OpenHD: A GPU-Powered Framework for Hyperdimensional Computing

Jaeyoung Kang, Behnam Khaleghi, Tajana Rosing, and Yeseong Kim

Abstract—Hyperdimensional computing (HDC) has emerged as an alternative lightweight learning solution to deep neural networks. A key characteristic of HDC is the great extent of parallelism that can facilitate hardware acceleration. However, previous hardware implementations of HDC seldom focus on GPU designs, which were also inefficient partly due to the complexity of accelerating HDC on GPUs. In this paper, we present OpenHD, a flexible and high-performance GPU-powered framework for automating the mapping of general HDC applications including classification and clustering to GPUs. OpenHD takes advantage of memory optimization strategies specialized for HDC, minimizing the access time to different memory subsystems, and removing redundant operations. We also propose a novel training method to enable data parallelism in the HDC training. Our evaluation result shows that the proposed training rapidly achieves the target accuracy, reducing the required training epochs by 4×. With OpenHD, users can deploy GPU-accelerated HDC applications without domain expert knowledge. Compared to the state-of-the-art GPU-powered HDC implementation, our evaluation on NVIDIA Jetson TX2 shows that OpenHD is up to 10.5× and 314× faster for HDC-based classification and clustering, respectively. Compared with non-HDC classification and clustering on GPUs, HDC powered by OpenHD, is 11.7× and 53× faster at comparable accuracy.

Index Terms—Brain-inspired Hyperdimensional Computing, Machine Learning, Edge Computing

1 Introduction

The growing demand for real-time data analysis has accelerated the trend of on-device machine learning (ML) systems. Deep learning (DL) is a popular method to extract higher-level features but running those workloads on conventional systems results in high energy consumption and slow processing speed. Furthermore, mobile devices need to perform online learning tasks in various environments, but data can be easily corrupted or vulnerable to noise.

The limitations lead researchers to seek alternative computing methodologies for lightweight learning. Brain-inspired hyperdimensional computing (HDC) is such an alternative solution based on a long-term memory model, Sparse Distributed Memory (SDM) [1], which emulates human cognition with vector operations in a high-dimensional space. A key difference from the conventional computing method using boolean and numbers is that the HDC works based on a vector type, called hypervector. A hypervector includes the pattern of information encoded with tens of thousands of dimensions. For example, when having multiple items in a dataset to learn, in HDC, we first encode each item with different hypervectors. We then can perform various cognitive operations with hypervector arithmetic, e.g., adding multiple hypervectors to memorize them and learning its dominant pattern with a single hypervector.

Researchers have been presented diverse learning applications based on the HDC method with high accuracy and efficiency, e.g., DNA sequencing [2], language recognition [3], robotics [4], activity identification and voice recognition [5], multimodal sensor fusion [6], and bio-signal processing [7]. Several companies are also actively studying HDC to implement more generalizeable learning, e.g., Google [8], IBM [9], and Numenta [10].

Compared to the conventional DL, HDC shows several advantages: (i) the algorithm is highly parallelizable [11], (ii) it can perform training and inference with the lightweight computation, and (iii) the hypervector representation is robust against the noise [12]. However, since HDC deals with large hypervectors, the conventional CPU-centric architecture is not a suitable system to run such computation. HDC is easily parallelizable and can benefit from hardware accelerators. Earlier work has shown new hardware accelerator designs such as ASIC [3], [11], [13], and in-memory computing accelerator [14]. These works have shown promising speedup and energy efficiency over the CPU-based HDC.

Nevertheless, the hardware designs are not commonly available and need a relatively long period to synthesize and fabricate after deriving the new applications. HDC applications require flexibility, e.g., hypervector lengths, precision of operations, and encoding algorithm, which are infeasible in ASIC [11] and PIM [15] implementations. It is also essential to support high-precision numbers as the state-of-the-art HDC algorithms benefit from floating-point encoding to improve accuracy [16], [17], [18] which significantly reduces FPGAs efficiency. Besides, GPU can offer programmability with high parallelism. Even though GPU generally shows high power consumption, the off-the-shelf low-power embedded GPU platform is available in the market, such as NVIDIA Jetson.

