tiny-HD: Ultra-Efficient Hyperdimensional Computing Engine for IoT Applications

Behnam Khaleghi, Hanyang Xu, Justin Morris, Tajana Šimunić Rosing
CSE Department, UC San Diego, La Jolla, CA 92093, USA
Email: {bkhaleghi, hax032, j1morris, tajana}@ucsd.edu

***Abstract*—Hyperdimensional computing (HD) is a new brain-inspired algorithm that mimics the human brain for cognitive tasks. Despite its inherent potential, the practical efficiency of HD is tied to the underlying hardware, which throttles the efficiency of HD in conventional microprocessors. In this paper, we propose tiny-HD, a light-weight dedicated HD platform that targets low power, high energy efficiency, and low latency, while being configurable to support various applications. We leverage an enhanced HD encoding that alleviates the memory requirements and also simplifies the dataflow to make tiny-HD flexible with an efficient architecture. We further augment tiny-HD by pipelining the stages and resource sharing, as well as a data layout that enables opportunistic power reduction. We compared tiny-HD in terms of area, performance, power, and energy consumption with the state-of-the-art HD platforms. tiny-HD occupies ∼0.5 mm^2, consumes 1.6 mW standby and 9.6 mW runtime power (at 400 MHz), with a 0.016 ms latency on a set of IoT benchmarks. tiny-HD consumes average per-query energy of 160 nJ, which outperforms the state-of-the-art FPGA and ASIC implementations by 95.5× and 11.2×, respectively.**

I. Introduction

Internet of Things (IoT) devices are nowadays generating an ever-increasing volume of data that requires immediate use of machine learning algorithms to generate useful insights [1]. Although state-of-the-art machine learning techniques achieve high accuracy, they demand substantial compute and memory requirements that are beyond the capability of many battery-operated IoT devices [2]. Cloud-based analysis might be an alternative, but that is not always applicable due to communication (latency and reliability), storage, and privacy or security challenges [3].

Brain-inspired hyperdimensional computing (HD for short) is an emerging efficient machine learning technique that builds upon the evidence that brain maps the low-dimensional sensory input to high-dimensional distributed representation for subsequent analysis [4], [5]. By using a simple and parallelizable set of operations, HD computing maps raw data to high-dimensional vectors of length $\mathcal{D}_{hv}$, typically in the range of 2–4,000. Thereafter, these high-dimensional vectors, called *hypervectors*, that belong to the same category are bundled to represent the *set* of that category. This enables *classification* by mapping a query input to a hypervector using the same procedure and calculating the similarity of the query hypervector with all class hypervectors (sets) by using a suitable similarity metric. A growing number of studies have extended HD to various algorithms and applications such as speech recognition [6], bio-signal processing [7], robotics sensorimotor control [8], activity prediction [9], and clustering [10].

HD computing has a variety of appealing characteristics, particularly for the IoT domain. HD can work with simple vector-wise add and multiplication, which makes the operations parallelizable over all $\mathcal{D}_{hv}$ coordinates. HD is highly robust to error (bit-flip or noise), e.g., in an in-memory implementation, HD accuracy is degraded by only 4% in the presence of 30% bit error [11]. HD also enjoys several algorithmic characteristics that make it suitable for efficient online learning, [12], privacy-preserved [13] as well as secure federated learning [14].

Although HD computing is inherently efficient, the achievable benefit is tied to the underlying hardware. For instance, on CPU-based systems, the performance of HD is within the same range of a shallow neural network [7] or just $\sim 2-3\times$ of a conventional ML technique such as SVM or Naive Bayes [15]. This is due to the small cache size of microprocessors that makes the hypervectors movement bottleneck. The functional units of CPUs are also not optimized for low-bit operands such as binary popcount that is prevalent in HD. On the other hand, FPGA-based [16]–[19] and in-memory [20], [21] implementations of HD have shown remarkable performance and energy efficiency over CPU implementation by offering high parallelism and bit-granularity. Nonetheless, FPGA devices typically have a large form-factor and consume a considerable standby (leakage) power which drains a typical battery within a few hours. The recent in-memory implementation of HD [21] has shown promising improvement over CMOS implementations, however, in-memory computing is not yet a mature technology for mass production.

A few studies have focused on dedicated circuits for HD [22]–[25]. Nevertheless, these studies either proposed non-configurable ASICs that only implement a single application [22], [23], and/or low-power processors that still fail to compete with FPGAs in energy-per-query efficiency [24]. To the best of our knowledge, the work in [25] is the only custom HD platform which is, to a certain degree, programmable in terms of the length of hypervectors and supports different applications. Although the design achieves a relatively small energy-per-query of less than 7.5 µJ, it demands a lot of power ($267-1{,}335$ mW) which is comparable with an edge DNN accelerator.

In this paper, we propose tiny-HD, a small, low-power, and dedicated yet configurable HD engine, that can augment IoT devices by enabling ultra-efficient classification on the edge with consuming significantly less per-query energy consumption compared to the state of the art platforms. More specifically, we make the following key contributions.

(1) We analyze different HD algorithms and realize an efficient HD algorithm that helps us design tiny-HD ASIC architecture with minimal resources.

