed by chaining `hvperm` operations with varying shift parameters, where—as discussed in Section 3.2.1—shifts are constrained to block-granular multiples of $32 \times \text{SIMD}$ bits to match the architecture of the hardware permutation unit.

This software-orchestrated approach decouples algorithmic flexibility from hardware implementation. Modifying encoding strategies (e.g., switching from linear to thermometer encoding), adjusting bundling thresholds or implementing novel training procedures requires only recompiling the host application, without regenerating the FPGA bitstream. The trade-off is increased communication overhead between the CPU and accelerator compared to fully pipelined hardware datapaths; however, this is mitigated by efficient DMA transfers and the ability to batch operations within SPM boundaries before host synchronization is required.

5. Experiments

This section presents a comprehensive evaluation of the proposed AXI-based HDC accelerator. We begin by detailing the experimental platform and analyzing the hardware implementation results, focusing on the trade-offs between resource utilization and operating frequency across various parallelism configurations. Subsequently, we quantify the computational efficiency through a two-tiered benchmarking approach: micro-benchmarks targeting individual BSC primitives to assess peak throughput, and macro-benchmarks on

standard datasets to evaluate end-to-end training and inference acceleration relative to the embedded software baseline.

5.1. Experimental Setup

The complete system, schematically depicted in Figure 1, was implemented on a Digilent Zybo Z7-20 development board (Digilent Inc., Pullman, WA, USA). The target device is a Zynq-7000 XC7Z020-1CLG400C SoC (AMD Xilinx, San Jose, CA, USA), which integrates a dual-core ARM Cortex-A9 processor (Arm Ltd., Cambridge, UK) (Processing System, PS) and Artix-7 based Programmable Logic (PL).

As detailed in Table 4, the experimental platform adopts a hardware–software co-design approach:

- **Processing System (Host):** The ARM processor executes the software stack atop an embedded Linux distribution generated via PetaLinux 2022.1 (AMD Xilinx, San Jose, CA, USA) with Linux kernel version 5.15.19-xilinx-v2022.1. This partition manages the high-level orchestration, leveraging the mmap-based userspace driver and standard userspace I/O interfaces to interact with the hardware. The C++ software stack used for all experiments corresponds to GitHub commit 3ae3b46.
- **Programmable Logic (Device):** The PL partition hosts the HDC accelerator instantiated alongside the AXI DMA engine (AMD Xilinx, San Jose, CA, USA). Interconnects are established via the AXI4-Lite protocol for low-latency register configuration and the AXI4-Stream protocol for high-bandwidth data ingestion from the main memory. The hardware design was synthesized and implemented using Vivado Design Suite 2022.1 (AMD Xilinx, San Jose, CA, USA).

Table 4. Experimental Platform Specifications and Toolchain Details.

Parameter	Specification
Host System (PS)	
Processing Unit	Dual-core ARM Cortex-A9 @ 667 MHz (Arm Ltd., Cambridge, UK)
Operating System	PetaLinux 2022.1 (AMD Xilinx, San Jose, CA, USA)
Kernel Version	5.15.19-xilinx-v2022.1
Driver Interface	Userspace I/O (via /dev/mem)
Accelerator Domain (PL)	
Target Device	Zynq-7000 XC7Z020 (AMD Xilinx, San Jose, CA, USA)
Synthesis Tool	Vivado Design Suite 2022.1 (AMD Xilinx, San Jose, CA, USA)
Clock Frequency	100 MHz (System), Async HDC Core
Data Interface	AXI4-Stream (via AXI DMA)
Control Interface	AXI4-Lite (Memory Mapped)

To evaluate the architecture under realistic workloads, we selected three standard datasets widely used in the HDC literature, namely ISOLET [34], IRIS [35], and UCI-HAR [36] (Table 5). All benchmarks are formulated as classification tasks. For all selected datasets (ISOLET, IRIS, UCI-HAR), input features are normalized to the $[0, 1]$ range and quantized into L discrete levels (typically $L = 10$) to map them to continuous item memory level vectors.

Table 5. Benchmark Datasets Characteristics.

Dataset	Features	Classes	Total Samples	Task Type
ISOLET [34]	617	26	7797	Classification
IRIS [35]	4	3	150	Classification
UCI-HAR [36]	561	6	10,299	Classification

5.2. Hardware Implementation Results

Tables 6 and 7 summarize post-implementation resource utilization on the Zynq XC7Z020 FPGA across varying parallelism levels (SIMD ranging from 4 to 32). The reported figures refer to the *complete block design implemented in the PL*, including the HDC accelerator module, AXI SmartConnect interconnect, AXI DMA engine and the AXI-Stream infrastructure (e.g., CDC/AXIS FIFOs). The parameter SIMD scales the internal datapath and stream width ($W_{\text{stream}} = \text{SIMD} \times 32 \text{ bits}$), increasing pressure not only on the compute logic but also on the transport fabric required to sustain line-rate transfers.

Table 6. Post-implementation resource utilization on the Zynq XC7Z020 for different SIMD configurations. Fixed parameters: ADDR_WIDTH = 16, COUNTER_BITS = 16. f_{impl} denotes the maximum frequency at which timing closure is achieved.

