Efficient Design of a Hyperdimensional Processing Unit for Multi-Layer Cognition*

Mohamed Ibrahim, Youbin Kim, Jan M. Rabaey
Berkeley Wireless Research Center, EECS Department, University of California, Berkeley
Corresponding Author Email: mohamed.ibrahim@berkeley.edu

***Abstract*—The methodology used to design and optimize the very first general-purpose hyperdimensional (HD) processing unit capable of executing a broad spectrum of HD workloads (called "HPU") is presented. HD computing is a brain-inspired computational paradigm that uses the principles of high-dimensional mathematics to perform cognitive tasks. While considerable efforts have been spent toward realizing efficient HD processors, all of these targeted specific application domains, most often pattern classification. In contrast, the HPU design addresses the multiple layers of a cognitive process. A structured methodology identifies the kernel HD computations recurring at each of these layers, and maps them onto a unified and parameterized architectural model. The effectiveness in terms of runtime and energy consumption of the approach is evaluated. The results show that the resulting HPU efficiently processes the full range of HD algorithms, and far outperforms baseline implementations on a GPU.**

I. Introduction

Hierarchical cognition has emerged as an essential approach for developing dynamic intelligent systems. This approach involves developing a computation model that integrates three interactive neuro-symbolic functionalities: sensory perception, reasoning, and control. Various application domains stand to benefit from this cognitive model, such as intelligent mobile robots and adaptive prosthetic systems [1], [2].

Resembling the sense-reason-act cognitive cycle can be computationally modeled through a multi-layer framework [2], as illustrated in Fig. 1. The *perception* layer fuses multi-modal sensory inputs and maps them to high-level observations or classes. The *reasoning* layer constructs an optimization model, which is used to understand the semantics of the environment by factorizing joint perceptual representations into distinct items and applying rule-based knowledge search [3]. Feedback from this layer is used to adjust the parameters of the other two layers. The *control* layer facilitates stateful motor control using the recall of reactive behavior paradigm [4].

Realizing these layers through a unified framework is a challenging problem. Traditionally, neural networks are often used for perception, whereas logic programming is widely adopted for symbolic reasoning. These two are different computing methodologies that work in isolation. A new computing paradigm referred to as Hyperdimensional Computing (HDC) has recently emerged to bridge this gap [5]. In HDC, computational elements, such as scalars and objects, are represented using high-dimensional holographic vectors. These vectors can be manipulated by a set of algebraic operations, which together form a mathematical framework for implementing cognitive functions [6]. HDC has therefore proven to be highly effective, particularly showing significant advantages in robustness, energy efficiency, latency, and memory footprint [7]–[9].

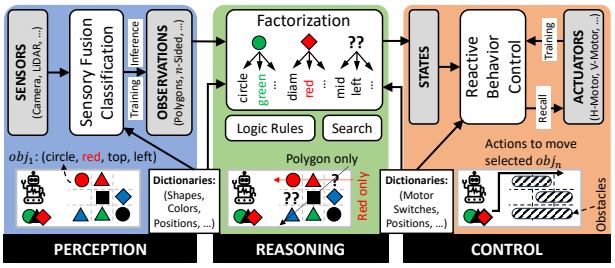


Fig. 1. A multi-layer representation of the sense-reason-act cognitive process.

There is, however, a side effect that arises due to HDC's high-dimensional nature. That is, HDC is not well-suited for implementation on classical compute platforms, such as GPUs. These platforms do not provide capabilities for accessing very wide vectors across distributed memory spaces, thereby causing inefficiency when implementing HD operations. Moreover, their dataflows are not optimized for on-chip dimensionality scaling, which is important for processing HD vectors with arbitrary sizes. Such limitations have sparked research on efficient hardware design for selected HDC algorithms [9]–[12].

On the other hand, these HDC processors lack programmability and architectural primitives that are necessary to extend the realization of HDC for multi-layer cognition. First, these processors cannot implement arbitrary logical primitives (e.g., state machines) or support factorization operations. Second, their dataflows do not typically involve on-chip dimensionality scaling of vectors. Third, the performance characteristics of these processors were optimized for selected tasks; therefore, no control knobs were provided to regulate the balance between the cost and performance of HD computations. These shortcomings highlight the need for a new design methodology and processor architecture that implements various HDC functions while allowing user control over performance tradeoffs.

This paper presents a design method for HPU (Hyperdimensional Processing Unit), the first general-purpose HDC processor that is capable of implementing multi-layer cognition. The proposed method exposes essential hardware features to the software, allowing for a large degree of configuration. The main contributions of this paper are summarized as follows:

- We identify the specifications of HDC-based multi-layer cognition and propose a structured design method that transforms these specifications into kernels and dataflow representations.
- We describe and evaluate the hardware design of HPU along with its control framework, showing a method to tune the performance based on this framework.