(2) tiny-HD is configurable in terms of the hypervector length and number of classes, thus it supports various applications,

978-3-9819263-5-4/DATE21/©2021 EDAA

but also the same application with different configurations.
(3) Leveraging the data layout of tiny-HD, we effectively augment tiny-HD with power-gating for opportunistic power efficiency.
(4) At 22 nm node, tiny-HD has an area of 0.496 mm^2 and consumes 1.6 mW standby power on a set of IoT benchmarks. tiny-HD achieves an average per-query energy of 160 nJ with a total runtime power of 9.6 mW, surpassing the state-of-the-art ASIC-HD [25] and FPGA [17] energy by 11.2× and 95.5×, respectively.

II. Hyperdimensoinal Computing Background

The human brain computes with the pattern of neural activity which can be explained by operations on hypervectors. The typical length of HD hypervectors, $\mathcal{D}_{hv}$, is around 4,000 dimensions. The large dimensionality of hypervectors makes any randomly chosen pair of points to be orthogonal, that is, their inner-product is zero. Consequently, for vectors V_α, V_β, and V_γ, their *bundling* (element-wise sum) $V_\Sigma = V_\alpha + V_\beta + V_\gamma$ is closer to V_α, V_β and V_γ than any other hypervector. This facilitates classification by encoding low-dimensional raw data into hypervectors and obtaining representative class (set) hypervectors by bundling the related hypervectors. An *encoded* query is more similar to its class hypervector than other classes under a well-defined similarity metric.

(1) Encoding: The first step of HD is mapping (encoding) the input signal (e.g., an image or a time-series window) to high-dimensional vectors. This step is common between all HD algorithms. Assume the input is represented by the vector $V_{iv} = \langle v_1, v_2, \cdots, v_{d_{iv}} \rangle$ where v_is show the features and d_{iv} is the length of input vector. Thus, enc : $\mathbb{R}^{d_{iv}} \rightarrow \mathbb{Z}^{\mathcal{D}_{hv}}$ is some encoding function that maps up a low-dimensional input vector into a high-dimensional vector of integers. There are several encoding algorithms with different memory-compute trade-offs, namely, permutation [22], base-level (a.k.a position-ID) [6], and random projection [26]. In both permutation and base-level encoding, the features (scalar values, e.g., image pixels) are represented by *level* hypervectors. To this end, a level hypervector $\mathcal{L}_1$ of length $\mathcal{D}_{hv}$ with bipolar (± 1) or binary coordinates is randomly generated and associated with the minimum value of features. Having Q distinct values for features (continuous data is also quantized into Q bins), each level hypervector $\mathcal{L}_k$ is generated by randomly flipping $\frac{\mathcal{D}_{hv}}{2Q}$ of vector $\mathcal{L}_{k-1}$. This transfers the similarity in low dimension to hyperspace, i.e., two close features in lower dimension also have high similarity (larger inner-product) in hyperspace, making $\mathcal{L}_1$ and $\mathcal{L}_Q$ have the highest distance, in expectation, $\mathcal{L}_1 \cdot \mathcal{L}_Q \simeq 0$.

To account for the temporal or spatial locality of features (e.g., the position of pixels in an image or arrival time of samples in time-series), permutation and base-level encoding follow different approaches. According to Equation (1), in the permutation encoding, for each feature v_i in an input, the proper level hypervector $\mathcal{L}(v_i)$ is selected. Then, to account the locality, $\mathcal{P}^{(i)}$ applies over $\mathcal{L}(v_i)$ and performs a circular shift of by $i-1$ indexes. The base-level encoding, shown in Equation (2), uses a different approach to account for positions. Per each feature position, it associates a *base* hypervector with the same dimensionality of the levels, i.e., $\mathcal{D}_{hv}$. The base hypervectors are random with bipolar (or binary) coordinates, so for $\mathcal{B}_i \cdot \mathcal{B}_j \simeq 0$ for $i \neq j$. All d_{iv} base hypervectors remain fixed for an HD model.

$$\mathcal{H}_p(V_{iv}) = \sum_{i=1}^{d_{iv}} \mathcal{P}^{(i)}(\mathcal{L}(v_i)) \tag{1}$$

$$\mathcal{H}_{bl}(V_{iv}) = \sum_{i=1}^{d_{iv}} \mathcal{B}_i \cdot \mathcal{L}(v_i) \tag{2}$$

Random projection, shown by Equation (3), also uses base hypervectors to maintain the locality information. However, instead of level hypervectors, it directly uses the scalar values of (quantized) input features.

$$\mathcal{H}_{rp}(V_{iv}) = \sum_{i=1}^{d_{iv}} \mathcal{B}_i \times v_i \tag{3}$$

(2) Training: For HD training, we first encode all input samples into hyperdimensional vectors, where $\mathcal{H}^\ell = \langle h_1, h_2, \cdots, h_{\mathcal{D}_{hv}} \rangle^\ell$ represents a hypervector of an input sample with label ℓ. Afterwards, we bundle all $\mathcal{H}^\ell$ (i.e., encoded hypervectors with the same label) together to create the representative class $\mathcal{C}^\ell$ for label ℓ as shown by Equation (4). V_i is the ith input, and y_i is its label.