SIMD	f_{impl} [MHz]	#Slice LUTs	#LUTRAM	#Flip-Flops
4	110	10,185 (19.14%)	843 (4.84%)	11,258 (10.58%)
8	100	14,664 (27.56%)	1238 (7.11%)	15,847 (14.89%)
16	90	24,409 (45.88%)	1982 (11.39%)	24,753 (23.26%)
32	77	44,347 (83.36%)	3454 (19.85%)	42,435 (39.88%)

Table 7. Post-implementation resource utilization on the Zynq XC7Z020 for different SIMD configurations for Accelerator Unit, DMA unit and interconnect network of the PL. Fixed parameters: ADDR_WIDTH = 16, COUNTER_BITS = 16.

SIMD	Component	Resource Utilization			
		LUT as Logic	#FF	#LUTRAM	#BRAM
4	HDC Unit	4600	3028	0	64
	DMA Unit	1871	3091	117	5
	Interconnect	2249	4286	664	0
8	HDC Unit	7407	4501	0	64
	DMA Unit	2894	4668	146	9
	Interconnect	2496	5827	1030	0
16	HDC Unit	14,030	7429	0	64
	DMA Unit	4773	7613	152	18.5
	Interconnect	2990	8858	1768	0
32	HDC Unit	27,298	13,241	0	64
	DMA Unit	8336	13,379	156	34
	Interconnect	4586	14,962	3236	0

As expected, overall utilization increases monotonically with SIMD. Slice LUT usage rises from 19.14% at SIMD = 4 to 83.36% at SIMD = 32, while Flip-Flop usage scales from 10.58% to 39.88%. Importantly, the scaling trend is dominated by the accelerator itself: the accelerator instance grows from 4600 LUTs at SIMD = 4 to 27,298 LUTs at SIMD = 32, accounting for an increasing fraction of the total logic (approximately 45% to 62% of system LUTs across the sweep). In addition, the supporting infrastructure also scales with the stream width: AXI DMA increases from 1988 to 8522 Slice LUTs, and AXI SmartConnect

from 2913 to 7822 Slice LUTs, reflecting the higher routing and buffering complexity of wide AXI datapaths.

LUTRAM consumption becomes significant at higher parallelism levels (up to 19.85%), but the utilization breakdown shows that it is almost entirely attributable to the interconnect fabric rather than to the HDC core (the accelerator itself uses 0 LUTRAM in all configurations). At SIMD = 32, for example, AXI SmartConnect alone accounts for 3236 LUTRAM primitives out of the total 3454. This increase contributes to routing congestion and exacerbates timing closure at high utilization.

The maximum implemented operating frequency (F_{impl}) degrades as the design widens: timing closure is achieved at 110 MHz for SIMD = 4, decreasing to 77 MHz for SIMD = 32. This behavior is primarily due to increased logic density and routing complexity when distributing wide control signals and routing wide datapaths through the interconnect and DMA subsystems.

Regarding the memory subsystem, the scratchpad depth was fixed by keeping ADDR_WIDTH = 16 constant across all configurations; therefore, the accelerator BRAM footprint remains invariant at 64 BRAM tiles. However, *system-level* BRAM usage increases with SIMD due to AXI DMA buffering and AXI-Stream FIFOs, growing from 73 BRAM tiles (SIMD = 4) to 127 BRAM tiles (SIMD = 32). The breakdown confirms this trend: at SIMD = 32, BRAM usage is composed of 64 (SPMs in the HDC core) + 34 (AXI DMA) + 29 (AXIS FIFO infrastructure, reported as 14.5 + 14.5 due to BRAM18 granularity). Since FIFO depths are synthesis-time parameters, this overhead can be tuned to trade buffering capacity for BRAM footprint depending on system constraints.

5.3. Performance Evaluation Methodology

To evaluate the computational efficiency of the proposed architecture, we measure the speedup S relative to a software baseline executed on a single ARM Cortex-A9 core embedded in the Xilinx XC7Z020 SoC. The speedup is defined as:

$$S = \frac{T_{\text{SW}}}{T_{\text{HW}}}, \quad (7)$$

where T_{SW} denotes the execution time of a BSC operation on the CPU, and T_{HW} represents the time required by the accelerator to execute the same BSC primitive. Specifically, T_{SW} is measured in Linux user space utilizing high-resolution POSIX timers (CLOCK_MONOTONIC), whereas T_{HW} is obtained via cycle-accurate performance counters integrated into the control logic of the accelerator. These counters record the total clock cycles elapsed between the assertion and deassertion of the enable signal of the requested FU. The physical time is derived as $T_{\text{HW}} = N_{\text{cycles}}/F_{\text{impl}}$, where F_{impl} denotes the synthesized clock frequency for a given hardware parallelism level. Data transfers duration to and from scratchpad memories depends on the SIMD parameter and on the size of the hypervectors. The maximum throughput of the scratchpad memories is $\text{SIMD} \cdot 4 \cdot F_{\text{impl}}$ B/s for both read and write operations, supposing there aren't access conflicts that would halt the operations and the input/output FIFOs are always ready to transmit/receive data.

The experimental campaign is structured into two phases:

- **Micro-benchmarks:** We assess the speedup achieved on the fundamental operations underlying the BSC model (binding, bundling, clipping, permutation, similarity and associative search) to characterize the efficiency of the accelerator at the primitive level.
- **Macro-benchmarks:** We evaluate the aggregate speedup obtained on complete training and inference phases by implementing a standard accumulation-based HDC classification algorithm [27] on the datasets described in Table 5. In this phase, the initialization cost associated with transferring persistent hypervectors (e.g., Base Vec-

tors, Level Vectors and Class Prototypes) from the main DDR memory to the SPMs is excluded from the speedup calculation, thereby reflecting a typical edge scenario where model parameters are cached on-chip and reused across multiple training/classification iterations.

