uHD: Unary Processing for Lightweight and Dynamic Hyperdimensional Computing

Sercan Aygun, Mehran Shoushtari Moghadam, M. Hassan Najafi
School of Computing and Informatics, University of Louisiana at Lafayette, LA, USA
{sercan.aygun, m.moghadam, najafi}@louisiana.edu

***Abstract*—Hyperdimensional computing (HDC) is a novel computational paradigm that operates on long-dimensional vectors known as hypervectors. The hypervectors are constructed as long bit-streams and form the basic building blocks of HDC systems. In HDC, hypervectors are generated from scalar values without considering bit significance. HDC is efficient and robust for various data processing applications, especially computer vision tasks. To construct HDC models for vision applications, the current state-of-the-art practice utilizes two parameters for data encoding: pixel intensity and pixel position. However, the intensity and position information embedded in high-dimensional vectors are generally not generated dynamically in the HDC models. Consequently, the optimal design of hypervectors with high model accuracy requires powerful computing platforms for training. A more efficient approach is to generate hypervectors *dynamically* during the training phase. To this aim, this work uses *low-discrepancy* sequences to generate intensity hypervectors, while avoiding position hypervectors. Doing so eliminates the multiplication step in vector encoding, resulting in a power-efficient HDC system. For the first time in the literature, our proposed approach employs lightweight vector generators utilizing *unary bit-streams* for efficient encoding of data instead of using conventional comparator-based generators.**

I. Introduction

Traditional computing systems based on positional binary radix encounter practical limitations in the efficient hardware design of today's big data applications. These systems suffer from extremely high power and memory consumption, particularly for cognitive tasks with iterative and complex learning procedures. Emerging computing technologies such as Hyperdimensional Computing (HDC), Stochastic Computing (SC), Unary Bit-stream Computing (UBC), Quantum Computing (QC), and Approximate Computing (AC) are shaping the next generation of computing systems. Among these, HDC has recently gained significant attention due to its lightweight, robust, and efficient solutions for various learning and cognitive tasks [1], [2], particularly for natural language processing [3] and image classification [4]. HDC encodes information using holographic hyperdimensional vectors, known as *hypervectors*, consisting of randomly distributed binary values of -1 (logic-0) and $+1$ (logic-1). This unconventional representation enables fast, robust, efficient, and fully parallel processing of large sets of data [5].

For high-quality HDC, hypervectors are expected to be *orthogonal*, i.e., uncorrelated with each other. By generating *pseudo-random* vectors, prior works encode data to hypervectors that are only *nearly* orthogonal. This work introduces a novel hypervector encoding scheme that radically differs from the encoding methods currently used in HDC systems. We propose a simpler and more effective method to achieve orthogonality by drawing an analogy between HDC and SC [6]. Instead of relying on pseudo-randomness, we leverage *quasi-randomness* provided by low-discrepancy (LD) sequences [7] to generate high-quality hypervectors. In addition, for the first time, to the best of our knowledge, we take advantage of UBC and its unary data representation [8] for the lightweight design of HDC systems. In what follows, we summarize the primary contributions of this work.

① Utilizing quantized LD sequences for hypervector encoding for the first time in the literature.
② Eliminating position hypervectors in HDC system, alleviating the total memory consumption, vector generation load, and arithmetic operations.
③ Developing **uHD**, a hybrid HDC system integrating unary bit-streams and hypervector processing.
④ Developing a lightweight combinational logic to compare unary bit-streams for dynamic generation of hypervectors.
⑤ A new circuitry for the binarization operation needed in HDC systems.
⑥ Achieving a higher image classification accuracy compared to the baseline HDC with pseudo-random hypervectors.

II. Background and Motivation

HDC maps raw input data into a high-dimensional space with hypervectors of $+1$s and -1s [9]. Each dimension in this space corresponds to a feature or attribute in the original data. HDC consists of two primary steps: hypervector *generation* and *encoding*, of which the latter creates another hypervector. While the encoding step has been extensively discussed in the literature [2], [10], vector generation is typically left to the performance of pseudo-randomness [11]. When a **scalar** value X is to be represented using a hypervector, its numerical value can be used for vector generation. However, when X is **symbolic** data (e.g., a *letter*), a *proper* vector should be attributed to the symbol. The term *proper* emphasizes the importance of orthogonality, as each symbol without numerical information should be treated equally and embedded in hypervectors without any bias towards one symbol over another. In other words, each hypervector should have an equal number of $+1$s and -1s with an independent random distribution. This representation requires good randomness to ensure hypervectors remain uncorrelated with each other. An important target of this work is to produce hypervectors with ideal orthogonality. For the scalar case, X can be a grayscale pixel value [4] ($0 \le X \le 255$ for 8-bit representation), the amplitude of a discrete signal [2], or a numerical feature of data [9]. This work will follow the convention for image classification, so we assume X is a pixel value.