$$\mathcal{C}^\ell = \sum_{i \text{ s.t. } y_i = \ell} \text{enc}(V_i) \tag{4}$$

(3) Inference: In inference, the query input V_q is first encoded to a hypervector $\mathcal{H}_q$ using the same encoding algorithm. Next, the similarity of $\mathcal{H}_q$ with all available class hypervectors is obtained using a similarity metric δ (e.g., cosine) represented by Equation (5). Similarity checking is also referred to as associative search.

$$\delta(\mathcal{H}_q, \mathcal{C}^\ell) = \frac{\mathcal{H} \cdot \mathcal{C}^\ell}{\|\mathcal{H}_q\|_2 \cdot \|\mathcal{C}^\ell\|_2} \tag{5}$$

III. Proposed tiny-HD

The goal of tiny-HD is to achieve a configurable HD platform by a dedicated hard-wired architecture. This entails algorithmic-hardware co-optimization to obtain a deterministic dataflow with simple control logic. In the following, we begin with devising an efficient encoding with a straightforward dataflow, by which we realize an efficient hardware architecture, and eventually calibrate it for maximal efficiency.

A. Encoding Algorithm

A major drawback of permutation (adopted by ASIC HD in [25]) and base-level encoding is the storage of level hypervectors. For a dimensionality of $\mathcal{D}_{hv}$=10,000 and having 10 bit features (2^{10} levels), HD needs a severe 10 Mbit of memory just to store the level hypervectors, which incurs a large cost and power. Some previous works such as the FPGA implementations in [16], [17] use larger quantization bins to limit the number of levels. This, however, degrades the accuracy, e.g., by up to 10% in time-series applications [7]. On the other hand, the random projection encoding (as well as base-level) needs base hypervectors (also called base hypervectors), one per feature. That is, for an application with as much as d_{iv}=1,000 features per input, we need 1,000 distinct bases, which also results in 10 Mbit memory.

To avoid the severe memory requirement of level hypervectors, in tiny-HD we leverage an enhanced random projection encoding which also eliminates the need for a large number of base hypervectors. Back in Equation (3), we can perceive that a dimension d of the encoded hypervector is obtained by summing up the same dimension of all base hypervectors multiplied to their associated feature, i.e., $\mathcal{H}_d = \sum_{i=1}^{d_{iv}} \mathcal{B}_{i,d} \cdot v_i$ where $\mathcal{B}_{i,d}$ indicates coordinate d of base $\mathcal{B}_i$. Therefore, we can formulate the random projection encoding as matrix-vector multiplication shown by Equation (6). $B_{\mathcal{D}_{hv} \times d_{iv}}$ is the projection matrix where each column i is the ith base hypervector.

$$\begin{bmatrix} H_1 \\ H_2 \\ H_3 \\ \vdots \\ H_{D_{hv}} \end{bmatrix}_{\boldsymbol{H}} = \begin{bmatrix} B_{1,1} & B_{2,1} & \cdots & B_{d_{iv},1} \\ B_{1,2} & B_{2,2} & \cdots & B_{d_{iv},2} \\ B_{1,3} & B_{2,3} & \cdots & B_{d_{iv}3} \\ \vdots & \vdots & \vdots & \vdots \\ B_{1,D_{hv}} & B_{2,D_{hv}} & \cdots & B_{d_{iv},D_{hv}} \end{bmatrix}_{\boldsymbol{B}} \times \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{d_{iv}} \end{bmatrix}_{\boldsymbol{V}} \quad (6)$$

Unlike the level hypervectors that need to keep the proximity of values, the only criteria for base hypervectors are randomness and orthogonality. If we shift/permute a randomly generated base hypervector, the resultant hypervector will be still orthogonal, i.e., $\mathcal{B}_i \cdot \mathcal{P}^{(k)}(\mathcal{B}_i) \simeq 0$ for $k \neq \mathcal{D}_{hv}$, where $\mathcal{P}^{(k)}$ is permutation by k indexes. Accordingly, we can generate the $\mathcal{B}_2$ from $\mathcal{B}_1$, and generally $\mathcal{B}_i$ from $\mathcal{B}_{i-1}$ by $k = 1$ permutation, i.e., $\mathcal{B}_i = \mathcal{P}^{(1)}(\mathcal{B}_{i-1})$. The new matrix $\mathcal{B}'$ is shown by Equation (7). Since all elements of matrix $\mathcal{B}'$ belong to base $\mathcal{B}_1$, we can simply denote $\mathcal{B}_{1,i}$s by $\mathcal{B}_i$ (as in Equation (8)). From storage perspective, $\mathcal{B}'$ compresses the base memory by a factor of d_{iv}, e.g., 10 Kbit instead of 10 Mbit.

$$\begin{bmatrix} H_1 \\ H_2 \\ H_3 \\ \vdots \\ H_{D_{hv}} \end{bmatrix}_{\boldsymbol{H}} \triangleq \begin{bmatrix} B_{1,1} & B_{1,2} & B_{1,3} & \cdots & B_{1,d_{iv}} \\ B_{1,2} & B_{1,3} & B_{1,4} & \cdots & B_{1,d_{iv}+1} \\ B_{1,3} & B_{1,4} & B_{1,5} & \cdots & B_{1,d_{iv}+2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ B_{1,D_{hv}} & B_{1,1} & B_{1,2} & \cdots & B_{1,d_{iv}-1} \end{bmatrix}_{\boldsymbol{B'}} \times \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{d_{iv}} \end{bmatrix}_{\boldsymbol{V}} \quad (7)$$