*This work was supported by TSMC under the HDC JDP program.

979-8-3503-4859-0/DATE24/© 2024 EDAA

II. Background

A. HDC Operations

We focus here on an HDC method that encodes atomic attributes and patterns using dissimilar HD vectors (*item hypervectors*) $x_i \in \{-1, +1\}^D$, with D in the range of thousands. Each hypervector x_i has an equal number of randomly placed $+1$s and -1s. Four operations can be performed on these hypervectors: (1) element-wise multiplication ($\otimes$), or binding, which creates a new hypervector that is quasi-orthogonal (dissimilar) to its constituents; (2) element-wise addition ($[+]$), or bundling, which combines hypervectors using element-wise majority count; (3) permutation (ρ), which rearranges the elements of a hypervector to preserve its order within a sequence; (4) scalar multiplication ($\times$), which scales hypervector elements with a scalar weight. The similarity between vectors is measured using a distance metric, such as the dot product [13].

B. HDC Realization of Multi-Layer Cognition

HDC enables an integrated realization of multi-layer cognition (Fig. 1), leveraging its effective implementation of key functional components such as HD encoding and decoding, associative memory, and optimization. These components are depicted in Fig. 2 and are described as follows.

HD Encoding: This component transforms input features into high-order models, often referred to as *prototype hypervectors* [14]. These models include observation or class vectors composed by the perception layer, data-structure representations embedded through the reasoning layer, and reactive-behavior models trained by the control layer. HD encoding is compositional; hence, it can generate new (hierarchical) representations of higher-order models based on existing ones, e.g., composing key-value pairs, sets, and sequences to map features of spatiotemporal signals into classes [10].

HD Decoding: This component performs the inverse of HD encoding, i.e., retrieves item vectors from composite data-structure vectors. This component is particularly important for reactive-control settings where actuation vectors need to be recalled based on specific reactive-behavior models [4]. The result of HD decoding is approximate due to the cross-talk noise coming from all other item vectors. Note that the adopted HDC model enjoys the self-inverse property, which allows both encoding and decoding to be implemented using the same algebraic operations [13].

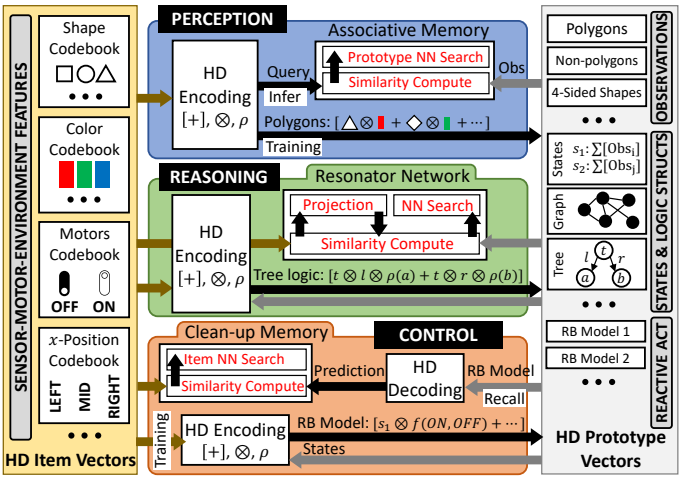


Fig. 2. An HDC implementation of the multi-layer cognition framework described in Section I (NN: Nearest Neighbor, RB: Reactive Behavior).

Associative Memory: This function, employed in the perception layer, performs pattern recognition for a given query vector. It determines the query's proximity to prototype (class) vectors and then searches through them to identify the closest vector.

Clean-up Memory: It operates on the approximate outcome of HD decoding, aiming to recall the exact item vector. It determines the outcome's proximity to all item vectors and searches through them to identify the closest vector.

Resonator Network: This component provides a unique capability for factorizing complex perceptual representations and data structures [3], [8]. Specifically, it seeks to decompose a complex HD model (e.g., an object vector) into its constituent items or features. Consider a composed object vector $f = a \otimes b \otimes c$, which is defined by three unknown factors: a for color, b for shape, and c for position. These factors are unknown because we only have access to the codebooks (A, B, and C), which represent the HD item sub-spaces for these factors. We use the resonator network algorithm to search for the original items of a, b, and c. The following state-space equations describe this search:

$$\hat{a}(t+1) = g\big(AA^\top (f \otimes \hat{b}(t) \otimes \hat{c}(t))\big; \quad A = [a_1\ a_2\ \ldots]$$
$$\hat{b}(t+1) = g\big(BB^\top (f \otimes \hat{a}(t) \otimes \hat{c}(t))\big; \quad B = [b_1\ b_2\ \ldots]$$
$$\hat{c}(t+1) = g\big(CC^\top (f \otimes \hat{a}(t) \otimes \hat{b}(t))\big; \quad C = [c_1\ c_2\ \ldots]$$

Here, $g(\cdot)$ is the sign function; t is a time step; $\hat{a}$, $\hat{b}$, and $\hat{c}$ hold the predicted values of the factors a, b, and c, respectively. The term $AA^\top \hat{x}$ represents the update role for $\hat{a}$. It first computes the dot-product similarity between the decoded vector $\hat{x}$ and all item vectors of A. Next, it uses the similarity results to scale the corresponding item vectors and bundle the weighted vectors to generate a new estimate; this last step is known as projection.