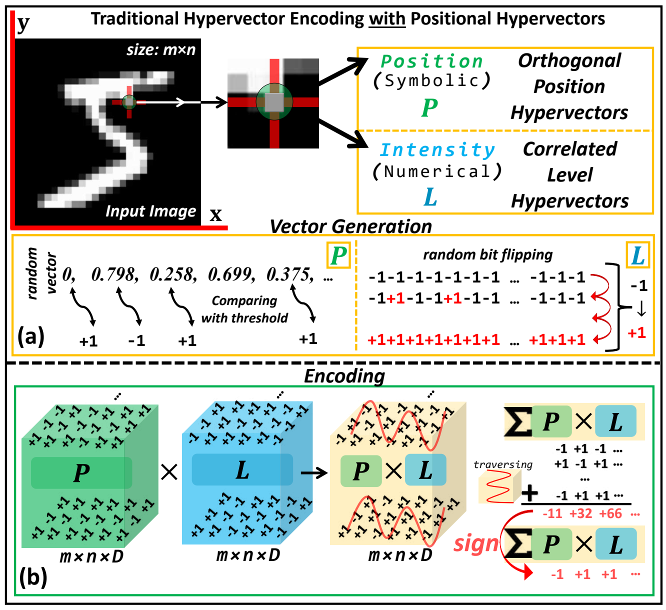


Fig. 1. Traditional hypervector design and encoding. (a) *Positional* and *Level* hypervector generation, (b) Binding, Bundling, and Binarization.

Fig. 1(a) shows a sample image pixel and its corresponding position ($\boldsymbol{P}$) and level ($\boldsymbol{L}$) hypervectors. Hypervectors are assigned with a dimension or size of D. $\boldsymbol{P}s$ are obtained from **symbolic** data, and $\boldsymbol{L}s$ from **scalar** values. $\boldsymbol{P}s$ are generated by comparing some random (R) numbers ($0 \leq R_{1..D} \leq 1$) with a threshold value ($t = 0.5$; *no-bias* point between 0 and 1). $\boldsymbol{L}s$ are typically generated by *bit flipping* [11]. Hence, closer numerical scalars have similar hypervectors, while different numerical scalars have more uncorrelated hypervectors. In generating both $\boldsymbol{P}$ and $\boldsymbol{L}$, a $+1$ or -1 value is returned for any hypervector position. If $R > t$, the corresponding position is set to -1; otherwise, it is set to $+1$ [2].

Fig. 1(b) illustrates the remaining encoding steps on the generated hypervectors. These steps compose the `class hypervector` ($\boldsymbol{C}$), which holds the overall representation of a class (e.g., Fig. 1 shows an image from class-*5* and its contribution to the corresponding `class hypervector`). All images in the training set contribute to building the `class hypervectors` by processing the hypervectors ($\boldsymbol{P}$ and $\boldsymbol{L}$) of all their pixels. The generated hypervectors are first multiplied element-wise (via bit-wise `XOR`). This is known as *binding*. The multiplied hypervectors coming from each pixel ($\boldsymbol{L} \oplus \boldsymbol{P}$) are accumulated by traversing positions. Hypervectors are added to each other by another element-wise processing (bit-wise `popcount`). This is known as *bundling* [12]. Then, the final values are evaluated for `class hypervectors` after scanning all data samples of the same class. Finally, a *binarization* operation is performed via a `sign` function (thresholding with a `comparator` or a `subtractor`) [3]. For each class in training, the labeled data are processed to build their corresponding `class hypervector`. This operation is performed only once, different from the conventional learning systems with iterative forward passes throughout the batches and epochs.

When all `class hypervectors` are defined ($\boldsymbol{C}_{1..q}$ with q-class dataset), the inference step measures the accuracy of the testing dataset. The same encoding steps are followed for any testing data to obtain a testing hypervector ($\boldsymbol{C}_{test}$). The final classification is performed using a similarity check between $\boldsymbol{C}_{test}$ vs. $\boldsymbol{C}_1$, $\boldsymbol{C}_2$, ..., and $\boldsymbol{C}_q$. In this work, we use cosine similarity. The highest similarity between $\boldsymbol{C}_{test}$ and one of the trained classes gives the classification decision [3].

Generating pseudo-random hypervectors with high orthogonality during training can be very time- and memory-consuming. To obtain a high classification accuracy, the best performing $\boldsymbol{P}$ and $\boldsymbol{L}$ random hypervectors are assigned iteratively. Hypervectors with different distributions are generated iteratively to find those with the highest orthogonality. One of our goals in this work is to minimize the number of vector operations. The bit-wise `XOR` operations in the binding process involve both $\boldsymbol{P}$ and $\boldsymbol{L}$ hypervectors. We use an encoding for level hypervectors that does not need iteration and provides accurate encoding deterministically [13]. Instead of pseudo-randomness, we provide high orthogonality via *quasi-randomness* ①. Our approach eliminates the need for position encoding and their corresponding multiplications ②. Thus, single-iteration vector optimization is guaranteed thanks to the properties of LD sequences [14].

As the first work of its kind, we use *unary bit-streams* in HDC systems ③. UBC utilizes unary (aka thermometer) coding, representing data using bit-streams with logic-1s (or logic-0s) aligned to the beginning or end of the bit-stream. For instance, $\begin{smallmatrix} X1 \rightarrow 0\,0\,0\,0\,0\,1\,1 \\ X2 \rightarrow 0\,0\,1\,1\,1\,1\,1 \end{smallmatrix}$ are two unary bit-streams of size N=7 representing 2 and 5. UBC can be exploited for the lightweight design of HDC systems. Hypervector generation in current HDC systems requires conventional binary `comparators`, which are complex and consume significant power. We employ UBC to design a new lightweight `comparator` logic for dynamic hypervector generation ④.