B. Encoding Architecture

The random projection encoding shown by Equation (7) requires $\mathcal{D}_{hv} \cdot d_{iv}$ add operations. Recall that $\mathcal{B}_i$s in the projection matrix are all bipolar (± 1), so a $\mathcal{B}_i \cdot v_j$ is just a selection between $+v_j$ and $-v_j$. To parallelize the operations, tiny-HD splits the operations into $r \times c$ windows, as shown by Equation (8). Multiplication of a $r \times c$ window by its corresponding c-element partial vector is done in one cycle. After that, the window slides horizontally by c columns to cover the next $r \times c$. When the window reaches the last column (in $\frac{d_{iv}}{c}$ cycles), r out of $\mathcal{D}_{hv}$ dimensions of the encoding hypervector is generated. We purposefully move the window horizontally as it helps to add up the partial sums of a specific coordinate without moving partial sum out and in.

$$\begin{bmatrix} B_1 & B_2 & \cdots & B_c & B_{c+1} & \cdots & B_{d_{iv}} \\ B_2 & B_3 & \cdots & B_{c+1} & B_{c+2} & \cdots & B_{d_{iv}+1} \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & B_{d_{iv}+2} \\ B_r & B_{r+1} & B_{r+2} & B_{r+c-1} & B_{r+c} & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ B_{D_{hv}} & B_1 & \cdots & \cdots & \cdots & \cdots & B_{d_{iv}-1} \end{bmatrix}_{\boldsymbol{B'}} \times \begin{bmatrix} v_1 \\ \vdots \\ v_c \\ v_{c+1} \\ \vdots \\ v_{2c} \\ \vdots \\ v_{d_{iv}} \end{bmatrix}_{\boldsymbol{V}} \quad (8)$$

With such a data flow, we can establish a datapath abstracted by Fig. 1. The key challenge is supporting the data flow of (8) within hard-wired connections. When processing a window, elements of each row are multiplied by the elements of the partial feature vector, which as alluded earlier, are realized by add operations since $\mathcal{B}_i$s are 1-bit bipolar. Therefore, we use r adders each of which has c input ports. In practice, the multi-port adders are implemented as a tree structure. Since all c elements of the feature vector are shared with (i.e., multiplied by) all rows of the window, in the encoding datapath, all fetched features ($v_1, \cdots, v_c$) are connected to all r adders. The projection bits ($\mathcal{B}_i$s) determine whether $+v_j$ or $-v_j$ should be accumulated.

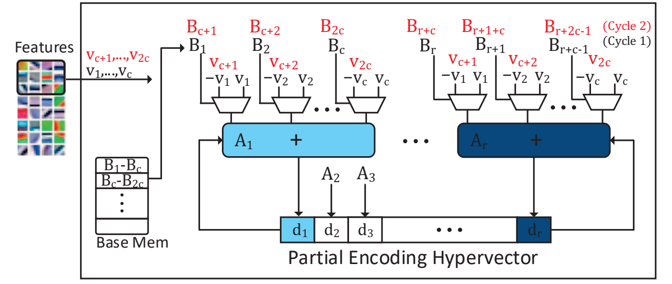

Fig. 1. Overview of the encoding datapath.

In Fig. 1, the feature and projection data for the first two cycles (cycle 1 in black, and cycle 2 by red color) are shown. We can observe that, at each cycle $k + 1$ we need to fetch c consecutive features: v_{1+kc} to v_{c+kc}. The ith fetched feature connects to the ith port of *all* the adders. Thus, we can hard-wire the output of feature memory to the adders' input ports. The controller, as a function of the current state, determines which c features should be read and activates the proper address port of the feature memory. Albeit the feature memory needs to deliver a bandwidth of $c \cdot w_v$ where w_v is the bit-width of input features. We will discuss the data layouts at the end of this section. The horizontal sliding of the window helps to store the partial sums until the encoding of r dimensions accomplishes. Thus, until a window reaches the last column, data of the encoding register is not moved.

Similar to the feature data, the base bits can be hard-wired to the adder module. At each cycle $k + 1$, we need $r + c - 1$ projection bits, more specifically, we need $\mathcal{B}_{1+ck}$ to $\mathcal{B}_{r+(k+1)c-1}$. The connection pattern can be perceived from Equation (8) and Fig. 1. Out of the $r + c - 1$ fetched base bits, the 1st to c^{th} bits connect, in order, to c multiplexers of adder A_1 (to select $\pm v_i$), the 2nd to $c + 1^{\text{th}}$ bases connect to adder A_2, and so on. Again, as a function of the current state, the controller determines fetching the proper base bits. We will show it can be simply enabled by a well-devised choice of the data striping and architecture parameters.

