GENERIC: Highly Efficient Learning Engine on Edge using Hyperdimensional Computing

Behnam Khaleghi, Jaeyoung Kang, Hanyang Xu, Justin Morris, Tajana Rosing
Department of Computer Science and Engineering, UC San Diego, La Jolla, CA 92093
{bkhaleghi, j5kang, hax032, j1morris, tajana}@ucsd.edu

Abstract

Hyperdimensional Computing (HDC) mimics the brain's basic principles in performing cognitive tasks by encoding the data to high-dimensional vectors and employing non-complex learning techniques. Conventional processing platforms such as CPUs and GPUs are incapable of taking full advantage of the highly-parallel bit-level operations of HDC. On the other hand, existing HDC encoding techniques do not cover a broad range of applications to make a custom design plausible. In this paper, we first propose a novel encoding that achieves high accuracy for diverse applications. Thereafter, we leverage the proposed encoding and design a highly efficient and flexible ASIC accelerator, dubbed GENERIC, suited for the edge domain. GENERIC supports both classification (train and inference) and clustering for unsupervised learning on edge. Our design is flexible in the input size (hence it can run various applications) and hypervectors dimensionality, allowing it to trade off the accuracy and energy/performance on-demand. We augment GENERIC with application-opportunistic power-gating and voltage over-scaling (thanks to the notable error resiliency of HDC) for further energy reduction. GENERIC encoding improves the prediction accuracy over previous HDC and ML techniques by 3.5% and 6.5%, respectively. At 14 nm technology node, GENERIC occupies an area of 0.30 mm^2, and consumes 0.09 mW static and 1.97 mW active power. Compared to the previous inference-only accelerator, GENERIC reduces the energy consumption by 4.1×.

ACM Reference Format:
Behnam Khaleghi, Jaeyoung Kang, Hanyang Xu, Justin Morris, Tajana Rosing. 2022. GENERIC: Highly Efficient Learning Engine on Edge using Hyperdimensional Computing. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530669

1 Introduction

Hyperdimensional Computing (HDC) is a novel brain-inspired learning paradigm based on the observation that brains perform cognitive tasks by mapping sensory inputs to high-dimensional neural representation [1–3]. It enables the brain to carry out simple, low-power, error-resilient, and parallelizable operations all in the hyperspace. Such characteristics of HDC make it appealing for a wide variety of applications such as IoT domain that generates an increasing amount of data with tight resource and energy constraints [4, 5].

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07.
https://doi.org/10.1145/3489517.3530669

HDC uses specific algorithms to encode raw inputs to a high-dimensional representation of hypervectors with $\mathcal{D}_{hv} \approx 2-5K$ dimensions. The encoding takes place by deterministically associating each element of an input with a binary or bipolar (±1) hypervector and bundling (element-wise addition) the hypervectors of all elements to create the encoded hypervector. Training is straightforward and involves bundling all encoded hypervectors of the same category. For inference, the query input is encoded to a hypervector in the same fashion and compared with all class hypervectors using a simple similarity metric such as cosine.

The bit-level massively parallel operations of HDC do not accord well with conventional CPUs/GPUs due to, e.g., memory latency and data movement of large vectors and the fact that these devices are over-provisioned for majorly binary operations of HDC. Previous works on custom HDC accelerators support a limited range of applications or achieve low accuracy. The authors of [6] and [7] propose custom HDC inference designs that are limited to a specific application. More flexible HDC inference ASICs are proposed in [8] and [9], but as we quantify in Section 3.2, the utilized encoding techniques achieve poor accuracy for particular applications such as time-series. The authors of [10] propose a trainable HDC accelerator, which yields 9% lower accuracy than baseline ML algorithms. An HDC-tailored processor is proposed in [11], but it consumes ∼1–2 orders of magnitude more energy than ASIC counterparts. The in-memory HDC platform of [12] uses low-leakage PCM cells to store hypervectors, but its CMOS peripherals throttle the overall efficiency.

In this paper, we propose GENERIC (highly efficient learning engine on edge using hyperdimensional computing) for highly efficient and accurate trainable classification and clustering. Our primary goal is to make GENERIC compact and low-power to meet year-long battery-powered operation, yet fast enough during training and burst inference, e.g., when it serves as an IoT gateway. To this end, we make the following contributions.

(1) We propose a novel HDC encoding that yields high accuracy in various benchmarks. Such a generic encoding is fundamental to develop a custom yet flexible circuit.

(2) We perform a detailed comparison of HDC and various ML techniques on conventional devices and point out the failure of these devices in unleashing HDC advantages.

(3) We propose the GENERIC flexible architecture that implements accurate HDC-based trainable classification and clustering.

(4) GENERIC benefits from extreme energy reduction techniques such as application-opportunistic power gating, on-demand dimension reduction, and error-resilient voltage over-scaling.

(5) Comparison of GENERIC with the state-of-the-art HDC implementations reveals GENERIC improves the classification accuracy by 3.5% over previous HDC techniques and 6.5% over ML techniques. GENERIC improves energy consumption by 4.1× and 15.7× compared to previous HDC accelerators [8] and [10], respectively.

