A Programmable Hyper-Dimensional Processor Architecture for Human-Centric IoT

Sohum Datta, *Student Member, IEEE*, Ryan A. G. Antonio, *Student Member, IEEE*, Aldrin R. S. Ison, *Student Member, IEEE*, and Jan M. Rabaey, *Fellow, IEEE*

***Abstract*—Hyper-dimensional Computing (HDC), a bio-inspired paradigm defined on random high-dimensional vectors, has emerged as a promising IoT paradigm. It is known to provide competitive accuracy on sequential prediction tasks with much smaller model size and training time compared to conventional ML, and is well-suited for human-centric IoT. In the post-Moore scaling era, where increasing variability has challenged traditional designers, its novel computing method based on randomness can be leveraged for continued performance. This work develops a complete, programmable architecture for ultra energy-efficient supervised classification using HD computing. Its simple construction follows from basic HD operations and its massively parallel, shallow datapath ($<$ 10 logic layers) resembles in-memory computing. The architecture also supports scalability: multiple such processors can be connected pralallely to increase effective HD dimension. A broad evaluation is performed by comparing HDC and 3 conventional ML algorithms on conventional architectures such as CPU and eGPU for instruction count, energy cost and memory requirements. Finally, a 2048-dim ASIC design is synthesized in a 28nm HK/MG process and benchmarked on 9 supervised classification tasks with varying complexity (such as language recognition and human face detection). The simulated chip exhibits energy efficiency $< 1.5\mu$J/pred. for the entire benchmark at about 2.5ns cycle time, with most applications requiring $<$ 700 nJ/pred. As a first complete design working with high dimensional stochastic signals, the main architectural decisions for similar systems harnessing variability in emerging devices (eg. CNFET and RRAM) are established. A fabricated system could be readily deployed for human-centric IoT applications.**

***Index Terms*—Brain-inspired computing, hyper-dimensional computing, holographic reduced representations, energy efficiency, Internet-of-Things, human-centric computing, body sensing (alternatively, body sensor networks).**

I. Introduction

TWO crucial events in the last decade determine the pace of technical innovation today. The first is the gradual slowdown in relentless miniaturization of semiconductor devices, known as Moore's law [1]. Its current phase of *equivalent scaling* of transistors (i.e. using novel gate materials and geometries) is likely to give way to *hyper-scaling* (functionality-aware beyond-Boltzmann transistors) after 2025 [2]. And as transistor dimensions approach 10nm, variability and reliability effects begin to dominate its deterministic behavior [3]. For continued miniaturization, new avenues of research into materials, semiconductor physics and organic chemistry for emerging devices have materialized [4].

Secondly, the rise of data-driven learning algorithms have completely changed the way businesses function [5]. Due to the widespread proliferation of sensory devices and improvements in connectivity, the huge amounts of data gathered must be processed for ensuring quality of services. Mobile devices (e.g. smartphones, tablets, sensor-nodes in sensor networks) function under limited bandwidth, battery and storage capacity, thereby requiring high energy efficiency in their computations [6].

Clearly, one way forward is to perform machine learning on emerging post-Moore devices with much lower energy footprints. This is especially useful for edge-based Internet-of-Things (IoT), where data is partially processed immediately after collection to reduce bandwidth usage and server workload. Emerging devices allow such computations to meet the strict energy constraints required. However, adapting emerging devices to the exact-computing paradigm is difficult due to their inherent variability [7]. As energy efficiency no longer scales with integration capacity, voltage reduction and near-threshold operation reduces power consumption at the expense of favorable signal-to-noise ratio (SNR) [8]. Finally, for conventional architectures such as CPUs and General-Purpose GPUs (GPGPUs), few applications today (including data mining and classification) have enough parallelism to completely utilize available hardware [9].

While challenges of using unreliable components have long been known [10], biology offers the most concrete inspiration. Consequently, **brain-inspired computing** could provide the required robustness and scalability for continued inprovements.

Hyper-Dimensional Computing (HDC) is one such nano-scalable paradigm [11], and is known to excel in body-based sensing/IoT applications [12]. It originated from a theoretical model of cognitive reasoning [13], [14] and is motivated by the fact that brains compute by transforming activation patterns of a *large population* of neurons. Hence, tolerance to variability is inherent: changes in activation of a few neurons do not affect the overall functionality. Its energy efficiency and robustness to noise (introduced by

Manuscript received April 15, 2019; revised July 9, 2019 and August 9, 2019; accepted August 10, 2019. Date of publication August 15, 2019; date of current version September 17, 2019. This work was sponsored by E2CDA-NRI, a funded center of NRI, through the SRC program co-sponsored by NSF. This article was recommended by Guest Editor K. Kailas. *(Corresponding author: Sohum Datta.)*

S. Datta and J. M. Rabaey are with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94704 USA (e-mail: sohumdatta@berkeley.edu).

R. A. G. Antonio and A. R. S. Ison are with the Electrical and Electronics Engineering Institute (EEEI), University of the Philippines Diliman, Quezon City 1101, Philippines.

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JETCAS.2019.2935464

TABLE I

THE HDC BENCHMARK SUITE: 9 SUPERVISED CLASSIFICATION TASKS WITH VARYING COMPLEXITY WERE CHOSEN TO EVALUATE THE GENERIC HD PROCESSOR (D = 2048). THE MULTI-PROCESSOR CONFIGURATION INCREASES EFFECTIVE DIMENSION AND IMPROVES ACCURACY SIGNIFICANTLY (FIG. 7(A)). (*)-MEMBERS ARE WELL SUITED FOR HUMAN-CENTRIC COMPUTING IN INTERNET OF THINGS (IoT) [35]

Applications	**Abbrev.**	**Encoding**	**HDC**	**Known State-of-the-Art Algorithm**
Language Recognition	LANG	4-gram	90.6 %	97.1 %, n-gram-based Nearest Neighbors [16]
EMG Hand-Gesture Recognition*	EMG	2-stage	**95.8** %	89.7 %, Support Vector Machine [27]
DNA Sequencing*	DNA	60-features	**96.2** %	93.7 %, knowledge-based Neural Networks [28]
Fetal State Classification (cardio.)*	CARDIO	21-features	**90.6** %	90.6 %, Support Vector Machine [29]
Page-block Classification	PAGE	10-features	**91.6** %	85.9 %, min-max Hyperplane Separation [30]
UCI Human activity Recognition*	UCIHAR	561-features	76.7 %	89.3 %, Support Vector Machines [31]
Spoken Letter Classification	ISOLET	617-features	75.9 %	97.1 %, boosted k-Nearest neighbors [32]
Human Face Detection*	FACE	608-features	66.0 %	96.1 %, HOG-based boosted Decision Trees [33]
MNIST Digit Classification	MNIST	784-features	75.4 %	99.7 %, Deep Convolution Neural Network [34]

reduced VDD) in the datapath was demonstrated for Language Recognition [15], [16] and tested on fabricated systems based on emerging devices: a hybrid of carbon nanotube field-effect transistors (CNFETs) and resistive RAM (RRAM) memory in [17], and a CMOS/vertical-RRAM (VRRAM) implementation in [18]. [18] also demonstrated the robustness of HDC to inherent RRAM variability in endurance cycles and wafer-level device-to-device characteristics.

However, both [18] and [17] are hard-wired to solve only Language Recognition on a specific dataset, and cannot be used for other datasets and applications. They also have low data-width (32 bits), requiring large amount of time-multiplexing to simulate the complete machine of full width ($>$ 1000 bits). Datapaths specific to applications other than Language Recognition have also been proposed [19], but a general HD system is yet to be developed ([11] presents a brief outline).

A **general** HDC architecture, which can be easily programmed to perform a variety of applications on different datasets, is crucial for evaluating its potential as a viable IoT paradigm [12]. This requires developing the fundamental architectural blocks that are configured and interconnected to produce a complete system. A comprehensive exploration of the above is the main goal of this work.

II. HYPER-DIMENSIONAL COMPUTING

HD computing defines random high-dimensional vectors ($D > 1000$) as its fundamental data type. It is a **holographic** computing framework: unlike arithmetic over numbers, no vector component contains more information than any other. Although vectors with elements from any algebraic field can be used (see Table I of [11]), we will consider only binary vectors as it results in the simplest hardware.

To compare vectors, a distance metric is required. *Hamming distance* (denoted by $d_H(a, b)$) is the number of dissimilar elements between vectors a and b. Two binary vectors x and y of dimension D are said to be orthogonal if $d_H(x, y) = D/2$. This definition is more familiar in bipolar code (0-valued elements replaced by integer -1): orthogonal x and y have zero inner product, $\langle x, y\rangle = 0$.

The underlying principle of HDC is **almost certain orthogonality** in high-dimensional spaces. For a rigorous demonstration, note that if x and y are chosen independently and uniformly from $\{0, 1\}^D$ (i.e. probability of any

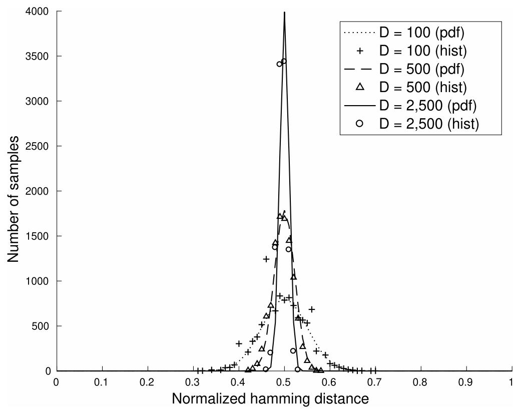


Fig. 1. **Orthogonality in High Dimensions**: Note the sharper concentration around 0.5 as D increases.

bit being 1 is $p = 1/2$), their hamming distance is binomially distributed: $d_H(x, y) \sim Bin(D, p = 1/2)$. Fig. 1 plots a histogram (hist) of $d_H(x, y)$ normalized by dimension D for 10,000 randomly-generated pairs (x, y). It also plots the density function (pdf) for Normal Approximation $N(Dp, Dp(1-p))$ of $Bin(D, p)$ scaled to have an area equalling sample size 10,000. The Normal Approximation helps in plotting and is very accurate for high dimensions: using the Berry-Essen bound (Theorem 10.4 in [20]) for $X \sim Bin(D, p)$, $Y \sim N(Dp, Dp(1-p))$, $p = 1/2$ and dimension $D \geq 1024$, the maximum error in cumulative distribution ($max_{0<t<1}|\Pr(X \leq t) - \Pr(Y \leq t)|$) is 0.025.

Then, it can be shown that (Theorem 1 of [21]):

$$\Pr\left[\left|\frac{d_H(x, y)}{D} - \frac{1}{2}\right| \geq \epsilon\right] < 2e^{-2D\epsilon^2} \tag{1}$$

Only high dimensions ($D > 1000$) result in a meaningful right-hand side in Eq. 1 [22]. Then random vectors x and y have normalized distance very close to 0.5. The *exponential* drop in probability beyond ϵ-deviation from the mean is the crucial property exploited here.