C. Associative Search Architecture

As explained above, a full horizontal window sliding takes $\frac{d_{iv}}{c}$ cycles and accomplishes r dimensions. Accordingly, while generating the next r encoding dimensions, tiny-HD concurrently computes the similarity of the already generated r dimensions with corresponding dimensions of the classes using dot-product. Since based on our setting $\frac{d_{iv}}{c} > n_C$ (n_C is number of classes), tiny-HD can compute the score with one

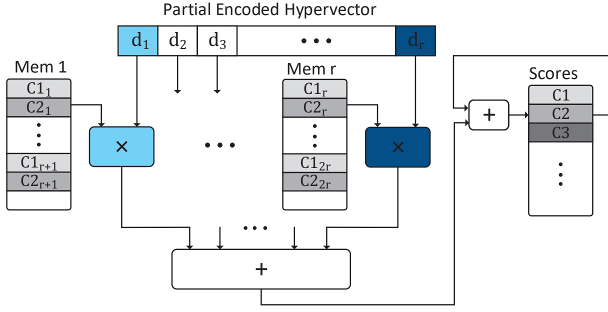

Fig. 2. Overview of the associative search (similarity checking) datapath. Ck_j indicates dimension j of kth class.

class at each cycle before the next r encoding dimensions are computed. Fig. 2 shows the associative search architecture. There are r memory buffers where 'Mem m' stores dimensions m, $m + r$, $m + 2r$, $\cdots$ of all classes by the illustrated data striping. Accordingly, the first cell of all buffers are allocated to elements 1st–r^{th} of class 1, their second cell is dedicated to class 2, and so on.

Using the implementation of Fig. 2, after the first r encoding dimensions are generated, we can read all r dimensions of class 1 in cycle 1, perform element-wise multiplication, and store the result in the score buffer. In the subsequent cycle, we repeat it for class 2 by reading its all first r elements from the second cell of all buffers. Therefore, the controller uses the same address for all buffers and increases by 1 at each cycle (when all n_C classes are visited, the counter halts until the next r dimensions are fully generated). The score buffer adds up the new partial similarity score of a class to the previously computed value. Evidently, tiny-HD efficiently shares the r multipliers between all classes. One of the input ports of a multiplier k is hard-wired to the encoding register d_k, and the other to logical buffer k, while the data layout and control flow enable supporting a different number of classes.

It is noteworthy that an alternative implementation could be storing each class in a separate buffer. However, it would use excessive resources as not only it requires a large bandwidth to read all r values from a buffer at once but also needs huge multiplexers in the multipliers ports so the right memory can access the encoding output. Besides, that would require using a large number of buffers to support applications with many classes. However, our proposed architecture supports an arbitrary number of classes for a fixed number of class buffers.

D. Architectural Details

Projection memory and adders: We aim to provide millisecond response even if the frequency is scaled to < 10 MHz. Therefore, we decide window sizes of 256 projection bits (see Equation (8)), i.e., $r \times c = 256$, thus, a whole inference including encoding and the pipelined similarity checking takes $\approx \frac{\mathcal{D}_{hv} \cdot d_{iv}}{256}$. Thus, for typical values of $\mathcal{D}_{hv} \simeq 4,000$ and $d_{iv} < 1000$, we can obtain millisecond inference in low frequencies.

For a given $r \times c$ operations per cycle, we prefer to minimize $r + c$ since tiny-HD reads $r + c - 1$ projection bits per cycle. Hence, we choose $r = c = 16$, thus at each cycle tiny-HD reads only 32 base bits. It also means tiny-HD uses sixteen 16-port adders, where each port has w_v bits. Accordingly, in the $(k+1)^{\text{th}}$ cycle, the index of the first base bit is $\mathcal{B}_{16k+1}$ and the last bit is $\mathcal{B}_{16k+31}$ (e.g., we read base bits 1–31 in cycle one, and 17–47 in cycle two). To realize that, we use a dual-read-port base buffer (alternatively, it can be implemented using two single-port buffers). With 16-bit data width, so we can read 32 bits with $16k + 1$ starting indexes. The projection buffer is only loaded once when the device starts up.

Feature memory: tiny-HD supports up to 1024-feature inputs, where each feature is (up to) eight bits. Continuous signals are quantized to 256 bins. As $c = 16$, we need to read 16 features per cycle. We use two dual-read buffers with 32-bit data width, so four features can be stored in a cell (row). Thus, by reading two cells of each buffer, we can read 16 features per cycle. Before starting the inference, features are loaded into the buffers with a layout similar to Fig. 2. That is, features 1–4 and 5–8 are loaded into the 1st and 2nd cells of the first feature buffer, features 9–12 and 13–16 are stored in the same cells of the second feature buffer, and so on. Hence, by reading two consecutive cells of each buffer, we can fetch 16 features in order. It makes both buffers share the same read addresses. Each buffer is implemented as $128 \times 32b$, hence, overall 1024 features can be stored. Note that as discussed earlier, the outputs of buffers are hard-wired to the encoding datapath adders thanks to the deterministic dataflow of tiny-HD.

Class memory and multipliers: Class and encoding dimensions are not necessarily binary as they are created by bundling of other hypervectors. During the initial HD training, we quantize the class dimensions into eight bits. Indeed, recent works have shown that HD retains the accuracy by even four bits [27], however, we use eight bits to account for more complex datasets. According to Fig. 2, we use $r = 16$ class buffers with 8-bit data width. Note that the physical layout could be also different from Fig. 2, e.g., memories 1–4 can be merged so each cell holds four successive dimensions of the same class. To support $\mathcal{D}_{hv} = 4,000$ for up to 32 classes (so we can support applications such as non-English voice and language detection, as well), each buffer should have a capacity of 64 Kb. We further segmentize each buffer into four 8 Kb buffers, so tiny-HD, thanks to its data layout, can power-gate the unused buffers in applications demanding fewer classes and/or dimensions. Note that the $\mathcal{D}_{hv} = 4,000$ limit is when there are 32 classes. As explained in Section III-C, for smaller number of classes, $\mathcal{D}_{hv}$ scales up proportionally.