In addition to optimizing vector generation and minimizing the operations in encoding, we improve the hardware design of the final stage with accumulation and binarization. We propose a concurrent binarization during `popcount`ing; Processing over binary data allows using `popcount` to count only the number of logic-1s. The binary output is obtained after D cycles to be compared or subtracted from a *threshold* value. This requires a separate module for thresholding or subtraction. We simplify the binarization module to make the decision on the spot while performing popcount ⑤.

III. Efficient Hypervector Encoding with uHD

We call the new unary HDC system **uHD**. **uHD** enjoys a lightweight architecture by taking advantage of unary processing. It also provides a higher accuracy by exploiting the uncorrelation and recurrence properties of LD sequences.

uHD radically alters the encoding approach in HDC systems. Conventional HDC systems are bounded by the spatial information of discrete data. LD sequences provide built-in indexes to be used for the positional information. Fig. 2 depicts the encoding using LD Sobol [7] scalars and indexes. We eliminate $\boldsymbol{P}s$ and only encode $\boldsymbol{L}s$ by using Sobol scalars. As shown in Fig. 2, for encoding image data, we compare LD Sobol sequences ($\boldsymbol{S}_i$) with image intensity values. We do not encode positions; instead, we use the corresponding index of any Sobol sequence ($\boldsymbol{S}_i$) ranging from $\boldsymbol{S}_1$ to $\boldsymbol{S}_{\texttt{row×column}}$.

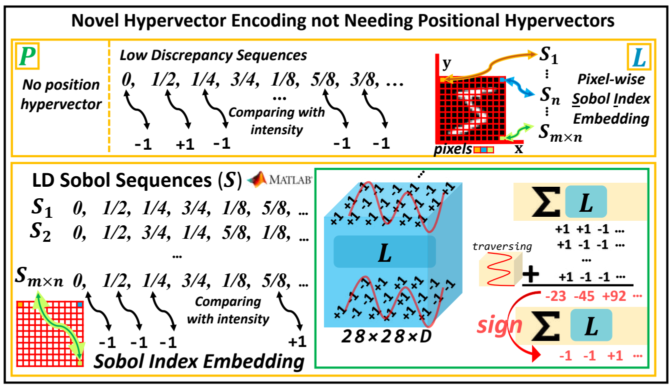
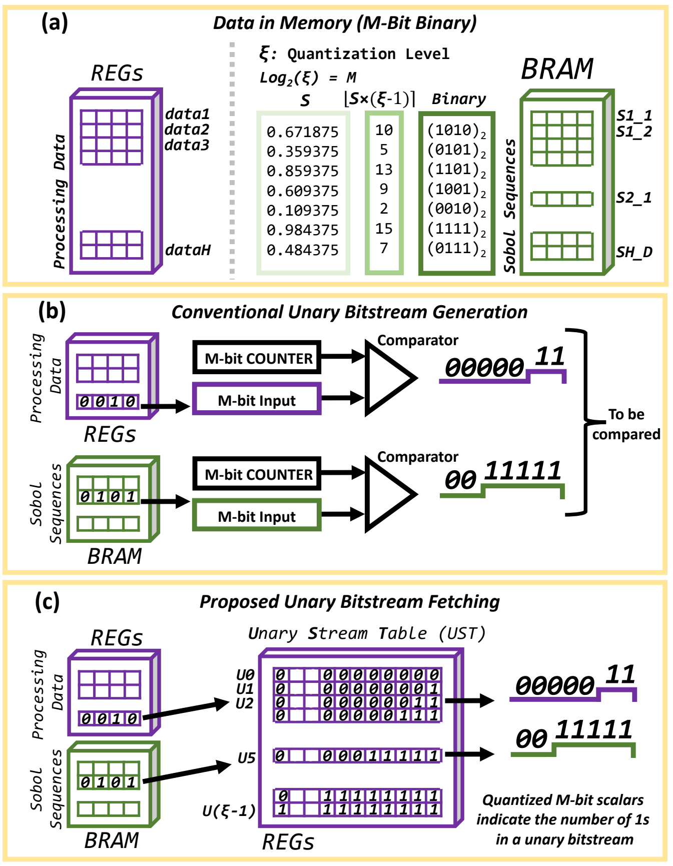


Fig. 2. Hypervector generation using Sobol sequences [15].

Finally, the non-binary image hypervector formula turns into $\Sigma_{i=1}^{N}(\boldsymbol{L}_i)$. Any pixel intensity is encoded based on the pixel position corresponding to the Sobol index. The normalized intensity value (by D) is compared with each element in the corresponding Sobol sequence. If the normalized intensity is smaller than the Sobol number, the hypervector position gets -1; otherwise, it gets $+1$. After obtaining $\boldsymbol{L}$, we perform the accumulation without the encoding's multiplication step. Thus, our novel approach achieves a *multiplier-less* vector encoding for HDC.