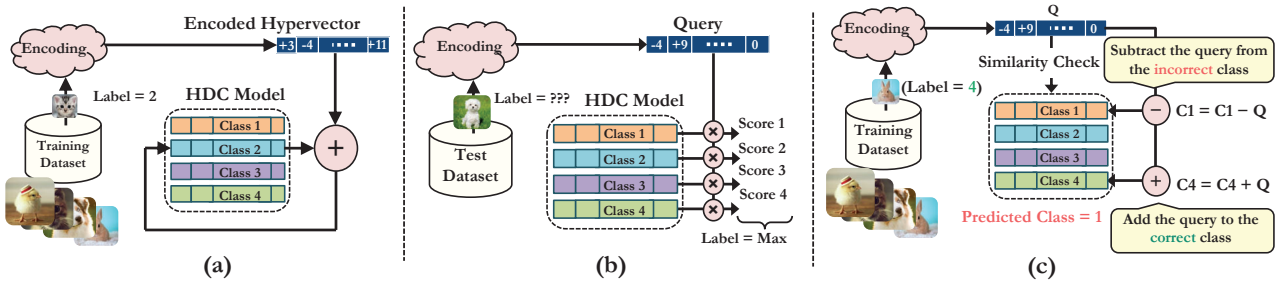

Figure 1: (a) HDC model training (initialization), (b) inference, and (c) retraining.

2 Hyperdimensional Computing

2.1 Learning with HDC

Figure 1 demonstrates the HDC training and inference. During training, each input $\mathcal{X}$ is encoded to a hypervector $\mathcal{H}(\mathcal{X})$ and added up to its class hypervector. In the inference, the query is likewise encoded and compared with class hypervectors. The class index with the highest similarity score is returned as the prediction result. We use cosine distance of the query and class hypervectors as the similarity metric. The accuracy of an HDC model can be improved by retraining iterations where the encoded train data are compared with the HDC model, and in case of misprediction, the model is updated by subtracting the encoded hypervector from the mispredicted class and adding it to the correct class.

The similarity of hypervectors indicates their proximity [1], which can be used to cluster data in the hyperspace [13]. Initially, k encoded hypervectors are selected as clusters centroids. At each iteration, all encoded inputs are compared with the centroids and added to the closest (highest score) centroid hypervector. In classification, the model is updated right away. However, in clustering, the model is fixed and used for finding the similarities, and a new model is created from scratch, which replaces the current model in the next iteration.

2.2 Encoding

Encoding is the major step of HDC; hence, previous works have proposed several encoding techniques to map the inputs to high-dimensional space. Most encodings associate hypervectors with the raw input features (elements), called *level* hypervector (see Figure 2(a)), which are hyperspace representative of scalar elements. Usually, inputs are quantized into bins to limit the number of levels. If there is a meaningful distance between the input elements (as in the values of white and black pixels), this distance is also preserved when generating the levels.

Encoding of an input is accomplished by aggregation the level hypervectors of its elements. To handle the positional order of elements, which is essential in most datasets such as image or voice, HDC uses variants of *binding*. The permutation encoding of Figure 2(b) carries out binding by circular shift of the level hypervectors; the level hypervector of m^{th} feature is permuted by m indexes. Some other encodings such as random projection (RP), shown in Figure 2(c), or level-id use *id* hypervectors for binding. In these encodings, each input index has a random (but constant) binary *id*, which is multiplied (XOR in the binary domain) with its level, and the result vector is aggregated with that of other indexes.

3 Proposed HDC Encoding

3.1 GENERIC Encoding

The encoding techniques discussed in Section 2.2 achieve low accuracy for certain datasets such as language identification which generally need extracting local subsequences of consecutive features, without considering the global order of these subsequences (see subsection 3.2). Previous studies use *ngram* encoding for such datasets [6, 7, 14]. Ngram encoding extracts all subsequences of length n (usually $n\in\{3-5\}$) in a given input, encodes all these subsequences and aggregates them to produce the encoded hypervector. However, ngram encoding achieves very low accuracy for datasets such as images or voices in which the spatio-temporal information of should be taken into account.

We propose a new encoding, dubbed GENERIC, to cover a more versatile set of applications. As shown in Figure 2(d), our encoding processes sliding windows of length n by applying the permutation encoding. That is, for every window consisting of elements $\{x_k, x_{k+1}, x_{k+2}\}$ (for n=3), three level hypervectors are selected, where $\ell(x_k)$, $\ell(x_{k+1})$, and $\ell(x_{k+2})$ are permuted by 0, 1, and 2 indexes, respectively. The permuted hypervectors are XORed element-wise to create the *window hypervector*. The permutation accounts for positional information within a window, e.g., to distinguish "abc" and "bca". To account for *global* order of features, we associate a random but constant *id* hypervector with each window, which is XORed with the window hypervector to perform *binding*. To skip the global binding in certain applications, *id* hypervectors are set to $\{0\}^{\mathcal{D}_{hv}}$. Equation (1) formalizes our encoding, where $\rho^{(j)}$ indicates permutation by j indexes, $\prod$ multiplies (XOR in binary) the levels of i^{th} window, id_i applies the binding id, and $\sum$ adds up the window hypervector for all windows of d elements.

$$\mathcal{H}(\mathcal{X}) = \sum_{i=1}^{d-n+1} \left(id_i \cdot \prod_{j=0}^{n-1} \rho^{(j)}\left(\ell(x_{i+j})\right)\right) \tag{1}$$

We use n=3 as it achieved the highest accuracy (on average) for our examined benchmarks (see subsection 3.2), however, GENERIC architecture can adjust the value of n for every application.