A. The Binary HDC Subset

All high-dimensional binary vectors used in a given computation will be called **hyper-vectors**. When the context is obvious, hyper-vectors and vectors will be used interchangeably.

In addition to high dimensionality, a set of operations are required that preserve near orthogonality. Although there are many sets of operations with equivalent performance, (see Table I of [11]), the **Multiply-Add-Permute (MAP)** framework is most suitable as it maps directly to logic gates. These **encoding operations** allow representation of complex structures such as sequences, lists and trees [14], [23], and are fundamental to the paradigm:

- **Multiplication/Binding** is useful for forming *associations* among related vectors. X and Y are bound together to form $C = X \oplus Y$ orthogonal to both its constituents. It is implemented by element-wise XOR.
- **Addition/Superposition** is the primary *conjunctive* operation. Based on Hebbian learning [23], the goal is to find a vector z representing the set of operand hyper-vectors $\{x_1, x_2, \ldots x_n\}$. It is denoted by $z = [x_1 + x_2 + \ldots + x_n]$ and implemented by performing vector sum of operands and **thresholding** each element at the mean (0 for bipolar code).
- **Permutation** is a unary operation such that the permuted vector (denoted by $\rho(x)$) is nearly orthogonal to a randomly-generated hyper-vector x. Any cyclic permutation which does not have fixed points is a valid candidate. While all candidates result in the same performance, we use circular shift as it is easiest to implement in hardware. Note that the chosen candidate is a constant operation to be applied everywhere (i.e. all applications and datasets). Hyper-vector x permuted n times is denoted as $\rho^n(x)$.

As Fig. 1 shows, it is very rare for random hyper-vectors to deviate much from orthogonality. The addition operation above generates non-orthogonal vectors from random operand vectors. This allows us to encode meaning, as significant deviation from orthogonality implies common membership or dependency. Therefore, an **associative search** to find the closest match of the MAP-encoded hyper-vector to stored *class hyper-vectors* is also a crucial operation.

B. Example: Language Recognition

Language recognition from text is an ideal example for illustrating HDC in supervised classification, as the state-of-the-art algorithm can be directly mapped to this framework. A corpus of 21 Indo-European languages transliterated to English forms the training set, and new sentences are queried as tests [16].

1) Baseline: n-Gram Character Model: In state-of-the-art algorithms, a language is modeled as a probability distribution on character sequence of length n (also called n-grams). More sophisticated models such as dictionary of words, phrases, etc., increase complexity with negligible gains [24], [25].

While training a language, raw n-gram frequency counts are generated from a large corpus and iteratively smoothened [26] to remove outlier artifacts. The resulting n-gram distribution is the trained language model. The steps are repeated for a test query, and the trained model with the closest distribution is the language prediction.

2) HDC Setup and Algorithm: To make use of HD computing, the first step is to map random vectors to meaningful entities. In this case, characters from the Latin alphabet are assigned to uniformly generated hyper-vectors of dimension $D = 10,000$. The HD algorithm uses them to encode the training data and generate a single hyper-vector for each language.

A direct equivalent of frequency counting is the superposition of hyper-vectors representing each occurring n-gram in the text. Permutation and Multiplication are used to generate an n-gram vector from constituent letter hyper-vectors. For example, "abc" is encoded as $V_{abc} \triangleq \rho^2(V_a) \oplus \rho(V_b) \oplus V_c$, where V_z is hyper-vector representing symbol z. Due to properties of HD operations (Section II-A), all n-gram and character hyper-vectors are nearly orthogonal to each other.

The test hyper-vector is computed similarly, and the language with closest hyper-vector is returned as the prediction. Since the superimposed language vector is in the linear space spanned by the set of all n-gram vectors, the class with the closest n-gram distribution from baseline equivalently has the smallest distance in HDC.

HDC has an accuracy of *96.7 %* against a baseline of *97.1 %* [16] for $n = 4, D = 10000$. However, it is an online algorithm requiring a **single iteration** though the dataset. The deviations from orthogonality (Fig. 1) during encoding operations automatically smoothen the superimposed multi-set. Finally, the HD model size (1 vector/class) is fixed with n-gram size, but **grows exponentially** in the baseline. Indeed, for $n = 4$, the HDC model is **20× smaller** than the baseline model.

III. Benchmark Applications

HDC, also known as Holographic Reduced Representations (HRR), has been applied to a variety of problems such as simulation of finite automata, logical and analogical reasoning [23]. In the simplest HDC algorithms, random vectors are combined according to an application-specific expression of MAP operations **in a single pass** through the dataset. This approach is viable for expert systems (as in Sec. II-B), where the generating process of the data is already known. For simple applications with unknown generating models, perceptron-based re-training of the closest known expert model can improve HDC accuracy [36]. During iteration, if the validation data-point with encoded hyper-vector v of correct class with vector $C_{correct}$ results in a wrong prediction (class with hyper-vector C_{wrong}), we update them as: $C_{wrong} \leftarrow C_{wrong} - v,\ C_{correct} \leftarrow C_{correct} + v$.

HDC is known to work very well for sequence prediction problems [37], especially when the generating process is Markov of finite order with a known upper bound [38], [39]. For finite-order Markov time series, where the label of the current time instant depends only on a few instants of the recent past (unlike the more general arbitrary dependence on the entire past), a **single-pass HDC** algorithm is equivalent in performance to the optimal classifier (mixed-order Markov Model) [38]. Such problems are routinely encountered in sensor-based IoT applications, such as body-based sensing and industrial fault isolation [40]. Body-sensing is an especially attractive application, as the benefits of wearable electronics and HDC can be integrated into a common platform for seamless processing of medical data [41], human-computer interaction, and human-centric IoT [27], [35]. Consequently, such applications have been extensively studied for HDC (see [12] for a summary). Single-pass HDC suffices compared to conventional ML techniques that require much more resources with negligible accuracy gains: in EEG-based medical applications, multi-layer Neural Networks (NN) require 5-13× more memory than HDC [42], Long-Short Term Memory (LSTM) and Convolutional Neural Network (CNN) require 1400× and 50× more operations [43] than HDC respectively to achieve similar accuracy on the same dataset.

For an arbitrary prediction problem, it is unknown whether HDC can achieve optimal performance, especially relative to known ML algorithms. Although it is too early to ascertain, we feel it is unlikely that HDC alone could tackle harder problems that have a richer structure than finite-order Markov processes. For such problems, a hybrid system of HDC and ML/deep-learning algorithm is perhaps superior to either approach separately. Some recent work has been done in this direction, such as using a hybrid HDC-NN system for storage and inference on knowledge graphs [44], a HDC back-end in the vision engine for active perception in robots [45], and training multiple NNs simultaneously in a single model using near orthogonality of Sec. II [46].

To evaluate the designed architecture, a set of applications must be chosen to faithfully represent the state of the art. The following 9 applications were finally chosen as the benchmark. Members which are well suited for processing on ultra energy-efficient sensor nodes in human-centric IoT [35] are *-marked.

Language Recognition (LANG) is described in Section II-B [16]. EMG Hand-Gesture Recognition (EMG) classifies 64-channel electromyography signals recorded from a subject's hand into a set of hand-gestures [27]. DNA Sequencing (DNA) predicts the presence of Exon/Intron or Intron/Exon boundaries in a strand of DNA [28]. Fetal State classification (CARDIO) uses measurements of heart-rate and uterine pressure during pregnancy to classify fetal condition before delivery [29]. Page-block classification (PAGE) finds all blocks of the page layout in a document that has been detected by a segmentation process [30]. UCI Human-activity Recognition (UCIHAR) classifies recordings of 30 subjects performing activities of daily living while carrying a waist-mounted smartphone with embedded inertial sensors [31]. Spoken Letter Classification (ISOLET) predicts the English letter spoken from voice recordings of subjects. Face Detection (FACE) determines whether a human face is present within a given picture frame [33]. MNIST Digit Recognition (MNIST) classifies the digit from images of drawn digits [34].

Table I compares the accuracy of single-pass HDC with the best ML models for each benchmark dataset from the literature. **Bold** indicates better or equal accuracy over the best-known ML algorithm. The list is non-exhaustive but contains representative datasets from human-centric IoT summarized in [12]. It is also **balanced** overall: MNIST, FACE and ISOLET represent the fact that known HDC algorithms alone applied on raw features cannot compare to ML for even simple speech and vision problems.

IV. Profiling on Conventional Architectures

Before proposing a specialized architecture tailored to HDC, it is prudent to look at its performance on conventional architectures with standard compilers and compare against conventional ML algorithms. In this section, a **broad evaluation** of the costs of doing HDC on a CPU and embedded GPU (eGPU) is performed for 3 supervised classification tasks, and the results are compared with conventional ML approaches for the same.

While [12] compares HDC and conventional ML algorithms on body-sensing applications alone, experiments here are conducted on popular datasets of the ML algorithm to study the effectiveness of CPU and eGPU architectures for more general tasks. It is shown that although compilers cannot discover the underlying parallelism present in HDC in general, HDC still produces more efficient code than ML occasionally. Finally, a comparison across platforms reveals that an ASIC implementation offers orders-of-magnitude improvements in classification costs, with energy/inference of few μJ necessary for deployment in sensor-based IoT.

A. CPU Profiling of HDC

LANG, MNIST and EMG was profiled on an ARM Cortex A57 onboard a NVIDIA Jetson TX2 embedded platform [47]. The main goal here is to gain a **preliminary comparison** of energy expended per prediction, number of cycles, instructions executed, number of loads/stores and memory footprint (maximum size of pages allocated).

1) Code Setup: For LANG, HD algorithm and the compared k-Nearest Neighbors (kNN) base-line were written in C [16]. The text-histograms for kNN was generated using standard hash-map [48], also written in C.

For EMG, the HD algorithm was written in C as described in [49], and the corresponding Support Vector Machine (SVM) algorithm implemented using `LIBSVM` library [50].

For MNIST, feature-superposition (Section V-B) is used for the 28×28 gray-scale frames. Since the pixel values are bimodal, its values are thresholded first to a 0 or 1. The Convolutional Neural Network compared for MNIST has 2 convolutional layers with 32 3×3 kernels and ReLU activation, followed by 2×2 max-pooling layer and finally a 128-node dense layer (total $\approx 0.5 \times 10^6$ parameters, code modified from [51]). This network was used as it was much smaller but with similar accuracy to cutting-edge MNIST classifier (99.3% vs. 99.7% [34]). However, as shown in Table II, HDC ($D = 2048$) is more resource efficient that even the simple CNN profiled. Clearly, HDC would be much more resource efficient than the far larger cutting-edge MNIST classifier [34].

All HDC algorithms for CPU were written in C where hyper-vectors are represented as 32-bit integer arrays. The associative search through class vectors was implemented as nested for-loops, where LANG, MNIST used hamming distance and EMG used cosine-similarity as in [49].