It is important to note that the focus of this analysis is not to assess the accuracy achieved by the accelerator, since every primitive operation implemented in hardware, as well as the full encoding, training and inference flow, has been rigorously validated against a software “golden reference”. This verification ensures the accelerator delivers the exact same accuracy as an HDC software model. Consequently, accuracy is determined solely by the application-level choices (encoding and learning rule) and is independent of the underlying execution engine.

5.3.1. Performance on BSC Operations

Table 8 reports the speedup of each BSC primitive over the ARM Cortex-A9 baseline. Overall, the accelerator delivers substantial gains across all operations, and the speedup generally increases with the SIMD width. However, scaling is not perfectly proportional because wider SIMD configurations achieve lower implemented clock frequencies (from 110 MHz at SIMD = 4 down to 77 MHz at SIMD = 32).

Table 8. Speedup (S) of atomic BSC operations vs. software baseline on ARM Cortex-A9 @ 667 MHz. FPGA results are reported for the implemented frequencies of each SIMD configuration (110/100/90/77 MHz for SIMD = 4/8/16/32).

Operation	HV Size [bit]	Speedup (S)			
		Hardware Parallelism (SIMD)			
		4	8	16	32
Binding	1024	30.00	42.85	60.00	57.69
	4096	12.50	21.05	36.36	43.95
	8192	6.63	11.42	19.13	27.97
Bundling	1024	30.22	189.47	354.54	428.57
	4096	32.26	225.37	431.42	414.2
	8192	64.77	115.26	204.88	331.86
Clipping	1024	103.77	173.68	300.00	362.63
	4096	93.27	165.67	317.14	343.19
	8192	47.61	84.73	150.61	243.95
Similarity	1024	59.32	66.66	85.71	76.92
	4096	44.11	71.42	115.38	76.92
	8192	24.15	40.54	64.93	88.75
Search	1024	54.94	68.18	107.14	115.38
	4096	43.01	57.14	105.26	120.19
	8192	16.96	29.85	51.94	80.97
Permutation	1024	73.17	120.00	200.00	n/a
	4096	36.66	64.70	122.22	123.07
	8192	18.8	33.33	58.82	94.01

Note: n/a indicates a non-admissible permutation configuration (HV fits in a single SIMD block).

The magnitude of the benefit is strongly operation-dependent. *Bundling* and *clipping* consistently achieve the highest accelerations, reaching up to $431\times$ (bundling, HV size 4096, SIMD = 16) and up to $363\times$ (clipping, HV size 1024, SIMD = 32). This behavior is expected, as both primitives are dominated by wide vote/reduction patterns that map efficiently onto the FPGA datapath. *Similarity* and *search* also obtain robust improvements (up to $115\times$ and $120\times$, respectively), since the accelerator computes hamming-distance reductions in

a highly parallel manner and further accelerates the repeated similarity evaluations over multiple candidate HVs.

Permutation benefits from the architectural choice of virtualizing the operation through SPM addressing rather than dedicated bit-level shuffling logic (see Section 3.2.1), achieving up to $200\times$ speedup (HV size 1024, SIMD = 16) and up to $123\times$ at larger dimensions (HV size 4096, SIMD = 32). Notably, the configuration SIMD = 32 with HV size 1024 is *not admissible* for permutation (reported as n/a in Table 8) because, with the adopted block-granular scheme, $W_{\text{data}} = \text{SIMD} \cdot 32 = 1024$ bit and the hypervector occupies a single block ($N = D/W_{\text{data}} = 1$), making any block-cyclic permutation degenerate and thus not meaningful to evaluate. In contrast, *Binding* exhibits the lowest gains (up to $60\times$), which is expected because BSC binding reduces to a simple bitwise XOR that is already efficiently executed on the ARM core and offers limited headroom for acceleration compared to reduction-dominated primitives.

Figure 4 evaluates how the latency of the associative-search primitive scales with the number of stored class prototypes. For a fixed hypervector dimension of $D = 8192$, the experiment measures the time required to compute and return the minimum Hamming distance between one query hypervector and a set of P class prototypes, with $P = \{10, 20, 30\}$. The results quantify the impact of increasing the number of similarity evaluations and of the available hardware parallelism, showing the expected near-linear growth of latency with P and the consistent reduction obtained with wider SIMD configurations.

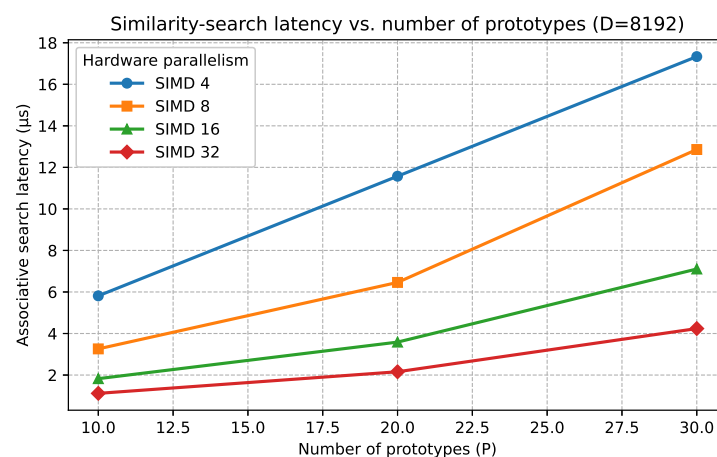


Figure 4. Similarity-search latency scaling with the number of class prototypes for a fixed hypervector size of $D = 8192$. The reported latency is the time required to return the minimum Hamming distance between a query hypervector and P stored class prototypes.