3.2 Accuracy Comparison

We compiled eleven datasets from different domains, consisting of the benchmarks described in [10], seizure detection by skull surface EEG signals, and user activity recognition by motion sensors (PAMAP2) [15]. We implemented the HDC algorithms using an optimized Python implementation that leverages SIMD operations. For ML techniques, we used Python scikit-learn library [16]. We discarded the results of logistic regression and k-nearest neighbors as they achieved lower accuracy. For DNN models of benchmarks, we used AutoKeras library [17] for automated model exploration.

Table 1 summarizes the accuracy results (RP: random projection, MLP: multi-layer perceptron, SVM: support vector machine, RF: random forest). The proposed GENERIC encoding achieves 3.5% higher accuracy than the best baseline HDC (level-id), 6.5% higher than best baseline ML (SVM), and 1.0% higher than DNN. The

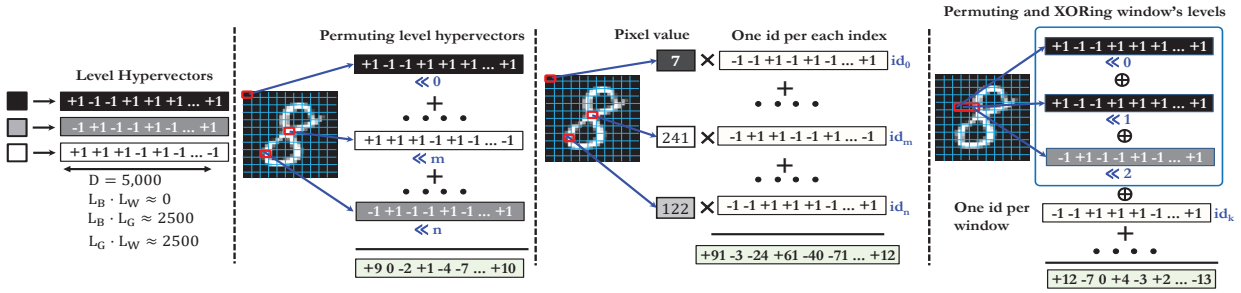

Figure 2: (a) Level hypervectors, (b) permutation encoding, (c) random projection encoding, (d) proposed GENERIC encoding.

Table 1: Accuracy of HDC and ML algorithms.

Dataset	HDC Algorithms					ML Algorithms			
	RP	level-id	ngram	permute	GENERIC	MLP	SVM	RF	DNN
CARDIO	83.0%	88.1%	88.1%	88.2%	**91.8%**	86.4%	86.4%	95.3%	90.1%
DNA	99.3%	99.3%	99.7%	99.3%	**99.7%**	99.5%	99.5%	99.5%	99.8%
EEG	46.8%	77.5%	83.1%	78.3%	**83.1%**	56.8%	75.4%	80.1%	60.2%
EMG	53.6%	90.9%	90.8%	91.1%	**90.9%**	91.0%	89.2%	83.6%	89.4%
FACE	95.3%	95.0%	73.3%	96.1%	**95.7%**	95.5%	97.3%	92.5%	96.7%
ISOLET	93.2%	93.5%	38.9%	93.5%	**93.1%**	95.0%	96.0%	92.2%	94.4%
LANG	8.2%	75.9%	100.0%	52.8%	**100.0%**	5.4%	30.8%	10.3%	99.9%
MNIST	94.6%	89.4%	53.0%	89.3%	**94.0%**	96.7%	97.9%	96.0%	99.1%
PAGE	96.1%	91.6%	91.7%	91.7%	**91.8%**	96.5%	96.9%	97.4%	95.8%
PAMAP2	83.0%	94.6%	60.9%	95.8%	**93.8%**	92.9%	91.9%	95.6%	96.1%
UCIHAR	93.4%	94.6%	64.9%	94.7%	**94.9%**	94.6%	95.8%	95.6%	96.5%
Mean	77.0%	90.0%	76.8%	88.3%	**93.5%**	82.8%	87.0%	85.3%	92.5%
STDV	27.5%	6.9%	19.2%	12.4%	**4.4%**	26.9%	19.0%	24.4%	10.8%

RP encoding fails in time-series datasets that require temporal information (e.g., EEG). As explained in subsection 3.1, the ngram encoding [6, 14] do not capture the global relation of the features, so it fails in datasets such as speech (ISOLET) and image recognition (MNIST). Except for the ngram and the proposed GENERIC, other HDC techniques fail in the LANG (text classification) as they enforce capturing sequential information and ignore subsequences.