TABLE II

CPU PROFILING RESULTS: ROWS 1 - 9 COMPARE 2048-DIM HDC WITH KNN, SVM AND CNN. BOLD NUMBERS ARE RATIOS OF METRICS FOR THE NON-HDC ALGORITHM TO 2048-DIM HDC; LARGER THAN 1 INDICATE HDC IS BETTER. ROWS 10 - 15 COMPARE $D = 10000$ AGAINST $D = 2048$; RATIO LARGER THAN 1 INDICATE $D = 2048$ IS MORE EFFICIENT

#	Application+Algo.	Instructs. Executed	Loads & Stores	No. of Cycles	Total Time	Max. Pages	Avg. Power	Total E/pred.	CPU E/pred.	DDR E/pred.
1	LANG+kNN	35.5×10^9	24.7×10^9	100×10^9	49.7 s	54.0 MB	2.84 W	672 mJ	414 mJ	258 mJ
2	LANG+HD(2k)	12.0×10^9	6.93×10^9	8.46×10^9	4.20 s	3.62 MB	1.82 W	36.4 mJ	23.4 mJ	13.0 mJ
3	**kNN/HD(2k)**	**2.96**	**3.56**	**11.8**	**11.8**	**14.9**	**1.56**	**18.4**	**17.7**	**19.8**
4	EMG+SVM	4.84×10^9	1.89×10^9	4.95×10^9	2.46 s	18.1 MB	1.71 W	0.376 mJ	0.228 mJ	0.148 mJ
5	EMG+HD(2k)	13.5×10^9	5.94×10^9	7.48×10^9	3.72 s	2.90 MB	1.85 W	0.617 mJ	0.407 mJ	0.210 mJ
6	**SVM/HD(2k)**	**0.359**	**0.318**	**0.662**	**0.661**	**6.24**	**0.922**	**0.610**	**0.560**	**0.705**
7	MNIST+CNN	389×10^9	129×10^9	261×10^9	129 s	2.99 MB	2.14 W	27.7 mJ	17.8 mJ	9.83 mJ
8	MNIST+HD(2k)	46.7×10^9	28.18×10^9	46.7×10^9	23.2 s	41.5 MB	2.07 W	4.80 mJ	2.97 mJ	1.83 mJ
9	**CNN/HD(2k)**	**8.33**	**4.58**	**5.59**	**5.58**	**0.072**	**1.03**	**5.76**	**6.01**	**5.36**
10	LANG+HD(10k)	58.6×10^9	33.8×10^9	41.3×10^9	20.5 s	12.0 MB	2.00 W	195 mJ	125 mJ	70.2 mJ
11	**LANG+HD(10k/2k)**	**4.88**	**4.88**	**4.88**	**4.88**	**3.32**	**1.09**	**5.36**	**5.34**	**5.40**
12	EMG+HD(10k)	70.1×10^9	28.9×10^9	38.1×10^9	18.9 s	3.50 MB	2.04 W	3.46 mJ	2.25 mJ	1.20 mJ
13	**EMG+HD(10k/2k)**	**5.19**	**4.87**	**5.09**	**5.08**	**1.21**	**1.10**	**5.60**	**5.54**	**5.72**
14	MNIST+HD(10k)	376×10^9	131×10^9	219×10^9	109 s	66.4 MB	2.21 W	24 mJ	14.9 mJ	9.1 mJ
15	**MNIST+HD(10k/2k)**	**8.05**	**4.65**	**4.69**	**4.69**	**1.60**	**1.06**	**5.01**	**5.02**	**4.99**

TABLE III

PERFORMANCE ON GPU OF HDC & NON-HDC: ARM CORTEX A57 (H) AND NVIDIA 256-CORE PASCAL (G) PERFORMANCE METRICS ARE DISPLAYED. HD DIMENSION IS $D = 2048$. BOLD NUMBERS ARE RATIO OF NON-HDC TO HDC; LARGER THAN 1 INDICATE HDC IS BETTER

App.+Algo.	Time H→ G	Time G→ H	Bytes H→ G	Bytes G → H	Instructions Executed	L2 Load Transactions	L2 Store Transactions	Global Load Transactions	Global Store Transactions
LANG+kNN	5.38 s	7.21 ms	12 GB	12 KB	368×10^6	581×10^6	185×10^6	736×10^6	184×10^6
LANG+HDC	3.50 ms	8.50 ms	6.1 MB	24 MB	2.60×10^6	3.70×10^5	1.15×10^6	1.16×10^6	1.13×10^6
kNN/HDC	**1540**	**0.848**	**1970**	**0.005**	**142**	**1570**	**161**	**634**	**163**
EMG+SVM	65.0 μs	9.50 μs	62 KB	23 KB	2.29×10^6	8.26×10^5	8.70×10^4	1.14×10^6	8.71×10^5
EMG+HDC	12.8 ms	2.7 μs	37 MB	2 KB	9.67×10^6	2.51×10^6	1.19×10^6	3.62×10^6	0.998×10^6
SVM/HDC	**0.005**	**3.52**	**0.002**	**11.5**	**0.237**	**0.329**	**0.073**	**0.315**	**0.873**
MNIST+CNN	8.02 ms	2.34 ms	16 MB	40 KB	259×10^7	698×10^6	87.4×10^6	122×10^7	86.6×10^6
MNIST+HDC	45.8 ms	68.9 ms	46 MB	172 MB	20.8×10^6	3.20×10^6	7.92×10^6	7.88×10^6	7.78×10^6
CNN/HDC	**0.175**	**0.034**	**0.348**	**0.00023**	**125**	**218**	**11**	**156**	**11.1**

2) Profiling Method: NVIDIA Jetson TX2 has an on-board power monitor `INA226` [52] used for energy and power measurements. The monitor measures power drawn from separate supply-lines of CPU and DDR as well. For measuring the maximum OS page-memory, *Valgrind's* `massif` utility was used to track all pages allocated to the process by system calls rather just for dynamic heap. Finally cycles, instructions, and number of loads and stores were profiled by sampling PMU registers through UNIX's `perf` utility. Since it is sampling-based, interference by OS code or other processes during the total elapsed time can lead to variations in measurements. Multiple runs were performed to reduce empirical variance in measurement.

Some observations from the results in Table II:

- $D = 2048$ is about 5-times more efficient than $D = 10000$. This is expected as most instructions, loads and stores and execution time scale linearly with dimension.
- kNN and MNIST are inefficient compared to HDC ($D = 2048$), even though hyper-vectors in C used `int32` to store 1 bit. kNN is especially inefficient as it needs to compare histograms with $28^4 = 5.31 \times 10^5$ dimensions versus 2048 for HDC.
- SVM is uniformly better than HDC ($D = 2048$). The lack of bit-level instructions for the compiler along with lower dimensions of support-vectors (64 dimensions for SVM vs. 2048 for HDC) explains this effect.

This clearly demonstrates promise for HDC: a better bit-level mapping and more efficient instruction sequences will make HDC better than all of these applications. Also note that kNN, CNN and SVM require multiple iterations while training (against a single-pass for HDC). Therefore, their energy consumption for training will be far higher than HDC. A more detailed study in the future could include dynamic binary instrumentation from source [53], effects of bit-instructions and optimal assembly-code on HDC performance, and profiling training with a reasonable setting of hyperparameters for kNN, CNN, SVM.

B. GPU Profiling for HDC

NVIDIA Jetson TX2 houses an embedded General-Purpose GPU (GPGPU) tailored for ultra-low-power ML applications [47]. The same steps as Section IV-A were repeated on the Arm Cortex A57 CPU - GPU system onboard Jetson TX2. We also measured memory transfers between the CPU host ("H" in Table III) and GPU Device ("G" in Table III).

1) Code Setup: HDC code in Section IV-A were re-written in TensorFlow [54]. APIs for HDC were developed, including

TABLE IV
ENERGY PROFILING FOR GPU IMPLEMENTATION OF HDC VS. NON-HDC

App.+Algo.	Total Power	Total Time	Total E/pred.	CPU E/pred.	DDR E/pred.	GPU E/pred.
LANG+kNN	1.87 W	31.2 s	927 mJ	404 mJ	425 mJ	98.1 mJ
LANG+HDC	2.10 W	8.66 s	289 mJ	153 mJ	115 mJ	21.0 mJ
kNN/HDC	**0.891**	**3.6**	**3.21**	**2.65**	**3.69**	**4.66**
EMG+SVM	1.41 W	0.863 s	5.32 mJ	2.49 mJ	2.25 mJ	0.583 mJ
EMG+HDC	2.05 W	7.4 s	65.1 mJ	33.4 mJ	26.3 mJ	5.39 mJ
SVM/HDC	**0.686**	**0.082**	**0.702**	**0.074**	**0.086**	**0.108**
MNIST+CNN	1.59 W	19.8 s	31.5 mJ	13.4 mJ	13.4 mJ	4.65 mJ
MNIST+HDC	2.70 W	97.5 s	263 mJ	148 mJ	99.5 mJ	14.9 mJ
CNN/HDC	**0.590**	**0.203**	**0.120**	**0.0903**	**0.135**	**0.312**

creation of data-*tensors* and execution graphs for computing dependant tensors for the final accuracy. For LANG, *k-Nearest Neighbors* (kNN) was implemented with hashing supported by *Python*'s standard dictionaries. For EMG, the *Support Vector Machine* (SVM) algorithm was implemented using `ThunderSVM` library [55]. For MNIST, the same CNN as in CPU was implemented in TensorFlow. HDC tensors use `int32` to store hyper-vectors.

2) Profiling Method: NVIDIA Jetson TX2's on-board power monitor `INA226` can access measure GPU supply-lines as well [52]. For the remaining metrics, NVIDIA's proprietary `nvprof` utility was used. The *summary* log was used for reporting total time spent in copying memory from CPU to GPU or vice-versa. The *API-trace* log was used to calculate the total data (in Bytes) transferred between CPU and GPU. Finally, all remaining metrics were profiled by `nvprof` by re-running Kernels.

Unlike in Section IV-A, the comparison results of HDC with non-HDC when using GPU is not uniform (see Tables III, IV). kNN doesn't have enough DLP to efficiently utilize GPU parallelism. It transfers huge amount of data from CPU to GPU before the computation can begin, thereby executing a large number of loads and instructions on the GPU. Hence, it is highly inefficient and expensive.

SVM shows similar results on GPU as on CPU. A downside of using `float32` to store bit-elements is the use of complex ALUs. Using bit-operations supported on GPU could greatly improve HDC performance.

Finally, CNN is about 10X more efficient than HDC on the GPU. This is most likely because of the highly-optimized APIs for CNNs supplied by TensorFlow. However, the fact that far more instructions are executed for CNN than HDC indicates that actual computation cost is likely to be small.

A future experiment should instrument a carefully optimized code (preferably written in lower-level CUDA) that can handle multiple-batch sizes and compute them in a *single kernel-session*.

C. Summary

Fig. 2 summarizes the energy/inference costs for the HDC experiments performed in this section, and also shows the cost comparison with the synthesized ASIC processor described in Sec. VI. Clearly, eGPU and CPU require much higher energy expenditure than the ASIC design.