To run the HDC on the GPU, one could integrate HDC with the parallel computing models oriented to vector processing, e.g., TensorFlow. Recent work [11] used TensorFlow on embedded GPU, but it showed a comparable speed to

- *J. Kang, B. Khaleghi, and T. Rosing are with the University of California San Diego.*
- *Y. Kim is with Daegu Gyeongbuk Institute of Science and Technology.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2022.3179226, IEEE Transactions on Computers

CPU-based implementation. In our experiment, TensorFlow XLA-based HDC classification running on NVIDIA Jetson TX2 was only 1.74× faster than CPU, with a low GPU utilization of at most 40%. It creates a lot of HDC data transfer between CPU and GPU, which causes a memory bottleneck. Furthermore, TensorFlow-based HDC often fails to effectively leverage the memory hierarchy on the GPU since HDC data are large-sized and have specific memory access patterns.

Our previous work, XCelHD [19], introduced a way to optimize HDC classification on GPU, achieving up to 35× speedup over the TensorFlow-based HDC implementation. However, it fails to achieve optimal performance when data characteristics, device architecture or HDC algorithm change. A hand-optimizing kernel requires a lot of effort and is not flexible to data characteristics, HDC algorithm, and hardware changes. As such, it is critical to find optimal configurations and parameters by adaptively reacting to data, hardware, and HDC design rather than relying on a hand-optimized kernel. To this end, it is essential to develop a new library specifically designed for HDC, to leverage the parallel computation pattern in an efficient, flexible, and automated manner.

In this paper, we present a GPU-powered HDC framework, dubbed OpenHD. Unlike existing HDC accelerators based on FPGA and ASIC, OpenHD supports various types of HDC-based algorithms including classification and clustering. The proposed solution offers a software interface capable of automatically generating optimized CUDA code of commonly used HDC applications with the JIT compilation to ease the development and optimizing efforts. OpenHD adopts various optimization strategies that effectively mitigate the low parallelization and memory access issues of the existing HDC-based applications. Furthermore, we propose a highly parallel HDC training, called PARTRAIN that elevates the GPU parallelism by redesigning the HDC training algorithm. It converts the sequential HD training process into multiple simultaneous subtasks, each of which learns partial models with different training samples while updating the final model based on the partial models. Our contributions are summarized as follows:

- We present OpenHD for GPUs, which supports various HDC-based learning tasks. In this paper, we apply OpenHD to two representative HDC-based algorithms: classification and clustering. OpenHD efficiently parallelizes the HDC learning procedure in a GPU-friendly way while intelligently optimizing memory allocation/access patterns.
- Our OpenHD framework automatically extracts the HDC-related parts from a given Python program, maps them to highly-optimized CUDA code, and compiles them using JIT compiling. It allows users to implement various GPU-accelerated applications with HDC philosophy even when they are unaware of the detailed acceleration mechanism on the underlying GPU subsystems.
- We propose a novel HDC training method for high parallelism. It enables full utilization of hardware resources and reduces the number of required training epochs by $4\times$ on top of the GPU-accelerated HDC.
- Our strategies specially designed for GPU maximize data reuse and enhance cache utilization, additionally accelerating the prediction (up to $12\times$) and the encoding step (up to $1.94\times$) in HDC with simple parallelization.
- We show that OpenHD achieves comparable accuracy to the state-of-the-art deep neural network (DNN)-based classification and clustering (K-means) algorithms running on the same GPUs, but at much higher speed.

We implemented and evaluated OpenHD with various datasets using NVIDIA Jetson TX2, which aims to low-power edge devices. Our evaluation results show that HDC-based classification has a 9× smaller model and runs 11.7× faster while offering comparable accuracy to DNN-based method. HDC-based clustering is 53× faster than the K-means algorithm with comparable quality. Moreover, our results indicate that OpenHD is up to 10.5× and 314× faster than the state-of-the-art classification and clustering HDC implementation running on the GPU, respectively.

The rest of the paper is organized as follows: In Section 2, we elaborate on the basics of HDC with a discussion of HDC applications. We then describe the software interface of OpenHD in Section 3 with the usage. Subsequently, in Section 4, we provide the details of our optimization strategies for GPU-based HDC. Next, we present experimental results of the OpenHD framework in accelerating the HDC applications with discussions for the state-of-the-art ML/HDC methods in Section 5. In Section 6, we introduce existing works that deal with automated GPU acceleration and HDC acceleration on various hardware platforms. Finally, Section 7 concludes the paper with discussion.