C. Prior Hardware Implementations of HDC

Early research on HDC processors focused on hardwired dataflows for language classification [12] and activity recognition [14]. Each of these designs supports only a specific configuration of HD encoding and associative memory and operates with a fixed vector length. Support for extending vector dimensionality was recently enabled using time multiplexing [10]. This solution divides vectors into sub-vectors called "folds" and processes each fold individually. Additionally, the item memory is replaced with a cellular automaton (CA Rule 90) implementation, which generates new random folds on-the-fly using XOR gates and shift operations.

Only two designs offering programmable HD operations were proposed [9], [15]. Datta et al.'s design [9] enables high-throughput HD processing but has a hardwired dataflow with a fixed number of configurable stages, thereby limiting its application domain. Eggimann et al.'s design [15] is more generic but lacks support for reasoning and control components.

Overall, all prior hardware implementations of HDC are limited because they are optimized solely for selected perception algorithms and lack scalability in performance. To fully realize HDC's potential, a new design methodology and programmable architecture are needed to support various applications.

III. HPU Kernel and Dataflows

A. HPU Kernel Formulation

We present a description of HPU's operations and programmability features using a formal representation, that is the kernel function. We express this kernel function as $O := F(y, s)$, where $F(\cdot)$ integrates an array of kernel sub-functions f_i that together cover the whole domain of HPU operations, and $y = \{y_1, y_2, \ldots\}$ represents an array combining all item and prototype vectors used in computation. The argument s is defined by a group of conditional variables $s = (s_1, s_2, \ldots)$, which together are used to draw the sub-domains associated with the sub-functions f_i.

The kernel functionality integrates computations for encoding and decoding, memory, and reasoning (Fig. 2). We next formulate sub-functions f_i that describe these computations.

Encoding and Decoding Kernel: To facilitate the encoding and decoding of arbitrary HD models, the kernel function must be designed in a way that allows for flexible configuration of the basic HD operations (binding, bundling, and permutation). Developing a kernel that achieves such a high degree of flexibility can be challenging; nevertheless, we take into account that binding (and its inverse) can be distributed over bundling [13]. Considering all the above, we introduce the following function:

$$a(y,(s_1,s_2)) := \begin{cases} b(y,(s_2)); & s_1 = 0 \\ \sum_i \left[b(y_i,(s_2))\right]; & s_1 = 1 \end{cases}$$

$$b(y,(s_2)) := \begin{cases} y; & s_2 = 0 \\ \bigotimes_j (y_j); & s_2 = 1 \\ \rho_j(y_j); & s_2 = 2 \\ \bigotimes_j \rho_{(j-1)}(y_j); & s_2 = 3 \end{cases}$$

$$\forall \{i,j\} \subset \mathbb{N}$$

Here, ρ_j means that the permutation operation (ρ) is repeated j times, i.e., $\rho_3(x) = \rho(\rho(\rho(x)))$. Likewise, when $j = 3$, the term $\bigotimes_j (x_j)$ becomes equivalent to $(x_1 \otimes x_2 \otimes x_3)$, and also the term $\bigotimes_j \rho_{(j-1)}(x_j)$ becomes equivalent to $(x_1 \otimes \rho(x_2) \otimes \rho(\rho(x_3)))$.

Resonator-Network Kernel: Recall that the operation of the resonator network involves iterative steps for similarity evaluation and projection (refer to Section II-B). The kernel function used for projection can be defined as follows: $c(y) := \sum_i [n_i \times y_i]; \quad \forall i \in \mathbb{N};\ n_i \in \mathbb{Z}$. Here, $c(y)$ calculates a weighted sum of the vectors in y. The weight n_i is given by a function $d(y_i, \bar{y})$, which measures the similarity between items y_i and an estimate vector $\bar{y} \in y$.

Nearest-Neighbor Search Kernel: The similarity function $d(y_i, \bar{y})$ also serves as the basis for identifying the closest vector to a query $\bar{y} \in y$ among an array of vectors $y = \{y_1, y_2, \cdots\}$. Note that the array y represents item vectors when performing a clean-up memory search, and prototype vectors when performing an associative memory search. To describe this operation, we use a kernel function defined as follows: $e(y) := \text{argmax}_i\ d(y_i, \bar{y})$.