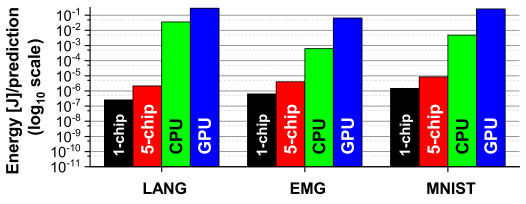


Fig. 2. **Energy Efficiency across platforms**: Shown energy consumed [J] in HDC across platforms CPU, eGPU and ASIC. *1-chip* denotes energy cost on the synthesized 28nm ASIC processor using 2048-dim vectors (see Sec. VI). *5-chip* denotes energy cost on a system of 5 interconnected ASIC processors with effective dimensionality of 10240.

HDC algorithms have been profiled on other platforms such as FPGAs [56] and ML accelerators [12]. Since the proposed ASIC design in Sec. VI is optimized to HDC, it is likely to have smaller overhead than such platforms. ASIC is most suitable for severely energy-constrained environments of IoT. As shown in later sections, it is indeed possible to design a simple and small architecture made of simple building blocks with small programming overhead.

V. GENERIC ARCHITECTURE

This section constructs the generic HD architecture from the properties of MAP operations (Section II-A). For the high energy efficiency necessary in IoT, it is crucial that the architecture is optimal. Therefore, ensuring component simplicity, using least resources and reducing redundant computation are the unifying principles used here.

A generic architecture must be able to map application-specific data after suitable pre-processing. The first step towards a general design is to *abstract essential elements* of HD algorithms. For supervised classification, a clear structure emerges.

A. Structure of HD Programs

1) Value Representation: To allow consumption by a discrete-time (clocked) digital system, the input data must be quantized into discrete states and sampled with a finite frequency. The choice of quantization scheme and sampling rates are important [57], [58] and is assumed to be pre-determined by a domain expert. Hence, a **common symbol set** $\mathbb{X}'$ representing values in the feature space always exists. Amulti-channel input stream can be serialized with a suitable policy (for example, samples from 2 channels $x_1, x_2, \ldots$ and $y_1, y_2, \ldots$ combined to $x_1, y_1, x_2, y_2, \ldots$).

Therefore, the input may be modeled as a single finite-length time-series of symbols (say, $I_{serial} \triangleq (x_1, x_2, \ldots, x_T)$) without losing generality. Once the set of classes, representation space, sampling rate, quantization and channel ordering are established, a supervised classification task is ready to be processed in HD.

2) Stages in HD Algorithms: Let $\mathbb{N}_n \triangleq \{1, 2, \ldots n\}$ be the set of all natural numbers up to n. The first step is to assign a random hyper-vector (**item**) to each symbol in $\mathbb{X}'$ to obtain the common item-set $\mathbb{X}$. Then the input stream of symbols in $I_{serial} = (x_t | t = 1, 2, \ldots, T)$ (where each $x_t \in \mathbb{X}'$) is replaced by the assigned hyper-vectors, to obtain the corresponding input sequence of hyper-vectors $\mathbb{I} \triangleq (X_t | t = 1, 2 \ldots, T)$ (where each $X_t \in \mathbb{X}$). Any collection of input values can now be specified by its set of indices in $\mathbb{I}$. Since only supervised classification is considered, a given $\mathbb{I}$ belongs to a single class to be trained or tested. For both cases, the exact same algorithm is applied for processing the hyper-vector sequence.

Therefore, the application-specific encoding stage is an **expression** of hyper-vectors $X_t, t \in \mathbb{N}_T$ and MAP operations (Section II-A). It is important to realize that superposition is the last operation in such expressions. This is because permutation and multiply distributes over superposition.

- $\rho([A + B]) = [\rho(A) + \rho(B)]$ for all permutations $\rho()$ and vectors A, B.
- $C \oplus [A + B] = [(C \oplus A) + (C \oplus B)]$ for all vectors A, B, C since threshold in superposition is the mean 0.

Using these distribution laws, any HD expression can be transformed to have superposition as the last operation.

A **single-stage algorithm** is defined as any HD algorithm where superposition is used only once. All operands of the final superposition are products of inputs and their permutations only. In other words, the encoded result is $S = [\sum_{i=1}^{K} f_i(\mathbb{I})]$, where i^{th} term is

$$f_i(\mathbb{I}) = (X_{p_1} \oplus X_{p_2} \ldots \oplus X_{p_m}) \oplus(\rho^{u_1}(X_{q_1}) \oplus \rho^{u_2}(X_{q_2}) \ldots \oplus \rho^{u_n}(X_{q_n})) \quad (2)$$

Each term $f_i(\mathbb{I})$ depends on specific input values: some occurring as is (set of positions denoted by $P_i \triangleq \{p_1, p_2, \ldots p_m\} \subseteq \mathbb{N}_T$), and others *permuted* (set of positions denoted by $Q_i \triangleq \{q_1, q_2, \ldots q_n\} \subseteq \mathbb{N}_T$) where permutation powers $(u_1, u_2, ..u_n)$ are positive integers. Note that a few inputs may occur both with and without permutation in the term (i.e. $P_i \cap Q_i \neq \phi$).

A **dual-stage algorithm** has terms composed of products of inputs, outputs of a single-stage algorithm and their permutations. Similarly, one can build **any multi-stage** HD program by hierarchically combining outputs of smaller-stage algorithms.

3) The 'Generic' Model: The main complexity is the generation of K term vectors $f_i(\mathbb{I})$. Each term requires only a few specific inputs $A_i = P_i \cup Q_i$, usually a small part of of the entire stream $\mathbb{I}$. In the most general case, expressions for distinct terms may have very different inputs and result expressions, and separate hardware would be dedicated to each of them. However, all known HD algorithms for expert systems and fixed-order Markov sequences (including Table I) have a much simpler form. Specifically, the following conditions are satisfied:

1) The number of dependent inputs $|A_i|$ is constant for all terms $i \in \mathbb{N}_K$. Let this be L.
2) All K terms have the **same HD expression**. If $f_i(z_1, z_2, \ldots, z_L)$ denotes the expression of the i^{th} term in terms of L *input variables* $z = (z_1, z_2, \ldots, z_L) \in \mathbb{X}^L$, then $f_i(z) = f_j(z)\ \forall i, j \in \mathbb{N}_K$. This *common expression* will be called $f(z)$.
3) The set of dependent inputs for the i^{th} term are **translations of a fixed subsequence** of input stream. That is, for some increasing sequence $t_i \in \mathbb{N}_T$ with $t_1 = 0$; the i^{th} input dependency set is $A_i = A_1 + t_i$. Here, $A_1 + t$ denotes the set obtained by adding t to each element of A_1. Note that the first term input A_1 captures the essential pattern of input dependencies for all terms.

These conditions limit the possibilities of single-stage HD expressions. An architecture designed to handle all such expressions shall be called **generic** as opposed to "*general-purpose*" or "*general*". Since input stream is a time series, the entire programming complexity of such a machine is only for computing $f(z)$.

Property 3 above ensures a sequential generation of all K term vectors. Assume a pipelined hardware for $f(z)$ with fixed latency and throughput equal to input rate and let the input hyper-vectors $\mathbb{I}$ be loaded in sequence. Then, the i^{th} term is produced t_i steps after the first term. The *final superposition* $S = [\sum_{i=1}^{K} f_i(\mathbb{I})]$ can be computed by accumulating these terms $f_i(\mathbb{I})$ at the required time-steps $(t_1, t_2, \ldots)$. A T-bit register can mark these time-steps when the accumulator must be enabled.

B. Common Algorithmic Kernels

Only a few basic expressions are used repeatedly in most HD algorithms. In fact, all applications in the benchmark in Table I require variations of the following 2 kernels. EMG encodes 4-grams for 64-feature samples.

- ***n*-gram Sequence Encoding**: As mentioned in Section II-B, n-grams or the multi-set of n-sequences can be very useful for modelling sequences. Characters in an *symbol set* $\mathbb{A} \triangleq \{a_1, a_2, \ldots, a_N\}$ are mapped to random hyper-vectors $Y_{a_1}, Y_{a_2}, \ldots, Y_{a_N}$ and the n-sequence $x_1, x_2, \ldots x_n$ (where $x_i \in \mathbb{A}$) is encoded as $\rho^{n-1}(Y_{x_1}) \oplus \rho^{n-2}(Y_{x_2}) \oplus \rho^{n-3}(Y_{x_3}) \ldots \oplus Y_{x_n}$. Finally, all occurring n-sequences in the input are encoded and superimposed to form the final hyper-vector.
- **Feature Superposition**: This is used to map input feature vectors into a hyper-vector. Let the feature vector of d dimensions be $V = (v_1, v_2, \ldots, v_d)$. For each *vector position* $i \in \mathbb{N}_d$, a random hyper-vector C_i is generated and as discussed in Section V-A.1, all possible input values v are assigned hyper-vectors Y_v. Then V is encoded as $[\sum_{i=1}^{d}(C_i \oplus Y_{v_i})]$ and a collection of n samples (a matrix U with sample vectors as rows) is encoded as the superposition $[\sum_{i=1}^{n} \sum_{j=1}^{d}(C_j \oplus Y_{U_{ij}})]$.

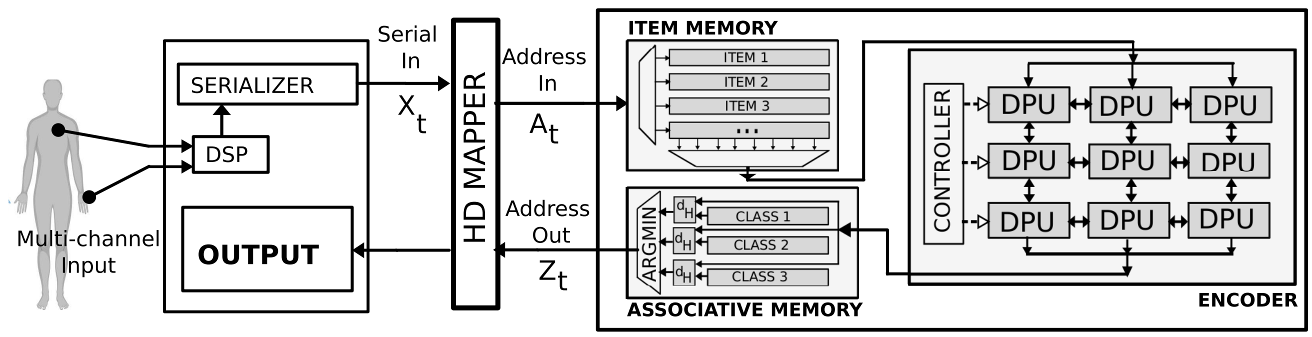


Fig. 3. **The Generic HD Processor**: major components and dataflow during test are shown. A body-sensing application is illustrated with two-channel input streams. All application-specific peripherals are to the left of *HD Mapper*. The Encoder is the only programmable component, and systolic array is the most suitable architecture.