From efficient hypervector encoding to complete hardware design, we focus on extended design perspectives for efficient HDC system design. Most prior works present hardware design for the inference. However, training on edge devices is a more challenging task. For high accuracy, the baseline HDC requires iterative hypervector generation and processing. Hence, single-pass data processing can significantly reduce runtime and energy consumption. Our proposed encoding achieves this benefit with a deterministic and reliable one-time iteration.

uHD reads two sets of data from memory: (i) *processing data* such as image pixels or features and (ii) *Sobol sequences*. We quantize both input data and Sobol scalar values in the proposed approach. Utilizing Sobol scalar and index encoding removes the need for $\boldsymbol{P}s$ and corresponding multiplications. We use unary bit-streams instead of the conventional binary radix encoding, bringing UBC into HDC systems for the first time. Now, let us take a look at the overall design. With M-bit quantization, only M-bit data is stored in memory. The input data size depends on the *features* or *raw data* size, such as the image's `row×column`. Each Sobol sequence has a length of D (i.e., has D Sobol numbers), where D is typically in the range of $1K$ to $10K$. Storing all Sobol data in registers may not be possible as they may exceed the memory size of the resource-constrained devices. Therefore, we use block `RAM` (`BRAM`) in a re-configurable design platform to store the quantized Sobol data. The processing data are relatively lighter in size, so we keep them in registers. Fig. 3(a) illustrates how we keep the data. The data are in quantized binary format (e.g., M=4) in registers (`REGs`). For Sobol scalars, a `BRAM` module holds binary values in M=4-bit (holding N=16-bit unary bit-streams. Each scalar shows the total count of logic 1s in the bit-stream). Fig. 3(a) shows an example of quantizing Sobol sequences. Here, ξ=16-level quantization is applied to obtain the to-be-stored binary values. We note that this



S	⌊S×(ξ-1)⌉	Binary
0.671875	10	$(1010)_2$
0.359375	5	$(0101)_2$
0.859375	13	$(1101)_2$
0.609375	9	$(1001)_2$
0.109375	2	$(0010)_2$
0.984375	15	$(1111)_2$
0.484375	7	$(0111)_2$

Fig. 3. Getting ready to hypervector generation. (a) Data represented in memory for unary bit-stream processing, (b) Conventional unary stream generation, and (c) proposed associative stream fetching.

data quantization does not affect the system's accuracy. Even though hypervector generation may experience some flipped bits ($+1$ instead of -1, or vice versa), the accumulated values yield large scalars (non-quantized `class hypervector`), and the `sign` of accumulation is not easily affected.

Let us now discuss how we convert the data to unary bit-streams. Unary bit-streams are conventionally generated by using a pair of M-bit binary `counter` and `comparator` [16], as shown in Fig. 3(b). This design is compact, especially for dynamic bit-stream generation with large sizes. However, our HDC design works only on N=16-bit sequences, and so all possible sequences can be pre-stored in memory. The data in memory are converted to unary bit-streams on the fly. Fig. 3(c) shows how we fetch the pre-stored unary bit-streams from an associative memory. The binary scalar in `REGs` or `BRAM` points to the corresponding index of a Unary Stream Table (UST), and the target bit-stream is fetched. We put the first design checkpoint here ❶ and compare energy consumption. We synthesize **uHD** and the baseline design using Synopsys Design Compiler with a 45-nm cell library. We compare the energy consumption of the two approaches for generating one bit of the hypervector. We observed that **uHD** consumes $0.77fJ$ energy while the baseline design consumes $0.167pJ$ (both designed for D=$1K$).

We generate hypervectors by comparing the data and Sobol scalars. Instead of directly comparing quantized scalars via conventional `comparators`, we use a novel unary bit-stream