3.3 Efficiency on Conventional Hardware

HDC's operations are simple and highly parallelizable, however, conventional processors are not optimized for binary operations such as one-bit accumulation. Also, the size of hypervectors in most settings becomes larger than the cache size of low-end edge processors, which may impose significant performance overhead. For a detailed comparison, we implemented the HDC and ML algorithms on the datasets of subsection 3.2 on a Raspberry Pi 3 embedded processor and NVIDIA Jetson TX2 low-power edge GPU, and also a desktop CPU (Intel Core i7-8700 at 3.2 GHz) with a larger cache. We used Hioki 3334 power meter to measure the power of the Raspberry Pi.

Figure 3 compares the training and inference (a) energy consumption and (b) execution time of the algorithms, reported as the geometric mean of all benchmarks (for eGPU, we omitted the results of conventional ML as it performed worse than CPU for a variety of libraries we examined). We can observe that (i) conventional ML algorithms, including DNN, unanimously consume smaller energy than HDC on all devices, (ii) GENERIC encoding, due to processing multiple hypervectors per window, is less efficient than other HDC techniques, and (iii) our eGPU implementation, by data packing (for parallel XOR) and memory reuse, significantly improves the HDC execution time and energy consumption. For instance, eGPU improves the energy usage and execution time of GENERIC inference by 134× and 252× over running on low-end Raspberry Pi (70× and 30× over CPU). However, GENERIC running on eGPU still consumes 12× (3×) more inference (train) energy,

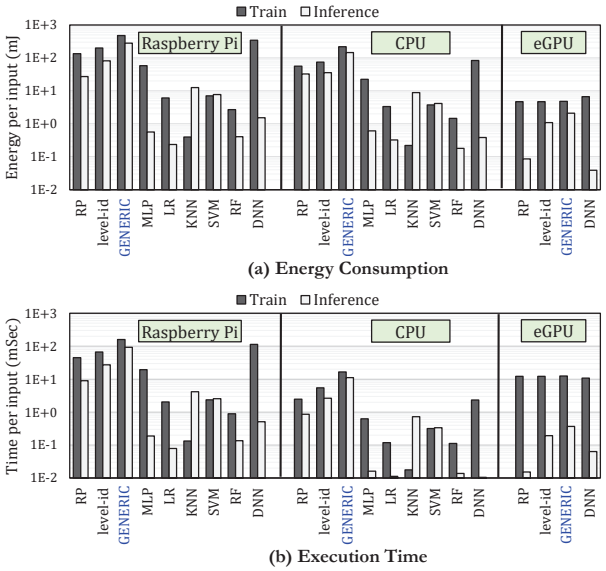

Figure 3: (a) Energy consumption and (b) execution time of HDC and ML algorithms on different devices.

with 27× (111×) higher execution time than the most efficient baseline (random forest). Nonetheless, eGPU numbers imply substantial energy and runtime reduction potential for HDC by effectively taking advantage of low-precision operations (achieved by bit-packing in eGPU) and high parallelism.

4 GENERIC Architecture

4.1 Overview

Figure 4 shows the main components of GENERIC architecture. The main inputs include (i) *input* port to read an input (including the *label* in case of training) from the serial interface element by element and store in the input memory before starting the encoding, (ii) *config* port to load the level, *id*, and class hypervectors (in case of offline training), and (iii) *spec* port to provide the application characteristics to the controller, such as $\mathcal{D}_{hv}$ dimensionality, d elements per input, n length of window, n_C number of classes or centroids, bw effective bit-width, and *mode* (training, inference, or clustering). *Output* port returns the labels of inference or clustering.

The controller, by using *spec* data, handles the programmability of GENERIC and orchestrates the operations. For instance, the encoder generates m=16 (architectural constant) partial dimensions after each iteration over the stored input, where the variable $\mathcal{D}_{hv}$ signals the end of encoding to finalize the search result, d denotes the number of input memory rows to be proceeded to fetch features (i.e., the exit condition for counter), n_C indicates the number

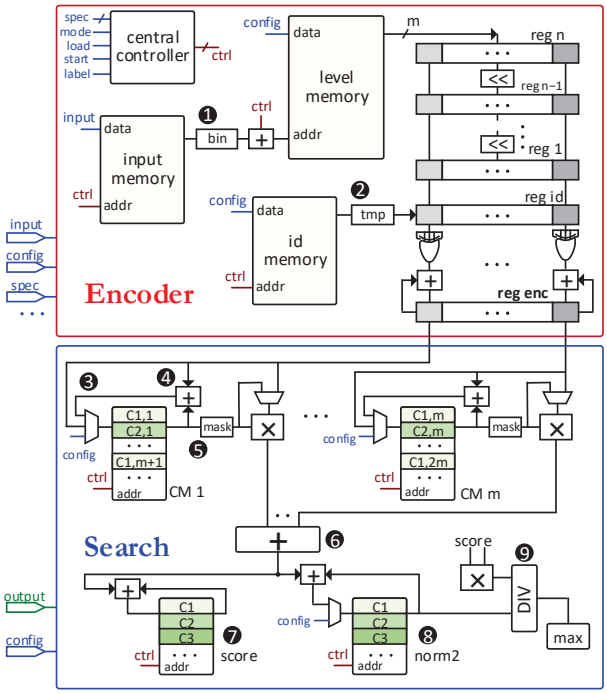

Figure 4: Overview of GENERIC architecture.