C. Major Components

A generic HDC processor for supervised classification requires **three major components** corresponding to the major steps of processing:

- **Item Memory (IM)** stores a repertoire of random hyper-vectors (**items**). A sufficiently large collection of such vectors can be re-used for many applications. This storage requirement cannot be avoided as the map of symbols to items must be the same during training and testing.
- **Encoder** combines the input hyper-vectors sequence according to an application-specific algorithm to form single vector per class.
- **Associative Memory (AM)** stores the trained class hyper-vectors. During testing, the class hyper-vector closest to the encoded test hyper-vector is returned as the final prediction.

Fig. 3 shows a diagram for the complete system. All application specific pre-processing, sampling and quantization is done before combining input streams. The peripheral **HD Mapper** assigns incoming symbols to an **item** in Item Memory and class labels to AM addresses. This mapping is retained for all sessions of training and testing. Therefore, the actual input to the generic processor is a time series of IM addresses. Fig. 3 also shows the operation during testing. The IM fetches input hyper-vectors and the Encoder generates the test hyper-vector. The AM returns the address of the closest class vector. The actual label is substituted back by HD Mapper for further consumption.

While there could be multiple ways to interconnect these fundamental blocks depending on the complexity of the HDC system, the simplest system to consider is the generic model of Sec. V-A.3. When the processor is abstracted in this manner, two crucial properties emerge:

1) **Uni-directional Dataflow:** For all applications, hyper-vectors always flow from IM to Encoder to AM during both training and testing. There are **no iterations** over the input sequence I_{serial}: symbols x_t are fed in order only once.
2) **Single programmable component:** Only the Encoder needs to be programmed for an application. The operation of both memories always remain the same and do not change with applications.

Therefore, all major architectural decisions principally concern the Encoder. Since it performs only MAP operations, it is important to note the parallelism of each operation. For superposition and multiply, an element of the result vector depends only on corresponding elements of its operands. Permutation is the only operation with **dependency across vector elements**, where a result element depends on a neighbouring operand element.

To begin with, the generic model of Section V-A.3 explicitly encodes dependencies that are not mapped into instructions optimally by compilers. Finally, hardware required to extract dynamic ILP and DLP add to energy costs. Section IV-A and IV-B and Fig. 2 show that conventional architectures such as CPU & embedded GPUs are very energy-inefficient for HDC.

Clearly, a **dataflow architecture** is the most suitable candidate. Here, the Encoder is comprised of a regular network of simple **Data Processing Units** (DPU), and inter-DPU communication for dependencies is restricted to neighbors no more than a fixed distance away [59]. Though several attempts have been made to map common workloads to DPUs [60]–[64], only a few of them where dependency patterns can be expressed as a regular graph have been successful [59], [65], [66]. HD algorithms perfectly fit these conditions. All HD operations can be implemented with a few gates. The sequential input model and the generic abstraction of Section V-A.3 enables us to map algorithms to DPUs explicitly.

D. Encoder Architecture

The Encoder is crucial for the overall programmability of the processor and has the largest activity and wiring complexity. Hence, efficient design of its architecture is important for optimal implementation.

1) Organization of the Encoder: Generic algorithms can be decoupled into the generation of terms and super-position. Clearly, the DPU network only needs to generate all necessary terms, hence it suffices to only implement multiply and permute in them.

Fig. 4(a) shows the **Hyper-dimensional Logic Unit (HLU)**, the simplest (single-bit) DPU with only a register and gate. Since permute has intra-word dependencies, *D* HLUs can be connected together to form a module operating on an entire

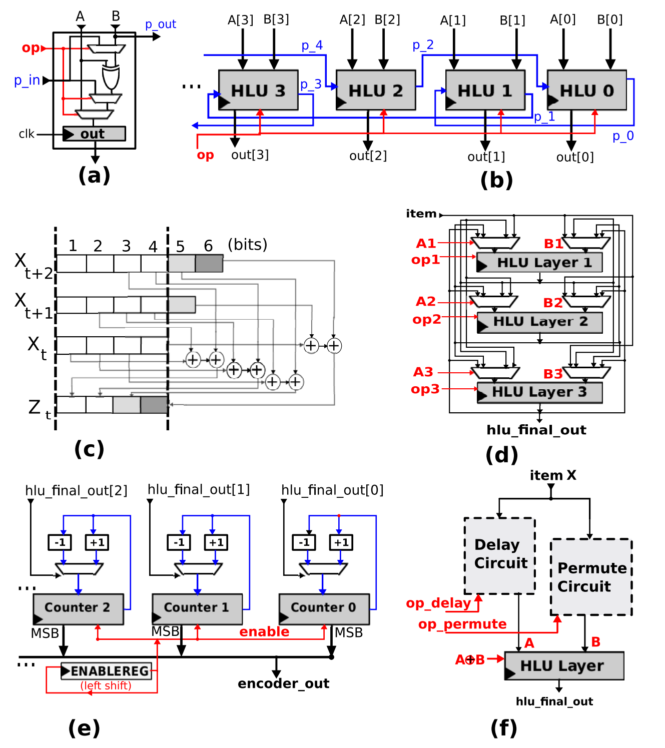


Fig. 4. **Encoder organization**: (From Top-Left): (a) Hyper-Dimensional Logic Unit (HLU), (b) connecting single-bit HLUs to create a HLU Layer, (c) encoding steps required for a 3-gram Z_t of hyper-vectors X_t, X_{t+1}, X_{t+2} and across-subword dependency, (d) inter-connecting multiple HLU Layers to generate terms, (e) accumulator for superposition, (f) for the generic model, the *delay partial term* and *permute partial term* can be separately computed and combined to get the final term for superposition.

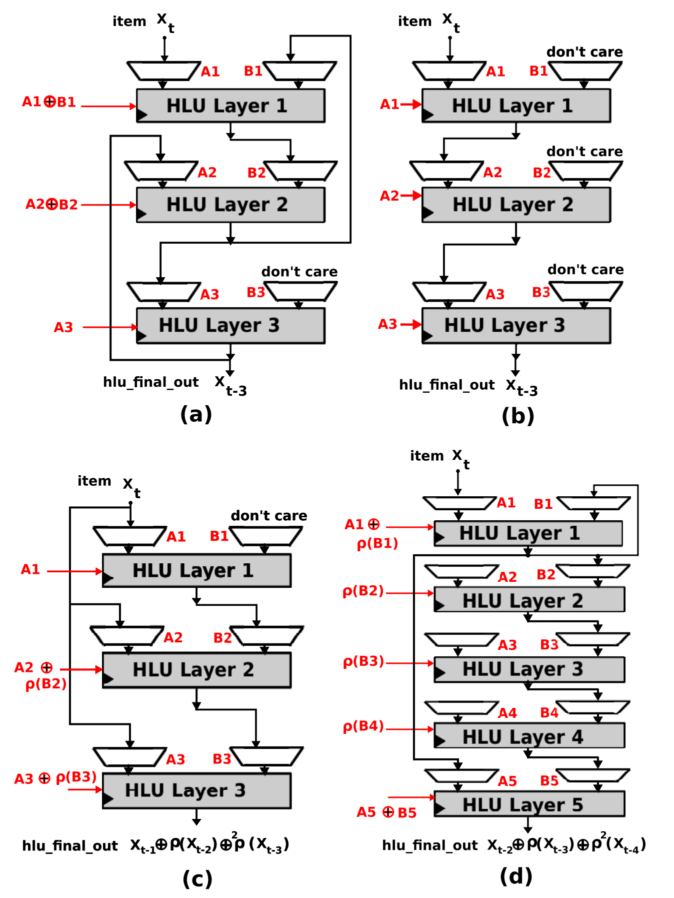


Fig. 5. **Examples of HLU Interconnections**: (a) A feedback implementation of input X_t delayed by 3 cycles. (b) A feedforward implementation of input X_t delayed by 3 cycles. (c) An optimal implementation of 3-gram, using the minimum number of HLU Layers. (d) A non-optimal implementation of 3-gram with 5 HLU Layers.

hyper-vector (Fig. 4(b)). This coherent unit will be called **HLU Layer**. It takes two hyper-vector operands A and B and can **multiply** ($C = A \oplus B$), **permute** ($C = \rho(A)$), **delay** ($C = A$) or **permute-and-multiply** ($C = A \oplus \rho(B)$). Each constituent HLU performs the *same* operation on input bits. Permute is a single-cycle derangement, hence any Hamiltonian path connection through `p_in` and `p_out` visiting all HLUs is valid. Fig. 4(b) illustrates a scheme where alternate HLUs (except first and last) are connected to minimize length of longest wire.

HLU Layers can be interconnected among themselves, generating an overall output `hlu_final_out` by transforming the stream of inputs (`item`) from the IM (Fig. 4(d)).

Finally, a simple array of accumulators perform the super-position (Fig. 4(e)). A counter at a hyper-vector position increments or decrements according to corresponding bits of `hlu_final_out` being 1 or 0. When the encoding completes, the vector formed by the MSB of the counters is the required superposition. Note that the accumulator takes in hyper-vectors (`hlu_final_out`) calculated by the **last HLU Layer**. In Fig. 4(e) the shift-register `ENABLEREG` is programmed to mark the cycles of arrival of required terms and enable the counters for accumulation (see Section V-A.3). Its contents are shifted as the design encodes, accumulating **only the required** terms.

2) Programmability: A crucial factor for array architectures is the choice of interconnection network. More general networks allow efficient mapping of algorithms at the cost of increased hardware complexity and power consumption.

For the Encoder, HLU Layers are the only independent processing elements. Therefore, the major decision here is the set of allowed operands to each HLU Layer. The most general network would allow output of any HLU Layer or IM to be either operand for all HLU Layers.

Fig. 4(d) provides an example Encoder with 3 layers. Signals `op1`, `op2`, `op3` program the operation carried out by HLU Layer 1, 2, 3 respectively. For generic programs, the *term expression* is constant, hence these control signals stay the same throughout the encoding. Operand-select signals `A1`, `B1`, `A2`, `B2`, `A3`, `B3` decide the actual interconnections of HLU Layers. Note that in this setting, **feedback** is allowed: a HLU Layer's output may be its own input in the next cycle.

Feedback offers greater variety for most term expressions. Fig. 5(a) and 5(b) are two HLU networks with feedback and feedforward configurations respectively. Both of them transform the input sequence X_t to the same final output `hlu_final_out` $= X_{t-3}$. Equivalent networks with feedback can have non-trivial dependency patterns and usually requires an exhaustive search through all possible interconnections.

However, operand-MUX widths and wiring complexity grow quadratically with the number of HLU Layers, making general interconnections infeasible for large designs. It also complicates the overall pipeline control, as flushing or filling a pipeline with arbitrary feedback connections is complex.