2 BACKGROUND

2.1 Preliminaries of HD Computing

Based on the understanding that brain events involve the simultaneous activity of a massive number of neurons, HDC models it with vectors in a high-dimensional space [20]. This high-dimensional vector is called *hypervector*, and HDC processes data in the unit of hypervectors.

Hypervector: The human memory associates different information and comprehends the relationship between them. HDC represents each datum with a *hypervector* and measures the correlation with the distance in the high-dimensional space. For example, let there exist two hypervectors with D-dimensionality, and each has random components between $+1$ and -1, i.e., $\{+1, -1\}^D$. In the vector space, the two bipolar hypervectors are near-orthogonal. We can represent two distinct items using the two randomly generated hypervectors. Generally speaking, finding the relationship and distinguishing information is done by measuring the similarity between a pair of hypervectors, $\delta(\overrightarrow{H}_1, \overrightarrow{H}_2)$ where $\overrightarrow{H}_1$ and $\overrightarrow{H}_2$ are two hypervectors. Existing works primarily use three different metrics: hamming distance for binary hypervectors, dot product similarity, and cosine similarity for non-binary hypervectors [21]. Note that the hypervector is a holographic representation of information in that there is no specific important dimension in a hypervector. This independence property of hypervector allows resiliency to corruption in components.

Addition (bundling) /Multiplication (binding): Using the element-wise addition and multiplication between hypervectors, we can *memorize* and *associate* different information. The element-wise addition operation, called bundling,

produces a hypervector that preserves all similarities of the combined members, mimicking the memorization functionality. The element-wise multiplication, called binding, results in a hypervector that associates two different hypervectors. The result hypervector represents new information, which is near-orthogonal (dissimilar) to both of the associated hypervectors.

Permutation: To process sequential information, the HDC has an operation called permutation. The permutation operation, which is denoted by $\rho^n(\overrightarrow{H})$, shuffles components of $\overrightarrow{H}$ with n-bit(s) rotation. The results of the operation creates a near-orthogonal, i.e., $\delta(\rho^n(\overrightarrow{H}), \overrightarrow{H}) \simeq 0$ for non-zero n. Also, the results of permutation is reversible, meaning that $\rho^{-n}(\rho^n(\overrightarrow{H})) = \overrightarrow{H}$. Hence, the permutation can be utilized to ordered information. For instance, $\rho^1(\overrightarrow{H}_1), \rho^2(\overrightarrow{H}_2)$ and $\rho^3(\overrightarrow{H}_3)$ may represent a sequence of three words.

Flip: To create hypervectors that maintain a certain degree of similarity, HDC uses the flip operation. Let $\overrightarrow{H}$ be a hypervector randomly sampled. If we flip $D/2$ elements, e.g., changing the sign bit of -1 and $+1$, the result hypervector is orthogonal to the original hypervector; flipping $D/4$ elements creates a hypervector which is 50% similar to the original $\overrightarrow{H}$ in the vector space. Using this property, we can represent different levels of information ranging from 0% to 100%. We first create a random hypervectors for 0%, which is denoted by $\overrightarrow{H}_0$. Then, for the target level p in percentage, $\overrightarrow{H}_p$, we can represent this by flipping $(D/2) \times (p/100)$ elements of $\overrightarrow{H}$.

With these simple hypervector operations, prior research has shown that it is possible to implement diverse cognitive tasks. In the following section, we discuss how they solve the classical ML problems based on HDC.

2.2 ML with HDC

Classification: In [2], [5], [22], [23], [24], the authors employ HDC to perform classification. It consists of the encoding, training, and inference stage. The classification was the first application tested using HDC. Some examples of various types of classification results are recently published in [5], [22], [23], [24]. It consists of the encoding, training, and inference stages.