of class memory rows that need to be read for dot-product and so on. The class memory layout of GENERIC also allows trade off between the hypervectors length D_{hv} and supported classes n_C. By default, GENERIC class memories can store D_{hv}=4K for up to n_C=32 classes. For an application with less than 32 classes, higher number of dimensions can be used (e.g., 8K dimensions for 16 classes). We further discuss it in subsection 4.3. These application-specific input parameters enable GENERIC the flexibility to implement various applications without requiring a complex instruction set or reconfigurable logic.

4.2 Classification and Clustering

4.2.1 ***Encoding and Inference:*** Features are fetched one by one from the input memory and quantized to obtain the level bin, and accordingly, m (16) bits of the proper level hypervector are read. The levels are stored as m-bit rows in the level memory. The stacked registers (reg n to 1) facilitate storing and on-the-fly sliding of level hypervectors of a window. Each pass over the input features generates m encoding dimensions, which are used for dot-product with the classes. The class hypervectors are distributed into m memories (CM 1 to CM m) to enable reading m consecutive dimensions at once. The dot-product of partial encoding with each class is summed up in the pipelined adder ❻, and accumulated with the dot-product result of previous/next m dimensions in the score memory ❼.

After $\frac{\mathcal{D}_{hv}}{m}$ iterations, all dimensions are generated, and the dot-product scores are finalized. We use cosine similarity metric between the encoding vector $\mathcal{H}$ and class C_i: $\delta_i = \frac{\mathcal{H}\cdot C_i}{\|\mathcal{H}\|_2 \times \|C_i\|_2}$; hence, we need to normalize the dot-product result with L2 norms. The $\|\mathcal{H}\|_2$ can be removed from the denominator as it is a constant and does not affect the rank of classes. In addition, to eliminate the square root of $\|C_i\|_2$, we modify the metric to $\delta_i = \frac{(\mathcal{H}\cdot C_i)^2}{\|C_i\|_2^2}$ without affecting the predictions. The norm2 memory of Figure 4 ❽ stores the *squared* L2 norms of classes, and similarly, the *squared* score is passed to the divider ❾. We use an approximate log-based division [18].

4.2.2 ***Training and Retraining:*** In the first round of training, i.e., model initialization, encoded inputs of the same class/label are accumulated. It is done through the adder ❹ and mux ❸ of all class memories. The controller uses the input label and the iteration counter to activate the proper memory row. In the next retraining epochs, the model is examined and updated in case of misprediction (see Figure 1). Thus, during retraining, meanwhile performing inference on the training data, the encoded hypervector is stored in temporary rows of the class memories (through the second input of mux ❸). If updating a class is required, the class rows are read and latched in the adder ❹, followed by reading the corresponding encoded dimensions from the temporary rows and writing the new class dimensions back to the memory. Hence, each update takes $3\times\frac{\mathcal{D}_{hv}}{m}$ cycles. Training also requires calculating the squared L2 norm of classes in the norm2 memory ❽. As it can be seen in Figure 4, the class memories are able to pass the output into both ports of the multipliers (one direct and another through the mux) to calculate and then accumulate the squared elements.

4.2.3 ***Clustering:*** GENERIC selects the first k encoded inputs as the initial cluster centroids and initializes k centroids in the class memories. It allocates two sets of memory rows for temporary data; one for the incoming encoding generated in the encoding module and another for the copy centroids (as mentioned in Section 2.1, clustering generates a new copy instead of direct update). Similarity checking of the encoding dimensions with the centroids is done pipelined similar to inference, but the encoded dimensions are stored to be added to the copy centroid after finalizing the similarity checking. After finding the most similar centroid, the copy centroid is updated by adding the stored hypervector (similar to retraining). The copy centroids serve as the new centroids in the next epoch.

4.3 Energy Reduction

We take advantage of the properties of GENERIC architecture and HDC for utmost energy efficiency. The following elaborates energy-saving techniques that benefit GENERIC. These techniques can also be applied to other HDC accelerators.

4.3.1 ***id Memory Compression:*** The *id* memory naturally needs 1K×4K=512 KB (for up to to 1K features per input, and $\mathcal{D}_{hv}$=4K dimensions) which occupies a large area and consumes huge power. However, GENERIC generates *id*s on-the-fly using a seed *id* vector, where k^{th} *id* is generated by permuting the seed *id* by k indexes. Therefore, the *id* memory shrinks to 4 Kbit, i.e., 1024× reduction. Permutation preserves the orthogonality. It is implemented by the *tmp* register in Figure 4 ❷, by which, for a new window, the reg id is right-shifted and one bit of *tmp* is shifted in. The *tmp* register helps to avoid frequent access to the *id* memory by reading m (16) bits at once and feeding in the next m cycles.