Fortunately, the benchmark applications do not need such large designs. Feedforward implementations for term expressions are easy to find and the generic abstraction from Section V-A.3 helps. Eq. 2 separates a term $f_i(\mathbb{I})$ into two products: the delay partial term $(X_{p_1} \oplus X_{p_2} \ldots \oplus X_{p_m})$ made up only of delayed inputs and the permute partial term $(\rho^{u_1}(X_{q_1}) \oplus \rho^{u_2}(X_{q_2}) \ldots \oplus \rho^{u_n}(X_{q_n}))$ made up of permuted and delayed inputs. They can be implemented by separate interconnections of the HLU **Delay Circuit** and **Permute Circuit** respectively, and combined to give the final output (Fig. 4(f)).

3) Depth and Width of Dataflow Network: For IoT and embedded applications, energy efficiency is the critical metric of success. This subsection describes the effect of Encoder depth (i.e. number of HLU Layers) and width (say $D_{Encoder}$) on energy efficiency.

a) Encoder depth: The HLU Layers are the principal computing resource in this architecture. They are also useful for intermediate storage as managing a high-dimensional register-file or data-cache is very inefficient. Consequently, a given encoding algorithm requires a minimum number of HLU Layers to be implemented. For example, 3-grams require at least 3 HLU Layers for storing the 3 operand hyper-vectors $X_{t-1}, X_{t-2}, X_{t-3}$ (Fig. 5(c) shows one such interconnection). Hence, an Encoder with 2 HLU Layers cannot encode n-grams of $n \geq 3$ as no interconnections with 2 Layers exist for them. Using more than the minimum Encoder layers is possible for any term $f(z)$ (while expending more energy), and there is no maximum (trivially, use Fig. 5(a) repeatedly to delay the input). Fig. 5(d) shows an interesting alternative interconnection to encode 3-grams using 5 HLU layers.

b) Encoder width: Note that both Item and Associative Memory must be full-width to avoid accesses to extremely expensive off-chip memory. Hence, the designer only has a choice of reducing the Encoder width. The following arguments suggest that sub-word Encoders ($D_{Encoder} < D$) are not energy-efficient:

- As Fig. 4(c) shows, permutation leads to additional dependencies: at each cycle, grey-coded output bits require operands from neighboring sub-words. This leads to redundancy as the external bits are re-computed for another sub-word. For example: n-gram uses $n(n-1)/2$ extra bits/cycle for each sub-word.
- Sub-words introduce multiple iterations on the dataset. This necessitates storing datasets and could require large memories: such as $\approx 10^6$ `char` in the training text for LANG that would require $\approx$ 1 MB. For a full-width Encoder, this is not necessary as a single-pass through the dataset suffices for both training and testing.
- A central property of HDC is that only a few MAP-operations need to be performed versus conventional ML algorithms. Therefore, as discussed in Section VII, a full-width Encoder is likely to consume less than half of the total power (i.e. $P_{FullEnc} < P_{Tot}/2$). So, the total power with a sub-word Encoder P'_{Tot} must have $P'_{Tot} > P_{Tot} - P_{FullEnc} > P_{Tot}/2$. Further, both memories remain unchanged and the Associative Memory's `popcount` module (Fig. 6(c)) is the critical-path (hence t_{CLK} is unchanged). Finally, the total cycles/prediction (say T_{Tot} for full-width) is dominated by the encoding cycles. Thus, sub-word total cycles/prediction satisfies $T'_{tot} \geq 2\ T_{tot}$ (at least 2 sub-words, or 2 iterations) and sub-word energy/prediction will be greater: $P'_{Tot} \times T'_{Tot} > (P_{Tot}/2) \times (2\ T_{Tot}) = P_{Tot} T_{Tot}$.

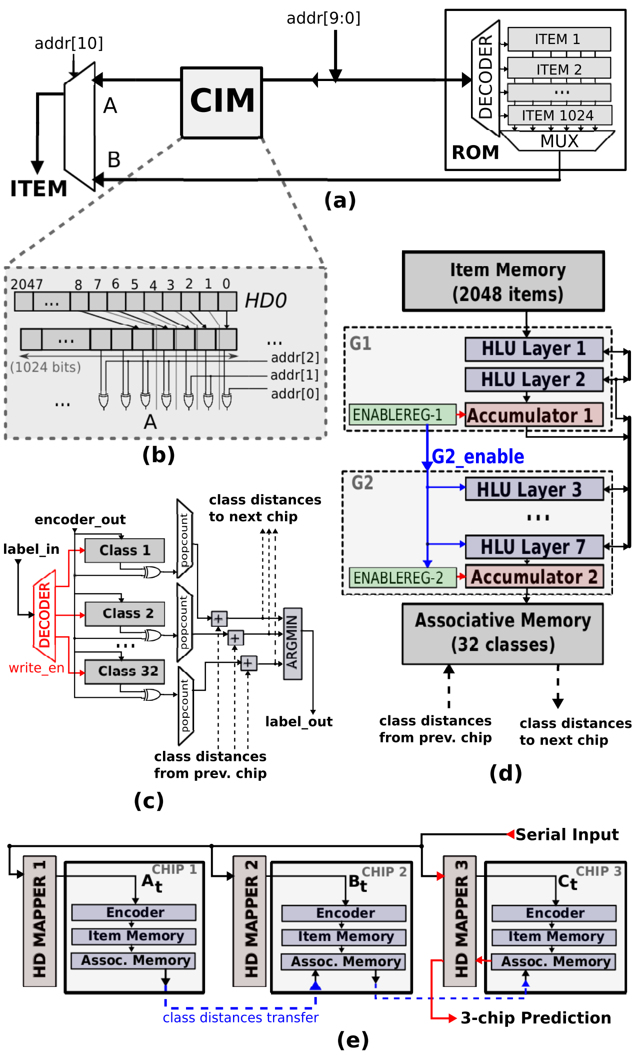


Fig. 6. (a) Item Memory block diagram (b) Continuous Item Memory (CIM) (c) Associative Memory block diagram (d) Summary of resources for the 2048-dim processor implemented (e) Interconnecting multiple processors to improve accuracy at the expense of energy and latency.

VI. ASIC Implementation

A completely digital ASIC implementation with standard logic cells, memory blocks and conventional Computer-Aided Design (CAD) toolflow is described.

A. Item Memory

***Continuous* Item Generation:** Assigning orthogonal vectors to integers or values from an ordered set may not be appropriate. Ideally, two close numbers should have a correspondingly strong correlation in their hyper-vectors. A possible solution (by [49]) is to assign points on a line connecting two *exactly* orthogonal vectors. Then any collection of 3 hyper-vectors A, B, C (representing integers $a \leq b \leq c$) will satisfy the triangle law $d_H(A, B) + d_H(B, C) = d_H(A, C)$. Equivalently, one can begin with a randomly generated *origin vector* `HD0` and a direction vector `Y` with $(D/2)$-bits being 1. The smallest integer, usually 0, is mapped to `HD0` and the largest integer, say M, is mapped to `HD0` $\oplus$ `Y` (Fig. 6(b)). For all other integers n, flip $D/2/M$ additional bits along direction `Y` from the hyper-vector assigned to $n-1$. The only restriction is that M divides $D/2$.

A **Read-Only Memory (ROM)** was chosen to store the constant item hyper-vectors generated offline (Fig. 6(a)). This is the simplest possible implementation suited for this preliminary design. For the applications in the benchmark (Table I), 1024 orthogonal items suffice. Item hyper-vectors are divided into 8-bit sub-words and stored separately in 256 instances of $ROM1024 \times 8$. Fig. 6(a) shows a 2048-dim Item Memory with an 11-bit address (`addr`); the address space is partitioned with ROM vectors in the lower half and continuous items in the upper half.

B. Associative Memory

The **Associative Memory** stores the learned vectors for a later comparison and retrieval (see Fig. 6(c)). During training, the `write_en` signal enables only the address location pointed by `label_in` for writing. The output from the Encoder (`encoder_out`) is saved into the corresponding register. During testing, the `encoder_out` contains the test hyper-vector whose hamming distance is computed to each class in parallel. The number of mismatches is computed by an element-wise XOR followed by counting the number of 1s using `popcount` logic.

A simple module was designed that can store 32 classes and each class vector has a separate `popcount` logic attached. `popcount` counts the number of 1s in a 256-bit sub-word in one cycle, and thus 8 cycles are required for the entire 2048-dim hyper-vector. Since the amount of storage required is small (32× 2048-dim hyper-vectors is about 8 KB), flip-flops were used instead of SRAMs.

C. Encoder

Fig. 6(d) provides a simplified block-diagram of the entire 2048-dim processor. The designed Encoder has only as much resources as required by the benchmark applications. A total of 7 HLU Layers are split into two groups `G1` and `G2`. HLU Layers 1 and 2 form group `G1`, where *Accumulator* 1 performs superposition of the Layer 2 outputs. HLU Layers 3 to 7 form group `G2` where *Accumulator* 2 performs superposition of the Layer 7 outputs. `ENABLEREG-1` and `ENABLEREG-2` are 256-bit shift registers implementing the control signals for enabling Accumulator 1 and Accumulator 2 respectively. To allow **dual-stage encoding**, `ENABLEREG-1` from group `G1` provides a **global enable** signal for all logic in `G2` as well. Note that the HLU Layer interconnections are not

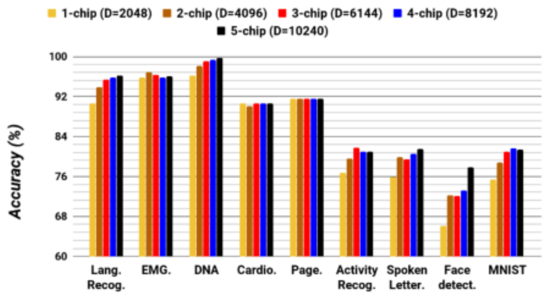


Fig. 7. Accuracy with multi-processor configuration.

fully general: the item hyper-vector can only go to Layer 1, Layers 1 and 2 are fully-connected, and Layers 3 to 7 are fully-connected with Layer 2 and Accumulator 1.

The control signals for the HLU Layers `op, A, B` are scanned into appropriate registers of the HLU Layer. To program the processor, all HLU Layers are configured within 130 cycles. `ENABLEREG` are also loaded before computation begins. Each HLU Layer requires 8 bits of control, which totals to 64 bits. Once the machine is programmed, inputs can be consumed sequentially to train/test on the generated hyper-vector.

D. Multi-Processor Configuration

A simple configuration of multiple processors results in increased effective dimension (see Fig. 7). To demonstrate this, consider LANG using 27 input characters (lower-case alphabets and whitespace). Suppose two instances of the 2048-dim processor (with the same items in the IM) are used for LANG. The 27 input symbols are randomly mapped to 27 of 1024 available items in the first processor's HD Mapper 1. The second processor's HD Mapper 2 chooses a **new map** of 27 symbols to 1024 items. On average, the two maps have about $27^2/1024 = 0.71$ common symbol-item pairs. Hence, they almost certainly compute with completely different items which is equivalent to the computation on a 4096-dim datapath with freshly generated items.