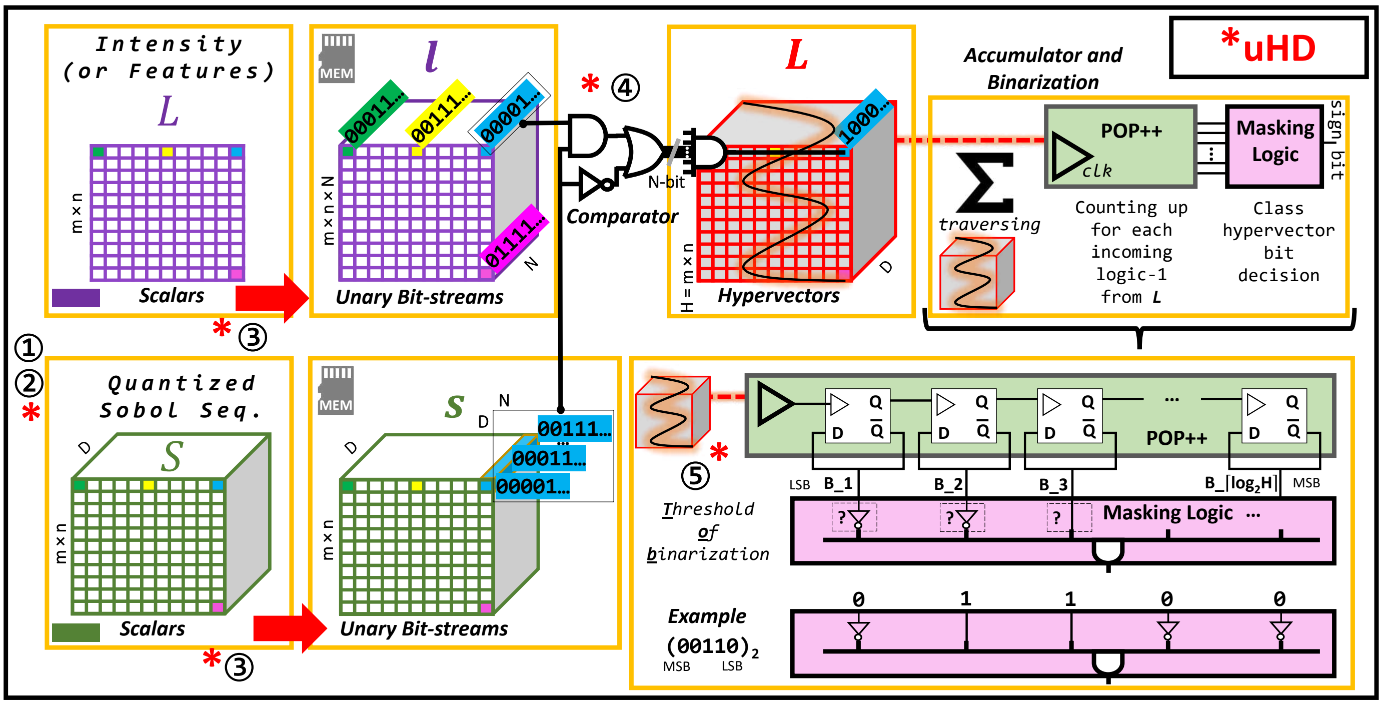


Fig. 4. Complete system overview of **uHD**. * indicates our proposals.

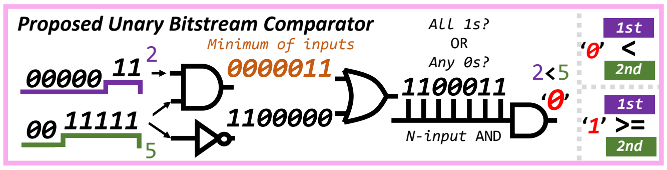


Fig. 5. Proposed unary bit-stream comparator.

comparator. Unary bit-streams (with the same length) are correlated, and ANDing them gives the minimum bit-stream. Fig. 5 illustrates the proposed unary comparator for comparing two N=7-bit unary inputs. The data- (🟪) and Sobol (🟩) unary bit-streams are compared to generate one bit of the hypervector: If the 1st operand (here, *data*) is greater than or equal to the 2nd operand (here, *Sobol value*), then the corresponding output is logic-1; otherwise, it is logic-0. The bit-wise AND operation determines the *minimum* input. The inverted unary Sobol is checked (by an OR gate) to see if the *minimum* equals this. The example in Fig. 5 compares two unary bit-streams corresponding to values 2 and 5. After finding the *minimum* (bronze color in Fig. 5), the inverted Sobol is checked to see if bit-wise ORing gives all-1s or at least one logic-0 at the output. If the Sobol number is the *minimum*, the OR operation returns a logic-1, and ANDing N consecutive bits at the output generates a logic-1 bit for the hypervector. If there is at least one logic-0 after OR, the final AND detects it and resets the bit of the hypervector. Since, in this example, the data (value of 2 in Fig. 5) is less than the Sobol number (value of 5 in Fig. 5), the output is not all-1s, and a logic-0 is generated. Here, we have the second design checkpoint ❷ to evaluate the energy consumption of hypervector generation when using the proposed unary comparator. The baseline HDC with conventional comparators consumes $2.49pJ$ while **uHD** consumes $0.24pJ$ (both designed for D=$1K$).

After encoding the level hypervectors (***L***s) using the steps above, the next step is the *accumulation* and *binarization*. Fig. 4 illustrates the overall design of **uHD**. The main contributions are underscored in this figure. The red-colored ***L*** is traversed for accumulation. For each hypervector bit, a popcount operation returns a binary output, counting the number of logic-1s. The D-type flip-flops are used in our design for this purpose. We propose a new binarization method that operates concurrently with the popcount instead of using an extra subtractor or comparator for the Threshold of Binarization (**TOB**). The size of the incoming data (the total pixel or feature count), H, is the maximum value to count up. Hence, a $\lceil log_2 H \rceil$-bit counter is required. $\frac{H}{2} = TOB$ is the critical threshold reached by popcount output for the decision on the logic-1s-in-majority. When the threshold is reached, the sign bit is set for the corresponding bit of the class hypervector; otherwise, it is 0. We propose to use a masking logic for capturing **TOB**, which is in binary: $(B_{\lceil log_2 H \rceil} ... B_2\ B_1)_2$. As shown in Fig. 4, the masking logic is hardwired, feeding $\lceil log_2 H \rceil$ bits to an AND gate. When popcount reaches **TOB**, this hardwired threshold guarantees logic-1 at the output of AND; otherwise, it remains logic-0 [17]. Here, we set another design checkpoint ❸ and compare the energy consumption of the baseline and **uHD** design for *accumulate-and-binarize* operation. We observe that **uHD** consumes $34.7pJ$ energy per feature of the incoming image, while for the same data, the baseline design consumes $68.7pJ$ energy (both designed for D=$1K$).