Kernel Support for Extended Vector Dimensions through Time-Multiplexing: A particular advantage of element-wise vector operations (binding, bundling, and permutation) is that they can process full-scale vectors and time-multiplexed folds similarly. In contrast, similarity operations in $d(y_i, \bar{y})$ require the time-multiplexed folds to be collapsed into a single vector representation. When a similarity quantity is computed using only a single fold, it represents a partial quantity. Therefore, to obtain the total similarity value, $d(y_i, \bar{y})$ needs to aggregate these partial quantities. We express this condition as follows: $d(y_i, \bar{y}) := \sum_k \left(y_{ik} \cdot \bar{y}_k\right)\ \forall\ i \in \mathbb{N};\ k \in \{1, 2, \ldots, L\}$. Here, L is the number of folds, and $\bar{y}_k$ and y_{ik} are the k-th folds of the vectors $\bar{y}$ and y_i, respectively. The dot product measures the similarity between these folds, and the sum over all k aggregates the similarities computed for all the folds.

Compact Kernel Formalism: Considering the information presented above, we present a compact and formal description of HPU's kernel functionality as follows:

$$F(y,(s_1,s_2,s_3)) := \begin{cases} a(y,(s_1,s_2)); & s_3 = 0 \\ c(y); & s_3 = 1 \\ e(y); & s_3 = 2 \end{cases}$$

In this definition, the control variables (s_1, s_2, s_3) are used to dynamically adjust the behavior of the kernel during runtime. Fig. 3 demonstrates how the kernel is adjusted to execute HD workloads. Performance results based on the mapping of these workloads and others are shown later in this paper.

B. HPU Dataflows

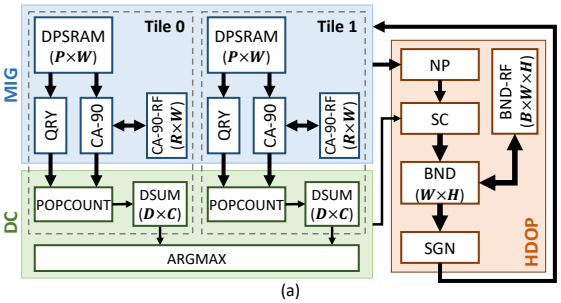
We present a method for constructing HPU's dataflows informed by the derived kernel. HPU consists of three subsystems: (1) memory and item-generation (MIG) subsystem, (2) HD operations (HDOP) subsystem, and (3) distance computation (DC) subsystem. A control unit is used to decode instructions and determine control configurations. Fig. 4(a) depicts these subsystems and their internal modules. A description of these modules and their correspondence with the kernel function are presented below.

MIG: This subsystem consists of four modules: a dual-port memory (DPSRAM), a logic unit that implements CA Rule 90 (CA-90), a register file (CA-90-RF), and a query register

Task	Algorithm	Description	Kernel (s_1, s_2, s_3)
Reactive behavior learning and recall [4]	(1) $s_j \leftarrow \sum_i [obs_i]$	• Superpose state information	(1, 0, 0)
	(2) $m_j \leftarrow \sum_k [a_k \otimes v_k^l]$	• Binding motor value with ID (a_k)	(1, 1, 0)
	(3) $b_j \leftarrow \sum_l [lc_l]$	• Gather environment data (labels)	(1, 0, 0)
	(4) $x \leftarrow \sum_j (s_j \otimes m_j \otimes b_j)$	• Learning reactive behavior model	(1, 1, 0)
	(5) $\widehat{v_x} \leftarrow x \otimes (s_j \otimes b_j \otimes a_k)$	• Decoding of a motor value	(0, 1, 0)
	(6) $\text{argmax}_l\ d(v_k^l, \widehat{v_x})$	• Clean-up memory	(−, −, 2)
Factoring – Single iteration [8]	(1) $x \leftarrow s \otimes (\hat{b} \otimes \hat{c} \otimes \hat{d})$	• Decoding step for factor a	(0, 1, 0)
	(2) $\hat{a} \leftarrow \sum_i [d(a_i, x) \times a_i]$	• Similarity against codebook of "a" and weighted bundling of vectors	(1, 0, 1)
	(3) $\text{argmax}_i\ d(a_i, \hat{a})$	• Finding the right item for factor "a" (following the last iteration)	(−, −, 2)

Algorithm Legend: *Encoding*; *Decoding*; *Clean-up/Associative Memory*; *Resonator Network*

Fig. 3. Illustration of how the kernel is programmed to implement workloads.



(a)

Data Parameters		Configuration Parameters	
W	HD bus (fold) width	K	Number of tiles
H	HDOP integer bit-width	R	Number of CA-90-RF registers
C	Distance integer bit-width	B	Number of BND-RF registers
		D	Number of DSUM registers

(b)

Fig. 4. Illustration of HPU architecture, including: (a) the dataflows and modules of the three subsystems, and (b) HPU design parameters.