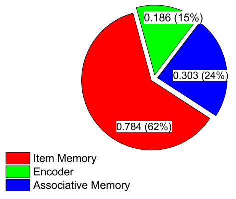
Fig. 6(e) shows a system of three 2048-dim processors interconnected to simulate $D_{eff} = 6144$. Class hyper-vectors are trained and stored locally in each processor. The mapping of labels to addresses for the AM must be the same for all three HD Mappers. During testing, the test-vector is encoded locally and the distances to the stored class hyper-vectors are computed locally by each processor. For each class, these local distances are added to generate a total distance. The class with smallest total distance is returned as prediction.

A Serial Peripheral Interface (SPI) module can be used for transmitting local distances to adjacent processors (see Fig. 6(e)). The AM could add local distances to those from its SPI slave before transmitting to its SPI master. The processors are connected linearly and the last processor computes the total class distances along with the final prediction.

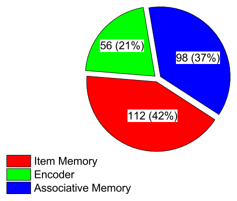
It is better to design with smaller dimensions, as one can easily increase the effective dimensions with interconnection (but not vice versa). All applications that are better than

TABLE V
QUALITY OF RESULTS (QoR) REPORT FOR 2048-DIM PROCESSOR

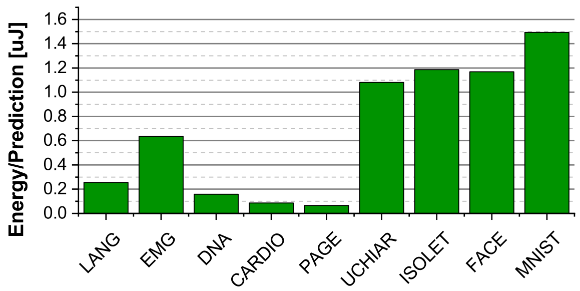
Property	Value
Technology	TSMC 28 HPM
Total Cell Area	1.27 sq. mm.
t_{CLK}	2.4 ns
Total estimated power	267 mW
Total number of Cells	4.73×10^5
Number of Buf/Inv Cells	37501



(a) Area [mm^2]



(b) Power [mW]



(c) Single-chip Energy [μJ/prediction]

Fig. 8. (a) Area and (b) Power breakdown. (c) 1-chip Energy/Prediction for the benchmark.

baseline for the 5-chip setting are also better for $D = 2048$. Hence, choosing $D = 2048$ is optimal.

VII. ASIC RESULTS

A. 28nm-Synthesized Processor

The HDC model was trained and tested for all applications in the benchmark. A System-Verilog RTL description of the datapath was synthesized using standard digital logic in the 28nm *High-K/Metal-gate* (HKMG) node provided by TSMC, under a clock-cycle constraint of $t_{CLK} \leq 2.4$ ns.

1) Hardware Complexity: The power consumption of the synthesized logic was estimated at the (0.8 V, 25°C, TT) corner for all apllications. Table V lists the main results of the designed hardware. The primary goal is to quantify the effects of the generic architecture *alone* on the processor performance. Hence no specialized library cells or circuits were utilized to optimize for area or power consumption.

Fig. 8 shows the component-wise breakdown of area and power consumption. The power consumption of LANG is shown in Fig. 8(b). Other applications resulted in a similar breakdown of power consumption. The Item Memory is the largest component as it stores *over a thousand* hyper-vectors. The principal contributor to the Encoder's size is the integer counters in its two *accumulators*. Each component of the accumulator hyper-vector needs a 22-bit register compared to single-bit registers elsewhere. The major contribution in the Associative Memory is the dedicated `popcount` logic for each class, adding to overall area and power consumption. Most importantly, the two **memory components** are the **main contributors** to overall power and area. More efficient implementations, such as analog content-addressable memory [15], would reduce the overall cost significantly.

2) Energy Efficiency: Fig. 8(c) shows the energy per prediction on a single-processor system. This suffices for LANG, EMG, DNA, CARDIO and PAGE with energy efficiency better than **0.8 μJ/pred**. For all applications, a single-processor system consumes less than 1.6μJ/pred. For comparison, [67] implements a 10,000-dim HD algorithm for EMG on the 4-core PULPv3 processor [68], operated at 0.5 V, to give an energy efficiency of 2.10 mW $\times$10ms/pred. = 21μJ/pred. A 5-chip implementation of EMG on this processor ($D_{eff} = 10240$) consumes only 4.03μJ/pred (5.2$\times$ improvement). However, PULPv3 operates at near-threshold range ($V_{DD} = 0.5$ V) and generates internal body-biasing to optimize for energy efficiency, whereas **no circuit optimizations** are used here. The total ROM size in this design (256 KB) also exceeds PULP's L2-cache of 64 KB.

VIII. CONCLUSION

The *generic abstraction* (Section V-A.3) shows that one can develop a simple architecture that can be programmed easily (only 8 HLU Layers requiring 130 cycles to program) to handle a large variety of supervised applications. A simple HDC system synthesized in 28nm predicts an energy efficiency better than **1.5 μJ/pred** for all benchmark applications. This indicates that a fabricated HDC chip could meet the extremely high energy-efficiency requirements necessary for human-centric IoT.

The choice of distance metric is a crucial factor in determining the overall performance. An advanced datapath using **cosine similarity** instead of hamming distance would improve accuracy significantly [28], [32]. Similarly, re-training [36] and using hierarchical HDC algorithms [19] can improve the accuracy of under-performing applications significantly. These could be incorporated in a future HDC system by adding additional control structures and features to the fundamental generic machine developed in this work.

As a preliminary design of a generic processor, both the memories are rudimentary. Reference [15] indicates an improvement of at least 10$\times$ in associative memory energy/pred. when using analog over digital circuits. ROM-based Item Memory is very expensive as it stores thousands of high-dimensional vectors. Onboard generation of pseudo-random vectors (such as using RRAMs in [17]) would be a great improvement as well.

The datasets used in the benchmark could be improved in the future. The benchmark datasets are from public repositories, as access to high-quality data, especially for body-sensing or industrial applications, requires special licensing, legal commitments and privacy assurances. Finally, an algorithmic exploration using newer and better ML algorithms than the most recent literature on the benchmark datasets could be done as well.

ACKNOWLEDGMENT

This work is sponsored by "Energy Efficient Computing: from Devices to Architecture" (E2CDA-NRI), a funded center of Nanoelectronics Research Initiative (NRI), a Semiconductor Research Corporation (SRC) program co-sponsored by the National Science Foundation (NSF). The authors are grateful to Taiwan Semiconductor Manufacturing Comparny Ltd. (TSMC) for providing the technology libraries used in this work, and to Commission on Higher Education (CHED)-Philippine-California Advanced Research Institutes (PCARI) for facilitating this joint work of collaboration between University of California Berkeley and University of Philippines Diliman.

The authors would like to thank Bruno Olshausen and Pentti Kanerva (UC Berkeley) for insightful discussions, Mohsen Imani (UC San Diego) for providing data files used in training and classification, Youbin Kim (UC Berkeley) for proofreading the final manuscript, and all the reviewers for providing thorough comments and reviews.

REFERENCES

[1] C. A. Mack, "Fifty years of Moore's law," *IEEE Trans. Semicond. Manuf.*, vol. 24, no. 2, pp. 202–207, May 2011.

[2] S. Salahuddin, K. Ni, and S. Datta, "The era of hyper-scaling in electronics," *Nature Electron.*, vol. 1, no. 8, pp. 442–450, 2018.

[3] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, Nov. 2005.

[4] T. N. Theis and H.-S. P. Wong, "The end of Moore's law: A new beginning for information technology," *Comput. Sci. Eng.*, vol. 19, no. 2, pp. 41–50, Mar./Apr. 2017.

[5] H. Chen, R. H. Chiang, and V. C. Storey, "Business intelligence and analytics: From big data to big impact," *MIS Quart.*, vol. 36, no. 4, pp. 1165–1188, 2012.

[6] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: Architecture, applications, and approaches," *Wireless Commun. Mobile Comput.*, vol. 13, no. 18, pp. 1587–1611, Dec. 2013.

[7] Y.-B. Kim, "Challenges for nanoscale MOSFETs and emerging nanoelectronics," *Trans. Elect. Electron. Mater.*, vol. 11, no. 3, pp. 93–105, Jun. 2010.

[8] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits," *Proc. IEEE*, vol. 98, no. 2, pp. 253–266, Feb. 2010.

[9] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proc. 38th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2011, pp. 365–376.

[10] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automat. Stud.*, vol. 34, no. 34, pp. 43–98, 1956.

[11] A. Rahimi *et al.*, "High-dimensional computing as a nanoscalable paradigm," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 64, no. 9, pp. 2508–2521, Sep. 2017.

[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," *Proc. IEEE*, vol. 107, no. 1, pp. 123–143, Jan. 2019.

[13] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cogn. Comput.*, vol. 1, no. 2, pp. 139–159, Oct. 2009.

[14] P. Kanerva, *Sparse Distributed Memory*. Cambridge, MA, USA: MIT Press, 1988.

[15] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, "Exploring hyperdimensional associative memory," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 445–456.

[16] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*, 2016, pp. 64–69.

[17] T. F. Wu *et al.*, "Brain-inspired computing exploiting carbon nanotube fets and resistive ram: Hyperdimensional computing case study," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2018, pp. 492–494.

[18] H. Li *et al.*, "Hyperdimensional computing with 3D VRRAM in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition," in *IEDM Tech. Dig.*, Dec. 2016, pp. 16.1.1–16.1.4.

[19] M. Imani, C. Huang, D. Kong, and T. Rosing, "Hierarchical hyperdimensional computing for energy efficient classification," in *Proc. 55th Annu. Design Autom. Conf. (DAC)*, 2018, pp. 108:1–108:6.

[20] A. DasGupta, "Normal approximations and the central limit theorem," in *Fundamentals of Probability: A First Course*. New York, NY, USA: Springer, 2010, pp. 213–242.

[21] M. Okamoto, "Some inequalities relating to the partial sum of binomial probabilities," *Ann. Inst. Stat. Math.*, vol. 10, no. 1, pp. 29–35, 1959.

[22] P. Kanerva, "Some properties of the space $\{0, 1\}^n$," in *Sparse Distributed Memory*. Cambridge, MA, USA: MIT Press, 1988, ch. 1, pp. 18–22.

[23] T. A. Plate, "Holographic reduced representations," *IEEE Trans. Neural Netw.*, vol. 6, no. 3, pp. 623–641, May 1995.

[24] T. Vatanen, J. J. Väyrynen, and S. Virpioja, "Language identification of short text segments with N-Gram models," in *Proc. LREC*, 2010, pp. 1–8.

[25] J. F. da Silva and G. P. Lopes, "Identification of document language is not yet a completely solved problem," in *Proc. Int. Conf. Comput. Intell. Modelling Control Automat. Int. Conf. Intell. Agents Web Technol. Int. Commerce (CIMCA)*, Nov. 2006, p. 212.

[26] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," *Comput. Speech Lang.*, vol. 13, no. 4, pp. 359–394, Oct. 1999.

[27] A. Moin *et al.*, "An EMG gesture recognition system with flexible high-density sensors and brain-inspired high-dimensional classifier," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.