TABLE I
ENERGY CONSUMPTION AND *Area* × *Delay* COMPARISON OF **uHD** & THE BASELINE HDC FOR EACH HYPERVECTOR (HV) AND IMAGE

Design Approach	**Energy Consumption (pJ)**			**Area × Delay (m^2× s)**		
	D=1K	**D=2K**	**D=8K**	**D=1K**	**D=2K**	**D=8K**
uHD per HV	0.79	1.58	6.32	40.60×10^{-12}	81.20×10^{-12}	324.80×10^{-12}
uHD per image (MNIST)	113.76	227.52	910.08	5.83×10^{-9}	11.67×10^{-9}	46.69×10^{-9}
Baseline per HV	171.42	415.41	4023.82	11.79×10^{-9}	25.55×10^{-9}	230.33×10^{-9}
Baseline per image (MNIST)	24.68×10^{3}	59.80×10^{3}	57.94×10^{4}	1.70×10^{-6}	3.70×10^{-6}	33.17×10^{-6}

IV. DESIGN EVALUATION AND RESULTS

We evaluate the performance, hardware cost, energy consumption, and area×delay of **uHD** compared to the baseline architecture and the prior state-of-the-art (SOTA). We utilize the standard MNIST dataset for accuracy evaluations [18]. We compare the hardware costs of the baseline and **uHD** architecture, specifically for the hypervector generation process. The baseline design follows the dynamic and independent training target. Linear-feedback shift registers (LFSRs) are used for hypervector generation in the baseline design. Table I compares the *energy consumption* and *area×delay* numbers to evaluate the hardware efficiency of the proposed design. While an iterative design is required for the baseline design (like i=100 different attempts to get the best-performing hypervectors), we credit it by assuming that hypervectors are the best, and only a single-run is sufficient for high accuracy. However, for realistic baseline training on an edge device, more iterations are needed for high accuracy, which accordingly increases the energy consumption of the baseline design. We estimate the energy consumption for generating each $\boldsymbol{P} \times \boldsymbol{L}$ hypervector in the baseline HDC (Fig. 1(b)) and for each $\boldsymbol{L}$ hypervector in the **uHD** design (Fig. 2). As can be seen in the reported numbers, our proposed **uHD** is more hardware-efficient than the baseline HDC.

Table II compares the SOTA HDC architectures implemented on a central processing unit or microprocessor. A thorough benchmarking is outlined in the HDC surveys by Hassan et al. [19] and Chang et al. [20]. This table ranks the top energy-efficient frameworks for overall architecture (including *hypervector generation*, *binding*, *bundling*, and *binarization*) from the aforementioned surveys and contrasts them with our proposed architecture, which benefits from a novel approach for hypervector generation. As it can be seen, our proposed HDC architecture provides remarkable energy efficiency by exploiting UBC.

Table III compares the accuracy performance of the baseline HDC and **uHD**. The baseline architecture is monitored at different iterations of generating hypervectors ($\boldsymbol{P}$ and $\boldsymbol{L}$). At each random hypervector assignment in the training phase (i=1...100), the test accuracy is recorded. The table reports the average accuracy values at different checkpoints of i. **uHD** utilizes LD Sobol sequences and completes its deterministic hypervector (only $\boldsymbol{L}$) assignment in a single iteration (i=1). The MNIST dataset is segmented to separate the training and testing images. For the sake of fair accuracy comparison between the two designs, there is no retaining, no neural network (NN) assistance, and no prior optimizations. Some prior work heavily relies on these optimizations; however, these optimizations and the use of other machine learning (ML) techniques affect the cost of the training hardware. The impact of using random vectors in the baseline HDC is reported in Fig. 6(a). The fluctuations in the testing accuracy underscore the importance of having an iterative process for selecting the best vectors. Fig. 6(b) reports the accuracy of prior SOTA HDC systems (the ones without NN assistance, complex optimizations in training, or multi-models - only with (w/) or without (w/o) the retraining efforts) in performing MNIST classification. As can be seen, **uHD** with single-pass learning achieves better accuracy compared to the baseline and SOTA designs.

To extend our evaluations, we utilized various image-based datasets, including CIFAR-10, BloodMNIST, BreastMNIST, FashionMNIST, and SVHN [30]. Table IV presents the accuracy comparison between the baseline HDC and our proposed **uHD**. We note that these accuracy results were obtained without employing any optimization (e.g., retraining, NN assistance, or transfer learning). The findings demonstrate the effectiveness and versatility of **uHD** across different datasets, showcasing its potential for various machine vision applications. The ISO-accuracy values presented show that **uHD** exhibits superior hardware efficiency compared to conventional learning frameworks (ML, deep neural networks), which often require resource-intensive hardware setups. As a result, **uHD** offers a more efficient and cost-effective solution to achieve the same accuracy level.