5.3.2. Performance on Real Datasets

We evaluated the accelerator on end-to-end classification pipelines (encoding + training/inference) using the datasets listed in Table 5. For these macro-benchmark experiments, the hardware accelerator was configured with SIMD = 8 and an operating frequency of 100 MHz. The bundling functional unit was set to 16-bit signed accumulation to safely support the intermediate sums required by feature-wise superposition during encoding and by class-wise accumulation during training. The memory subsystem was configured with 256 KB per SPM, which is sufficient to keep the item memories, intermediate HVs and the full associative-memory (AM) model on-chip for the considered workloads. The HV dimension was fixed to 1024 bits for all experiments.

We implemented a straightforward HDC classifier with single-pass (one-shot) training. During training, each labeled sample is encoded into the HV space and accumulated into its corresponding class prototype via binding and bundling operations (respectively,

element-wise XOR and element-wise integer addition). After all training samples have been processed, each class accumulator is binarized through a clipping operation (majority vote) to obtain the final binary prototypes stored in the AM. During inference, each unseen sample is encoded into a query HV, and the accelerator performs an associative search by computing similarity scores between the query and all class prototypes in the AM (via Hamming-distance-based matching for binary HVs). The predicted label is selected as the class with the best similarity score. HV transfers between the host and the accelerator are achieved via load and store operations, which are issued to both the accelerator and the DMA unit through memory-mapped registers. Before evaluating the speedup, the correctness of the end-to-end pipeline was verified. For every benchmark, the hardware-accelerated training and inference phases produced identical HVs and class predictions to the software model, confirming functional equivalence.

Table 9 reports the measured speedups with respect to the ARM Cortex-A9 software baseline. Overall, we observe substantial acceleration in both phases, with training speedups ranging from $58.58\times$ to $81.69\times$ and inference speedups ranging from $57.84\times$ to $151.83\times$. The largest gain is achieved on ISOLET during inference, where the higher class count amplifies the cost of associative search in software and allows the parallel search datapath to be exploited more effectively in hardware. Averaged across datasets, the accelerator provides $68.45\times$ speedup in training and $93.34\times$ speedup in inference; when accounting for the clock-frequency gap (667 MHz vs. 100 MHz), this corresponds to a cycle-level efficiency improvement of approximately $456\times$ and $623\times$, respectively.

Table 9. Overall speedup on benchmark datasets (Accelerator (SIMD = 8, HV_SIZE = 1024) vs. ARM Cortex-A9 software baseline).

Dataset	Training Speedup	Inference Speedup
IRIS	81.69	70.34
UCI-HAR	58.58	57.84
ISOLET	65.08	151.83

We also measured board-level energy during both training and inference for the same experimental configuration used in the macro-benchmarks (SIMD = 8, HV_SIZE = 1024, 100 MHz). Table 10 reports the measured energy per phase for each dataset.

Table 10. Board-level energy measured during training and inference (SIMD = 8, HV_SIZE = 1024, 100 MHz).

Dataset	Training Energy [J]	Inference Energy [J]
IRIS	3.25×10^{-4}	1.42×10^{-4}
UCI-HAR	2.295	0.983
ISOLET	1.737	0.745

At the beginning of each phase, there are overhead costs for loading all the required HVs into the scratchpad memories. This only happens in a cold start scenario, where Hypervectors have not been preloaded into the SPMs at synthesis time. Each dataset requires N feature HVs and L quantization HVs to start the training process. At the end of the training phase, C class HVs are retrieved from the scratchpad memories. The inference phase requires all the class, feature and level HVs to be loaded into the scratchpad memories, if not already present, and the result of each search operation needs to be transferred to the host memory. Table 5 shows the number of features and classes used for each dataset (N and C), while the number of quantization levels is set to 10 for every dataset ($L = 10$). With HV dimension set to 1024, SIMD = 8 and accelerator operating frequency set to 100 MHz,

the bandwidth of the scratchpad memories is $\text{SIMD} \cdot 4 \cdot 100 = 3200 \text{ MB/s}$ and each HV is transferred over $1024/(8 \cdot 32) = 4$ clock cycles. Table 11 shows the total execution time and overhead percentage for each dataset tested. It must be noted that the overhead is slightly underestimated, as it does not take into account the time needed to issue load and store instructions to the accelerator and the DMA unit; however, it is safe to assume this contribution can be neglected given that the CPU runs at a much higher frequency. From the data in Tables 9 and 11, we can see that, albeit there is an initial transfer cost, performing the benchmark on the HDC accelerator is much faster than performing it on a general-purpose processor and the overhead cost is minimal compared to the execution time of the algorithms. We also stress that in most application scenarios all the required Hypervectors can be preloaded at synthesis time, as they do not depend on the specific learning task; in our case, only the class vectors need to be counted in the overhead cost, after the learning phase.

Table 11. Total execution time and overhead percentage.