Encoding maps (encodes) real-world data into the HD space. There are various encoding methods. One of the most popular ones is the ID-level method [5], [14], [22], [25]. For a given feature vector, this encoding method represents each feature value using the flip operation and each feature position using a randomly generated hypervector. It then combines the hypervector pair of the feature position and value using the element-wise multiplication. In particular, we encode an input data to an *input hypervector* $\overrightarrow{I}$ as follows:

$$\overrightarrow{I} = \sum_f \overrightarrow{ID}_f \odot \overrightarrow{LV}_m \tag{1}$$

where $\odot$ indicates an element-wise multiplication. Here, $\overrightarrow{ID}_f$ indicates *ID hypervectors* which assigned to an individual fth index of feature. We generate *level hypervectors* $\overrightarrow{LV}_m$ to capture different values m in fth index of feature. The ID-level method is more complex but also encompasses all the steps of random projection (RP) encoding, which means that any implementation that can do ID-level can also implement the RP [5].

The **training** stage builds *HD model* (class hypervectors). We initialize HD model by combining all hypervectors $\overrightarrow{H}_i^k$ belonging each class k using the element-wise addition, i.e., $\overrightarrow{C}_k = \sum_i \overrightarrow{H}_i^k$ where $\overrightarrow{C}_k$ indicates class hypervector for class k. Iterative *training* is used to achieve higher accuracy. HDC calibrates class hypervectors based on the prediction using the current state of class hypervectors. To be specific, it checks the ground truth label is equivalent to the predicted class. We can make predictions by selecting a class that has the maximum similarity between the input hypervector and class hypervectors. We use either Hamming distance (binary hypervectors) or cosine similarity (non-binary data). If the model gives an incorrect prediction, we update the class hypervectors by subtracting the encoded hypervector from the wrongly predicted class and adding the pattern again to the correct target class hypervector to amplify the pattern. We can apply the learning rate ($\eta \in (0, 1]$) for scaling the input hypervector during the update [26]. Here, we define *epoch* as an iteration over all data.

The **inference** stage maps test data into query hypervectors using the same method in the encoding stage. The class that shows the maximum similarity to query hypervectors is selected as the prediction result.

Clustering: Earlier work [27] proposed an HDC-based clustering algorithm. After mapping raw data to the high-dimensional space, their algorithm performs clustering by iterating two stages: similarity computation and cluster center update. The algorithm aims to learn the representative hypervector for each cluster head. A cluster head with the highest similarity is assigned to each data. For the cluster center update stage, the element-wise addition among data that points to the same cluster head is performed. Then, each component of the hypervector is binarized, and that hypervector is assigned as a representative hypervector of the cluster head.

3 OpenHD Framework

OpenHD provides a runtime environment to implement HDC applications seamlessly integrating with GPU for high efficiency. The OpenHD software interfaces reduce the user's need for understanding the GPU details, and minimizes the need for hand-optimization. It also enables implementing various machine algorithms of HDC. The abstraction layer of OpenHD automates the memory allocation of GPU memory and execution of the accelerated code. OpenHD uses Python as the front0end with NumPy-style syntax. At runtime it automatically generates optimized CUDA code to accelerate the implemented HDC applications on the state-of-the-art GPU systems.

Programming interface: To ease the understanding of how users can implement HDC applications with OpenHD, Fig. 2 shows a Python example code automatically accelerated on GPU. As shown in the example code, users can create either hypermatrix, i.e., an array of hypervectors, or hypervectors, and performs hypervector arithmetic operations, regarding that (i) they are built in the OpenHD Python library and (ii) will be accelerated automatically on their system. When the `@hd.run` decorator is applied to the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2022.3179226, IEEE Transactions on Computers