(QRY). DPSRAM stores both random folds (seeds for item vectors) and full prototype vectors; therefore, it serves as both the source and destination for all data paths. The two ports of DPSRAM can be configured to perform load and store operations simultaneously. CA-90 implements the folding mechanism described in Section II-C, allowing new item folds to be generated on-the-fly. CA-90 may need to repeat many update cycles each time a specific fold is needed; therefore, newly generated folds can be held in CA-90-RF to avoid redundant activation of CA-90. In Section V, we show the impact of CA-90-RF on performance. QRY is used to hold data required by the similarity computation.

HDOP: HPU operations corresponding to the kernel functions $a(y,(s_1,s_2))$ and $c(y)$ are performed by HDOP. This subsystem consists of five logic units: a binding-permutation unit (NP), a bundling unit (BND), a register file (BND-RF), a scaling unit (SC), and a sign unit (SGN). NP and BND adopt different data representations, with NP using binary and BND utilizing integer formats. The conversion from binary to integer is managed by SC, which also executes element-wise scalar multiplication (to implement $c(y)$). Integer folds in BND can be held in BND-RF or converted to binary through SGN for storage in DPSRAM.

DC: This subsystem executes operations that implement the functions $d(y_i,\bar{y})$ and $e(y)$. DC consists of three logic units: POPCOUNT, DSUM, and ARGMAX. POPCOUNT uses XOR gates to compare two folds and an adder tree to return the difference between the number of 1's and the number of 0's, thus computing the dot-product similarity. Note that POPCOUNT generates a signed integer that represents a partial distance quantity. Therefore, distance accumulation over multiple folds is performed via DSUM, which in turn can perform this operation through one of multiple (D) distance registers. The controller of these registers allows reading and writing operations to be performed independently. DSUM also includes a multiplexing logic that allows a selected distance quantity to be transferred to HDOP for scalar multiplication. Searching for the nearest vector ($e(y)$) is performed based on the computed distances and is executed in ARGMAX.

Parameterized Multi-Tile Architecture: The integration of the above modules leads to a "single-tile" HPU architecture, which includes only a single instance of MIG and DC. We also propose a "multi-tile" architecture, which allows similarity computations to be distributed across multiple (K) tiles and exploits a single-instruction, multiple-data (SIMD) implementation to speed up the execution [Fig. 4(a)]. A multi-tile architecture also extends HPU's storage capabilities, providing a means to accommodate larger models. Tiles are also equipped with configuration registers, which allow tiles to be selectively activated (or deactivated) before issuing instructions.

Our design methodology is parameterized, and the performance impact of each parameter is not constant across different applications. We summarize the design parameters in Fig. 4(b) and evaluate their impact on performance in Section V.

IV. HPU Control Methodology

The HPU dataflow requires seven pipeline stages, with each stage associated with a certain type of operation, as shown in Fig. 5. This approach proposes a streamlined integration of dataflow and control-flow primitives, allowing different control methods to be applied. We particularly examine two control methods for HPU: *single-operation-per-cycle* (SOPC) and *multiple-operations-per-cycle* (MOPC).

SOPC simplifies programming and reduces power consumption since only one pipeline stage switches during each cycle. However, this approach increases runtime, making it unsuitable for high-throughput applications. Conversely, MOPC enables pipeline stages to perform operations simultaneously, thus increasing the number of *operations per cycle* (OPC). However, MOPC leads to increased power consumption, and furthermore, it requires a complex mapping framework (a compiler) to analyze program dependencies and optimize control activities. MOPC is better suited for high-throughput applications that require a balance between runtime and power consumption.

A. Comparing HPU Control Methods

To compare SOPC and MOPC, we consider an example of a resonator network described formally as follows:

$$\hat{a}(t+1) = g\big(AA^{\top}(f \otimes \hat{b}(t))\big); \quad \hat{b}(t+1) = g\big(BB^{\top}(f \otimes \hat{a}(t))\big)$$

With a two-tile HPU architecture, updating both factors, i.e., $\hat{a}$ and $\hat{b}$, in each iteration of the network requires 34 HPU operations. By using the MOPC approach, these operations can be executed within 19 clock cycles (with OPC equal to 1.8), as opposed to 34 clock cycles with SOPC. Fig. 6(a) depicts the activity of the HPU pipeline while executing these operations using MOPC. The increase in OPC nonetheless leads to a 44% increase in the average power consumption.

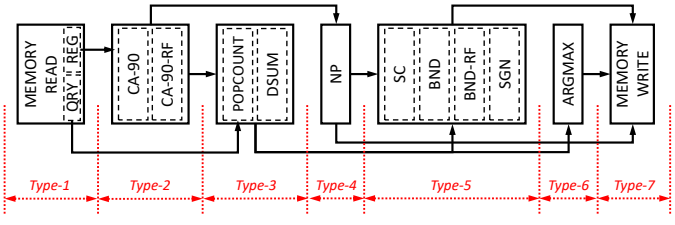


Fig. 5. HPU pipeline stages and operation types.