Dataset	Training Time [μs]	Inference Time [μs]	Training Overhead [%]	Inference Overhead [%]
IRIS	149.78	65.93	0.45%	1.03%
UCI-HAR	1,054,413.48	451,851.11	0.33%	<0.01%
ISOLET	798,189.416	342,166.277	0.22%	<0.01%

6. Discussion

This work targets a practical deployment gap that often emerges in HDC hardware acceleration: achieving high throughput on BSC primitives while remaining *system-integrable* and *retargetable* without redesign. By packaging the accelerator as a standalone AXI plug-and-play module with an AXI4-Lite control plane and an AXI4-Stream data plane, the proposed architecture can be driven by any AXI-capable host and can sustain bulk data ingestion at a bandwidth that scales with the internal datapath width. This design point is reflected in the evaluation, where the accelerator attains substantial speedups at both the primitive and end-to-end pipeline levels. In micro-benchmarks, speedups grow with SIMD and reach up to $431\times$ for bundling and $362\times$ for clipping at the largest configuration, while also accelerating similarity and associative search by up to $188\times$ and $177\times$, respectively. In macro-benchmarks on complete classification pipelines (encoding + training/inference), the accelerator delivers $58.58\times$ – $81.69\times$ training speedup and $57.84\times$ – $151.83\times$ inference speedup, with the highest inference gain observed on ISOLET due to the larger associative-memory search workload. A quantitative comparison with the closest Zynq-7000 FPGA-based HDC accelerators is summarized in Table 12.

6.1. Scalability Is a Full-System Problem

A central takeaway from the implementation sweep is that scaling vector-level parallelism is not only a compute-core design issue: it stresses the *entire* data-movement fabric. As SIMD increases, the stream width grows as $W_{\text{stream}} = \text{SIMD} \times 32 \text{ bits}$ and the post-implementation results show monotonic growth in Slice LUTs (19.14% at SIMD = 4 to 83.36% at SIMD = 32) and Flip-Flops (10.58% to 39.88%), alongside a frequency degradation from 110 MHz down to 77 MHz at SIMD = 32. Importantly, the breakdown highlights that a sizable fraction of the additional cost at high SIMD originates from the supporting infrastructure (e.g., interconnect and DMA), not just from the accelerator datapath. The same phenomenon appears in LUTRAM usage: at SIMD = 32, LUTRAM becomes significant (19.85%), but it is dominated by the AXI interconnect, indicating that routing and buffering for wide datapaths can become the timing and area limiter. These results suggest that,

for Zynq-7000 class devices, an “optimal” SIMD is often the one that balances compute throughput with routability and sustainable bandwidth (e.g., SIMD = 8/16 in this study), rather than the maximum achievable width.

6.2. Memory Subsystem Design and Host Decoupling

The proposed SPM-based organization is a key enabler of sustained throughput because it aligns storage parallelism with compute parallelism. The SPM controller is explicitly designed to (i) exploit dual-port BRAMs for simultaneous two-operand reads, (ii) arbitrate concurrent accesses (compute vs. DMA) and (iii) expose native AXI4-Stream connectivity for burst transfers directly into the SPMs. Crucially, address decoding and routing are absorbed into the accelerator-side memory controller, removing host-architecture dependencies and enabling true standalone use through standard AXI interfaces. This differs from many accelerator integrations that rely on host-side load/store or bespoke drivers to manage accelerator-local memories, and it is aligned with the “plug-and-play” goal of minimizing integration friction across systems.

6.3. Architectural Simplifications That Preserve HDC Semantics

Two design choices are especially relevant to the observed area/performance behavior. First, the permutation unit adopts a block-cyclic approach, trading exact bit-level shift granularity for lower routing complexity while retaining the bijective transformation needed for role/position encoding in typical HDC pipelines. Second, the clipping datapath is restructured to avoid variable bit-indexing by generating packed comparison blocks and reconstructing the output via a regular shift/append mechanism across a number of cycles corresponding to the counter bit-width, substantially reducing routing pressure while preserving the functional behavior and latency model. Together, these optimizations reinforce a broader point: many HDC primitives tolerate constrained implementations (e.g., coarse permutations, regularized datapaths) as long as the algebraic role of the operation is preserved.

6.4. Comparison with GP-HDCA

While GP-HDCA [30] targets FPGA-based acceleration on Zynq-7000, its architectural scope is primarily tailored to flexible *encoding* via an instruction-driven coprocessor. In contrast, our design acts as a full-pipeline accelerator, offloading not only the encoding but also the associative memory training and inference stages through dedicated stream-processing hardware. To ensure a fair resource comparison, we evaluate the total system footprint. GP-HDCA reports utilizing 24,760 LUTs for its Int32-V128 configuration (128-bit datapath), an overhead largely dominated by the complex control logic required to support its custom ISA. Conversely, our proposed design configured with an equivalent 128-bit datapath (SIMD = 4) consumes only $\approx 10,200$ LUTs for the *complete system* (including AXI DMA and SmartConnect). This demonstrates that our stream-based architecture is roughly $2.4\times$ more area-efficient than the instruction-based competitor at equivalent parallelism, allowing us to scale up to a SIMD = 32 configuration (1024-bit datapath) on the same device to achieve massive throughput. Furthermore, we offer superior usability and configurability: while GP-HDCA restricts flexibility to the limits of its custom ISA and requires low-level assembly-like programming, our IP provides extensive synthesis-time parametrization (e.g., SIMD width, memory depth) and is controlled via a high-level C++ software stack in Linux userspace. Finally, unlike GP-HDCA, which relies on estimated performance from a MATLAB emulator, we report measured latency from a physical implementation on the Zynq XC7Z020 and release our complete open-source hardware/software stack to foster reproducibility.