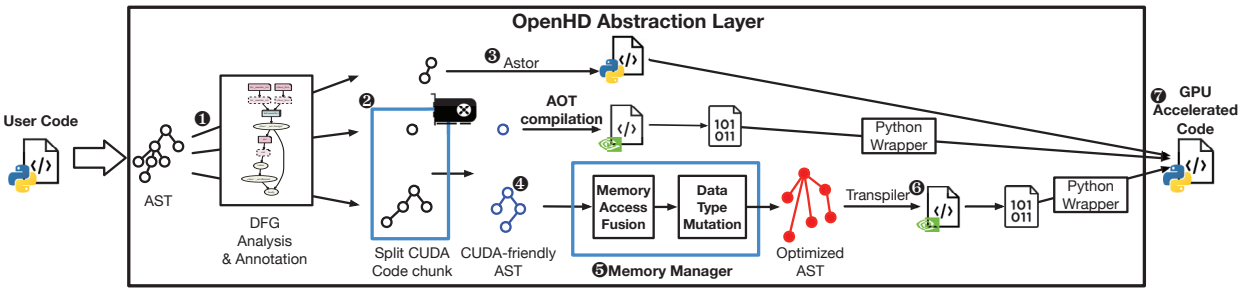


Fig. 1: Overall execution workflow of OpenHD

```
1  hd.init(D=D, context=global())
2  @hd.run
3  def add_row(F):
4    id_base_hx = hd.hypermatrix(F)
5    id_test = hd.draw_random_hypervector()
6    for f in range(F):
7      id_base_hx[f] += id_test
8      id_base_hx[f] += id_test
9    return id_base_hx, id_test
```

Fig. 2: Example implementation using OpenHD. `@hd.run` decorator passes the entire function component to the abstraction layer of OpenHD. `add_row` adds the hypervector `id_test` for each row of hypermatrix `id_base_hx`.

code (Line 1), the abstraction layer of OpenHD performs JIT compilation. It generates highly optimized CUDA code based on HDC characteristics and compiles using the CUDA compiler installed locally. Also, it finds the optimized parameters (kernel configuration) required for execution and binds the execution file to the Python front-end through PyCUDA [28]. Through this library, users can run GPU-accelerated HDC applications with less effort in a highly optimized fashion.

3.1 OpenHD Workflow

Fig. 1 illustrates how the abstraction layer of OpenHD handles the given Python code. Python code is transformed into abstract syntax trees (AST) by the built-in library. In the process of scanning the AST, OpenHD annotates each node of the AST to identify whether it is a JIT-compatible tree. Then, we build a data flow graph (DFG) using the annotated AST (❶). One of the goals of the abstraction layer is to decide the code is to be offloaded or not. In other words, we need to find code chunks that can be efficiently handled with the GPU (❷). Using the results obtained from the static analysis of the AST, we can distinguish those two. First, the code component that needs to be executed on the CPU is regenerated as python code from a node tree using Astor library [29] (❸). The remaining code is converted into the optimized CUDA code. OpenHD's CUDA code generator takes into account the GPU memory hierarchy (e.g., global memory, shared memory, constant memory) in order to maximize performance and efficiency.

The AST of the corresponding Python code fragment cannot be used as-is for the CUDA code generation (❹). It needs to be converted into a CUDA-understandable AST. Also, we need to consider the hardware specifications and input data characteristics to generate optimized code. After this step, AST addresses in the annotation nodes are invalidated. Note that the code fragment that handles HDC-specific operations such as permutation and shuffle is substituted with a single HDC operation. Before generating code, the memory manager of OpenHD (❺) considers access fusion to minimize global memory access. Furthermore, we apply data type mutation to save memory allocated to variables and maximize memory coalescing by analyzing and managing the minimum data type required by each variable in OpenHD.

CUDA-compatible AST is transpiled into actual compilable code by the transpiler of OpenHD (❻). Since the generated code cannot be executed standalone, the transpiler adds the required headers and definitions automatically. Then, the final CUDA code is wrapped with PyCUDA-based Python code. Finally, the abstraction layer binds the CPU side code and the GPU code (❼). Instead of running the code generation for all code wrapped with `@hd.run` every time, OpenHD caches the generated code for future reuses. If the same code requests for the JIT compilation, OpenHD intelligently utilizes previously compiled code.

3.2 Data Type Definition

HDC's data units are hypervectors. Thus, we defined two additional built-in data types in OpenHD: *hypervector* and *hypermatrix*. Hypervector is a high-dimensional vector with arbitrary D dimensionality. Hypermatrix corresponds to the multiple hypervectors; it can be used to express class hypervectors. HDC uses a fixed dimensionality during application execution. In OpenHD, users declare it in an initializing function that applies to the global scope, and it is used in the subsequent process. The result of an operation like the similarity computation may not follow the hypervector or hypermatrix. In OpenHD, users do not need to declare a variable to store the result explicitly. Instead, variables reside in GPU memory for later computation, and they can be copied to the host through the `to_numpy()` method.