IV. Experiments and Results

We implemented tiny-HD in SystemVerilog at the RTL level and used Synopsys Design Compiler with the 45 nm open-source NanGate cell library [28] to synthesize. To characterize the on-chip buffers, we used CACTI-P [29]. Under targeted area minimization constraint, tiny-HD achieved a clock period of 2.5 ns, i.e., 400 MHz. For a fair comparison with [25], we will scale the results to 22 nm by using 22 nm technology in CACTI for the memories area and power, and adopting the scaling trend of Intel [30] for the logic cells area. To scale the power, we first obtained the 45 nm power consumption reported by Synopsys Power Compiler, then, we obtained the scaling

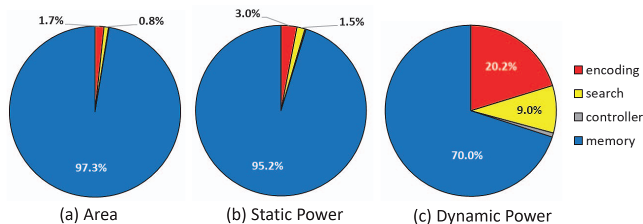

Fig. 3. Area and power breakdown of different components of tiny-HD.

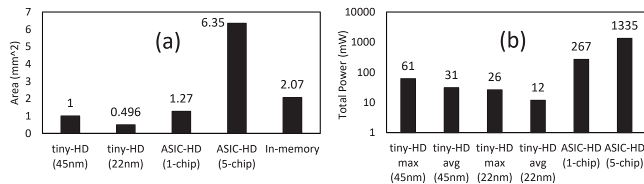

Fig. 4. Total area and power comparison of tiny-HD with state-of-the-art ASIC [25] and im-memory [21] HD designs.

trend using HSPICE simulations with Predictive Technology Model (PTM) [31] models. Our scaling factor ($\sim$3.5$\times$) concurs with [32].

A. Area

At the 45 nm technology node, tiny-HD occupies 1,004,642 μm^2 $\simeq$ 1.0 mm^2. The area breakdown is shown by Fig. 3(a). Memory modules contribute to 97.3% of the area, out of which 95.6% belongs to the 1 Mbit class buffers. Scaling the memory and logic to 22 nm technology shrinks tiny-HD area to 0.496 mm^2. Fig. 4(a) compares tiny-HD area with state-of-the-art programmable ASIC-HD [25] and in-memory [21] architectures. ASIC-HD [25] occupies 1.27 mm^2 (in TSMC 28 nm HKMG process) in the $\mathcal{D}_{hv}$ = 2000-dimension configuration, so their 5-chip configuration that supports 10K dimensions occupies 6.35 mm^2. tiny-HD, on the other hand, supports $\mathcal{D}_{hv}$ = 4000 dimensions for up to 32 classes, while, as explained in Section III-D, its data striping layout makes it flexible to scale up the dimensions linearly when an application has a smaller number of classes. In fact, the average number of classes for the benchmarks used in [25] is 10. For 10 classes, tiny-HD supports hypervectors of 12,800 dimensions which is larger than the 10K supported by [25]. Thus, compared to [25], tiny-HD reduces the area by 6.35$\times$ and 12.8$\times$ in the 45 nm and 22 nm nodes, respectively, while supporting larger dimensionality, on average. Compared to the PCM-based in-memory design of [21] (that occupies 2.07 mm^2 and supports 10K dimensions), tiny-HD improves the area by 2.07$\times$ and 4.17$\times$ in 45 nm and 22 nm processes, respectively.

B. Power

In the synthesized 45 nm node with 1.1 V supply voltage, tiny-HD consumes a *worst-case* 11.8 mW static power, and a *worst-case* 49.4 mW dynamic power running at 400 MHz. A significant portion of the power is dissipated by the memories (see Fig. 3(b)-(c)). In particular, class buffers consume respectively 93.2% and 65.5% of the total static and dynamic power. The reported powers, nevertheless, are pessimistic. As discussed in Section III-D, since class memories are large, each memory is implemented as four smaller buffers (multiplexed by the control unit using two address bits). Therefore, in applications with a smaller number of classes and/or small dimensionality, up to three out of four buffers are power-gated. Indeed, even using $\mathcal{D}_{hv}$ =10,000, on average, for the diverse datasets used in [25], only two active buffers per class memory is needed. Power-gating the unused buffers reduces the static power to 6.3 mW, and dynamic power to 33.2 mW. The dynamic power is still pessimistic as we assumed that

TABLE I
tiny-HD PERFORMANCE ON THE UTILIZED BENCHMARKS.