6.5. Comparison with EcoFlex-HDP

EcoFlex-HDP [31] proposes a Zynq-based co-processing architecture centered on a programmable Hyperdimensional Processing Unit (HPU) controlled via a custom ISA. While this approach offers flexibility through instruction-level programming, it suffers from severe resource bloat and incomplete pipeline acceleration. First, regarding area efficiency, EcoFlex-HDP reports a massive resource utilization of 47,122 LUTs, occupying 88.58% of the Zynq XC7Z020 logic capacity. This saturation effectively precludes the integration of additional user logic or peripherals. In stark contrast, our architecture demonstrates superior density: even in our high-parallelism SIMD=32 configuration (1024-bit datapath), our complete system utilizes only 27,300 LUTs ($\approx 51\%$ of the device), leaving ample headroom for system-level integration while delivering comparable or superior throughput. Second, regarding the acceleration scope, EcoFlex-HDP focuses primarily on the encoding primitives (bind, permutation, bundling), neglecting the critical associative memory search stage, which dominates inference latency. Our design addresses this bottleneck by integrating a dedicated streaming search engine alongside the encoding primitives, enabling full end-to-end hardware acceleration for both training and inference. It is also worth noting that EcoFlex-HDP does not report absolute latency figures for individual primitive operations, preventing a direct operator-level performance comparison; their evaluation is limited to relative speedups against a software baseline. Finally, rather than relying on a custom assembly-like ISA that complicates the software stack, we provide a synthesis-time parameterizable IP coupled with a standard C++ API, ensuring a smoother adoption path for edge-AI developers.

Table 12. Comparison with state-of-the-art FPGA-based HDC accelerators on Zynq-7000 platform. **Note:** n/s indicates information not stated in the referenced work.

	GP-HDCA [30]	EcoFlex-HDP [31]	This Work
Platform target	Zynq-7000	Zynq-7000	Zynq-7000
Host Interface	AXI4-Stream & AXI4-Lite	AXI4-Stream & AXI4-Lite	AXI4-Stream & AXI4-Lite
Supported Primitives	Bind, Bundle, Permute, Clip	Bind, Bundle, Permute	Bind, Bundle, Permute, Clip, Similarity, Search
HW Parallelism	32-bit to 128-bit	1024-bit	128-bit to 1024-bit
Frequency (MHz)	n/s	125	77–110
#LUTs	3137–24,760	47,122	10,185–44,349
#FFs	5863–20,076	50,707	11,258–42,437
#BRAM Tiles	n/s	92	73–127
Design Focus	Encoder Flexibility	Low-Power Multi-core	IP Scalability & Modularity
Availability	Closed Source	Open Source	Open Source

6.6. Limitations and Future Directions

First, macro-benchmark speedups are reported under a steady-state assumption and therefore exclude the one-time transfer of persistent HVs (e.g., base vectors, level vectors and class prototypes) from DDR to the SPMs. The cold-start overhead is quantified separately Table 11 and remains small for the considered workloads; however, in scenarios with frequent model swapping or limited on-chip capacity, transfer costs can become non-negligible, making overlap and caching policies more critical. Second, frequency degradation and LUTRAM growth at high SIMD confirm that interconnect and DMA scaling can become the dominant constraint; future work should investigate interconnect-minimized integration patterns, alternative streaming fabrics and multi-instance scaling (replicating moderate-SIMD engines) to improve routability and sustained throughput. Third, board-level energy is reported for training and inference in a representative configuration Table 10, but a complete energy/EDP characterization across the design space (e.g., different HV sizes, SIMD points and bandwidth regimes), including a finer-grained separation of accelerator and system-level power contributions, remains an important direction for future work.

7. Conclusions

This paper presented a general-purpose, open-source Hyperdimensional Computing accelerator designed to bridge the gap between high-performance hardware and software usability. By abandoning tight CPU-coupling in favor of a standalone, host-agnostic AXI4 architecture, we demonstrated that it is possible to achieve substantial acceleration without sacrificing portability across heterogeneous SoCs.

The proposed design, validated on a Xilinx Zynq-7000 platform, leverages a scalable SIMD datapath and an optimized memory subsystem to deliver primitive-level speedups of up to $761\times$ compared to an embedded ARM Cortex-A9 processor. Architectural optimizations in the permutation and clipping units proved effective in containing resource utilization, enabling high-throughput configurations (up to $\text{SIMD} = 32$) within the constraints of mid-range edge FPGA devices. At the system level, the integration of a multi-layer software stack allows the accelerator to seamlessly offload complete HDC pipelines, achieving average speedups of $68.45\times$ in training and $93.34\times$ in inference across standard classification benchmarks.

Our analysis highlighted that at high parallelism levels, the bottleneck shifts from computation to data movement and interconnect complexity, suggesting that future edge-AI architectures must co-optimize the accelerator datapath with the system-level transport fabric. By releasing the complete RTL and software stack as open-source hardware, this work provides a reusable and extensible foundation for the research community, facilitating the deployment of robust and energy-efficient HDC intelligence in real-world embedded systems.