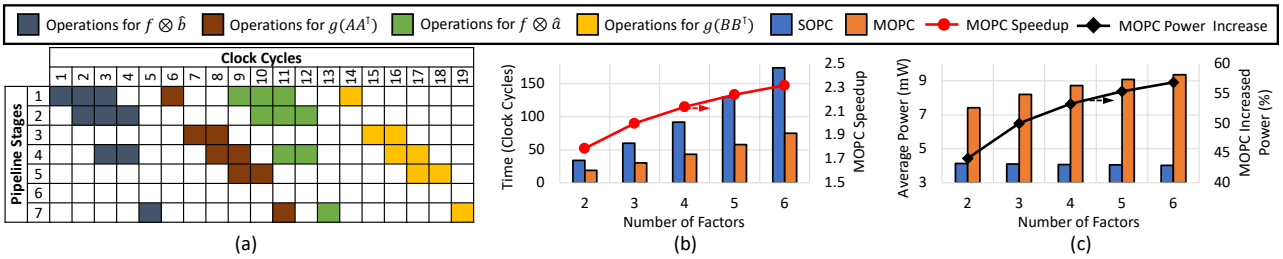


Fig. 6. Demonstration of HPU behavior and performance tradeoffs when two control methods (SOPC and MOPC) are used to execute the resonator-network algorithm: (a) HPU pipeline allocation when MOPC is used, (b) runtime results, and (c) power-consumption results.

OP_PARAM (57 bits)	Type_7 (3 bits)	Type_6 (3 bits)	Type_5 (3 bits)	Type_4 (2 bits)	Type_3 (3 bits)	Type_2 (3 bits)	Type_1 (2 bits)

Fig. 7. The *Instruction Word* format adopted in HPU.

To better understand the trends in MOPC speed-up and power consumption, we also examine the execution of four additional resonator networks, ranging in complexity from three to six factors. As shown in Fig. 6(b), as the network complexity increases from two to six factors, the speed-up gained by using MOPC increases from 1.8 to 2.3. However, using MOPC also increases power consumption compared to SOPC. The percentage of increased power by using MOPC grows from 44% to 57% as the network complexity increases; see Fig. 6(c). **Adopting MOPC:** We optimize HPU's control-flow based on MOPC as it offers two major advantages. First, MOPC allows HPU to speed up the execution of a given task, as opposed to SOPC. This is especially important when multiple heterogeneous tasks need to be executed simultaneously, e.g., layered cognition frameworks. Second, the speed-up gain can be flexibly configured based on power-consumption constraints. Consider the case of executing a six-factor resonator network using MOPC. If the speed-up is decreased from 2.3 to 2.0, the average power consumption decreases from 9.3 mW to 8.0 mW. In fact, we could aim to decrease the speed-up further until it converges to that of SOPC; that is 1.0, achieving a reduced power consumption of only 4.0 mW. In other words, SOPC is considered a special case of MOPC.

B. HPU Instruction Format

To realize the MOPC control method, we design an instruction-set architecture that employs a wide-word macro format, referred to as *Instruction Word*. Similar to a Very-Large Instruction Word (VLIW), a single *Word* consists of multiple operations, except that these operations are sequential in the pipelined dataflow and not parallel like VLIW architectures. As shown in Fig. 7, the *Word* format consists of seven `Type` fields, used to specify the operations to be executed in seven pipelined stages of HPU, and an `OP_PARAM` field, used to configure all or specific `Type` operations. This approach offers a high degree of flexibility and is commonly used with domain-specific processors. Details on the instruction fields and compiler optimization are omitted due to lack of space.

V. Simulation Results

HPU was implemented using SystemVerilog, and the design was synthesized using 28nm library. Fig. 8(a) shows the architectural parameters of the synthesized instances, which cover various conditions of the HPU configuration space. A sensitivity analysis was performed to explore the impact of the processor's integer bit-width on the accuracy and select the proper values of $\boldsymbol{H}$ and $\boldsymbol{C}$. HPU energy was measured using Synopsys PrimeTime PX. HD workloads were also simulated on NVIDIA Tesla V100 GPU, which is used as a baseline for comparison. Workloads were implemented on GPU using PyTorch, and GPU power was measured using the `nvidia-smi` utility.

The algorithms listed in Fig. 8(b) were used for evaluation. These algorithms incorporate all the components shown in Fig. 2. As such, this evaluation framework facilitates a comprehensive assessment of the effects of HPU configuration parameters on multi-layer cognition systems.