4.3.2 ***Application-opportunistic Power Gating:*** For an application with n_C classes and using $\mathcal{D}_{hv}$ dimensions, GENERIC stripes the dimensions 1 to m (16) of its 1st class vector in the 1st row of m class memories, the 2nd class vector in the 2nd row, and so on (see Figure 4). The next m dimensions of the 1st class vector are therefore written into $n_C + 1^{\text{th}}$ row, followed by the other classes. Thus, GENERIC always uses the first $\frac{n_C\times\mathcal{D}_{hv}}{32\times4K}$ portion of class memories.

The applications of Section 3.2, on average, fill 28% of the class memories (minimum 6% for EEG/FACE, and maximum 81% for ISOLET) using $\mathcal{D}_{hv}$=4K dimensions. Accordingly, GENERIC partitions

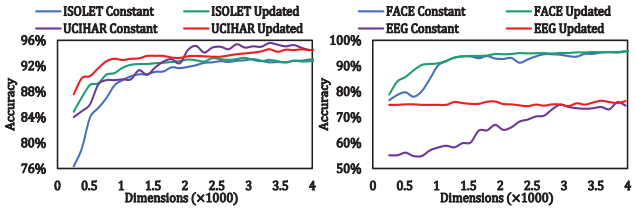

Figure 5: Accuracy with constant and updated L2 norm.

each class memory into four banks and power gates the unused banks. With four banks, 1.6 out of four banks are activated on average, leading to 59% power saving. With more fine-grained eight banks, 2.7 banks (out of eight) become active, saving 66% power. However, eight banks impose 55% area overhead compared to 20% of four banks (see Section 5.1 for setup). We concluded that the four-bank configuration yields the minimum area×power cost. Since the power gating is static (permanent) for an application, no wake-up latency or energy is involved.

4.3.3 ***On-demand Dimension Reduction:*** GENERIC can trade the energy consumption and performance with accuracy. Recall that GENERIC generates m dimensions of the encoding per iteration over the features. By feeding a new $\mathcal{D}_{hv}$ value as input, GENERIC can seamlessly use the new dimension count by updating the counter exit condition, so smaller hypervectors of the encoding and class hypervectors will be used. Nevertheless, GENERIC stores the squared L2 norms of the whole classes for similarity metric ($\delta_i = \frac{(\mathcal{H}\cdot C_i)^2}{\|C_i\|_2^2}$) while for arbitrary reduced encoding dimensions, only the corresponding elements (and their L2 norms) of the classes are needed. As Figure 5 shows, using the old (*Constant*) L2 values causes significant accuracy loss compared to using the recomputed (*Updated*) L2 norm of sub-hypervectors. The difference is up to 20.1% for EEG and 8.5% for ISOLET. To address this issue, when calculating the squared L2 norms during the training, GENERIC stores the L2 norms of every 128$^{\text{th}}$-dimension sub-class in a different row of the norm2 memory ❽.Thus, dimensions can be reduced with a granularity of 128 while keeping the norm2 memory small (2 KB for 32 classes).

4.3.4 ***Voltage Over-scaling:*** GENERIC has to use 16-bit class dimensions to support training. As a result, the large class memories consume ~80% of the total power. HDC exhibits notable tolerance to the bit-flip of vectors [19], which can be leveraged to over-scale the memory voltage without performance loss. Figure 6 shows the accuracy of select benchmarks (ISOLET and FACE) with respect to the class memory error. The static (*s*) and dynamic (*dyn*) power saving as a result of corresponding voltage scaling (without reducing clock cycle) is also shown in the right axis (based on the measured data of [20]). The figure shows the result of the HDC models with different bit-width (*bw* input parameter of GENERIC) of classes by loading a quantized HDC model (the mask unit ❺ in the architecture masks out the unused bits). As it can be seen, error tolerance not only depends on application but also on the bit-width. 1-bit FACE model shows a high degree of error tolerance (hence, power saving) by up to 7% bit-flip error rate, while ISOLET provides acceptable accuracy by up to 4% bit-flip using a 4-bit model. Quantized elements also reduce the dynamic power of dot-product.

Voltage over-scaling also depends on the application's sensitivity to dimension reduction and its workload. For instance, FACE has a higher tolerance to voltage scaling than dimension reduction (see Figure 5). On the other hand, ISOLET is more sensitive to voltage reduction but achieves good accuracy down to 1K dimensions (Figure 5), which means 4× energy reduction compared to 4K dimensions.

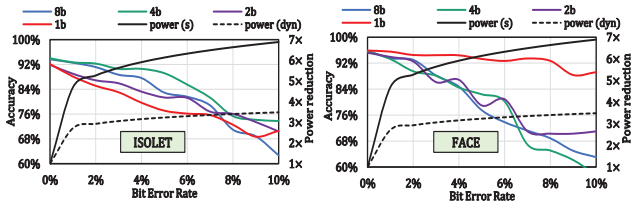

Figure 6: Accuracy and power reduction wrt memory error.