Deploying the hypermatrix/hypervector as built-in data types creates a serious design challenge – hypervectors usually have a large data size, incurring both high communication and computation costs on the GPU subsystems. In the rest of this section, we describe the representative optimization schemes applied in OpenHD.

3.3 Access Fusion

To improve kernel performance, it is essential to reduce latency by minimizing access to GPU off-chip global memory, i.e., GDDR. The TensorFlow-based implementation assumes

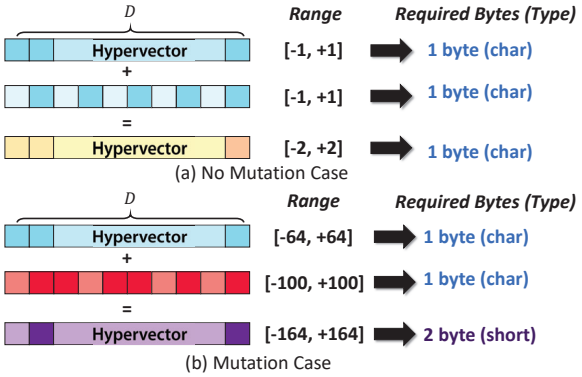


Fig. 3: The example of data type mutation

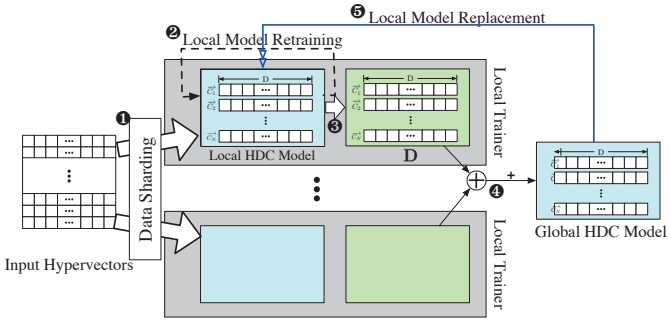


Fig. 4: Overview of the proposed parallel training method

a hypervector as a tensor, so each line of code is represented by a node. Therefore, access to the global memory occurs every time the line is executed, leading to reduced performance. OpenHD uses the NumPy-style syntax for ease of programmability. If the AST is used as-is, the same problem may arise. Therefore, in OpenHD, coping with how the user wrote the code, the global memory access corresponding to hypervector or hypermatrix fuses through *memory access fusion* to reduce the memory access. For example, Line 7–8 of the code in Fig. 2 is fused to a single kernel so that the two consecutive memory accesses are replaced with a global memory access and a register access.

Given the CUDA-understandable AST, OpenHD generates a control flow graph and traverses it. The variable node that reads or writes occurs is recorded individually, and we can check a propagable path. For example, if multiple reads or writes occur in the same variable without dependency, those nodes can be fused. We reflect changes to the corresponding node in the AST.

3.4 Automated Data Type Mutation

For some GPU-powered embedded devices, the CPU and GPU may share the same system memory through unified virtual addressing (UVA), and it is essential to reduce the memory footprint. In HDC, the operation is performed on a high-dimensional vector as a unit. If the components of the hypervector use a data type with a large size, the amount of memory occupied by a single hypervector will increase accordingly. To reduce the memory footprint caused by hypervectors, we applied *automated data type mutation* optimization technique that inferences for the data type of hypervector components.

Fig. 3 shows how OpenHD tracks the range of the hypervectors to verify when to mutate the data type, saying the no mutation case (a) and mutation case (b). The first hypervector created in the HDC application is $\{+1, -1\}^D$ where D is the dimensionality of the vector. Thus, it starts from a vector of `char` type (1 byte), and the required range changes as HDC addition or multiplication are performed in the subsequent process. For example, for a variable of `char` type, the minimum required data type may increase in the order of `short` (2 byte), `int` (4 byte), and `long` (8 byte) during iteration.[1] Therefore, this optimization module tracks the range of variables along with the dependency graph in kernel units. We hereby track three types: variable nodes, conditional statement, and the dominator. Finally, the data type in nodes (the operation inside the kernel, the return type of the kernel, and the input type of the subsequent kernel) in the AST is overridden with the inferred one.