Application	Features	Classes	Accuracy	Cycle	Throughput (K)
CARDIO	21	3	93.46%	756	529
FACE	608	2	93.62%	9652	41
IoT	115	2	99.80%	2029	197
ISOLET	617	26	93.46%	9905	40
UCIHAR	561	12	95.50%	9141	44

the pipelined search (similarity checking) is performed every cycle. However, for an average of 10 classes [25], the search module is only active on 58% of the runtime. This reduces the expected dynamic power to 24.3 mW, making the expected runtime power of tiny-HD to be $\sim$31 mW.

According to Fig. 4(b), in the worst case, tiny-HD consumes 22$\times$ (51$\times$) less power than the previous ASIC work [25] in the 45 nm (22 nm) node. In the average case, tiny-HD improves the power by 44$\times$ (115$\times$) in the 45 nm (22 nm) node. We could not compare the power versus the in-memory implementation [21] as it only reports the energy.

C. Performance and Energy

tiny-HD exhibits a different performance than previous works which affects the relative energy consumption. Therefore, for a holistic comparison, we obtain the energy consumption of tiny-HD using a set of IoT benchmarks, including a speech recognition dataset (ISOLET), face detection (FACE), fetal state classification (CARDIO), human activity recognition (UCIHAR), and IoT attack detection (IoT), briefly introduced in [25]. Table I summarizes the tiny-HD performance using $\mathcal{D}_{hv}$ = 4000. The *cycle* column represents the latency in terms of clock cycle (2.5 ns) that tiny-HD processes an input. Accordingly, operating at 400 MHz, the *throughput* column indicates the number of inputs ($\times 10^3$) tiny-HD can process in one second. For instance, tiny-HD is able to process 40K input phonemes per second, which translates to $\sim$510K words per minute. In other words, with a frequency of 120 KHz, tiny-HD can still process 150 words per minute (average person speak rate) with trivial dynamic power (compared to the standby power).

Fig. 5 compares the per-query energy consumption (including encoding and associative search) of tiny-HD with the previous ASIC [25], FPGA [17], and in-memory [21] works for $\mathcal{D}_{hv}$ = 4000 in which the accuracy of all applications saturates. Therefore, we used 2-chip configuration of [25] to make it support 4,000 dimensions. [25] reports the power for 1-chip configuration, so we scaled it for 2-chip. Also, [21] has only reported an average energy number. Thus, since the latency increases roughly linearly with the number of features, we scaled its energy accordingly. We extracted the FPGA energy numbers directly from [17] (and scaled to 4,000 dimensions).

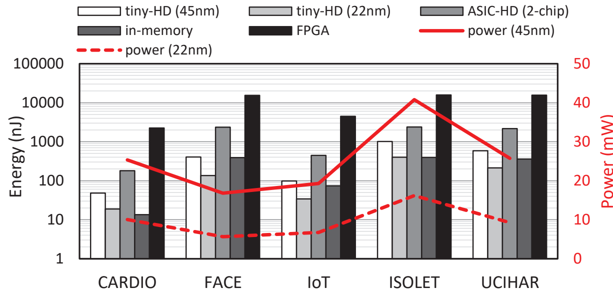

Fig. 5. Left axis (logarithmic): Energy-per-query comparison of tiny-HD with ASIC- [25] and in-memory HD [21]. Right axis: tiny-HD power consumption.

tiny-HD improves the per-query energy over the ASIC-HD [25] by 4.0× and 11.2×, respectively, on 45 nm and 22 nm technology. The 22 nm tiny-HD also improves the energy consumption by 1.7× compared as to the in-memory HD [21]. Compared to 45 nm tiny-HD, the in-memory implementation consumes 60% less energy. However, in-memory technology is immature and deep-nano FinFET remarkably surpasses state-of-the-art in-memory implementations [33]. E.g., ∼ 10× energy gain is expected migrating from High-k 45 nm to Multi-Gate 7 nm technology. Eventually, tiny-HD consumes 34.6× (95.5×) less energy than the state-of-the-art FPGA implementation [17] (implemented on Kintex-7 that is fabricated using TMSC's 28 nm high performance-low power process).

Fig. 5 also shows the power consumption of tiny-HD for different benchmarks, for both 45 nm and 22 nm technologies. The power consumption of applications depends on the number of features per input (d_{iv}) as well as the number of classes. For instance, ISOLET and FACE have a very close number of features, however, in ISOLET the search module is active in 67% of cycles, while in FACE (which has only two classes) the search is active in only 5% of the runtime. CARDIO, on the other hand, has three classes but has only 21 features per input. Therefore, after the second cycle (when $r = 16$ dimensions are produced), the encoding module halts for one cycle until all classes are compared, which imposes ∼33% performance (hence energy) inefficiency. The average power among five application is 9.6 mW (1.62 mW static and 7.94 mW dynamic).

V. Conclusion

We proposed tiny-HD, a low-power, energy-efficient, yet low-latency ASIC platform for hyperdimensional computing (HD). The intrinsic simplicity of HD together with our algorithmic innovation helped to reduce the memory requirement of tiny-HD and realizing a straightforward dataflow that makes tiny-HD configurable to support diverse applications by an efficient architecture. With an area of ∼0.5mm^2, tiny-HD outperforms the state-of-the-art FPGA, ASIC, and in-memory implementations of HD by 95.5×, 11.2×, and 1.7×, respectively. The latency of tiny-HD (0.025 *ms* in the slowest benchmark) is beyond the need of the majority of IoT applications.