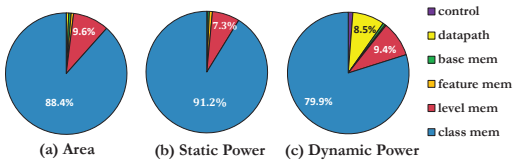

Figure 7: Accuracy and power reduction wrt memory error.

Thus, voltage over-scaling for ISOLET is only preferred in workloads with a higher idle time where the static power dominates (voltage scaling reduces the static power more significantly).

5 Results

5.1 Setup

We implemented GENERIC at the RTL level in SystemVerilog and verified the functionality in Modelsim. We used Synopsys Design Compiler to synthesize GENERIC targeting 500 MHz clock with 14 nm Standard Cell Library of GlobalFoundries. We used Artisan memory compiler to generate the SRAM memories. The level memory has a total size of 64×4K = 32KB for 64 bins, the feature memory is 1024×8b, and class memories are 8K×16b (16 KB each). We obtained the power consumption using Synopsys Power Compiler. GENERIC occupies an area of 0.30 mm^2 and consumes a *worst-case* static power of 0.25 mW when all memory banks are active. For datasets of Section 3.2, GENERIC consumes a static and dynamic power of 0.09 mW, and 1.79 mW, respectively (without voltage scaling). Figure 7 shows the area and power breakdown. Note that the level memory contributes to less than 10% of area and power. Hence, using more levels does not considerably affect the area or power.

5.2 Classification Evaluation

5.2.1 ***Training:*** Since previous HDC ASICs have not reported training energy and performance, we compare the per-input energy and execution time of GENERIC training with RF (random forest, most efficient baseline) and SVM (most accurate conventional ML) on CPU, and DNN and HDC on eGPU. Figure 8 shows the average energy and execution time for the datasets of Section 3.2. GENERIC improves the energy consumption by 528× over RF, 1257× over DNN, and 694× over HDC on eGPU (which, as discussed in Section 3.3, is the most efficient baseline device for HDC). GENERIC consumes an average 2.06 mW of training power. It also has 11× faster train time than DNN and 3.7× than HDC on eGPU. RF has 12× smaller train time than GENERIC, but as we mentioned, the overall energy consumption of GENERIC is significantly (528×) smaller than RF. Also, we used constant 20 epochs for GENERIC training while the accuracy of most datasets saturates after a few epochs.

5.2.2 ***Inference:*** We compare the energy consumption of GENERIC inference with previous HDC platforms from Datta et al. [10], and tiny-HD [8]. We scale their report numbers to 14 nm according to [21] for a fair comparison. We also include the RF (most efficient ML), SVM (most-accurate ML) and DNN on HDC on eGPU (most-efficient HDC baseline). Figure 9 compares the energy consumption

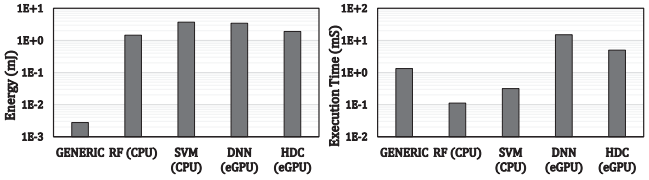

Figure 8: Training energy and execution time.

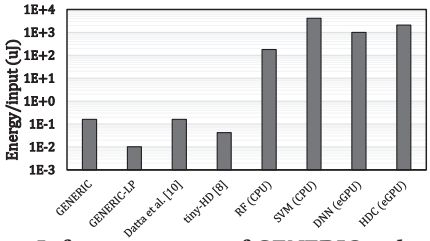

Figure 9: Inference energy of GENERIC vs baselines.

Table 2: Mutual information score of K-means and HDC.

	Hepta	Tetra	TwoDiamonds	WingNut	Iris
K-means	1.0	0.637	1.0	0.774	0.758
HDC	0.904	0.589	0.981	0.781	0.760

of GENERIC and aforementioned baselines. Since GENERIC achieves significantly higher accuracy than previous work (e.g., 10.3% over [10]), GENERIC–LP applies the low-power techniques of Section 4.3 to leverage this accuracy benefit. GENERIC–LP improves the baseline GENERIC energy by 15.5× through dimension reduction and voltage over-scaling. GENERIC–LP consumes 15.7× and 4.1× less energy compared to [10] and tiny-HD [8], respectively. Note that despite tiny-HD [8], GENERIC supports training which makes it to use larger memories. GENERIC is is 1593× and 8796× more energy-efficient than the most-efficient ML (RF) and eGPU-HDC, respectively.

5.3 Clustering Evaluation

Table 2 compares the normalized mutual information score of the K-means and HDC for the FCPS [22] benchmarks and the Iris flower dataset. On average, K-means achieves slightly (0.031) higher score, but for datasets with more features, the proposed GENERIC can better benefit from using windows (windows become less effective in a smaller number of features).

Figure 10 compares the per-input energy consumption of GENERIC with K-means clustering running on CPU and Raspberry Pi. GENERIC consumes only 0.068 μJ per input, which is 17,523× and 61,400× more efficient than K-means on Raspberry Pi and CPU. The average per-input execution time of Raspberry Pi and CPU is, respectively, 394 μSec and 248 μSec, while GENERIC achieves 9.6 μSec (41× and 26× faster than R-Pi and CPU, respectively).