TABLE II
ENERGY EFFICIENCY OVER BASELINE ARCHITECTURES [19], [20]

HDC Framework	**Platform**	**Energy Efficiency**
Semi-HD [21]	Raspberry Pi	12.60×
Voice-HD [22]	Central Processing Unit	11.90×
tiny-HD [23]	Microprocessor	11.20×
PULP-HD [24]	ARM Microprocessor	9.9×
Hierarchical-MHD [25]	Central Processing Unit	6.60×
AdaptHD [26]	Raspberry Pi	6.30×
Laelaps [27]	Central Processing Unit	1.40×
This work	**ARM Microprocessor**	**31.83×**

In a comprehensive survey conducted in [19] and [20], several HDC frameworks were benchmarked based on their energy efficiency in comparison to reference baseline models. This table lists the top energy-efficient architectures from [19] and [20]. All frameworks, including ours, report the overall (including memory read/write, hypervector generation, binding, and bundling) system energy consumption.

TABLE III
MNIST ACCURACY PERFORMANCE (%) OF *Baseline HDC* AND **uHD** ⑥

D	**Baseline HDC (Average)**						**uHD**
	i=1	***i=1..5***	***i=1..20***	***i=1..50***	***i=1..75***	***i=1..100***	***i=1***
1K	82.93	83.60	83.49	82.70	82.88	82.63	**84.44**
2K	86.24	86.58	87.05	86.35	86.37	86.53	**87.04**
8K	88.30	88.55	88.25	88.13	88.14	88.13	**88.41**

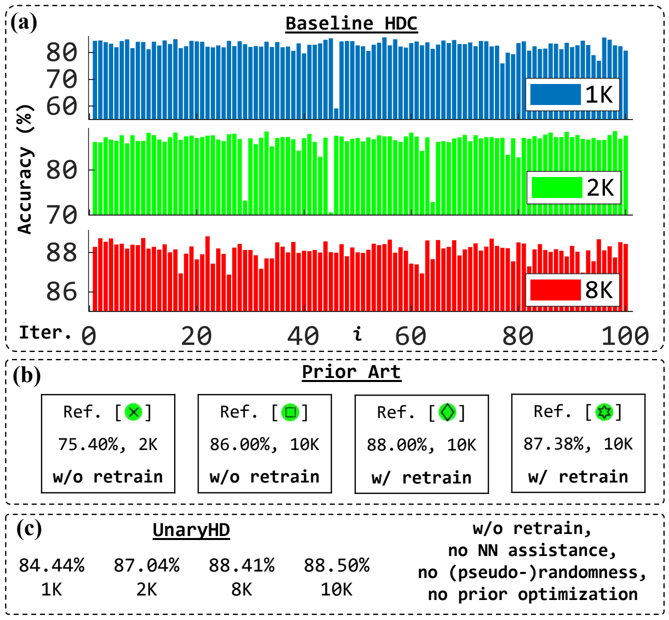


Fig. 6. Different accuracy monitoring of HDC designs. Accuracy fluctuations at each iteration of the baseline design with (pseudo-)randomness (a), MNIST dataset accuracy from prior HDC works; Ref.s ⊗ → [4], ▣ → [19], ◈ → [28], ✡ → [9], [29]) (b), and **uHD** (c).

TABLE IV
ACCURACY (%) COMPARISON OF THE BASELINE HDC AND **uHD** FOR DIFFERENT IMAGE DATASETS.

Datasets	D=1K		D=2K		D=8K	
	Ours	*Baseline*	*Ours*	*Baseline*	*Ours*	*Baseline*
CIFAR-10	**39.29**	38.21	**40.28**	40.26	**41.97**	41.71
Blood MNIST	**53.05**	48.52	**55.86**	51.20	**57.88**	51.82
Breast MNIST	**68.59**	68.47	**69.23**	69.11	**71.15**	70.93
Fashion MNIST	**68.60**	54.19	**70.06**	69.97	**71.37**	70.87
SVHN	**60.29**	60.06	**61.73**	61.24	**62.87**	62.82

For the Baseline HDC, the P and L hypervectors were generated using conventional random sequence generation.

V. Conclusions

This study proposed a hybrid HDC system, **uHD**, by employing UBC in HDC for the first time. The new design simplifies hardware implementation, providing significant hardware cost savings compared to the baseline HDC. **uHD** utilizes LD sequences for deterministic and high-quality generation of hypervectors. It achieves higher accuracy compared to the baseline HDC while offering single-iteration training. We propose a novel hypervector generator by representing data in the unary domain and comparing data using a novel unary comparator. The impact of adding the proposed modules is studied by comparing the energy consumption of the proposed design with the baseline HDC at different design checkpoints. We hope this work opens new avenues for HDC by employing the complementary advantages of emerging computing technologies such as SC and UBC in HDC.

Acknowledgments

This work was supported in part by National Science Foundation (NSF) grant #2019511, the Louisiana Board of Regents Support Fund #LEQSF(2020-23)-RD-A-26, and generous gifts from Cisco, Xilinx, and Nvidia.