Acknowledgements

This work was supported in part by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, in part by SRC Global Research Collaboration (GRC) grant, DARPA HyDDENN grant, and NSF grants #1730158, #1911095, and #2003279.

References

[1] U. S. Shanthamallu, A. Spanias *et al.*, "A brief survey of machine learning methods and their sensor and iot applications," in *2017 8th International Conference on Information, Intelligence, Systems & Applications (IISA)*. IEEE, 2017, pp. 1–8.
[2] M. Javaheripi, M. Samragh, T. Javidi, and F. Koushanfar, "Adans: Adaptive non-uniform sampling for automated design of compact dnns," *IEEE Journal of Selected Topics in Signal Processing*, vol. 14, no. 4, pp. 750–764, 2020.
[3] M. Javaheripi, M. Samragh, G. Fields *et al.*, "Cleann: Accelerated trojan shield for embedded neural networks," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.
[4] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive computation*, vol. 1, no. 2, pp. 139–159, 2009.
[5] A. Thomas, S. Dasgupta, and T. Rosing, "Theoretical foundations of hyperdimensional computing," *arXiv preprint arXiv:2010.07426*, 2020.
[6] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *International Conference on Rebooting Computing (ICRC)*. IEEE, 2017, pp. 1–8.
[7] F. Asgarinejad, A. Thomas, and T. Rosing, "Detection of epileptic seizures from surface eeg using hyperdimensional computing," in *International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, 2020, pp. 536–540.
[8] A. Mitrokhin, P. Sutor, C. Fermüller, and Y. Aloimonos, "Learning sensorimotor control with neuromorphic sensors: Toward hyperdimensional active perception," *Science Robotics*, vol. 4, no. 30, 2019.
[9] M. Imani, S. Bosch, M. Javaheripi *et al.*, "Semihd: Semi-supervised learning using hyperdimensional computing," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.
[10] M. Imani, Y. Kim, T. Worley, S. Gupta, and T. Rosing, "Hdcluster: An accurate clustering using brain-inspired high-dimensional computing," in *Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2019, pp. 1591–1594.
[11] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, "Exploring hyperdimensional associative memory," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 445–456.
[12] A. Cano, N. Matsumoto, E. Ping, and M. Imani, "Onlinehd: Robust, efficient, and single-pass online learning using hyperdimensional system," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021.
[13] B. Khaleghi, M. Imani, and T. Rosing, "Prive-hd: Privacy-preserved hyperdimensional computing," *arXiv preprint arXiv:2005.06716*, 2020.
[14] M. Imani, Y. Kim, S. Riazi, J. Messerly, P. Liu, F. Koushanfar *et al.*, "A framework for collaborative learning in secure high-dimensional space," in *12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 435–446.
[15] M. Imani, J. Morris, S. Bosch, H. Shu, G. De Micheli, and T. Rosing, "Adapthd: Adaptive efficient training for brain-inspired hyperdimensional computing," in *2019 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. IEEE, 2019, pp. 1–4.
[16] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, "F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 53–62.
[17] B. Khaleghi *et al.*, "Shearer: highly-efficient hyperdimensional computing by software-hardware enabled multifold approximation," in *ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 241–246.
[18] M. Imani *et al.*, "Revisiting hyperdimensional learning for fpga and low-power architectures," in *HPCA*. IEEE, 2021.
[19] S. Salamat, M. Imani, and T. Rosing, "Accelerating hyperdimensional computing on fpgas by exploiting computational reuse," *IEEE Transactions on Computers*, 2020.
[20] T. F. Wu, H. Li *et al.*, "Hyperdimensional computing exploiting carbon nanotube fets, resistive ram, and their monolithic 3d integration," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 11, pp. 3183–3196, 2018.
[21] G. Karunaratne, M. Le Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian, "In-memory hyperdimensional computing," *Nature Electronics*, pp. 1–11, 2020.
[22] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016, pp. 64–69.
[23] M. Imani *et al.*, "A binary learning framework for hyperdimensional computing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 126–131.
[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 *55th Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.
[25] S. Datta, R. A. Antonio, A. R. Ison, and J. M. Rabaey, "A programmable hyper-dimensional processor architecture for human-centric iot," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 439–452, 2019.
[26] M. Imani *et al.*, "Bric: Locality-based encoding for energy-efficient brain-inspired hyperdimensional computing," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.
[27] ——, "Quanthd: A quantization framework for hyperdimensional computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.
[28] Nangate open cell library. [Online]. Available: https://si2.org/open-cell-library/
[29] S. Li *et al.*, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2011, pp. 694–701.
[30] M. T. Bohr and I. A. Young, "Cmos scaling trends and beyond," *IEEE Micro*, vol. 37, no. 6, pp. 20–29, 2017.
[31] Predictive technology model. [Online]. Available: http://ptm.asu.edu/
[32] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm," *Integration*, vol. 58, pp. 74–81, 2017.
[33] X. Peng *et al.*, "Dnn+ neurosim: An end-to-end benchmarking framework for compute-in-memory accelerators with versatile device technologies," in *2019 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2019, pp. 32–5.