A. HPU Parameter-Space Exploration

Latency: We first evaluate the impact of varying HPU size on latency. Fig. 9(a) shows that HPU_4 provides speed-up of $1.3 - 1.8\times$ compared to HPU_2, highlighting resource under-provisioning in HPU_2. However, we observe that the benefits of scaling up the design from HPU_4 to HPU_8 are not equally realized by all algorithms. Specifically, only $1.16\times$ speed-up is achieved by MULT. This is because MULT typically performs HDOP-intensive computations for sequence encoding and thus its response to further increase in design size is minimal. On the other hand, REACT achieves $1.69\times$ speed-up when HPU_8 is used. This result is attributed to the fact that REACT performs extensive clean-up memory operations, which can be efficiently distributed across all tiles.

Next, we attempt to gain a better understanding of the impact of individual configuration parameters ($\boldsymbol{K}$, $\boldsymbol{R}$, $\boldsymbol{B}$, and

Instance Name	Bus Width (W)	# Tiles (K)	# CA-90-RF registers (R)	# BND-RF registers (B)	# DSUM registers (D)	Distance bit-width (C)	BND bit-width (H)	Memory Capacity
HPU_2	512	2	2	2	2	12	8	128 KB
HPU_4	512	4	4	4	4	12	8	256 KB
HPU_8	512	8	8	8	8	12	8	512 KB

(a)

Workload	Layer	Application	Problem Size (Complexity)
MULT	Perception	Multi-modal learning and inference [9]	300 samples, 120 item vectors, 16 prototype vectors (classes), 100 queries
TREE	Reasoning	Tree encoding and search [6]	70 tree structures, 9 items, 400 queries
FACT	Reasoning	Factorization of data sets [8]	60 iterations, 120 item vectors, 13 prototype vectors
REACT	Control	Motor learning and recall [4]	500 samples, 55 item vectors, 160 recalls

(b)

Fig. 8. Evaluation setup: (a) synthesized HPU configurations; (b) HD workloads used in simulation.

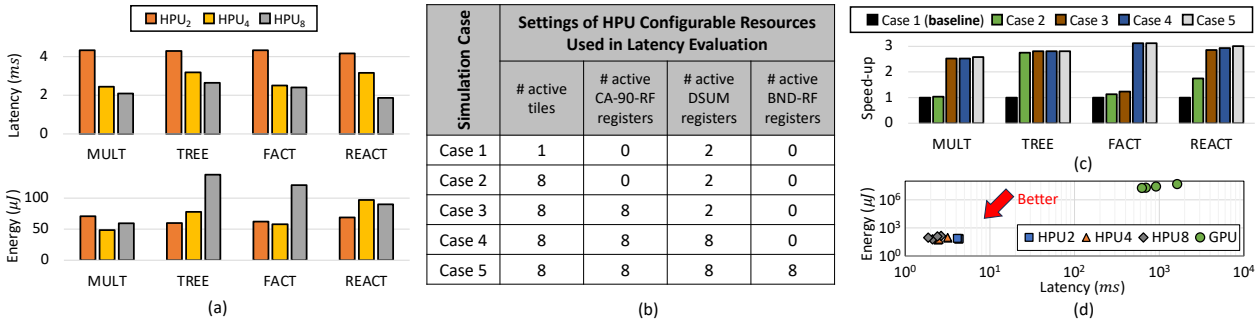


Fig. 9. Demonstration of HPU efficiency: (a) comparison between HPU_2, HPU_4, and HPU_8 in terms of latency and energy consumption, (b) parameter-space exploration (PSE) using HPU_8 as a platform, (c) PSE results, and (d) comparison between HPU and GPU (baseline).

D) on latency. To achieve this, we conducted an evaluation by mapping the workloads onto HPU_8 while selectively disabling certain resources. We considered five different settings of resource availability during simulation, which are depicted in Fig. 9(b). This approach allowed us to isolate the effects of each parameter on latency.

The results of this evaluation are shown in Fig. 9(c). We observe that MULT exhibits a major change in latency (achieving $2.45\times$ speed-up) only when we enable storing folds in CA-90-RF. In other words, its performance is nearly independent of the number of used tiles. On the other hand, TREE and REACT achieve a speed-up of $2.75\times$ and $1.75\times$, respectively, as the number of used tiles is increased. Both algorithms perform a clean-up memory search through a large number of item vectors, which benefits from increasing the number of DC instances. FACT implements a resonator network, which requires using six pipeline stages in each iteration. Hence, a significant speed-up can be observed only if all resources across all pipeline stages are fully utilized.

Energy Consumption: Fig. 9(a) shows that the energy efficiency does not exhibit systematic behavior as the size of HPU is varied. The reasons for this behavior are twofold: (1) The leakage power becomes increasingly significant when HPU size is increased. Our analysis shows that the leakage power increases from 1.7 mW to 5.2 mW (i.e., $3\times$ increase) when the design is scaled up from HPU_2 to HPU_8. (2) Each of HPU instructions has a unique effect on energy consumption, especially because instructions trigger circuit activity at different hardware modules.