References

[1] S. Aygun, M. S. Moghadam, M. H. Najafi, and M. Imani, "Learning from hypervectors: A survey on hypervector encoding," 2023.
[2] L. Ge and K. K. Parhi, "Classification using hyperdimensional computing: A review," *IEEE Circ. and Syst. Mag.*, 2020.
[3] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," 2016.
[4] S. Datta, R. A. G. Antonio, A. R. S. Ison, and J. M. Rabaey, "A programmable hyper-dimensional processor architecture for human-centric iot," *IEEE JETCAS*, 2019.
[5] O. Räsänen *et al.*, "Modeling dependencies in multiple parallel data streams with hyperdimensional computing," *IEEE SPL*, 2014.
[6] A. Alaghi, W. Qian, and J. P. Hayes, "The promise and challenge of stochastic computing," *IEEE TCAD*, vol. 37, no. 8, pp. 1515–1531, 2018.
[7] S. Liu and J. Han, "Energy efficient stochastic computing with Sobol sequences," in *DATE*, 2017, pp. 650–653.
[8] D. Wu, J. Li, R. Yin, H. Hsiao, Y. Kim, and J. S. Miguel, "UGEMM: Unary computing architecture for gemm applications," in *ISCA*, 2020.
[9] M. Imani, S. Bosch, S. Datta, S. Ramakrishna, S. Salamat, J. M. Rabaey, and T. Rosing, "QuantHD: A quantization framework for hyperdimensional computing," *IEEE TCAS*, vol. 39, no. 10, pp. 2268–2278, 2020.
[10] D. Kleyko, D. A. Rachkovskij, E. Osipov, and A. Rahimi, "A survey on hyperdimensional computing aka vector symbolic architectures, part I: Models and data transformations," *ACM Comput. Surv.*, Dec 2022.
[11] T. Basaklar, Y. Tuncel, S. Y. Narayana, S. Gumussoy, and U. Y. Ogras, "Hypervector design for efficient hyperdimensional computing on edge devices," 2021, arXiv 2103.06709.
[12] A. Rahimi, P. Kanerva, L. Benini, and J. M. Rabaey, "Efficient biosignal processing using hyperdimensional computing: Network templates for combined learning and classification of ExG signals," *Proceedings of the IEEE*, 2019.
[13] S. Aygun, M. H. Najafi, and M. Imani, "A linear-time, optimization-free, and edge device-compatible hypervector encoding," in *DATE*, 2023.
[14] S. Liu and J. Han, "Toward energy-efficient stochastic circuits using parallel sobol sequences," *IEEE TVLSI*, vol. 26, no. 7, 2018.
[15] M. S. Moghadam, S. Aygun, and M. H. Najafi, "No-multiplication deterministic hyperdimensional encoding for resource-constrained devices," *IEEE ESL*, 2023.
[16] M. H. Najafi, D. J. Lilja, M. D. Riedel, and K. Bazargan, "Low-cost sorting network circuits using unary processing," *IEEE VLSI*, 2018.
[17] S. Aygun, E. O. Gunes, and C. De Vleeschouwer, "Efficient and robust bitstream processing in binarised neural networks," *Electron. Lett.*, 2021.
[18] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.
[19] E. Hassan, Y. Halawani, B. Mohammad, and H. Saleh, "Hyper-dimensional computing challenges and opportunities for AI applications," *IEEE Access*, 2022.
[20] C.-Y. Chang, Y.-C. Chuang, C.-T. Huang, and A.-Y. Wu, "Recent progress and development of hyperdimensional computing (HDC) for edge intelligence," *IEEE JETCAS*, pp. 1–1, 2023.
[21] M. Imani, S. Bosch, M. Javaheripi, B. Rouhani, X. Wu, F. Koushanfar, and T. Rosing, "SemiHD: Semi-supervised learning using hyperdimensional computing," in *ICCAD*, 2019, pp. 1–8.
[22] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "VoiceHD: Hyperdimensional computing for efficient speech recognition," in *IEEE ICRC*, 2017.
[23] B. Khaleghi, H. Xu, J. Morris, and T. S. Rosing, "tiny-HD: Ultra-efficient hyperdimensional computing engine for IoT applications," in *DATE*, 2021.
[24] F. Montagna, A. Rahimi, S. Benatti, D. Rossi, and L. Benini, "PULP-HD: Accelerating brain-inspired high-dimensional computing on a parallel ultra-low power platform," in *DAC*, 2018.
[25] M. Imani, C. Huang, D. Kong, and T. Rosing, "Hierarchical hyperdimensional computing for energy efficient classification," in *DAC*, 2018.
[26] M. Imani, J. Morris, S. Bosch, H. Shu, G. D. Micheli, and T. Rosing, "Adapthd: Adaptive efficient training for brain-inspired hyperdimensional computing," in *BioCAS*, 2019, pp. 1–4.
[27] A. Burrello, L. Cavigelli, K. Schindler, L. Benini, and A. Rahimi, "Laelaps: An energy-efficient seizure detection algorithm from long-term human iEEG recordings without false alarms," in *DATE*, 2019.
[28] C.-Y. Hsieh, Y.-C. Chuang, and A.-Y. A. Wu, "FL-HDC: Hyperdimensional computing design for the application of federated learning," in *AICAS*, 2021.
[29] S. Duan, X. Xu, and S. Ren, "A brain-inspired low-dimensional computing classifier for inference on tiny devices," 2022.
[30] "Datasets," *Hugging Face Datasets*, https://huggingface.co/datasets?sort=trending.