6 Conclusion

We proposed GENERIC, a highly-efficient HDC accelerator that supports classification (inference and training) and clustering using a novel encoding technique that achieves 3.5% (6.5%) better accuracy compared to other HDC (ML) algorithms. GENERIC benefits from power-gating, voltage over-scaling, and dimension reduction for utmost energy saving. Our results showed that GENERIC improves the classification energy by 15.1× over a previous trainable HDC accelerator, and 4.1× over an inference-only accelerator. GENERIC HDC-based clustering consumes 17,523× lower energy with 41× higher performance than Raspberry Pi running K-means with similar accuracy, facilitating ultra-efficient continuous learning on edge.

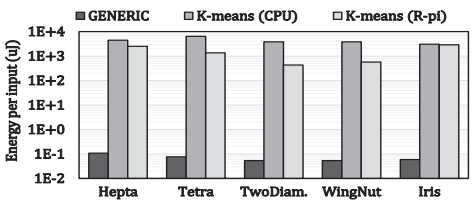

Figure 10: GENERIC and K-means energy comparison.

Acknowledgements

This work was supported in part by CRISP, one of six centers in JUMP (an SRC program sponsored by DARPA), SRC Global Research Collaboration (GRC) grant, and NSF grants #1911095, #1826967, #2100237, and #2112167. We would like to thank Amin Kalantar and Onat Gungor for helping in Raspberry Pi experiments.

References

[1] A. Thomas, S. Dasgupta, and T. Rosing, "Theoretical foundations of hyperdimensional computing," *Journal of Artificial Intelligence Research*, vol. 72, pp. 215–249, 2021.
[2] L. Ge and K. K. Parhi, "Classification using hyperdimensional computing: A review," *IEEE Circuits and Systems Magazine*, vol. 20, no. 2, pp. 30–47, 2020.
[3] 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.
[4] R. Aitken, V. Chandra, J. Myers, B. Sandhu, L. Shifren, and G. Yeric, "Device and technology implications of the internet of things," in *2014 symposium on VLSI technology (VLSI-technology): digest of technical papers*, pp. 1–4, IEEE, 2014.
[5] X. Yu, X. Song, L. Cherkasova, and T. Š. Rosing, "Reliability-driven deployment in energy-harvesting sensor networks," in *2020 16th International Conference on Network and Service Management (CNSM)*, pp. 1–9, IEEE, 2020.
[6] A. Rahimi, P. Kanerva, *et al.*, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *International Symposium on Low Power Electronics and Design*, pp. 64–69, 2016.
[7] A. Moin, A. Zhou, A. Rahimi, A. Menon, S. Benatti, G. Alexandrov, S. Tamakloe, *et al.*, "A wearable biosensing system with in-sensor adaptive machine learning for hand gesture recognition," *Nature Electronics*, vol. 4, no. 1, pp. 54–63, 2021.
[8] B. Khaleghi, H. Xu, J. Morris, and T. Š. Rosing, "tiny-hd: Ultra-efficient hyperdimensional computing engine for iot applications," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 408–413, IEEE, 2021.
[9] M. Eggimann, A. Rahimi, and L. Benini, "A 5μ w standard cell memory-based configurable hyperdimensional computing accelerator for always-on smart sensing," *arXiv preprint arXiv:2102.02758*, 2021.
[10] S. Datta *et al.*, "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.
[11] 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)*, pp. 1–6, IEEE, 2018.
[12] G. Karunaratne, M. Le Gallo, G. Cherubini, L. Benini, *et al.*, "In-memory hyperdimensional computing," *Nature Electronics*, pp. 1–11, 2020.
[13] M. Imani, Y. Kim, *et al.*, "Hdcluster: An accurate clustering using brain-inspired high-dimensional computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1591–1594, IEEE, 2019.
[14] P. Alonso *et al.*, "Hyperembed: Tradeoffs between resources and performance in nlp tasks with hyperdimensional computing enabled embedding of n-gram statistics," in *International Joint Conference on Neural Networks*, IEEE, 2021.
[15] "Uci machine learning repository." https://archive.ics.uci.edu/ml/datasets/.
[16] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.
[17] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1946–1956, 2019.
[18] J. N. Mitchell, "Computer multiplication and division using binary logarithms," *IRE Transactions on Electronic Computers*, no. 4, pp. 512–517, 1962.
[19] M. Imani, A. Rahimi, D. Kong, T. Rosing, *et al.*, "Exploring hyperdimensional associative memory," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 445–456, IEEE, 2017.
[20] L. Yang and B. Murmann, "Sram voltage scaling for energy-efficient convolutional neural networks," in *International Symposium on Quality Electronic Design (ISQED)*, pp. 7–12, IEEE, 2017.
[21] 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.
[22] A. Ultsch, "Clustering with som: U^* c," in *Proceedings of the workshop on self-organizing maps, 2005*, 2005.