These results underscore the critical role of compiler optimization in improving the energy efficiency of HPU. Specifically, they suggest that achieving maximum energy efficiency requires a unique optimization approach, which characterizes energy at both the module and system levels and employs targeted optimizations during compilation.

B. Comparison with GPU

We also compare all HPU instances with GPU in terms of workload latency and energy consumption. Fig. 9(d) shows that HPU is up to three orders of magnitude faster in executing HD workloads than GPU, despite using batch processing in our GPU implementation. This result suggests that the GPU-memory interface is not optimized for HD data transfer. In addition, Fig. 9(d) demonstrates that HPU operation is up to six orders of magnitude more energy efficient than GPU processing. These findings also align with comparisons reported in prior classification processors [9]. This performance gap is attributed to GPU's scalar architecture, which relies on complex SIMD arithmetic units to perform simple vector operations.

VI. Conclusion

This paper presented HPU, an HD processing unit that can implement diverse algorithms. We designed the HPU kernel and dataflow and proposed software control methods over performance tradeoffs. We demonstrated HPU's advanced capabilities in comparison with GPU. We also showed the effect of HPU parameters on the latency and energy consumption of a benchmark suite that implements multi-layer cognition.

References

[1] N. S.-Y. Dumont *et al.*, "Exploiting Semantic Information in a Spiking Neural SLAM System," *Front. Neurosci.*, vol. 17, p. 1190515, 2023.

[2] L. I. G. Olascoaga *et al.*, "A Brain-Inspired Hierarchical Reasoning Framework for Cognition-Augmented Prosthetic Grasping," in *Proc. CLeaR*, 2021.

[3] A. Renner *et al.*, "Neuromorphic Visual Scene Understanding with Resonator Networks," *arXiv preprint arXiv:2208.12880*, 2022.

[4] A. Menon *et al.*, "Shared Control of Assistive Robots through User-intent Prediction and Hyperdimensional Recall of Reactive Behavior," in *Proc. IEEE ICRA*, 2023, pp. 12 638–12 644.

[5] M. Hersche *et al.*, "A Neuro-Vector-Symbolic Architecture for Solving Raven's Progressive Matrices," *Nat. Mach. Intell.*, vol. 5, no. 4, 2023.

[6] D. Kleyko *et al.*, "Vector Symbolic Architectures as a Computing Framework for Emerging Hardware," *Proceedings of the IEEE*, vol. 110, no. 10, pp. 1538–1571, 2022.

[7] A. Hernandez-Cane *et al.*, "OnlineHD: Robust, Efficient, and Single-Pass Online Learning Using Hyperdimensional System," in *Proc. IEEE/ACM DATE*, 2021, pp. 56–61.

[8] E. P. Frady *et al.*, "Resonator Networks, 1: An Efficient Solution for Factoring High-Dimensional, Distributed Representations of Data Structures," *Neural Computation*, vol. 32, no. 12, pp. 2311–2331, 2020.

[9] S. Datta *et al.*, "A Programmable Hyper-Dimensional Processor Architecture for Human-Centric IoT," *IEEE JETCAS*, vol. 9, no. 3, 2019.

[10] A. Menon *et al.*, "A Highly Energy-Efficient Hyperdimensional Computing Processor for Biosignal Classification," *IEEE TBioCAS*, vol. 16, no. 4, pp. 524–534, 2022.

[11] S. Salamat *et al.*, "F5-HD: Fast Flexible FPGA-based Framework for Rrefreshing Hyperdimensional Computing," in *Proc. FPGA*, 2019.

[12] A. Rahimi *et al.*, "A Robust and Energy-Efficient Classifier Using Brain-Inspired Hyperdimensional Computing," in *Proc. ACM/IEEE ISLPED*, 2016, pp. 64–69.

[13] P. Kanerva, "What We Mean When We Say "What's the Dollar of Mexico?": Prototypes and Mapping in Concept Space," in *Proceedings of AAAI Fall Symposium: Quantum Informatics for Cognitive Social and Semantic Processes*, 2010, pp. 2–6.

[14] A. Moin *et al.*, "A Wearable Biosensing System with In-sensor Adaptive Machine Learning for Hand Gesture Recognition," *Nat. Electron.*, vol. 4, no. 1, pp. 54–63, 2021.

[15] M. Eggimann *et al.*, "A 5 μW Standard Cell Memory-Based Configurable Hyperdimensional Computing Accelerator for Always-on Smart Sensing," *IEEE TCAS-I: Regular Papers*, vol. 68, no. 10, pp. 4116–4128, 2021.