[28] M. Imani, T. Nassar, A. Rahimi, and T. Rosing, "HDNA: Energy-efficient DNA sequencing using hyperdimensional computing," in *Proc. IEEE EMBS Int. Conf. Biomed. Health Inform. (BHI)*, Mar. 2018, pp. 271–274.

[29] N. Chamidah and I. Wasito, "Fetal state classification from cardiotocography based on feature extraction using hybrid k-means and support vector machine," in *Proc. Int. Conf. Adv. Comput. Sci. Inf. Syst. (ICACSIS)*, Oct. 2015, pp. 37–41.

[30] A. M. Bagirov, J. Ugon, D. Webb, and B. Karasözen, "Classification through incremental max–min separability," *Pattern Anal. Appl.*, vol. 14, pp. 165–174, May 2011.

[31] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "Human activity recognition on smartphones using a multiclass hardware-friendly support vector machine," in *Ambient Assisted Living and Home Care (IWAAL)* (Lecture Notes in Computer Science), vol. 7657, J. Bravo, R. Hervás, and M. Rodríguez, Eds. Berlin, Germany: Springer, 2012.

[32] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "VoiceHD: Hyperdimensional computing for efficient speech recognition," in *Proc. IEEE Int. Conf. Rebooting Comput. (ICRC)*, Nov. 2017, pp. 1–8.

[33] Y. Kim, M. Imani, and T. Rosing, "ORCHARD: Visual object recognition accelerator based on approximate in-memory processing," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2017, pp. 25–32.

[34] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2012, pp. 3642–3649.

[35] J. M. Rabaey, "The human Intranet–Where swarms and humans meet," *IEEE Pervasive Comput.*, vol. 14, no. 1, pp. 78–83, Jan. 2015.

[36] Y. Kim, M. Imani, and T. S. Rosing, "Efficient human activity recognition using hyperdimensional computing," in *Proc. 8th Int. Conf. Internet Things (IOT)*, 2018, pp. 38:1–38:6.

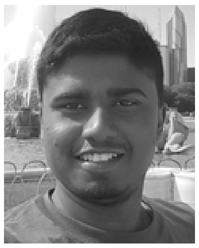
[37] J. Bose, S. B. Furber, and J. L. Shapiro, "An associative memory for the on-line recognition and prediction of temporal sequences," in *Proc. IEEE Int. Joint Conf. Neural Netw. (IJCNN)*, vol. 2, Jul. 2005, pp. 1223–1228.

[38] O. J. Räsänen and J. P. Saarinen, "Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 27, no. 9, pp. 1878–1889, Sep. 2016.

[39] O. Rasanen and S. Kakouros, "Modeling dependencies in multiple parallel data streams with hyperdimensional computing," *IEEE Signal Process. Lett.*, vol. 21, no. 7, pp. 899–903, Jul. 2014.

[40] D. Kleyko, E. Osipov, N. Papakonstantinou, and V. Vyatkin, "Hyperdimensional computing in industrial systems: The use-case of distributed fault isolation in a power plant," *IEEE Access*, vol. 6, pp. 30766–30777, 2018.

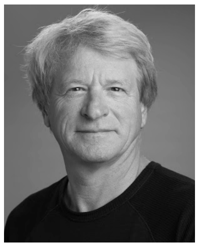
[41] Y. Khan *et al.*, "Flexible hybrid electronics: Direct interfacing of soft and hard electronics for wearable health monitoring," *Adv. Funct. Mater.*, vol. 26, no. 47, pp. 8764–8775, Dec. 2016.
[42] A. Burrello, K. Schindler, L. Benini, and A. Rahimi, "One-shot learning for iEEG seizure detection using end-to-end binary operations: Local binary patterns with hyperdimensional computing," in *Proc. IEEE Biomed. Circuits Syst. Conf. (BioCAS)*, Oct. 2018, pp. 1–4.
[43] 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 *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 752–757.
[44] Y. Ma, M. Hildebrandt, V. Tresp, and S. Baier, "Holistic representations for memorization and inference," in *Proc. UAI*, 2018, pp. 403–413.
[45] A. Mitrokhin, P. Sutor, C. Fermüller, and Y. Aloimonos, "Learning sensorimotor control with neuromorphic sensors: Toward hyperdimensional active perception," *Sci. Robot.*, vol. 4, no. 30, p. eaaw6736, 2019.
[46] B. Cheung, A. Terekhov, Y. Chen, P. Agrawal, and B. Olshausen, "Superposition of many models into one," Feb. 2019, *arXiv:1902.05522*. [Online]. Available: http://arxiv.org/abs/1902.05522
[47] D. Franklin. (2017). *NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge*. [Online]. Available: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/
[48] A. Troy and O. Hanson. (2017). *UT-HASH: A Hash Table for C Structures*. [Online]. Available: https://troydhanson.github.io/uthash/
[49] A. Rahimi, S. Benatti, P. Kanerva, L. Benini, and J. M. Rabaey, "Hyperdimensional biosignal processing: A case study for emg-based hand gesture recognition," in *Proc. IEEE Int. Conf. Rebooting Comput. (ICRC)*, Oct. 2016, pp. 1–8.
[50] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, 2011.
[51] (2018). *Kann*. [Online]. Available: https://github.com/attractivechaos/kann
[52] *INA226 High-Side or Low-Side Measurement, Bi-Directional Current and Power Monitor With I2C Compatible Interface*, Texas Instrum., Dallas, TX, USA, Jun. 2011.
[53] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the ARM architecture," in *Proc. Int. Conf. Compil., Archit., Synth. Embedded Syst.*, Seoul, South Korea, Oct. 2006, pp. 261–270.
[54] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," 2016, *arXiv:1603.04467*. [Online]. Available: https://arxiv.org/abs/1603.04467
[55] Z. Wen, J. Shi, Q. Li, B. He, and J. Chen, "ThunderSVM: A fast SVM library on GPUs and CPUs," *J. Mach. Learn. Res.*, vol. 19, no. 1, pp. 797–801, 2018.
[56] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, "F5-HD: Fast flexible FPGA-based framework for refreshing hyperdimensional computing," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2019, pp. 53–62.
[57] A. Coates and A. Y. Ng, "The importance of encoding versus training with sparse coding and vector quantization," in *Proc. 28th Int. Conf. Mach. Learn. (ICML)*, 2011, pp. 921–928.
[58] G. E. Batista, R. C. Prati, and M. Monard, "A study of the behavior of several methods for balancing machine learning training data," *ACM SIGKDD Explorations Newslett.*, vol. 6, no. 1, pp. 20–29, 2004.
[59] H. V. Jagadish, S. K. Rao, and T. Kailath, "Array architectures for iterative algorithms," *Proc. IEEE*, vol. 75, no. 9, pp. 1304–1321, Sep. 1987.
[60] U. Eckhardt and R. Merker, "Hierarchical algorithm partitioning at system level for an improved utilization of memory structures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 1, pp. 14–24, Jan. 1999.
[61] S. V. Rajopadhye, "Synthesizing systolic arrays with control signals from recurrence equations," *Distrib. Comput.*, vol. 3, no. 2, pp. 88–105, 1989.
[62] S. Borkar, R. Cohn, G. Cox, S. Gleason, and T. Gross, "Warp: An integrated solution of high-speed parallel computing," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 1988, pp. 330–339.
[63] K. K. Parhi, C.-Y. Wang, and A. P. Brown, "Synthesis of control circuits in folded pipelined DSP architectures," *IEEE J. Solid-State Circuits*, vol. 27, no. 1, pp. 29–43, Jan. 1992.
[64] P. Cappello, "A processor-time-minimal systolic array for cubical mesh algorithms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 1, pp. 4–13, Jan. 1992.
[65] T. Komarek and P. Pirsch, "Array architectures for block matching algorithms," *IEEE Trans. Circuits Syst.*, vol. 36, no. 10, pp. 1301–1308, Oct. 1989.
[66] H. V. Jagadish and T. Kailath, "A family of new efficient arrays for matrix multiplication," *IEEE Trans. Comput.*, vol. 38, no. 1, pp. 149–155, Jan. 1989.
[67] 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 *Proc. 55th Annu. Design Autom. Conf. (DAC)*, 2018, pp. 111:1–111:6.
[68] D. Rossi *et al.*, "A self-aware architecture for PVT compensation and power nap in near threshold processors," *IEEE Design Test*, vol. 34, no. 6, pp. 46–53, Dec. 2017.



Sohum Datta received the bachelors' degree in electrical engineering from IIT Kanpur, Kanpur, India, in 2015, and the masters' degree in computer science from the University of California at Berkeley (UC Berkeley) in 2018, where he is currently pursuing the Ph.D. degree in computer science with the Department of Electrical Engineering and Computer Sciences (EECS), advised by Prof. Jan M. Rabaey. His main interests are brain-inspired memory and cognitive models, stochastic computing, and energy-efficient implementation of such systems.

Ryan A. G. Antonio received the B.Sc. degree in computer engineering from the University of the Philippines Diliman (UP Diliman) in 2016, where he is currently pursuing the M.Sc. degree in electrical engineering with the Microelectronics and Microprocessors Laboratory. His current research interests are hardware architectures for hyperdimensional computing, machine learning, and asynchronous circuit designs.

Aldrin R. S. Ison received the B.Sc. degree in electronics and communications engineering from the University of the Philippines Diliman (UP Diliman) in 2016, where he is currently pursuing the M.Sc. degree in electrical engineering with the Microelectronics and Microprocessors Laboratory. His research interests include spectrum sensing and hardware architectures for edge computing.



Jan M. Rabaey is currently a Professor with the EECS Department, Graduate School, University of California at Berkeley (UC Berkeley), after being the holder of the Donald O. Pederson Distinguished Professorship at UC Berkeley for over 30 years. Before joining the Faculty of UC Berkeley, he was a Research Manager with IMEC, Belgium, from 1985 to 1987. He is also the founding Director of the Berkeley Wireless Research Center (BWRC) and the Berkeley Ubiquitous SwarmLab and has served as the Electrical Engineering Division Chair at UC Berkeley twice. In 2019, he became the CTO of the System-Technology Co-Optimization (STCO) Division, IMEC. He has been involved in a broad variety of start-up ventures, including Cortera Neurotechnologies, of which he was a Co-Founder. He has made high-impact contributions to a number of fields, including advanced wireless systems, low-power integrated circuits, mobile devices, sensor networks, and ubiquitous computing. His current interests include the conception of the next-generation distributed systems, as well as the exploration of the interaction between the cyber and the biological world.

Dr. Rabaey is a member of the Royal Flemish Academy of Sciences and Arts of Belgium. He received honorary doctorates from Lund University, Sweden, the University of Antwerp, Belgium, and Tampere University, Finland. He was a recipient of major awards, among which the IEEE Mac Van Valkenburg Award, the European Design Automation Association (EDAA) Lifetime Achievement Award, the Semiconductor Industry Association (SIA) University Researcher Award, and the SRC Aristotle Award.