Author Contributions: Conceptualization, R.M., A.M. and M.O.; methodology, R.M., M.P. and M.A.; hardware design, R.M., M.P., M.A. and A.M.; memory subsystem, M.P.; functional unit optimizations, M.A.; system integration, A.M. and R.M.; software, R.M.; validation, R.M., M.P. and M.A.; formal analysis, R.M.; investigation, R.M. and A.R.; resources, M.O.; data curation, R.M.; writing—original draft preparation, R.M.; writing—review and editing, R.M., M.P., M.A., M.B., A.M., A.R. and M.O.; paper organization and editorial guidance, M.B.; supervision, M.O.; project administration, M.O. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The complete RTL and software stack are released as open-source at: <https://github.com/RoMartino/AXI-HDC-Accelerator> (accessed on 15 January 2026). Publicly available datasets used in this study are cited in Table 5.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
BSC	Binary Spatter Code
CDC	Clock Domain Crossing
DMA	Direct Memory Access
EDP	Energy-Delay Product
FF	Flip-Flop
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array

HDC	Hyperdimensional Computing
HLS	High-Level Synthesis
HV	Hypervector
IP	Intellectual Property
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
LUT	Look-Up Table
PL	Programmable Logic
PS	Processing System
RTL	Register Transfer Level
SIMD	Single Instruction Multiple Data
SoC	System-on-Chip
SPM	Scratchpad Memory
UIO	Userspace I/O
VSA	Vector Symbolic Architectures

References

1. Kanerva, P. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cogn. Comput.* **2009**, *1*, 139–159. [\[CrossRef\]](#)
2. Gayler, R.W. Vector Symbolic Architectures Answer Jackendoff's Challenges for Cognitive Neuroscience. In Proceedings of the 4th ICCS International Conference on Cognitive Science and the 7th ASCS Australasian Society for Cognitive Science Conference, Sydney, Australia, 13–17 July 2003; pp. 133–138.
3. Kleyko, D.; Rachkovskij, D.A.; Osipov, E.; Rahimi, A. A survey on hyperdimensional computing aka vector symbolic architectures, Part I: Models and data transformations. *ACM Comput. Surv.* **2022**, *55*, 175. [\[CrossRef\]](#)
4. Kleyko, D.; Rachkovskij, D.; Osipov, E.; Rahimi, A. A survey on hyperdimensional computing aka vector symbolic architectures, part II: Applications, cognitive models, and challenges. *ACM Comput. Surv.* **2023**, *55*, 175. [\[CrossRef\]](#)
5. Schlegel, K.; Neubert, P.; Protzel, P. A comparison of vector symbolic architectures. *Artif. Intell. Rev.* **2022**, *55*, 4523–4555. [\[CrossRef\]](#)
6. Chang, C.-Y.; Chuang, Y.-C.; Huang, C.-T.; Wu, A.-Y. Recent progress and development of hyperdimensional computing (HDC) for edge intelligence. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2023**, *13*, 119–136. [\[CrossRef\]](#)
7. Kleyko, D.; Davies, M.; Frady, E.P.; Kanerva, P.; Kent, S.J.; Olshausen, B.A.; Osipov, E.; Rabaey, J.M.; Rachkovskij, D.A.; Rahimi, A.; et al. Vector symbolic architectures as a computing framework for emerging hardware. *Proc. IEEE* **2022**, *110*, 1538–1571. [\[CrossRef\]](#)
8. Kanerva, P. Fully Distributed Representation. In 1997 *Real World Computing Symposium (RWC '97)*; Real World Computing Partnership: Tsukuba, Japan, 1997; pp. 358–365.
9. Morris, J.; Fernando, R.; Hao, Y.; Imani, M.; Aksanli, B.; Rosing, T. Locality-based encoder and model quantization for efficient hyper-dimensional computing. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2021**, *41*, 897–907. [\[CrossRef\]](#)
10. Schmuck, M.; Benini, L.; Rahimi, A. Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory. *ACM J. Emerg. Technol. Comput. Syst.* **2019**, *15*, 32. [\[CrossRef\]](#)
11. Khaleghi, B.; Xu, H.; Morris, J.; Rosing, T.S. Tiny-HD: Ultra-efficient hyperdimensional computing engine for IoT applications. In Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 1–5 February 2021; pp. 408–413.
12. Eggimann, M.; Rahimi, A.; Benini, L. A 5 μ W standard cell memory-based configurable hyperdimensional computing accelerator for always-on smart sensing. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2021**, *68*, 4116–4128. [\[CrossRef\]](#)
13. Taheri, F.; Bayat-Sarmadi, S.; Hadayeghparast, S. RISC-HD: Lightweight RISC-V processor for efficient hyperdimensional computing inference. *IEEE Internet Things J.* **2022**, *9*, 24030–24037. [\[CrossRef\]](#)
14. Yu, T.; Wu, B.; Chen, K.; Zhang, G.; Liu, W. LAHDC: Logic-aggregation-based query for embedded hyperdimensional computing accelerator. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2025**, *44*, 119–129. [\[CrossRef\]](#)
15. Salamat, S.; Imani, M.; Khaleghi, B.; Rosing, T. F5-HD: Fast flexible FPGA-based framework for refreshing hyperdimensional computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium Field-Programmable Gate Arrays*; ACM: New York, NY, USA, 2019; pp. 53–62.
16. Imani, M.; Bosch, S.; Datta, S.; Ramakrishna, S.; Salamat, S.; Rabaey, J.M.; Rosing, T. QuantHD: A quantization framework for hyperdimensional computing. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 2268–2278. [\[CrossRef\]](#)

17. Yu, T.; Wu, B.; Chen, K.; Zhang, G.; Liu, W. Fully learnable hyperdimensional computing framework with ultratiny accelerator for edge-side applications. *IEEE Trans. Comput.* **2024**, *73*, 574–585. [\[CrossRef\]](#)
18. Kim, Y.; Imani, M.; Moshiri, N.; Rosing, T. GenieHD: Efficient DNA pattern matching accelerator using hyperdimensional computing. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*; IEEE: Piscataway, NJ, USA, 2020; pp. 115–120.
19. Khaleghi, B.; Kang, J.; Xu, H.; Morris, J.; Rosing, T.S. GENERIC: Highly efficient learning engine on edge using hyperdimensional computing. In *Proceedings 59th ACM/IEEE Design Automation Conference (DAC)*; ACM: New York, NY, USA, 2022; pp. 1117–1122.
20. Martino, R.; Angioli, M.; Rosato, A.; Barbirotta, M.; Cheikh, A.; Olivieri, M. Configurable hardware acceleration for hyperdimensional computing extension on RISC-V. *IEEE Trans. Comput.* **2026**, *75*, 653–664. [\[CrossRef\]](#)
21. Salamat, S.; Imani, M.; Rosing, T. Accelerating hyperdimensional computing on FPGAs by exploiting computational reuse. *IEEE Trans. Comput.* **2020**, *69*, 1159–1171. [\[CrossRef\]](#)
22. Morris, J.; Set, S.T.K.; Rosen, G.; Imani, M.; Aksanli, B.; Rosing, T. AdaptBit-HD: Adaptive model bitwidth for hyperdimensional computing. In *Proceedings of the 2021 IEEE 39th International Conference on Computer Design (ICCD)*, Storrs, CT, USA, 24–27 October 2021; pp. 93–100.
23. Sadeghipour Roodsari, M.; Krautter, J.; Meyers, V.; Tahoori, M. E3HDC: Energy efficient encoding for hyper-dimensional computing on edge devices. In *Proceedings of the 2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, Torino, Italy, 2–6 September 2024; pp. 274–280.
24. Li, H.; Liu, F.; Chen, Y.; Wang, Z.; Huang, S.; Yang, N.; Lyu, D.; Jiang, L. FATE: Boosting the performance of hyper-dimensional computing intelligence with flexible numerical data type. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*; Association for Computing Machinery: New York, NY, USA, 2025; pp. 1269–1282.
25. Zhang, T.; Salamat, S.; Khaleghi, B.; Morris, J.; Aksanli, B.; Rosing, T.S. HD2FPGA: Automated framework for accelerating hyperdimensional computing on FPGAs. In *Proceedings of the 2023 24th International Symposium on Quality Electronic Design (ISQED)*, San Francisco, CA, USA, 5–7 April 2023; pp. 1–9.
26. Martino, R. AXI-HDC-Accelerator: A General-Purpose, AXI4-Compliant Hyperdimensional Computing (HDC) Accelerator IP for Xilinx Zynq SoCs. GitHub Repository. 2026. Available online: <https://github.com/RoMartino/AXI-HDC-Accelerator> (accessed on 15 January 2026).
27. Vergés, P.; Heddes, M.; Nunes, I.; Kleyko, D.; Givargis, T.; Nicolau, A. Classification using hyperdimensional computing: A review with comparative analysis. *Artif. Intell. Rev.* **2025**, *58*, 173. [\[CrossRef\]](#)
28. Cumbo, F.; Chicco, D. Hyperdimensional computing in biomedical sciences: A brief review. *PeerJ Comput. Sci.* **2025**, *11*, e2885. [\[CrossRef\]](#)
29. Angioli, M.; Jamili, S.; Barbirotta, M.; Cheikh, A.; Mastrandrea, A.; Menichelli, F.; Rosato, A.; Olivieri, M. AeneasHDC: An automatic framework for deploying hyperdimensional computing models on FPGAs. In *Proceedings of the 2024 International Joint Conference on Neural Networks (IJCNN)*, Yokohama, Japan, 30 June–5 July 2024.
30. Asghari, M.; Le Beux, S. A general purpose hyperdimensional computing accelerator for edge computing. In *Proceedings of the 2024 22nd IEEE Interregional NEWCAS Conference (NEWCAS)*, Sherbrooke, QC, Canada, 16–19 June 2024; pp. 383–387.
31. Isaka, Y.; Sakaguchi, N.; Inoue, M.; Shintani, M. EcoFlex-HDP: High-speed and low-power and programmable hyperdimensional-computing platform with CPU co-processing. In *Proceedings of the 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Valencia, Spain, 25–27 March 2024.
32. Cheikh, A.; Sordillo, S.; Mastrandrea, A.; Menichelli, F.; Scotti, G.; Olivieri, M. Klessydra-T: Designing vector coprocessors for multi-threaded edge-computing cores. *IEEE Micro* **2021**, *41*, 64–71. [\[CrossRef\]](#)
33. Angioli, M.; Kymn, C.J.; Rosato, A.; Loutfi, A.; Olivieri, M.; Kleyko, D. Efficient Hyperdimensional Computing with Modular Composite Representations. *arXiv* **2025**, arXiv:2511.09708. [\[CrossRef\]](#)
34. Cole, R.; Fandy, M. ISOLET [Dataset]. UCI Machine Learning Repository. 1991. Available online: <https://archive.ics.uci.edu/dataset/54/isolet> (accessed on 15 January 2026).
35. Fisher, R.A. Iris [Dataset]. UCI Machine Learning Repository. 1936. Available online: <https://archive.ics.uci.edu/dataset/53/iris> (accessed on 15 January 2026).
36. Reyes-Ortiz, J.; Anguita, D.; Ghio, A.; Oneto, L.; Parra, X. Human Activity Recognition Using Smartphones. UCI Machine Learning Repository. 2013. Available online: <https://archive.ics.uci.edu/dataset/240/human+activity+recognition+using+smartphones> (accessed on 15 January 2026).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.