1. It may vary depending on the system environment.

3.5 Optimization of Built-in Functions

OpenHD supports HDC's standardized operations as an API (see Table 1). It reduces syntax analysis and code generation time. During the code generation process, these operations are unnested from AST and compiled in the ahead-of-time (AOT) as runtime library APIs.

Permutation/shuffle operation accompanies non-coalesced memory access if source data reside on global memory. For shuffle operations where many reading transactions can occur, we place the source hypervector in the constant memory. Even if a user uses multiple permutations, the memory access fusion module combines operations into a single kernel. Thus, it avoids performance degradation by storing intermediate results in registers.

4 OpenHD Optimization Techniques

4.1 PartTrain: Parallel Training for HDC Classification

The iterative training enhances the HDC model accuracy. In contrast to the HDC encoding, the existing iterative training process has limited parallelism in nature since it feds data sequentially. In other words, the hardware can be underutilized and become a bottleneck of the HDC application. We propose PartTrain, which enables a parallel iterative training process and expedites model refinement by maximizing hardware utilization.

Fig. 4 illustrates the design of PartTrain. We spawn *local trainers*. Multiple local trainers can be created until it fills up available hardware resources. Each local trainer contains two sets of C hypervectors, where C is the number of classes. The first set contains a local HDC model while the other stores the changes of the local model. We initialize a local HDC model using single-pass training and the other set with zeros. PartTrain aims to maximize the data parallelism during the training. We split the data equally according to the number of local trainers (❶). Each local trainer refines the local HDC model in parallel (❷) and accumulates changes of it to $\overrightarrow{D}$ (❸). After training with assigned datapoints, we update the global model by calculating $\overrightarrow{G} = \sum_i \overrightarrow{D}_i$ (❹). Note that i indicates ith local

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2022.3179226, IEEE Transactions on Computers

TABLE 1: The OpenHD API examples

Function	Semantics
`hd.init(D, context)`	Initialize HDC application with D dimensionality in designated context (e.g., `global()`)
`hd.hypervector()` / `hd.hypermatrix(N)`	Declare hypervector or hypermatrix with N hypervectors.
`hd.draw_random_hypervector()` / `hd.draw_gaussian_hypervector()`	Generate a random hypervector from uniform distribution or normal distribution
`hd.permute(T, N)`	Perform HDC permutation to hypervector T
`hd.shuffle(T, S)`	Randomly shuffles hypervector T with seed S
`hd.search(C, A)` / `hd.sim(C, A)`	Perform associative search between class hypervectors C and hypermatrix or hypervector A
`hd.cos(A)` / `hd.sign(A)`	Apply element-wise cosine / sign flip on hypermatrix or hypervector A

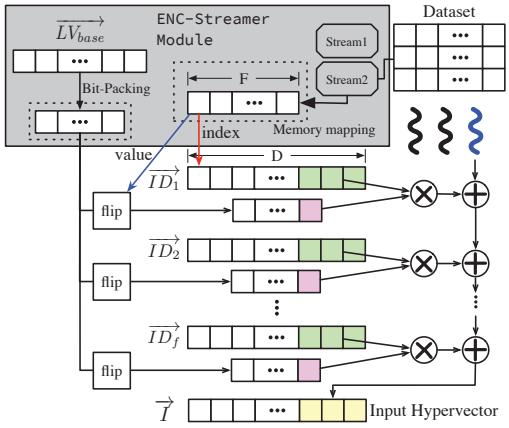


Fig. 5: Design of ENC-STREAMER-applied HDC encoding

trainer. The global model $\overrightarrow{G}$ replaces all local HDC models (❺). The whole process is defined as *one epoch*.

Note that PARTRAIN is different from data-parallel training used in DNNs wherein gradients are aggregated from multiple GPUs, which is equivalent to having a larger batch on a larger GPU. In contrast, PARTRAIN performs local model updates before the global model aggregation. Hence, the training result (model) of PARTRAIN is different from the sequential HDC training as all updates are postponed to the end of the epoch (even the local models are not being sequentially updated as opposed to the conventional HDC algorithm).

4.2 HDC-Based Memory Optimization

OpenHD uses two techniques to optimize memory transaction which is the main challenge in HDC that uses large-sized hypervectors. ENC-STREAMER optimizes the encoding stage of HDC and L2-RECYCLE optimizes the training (local trainer in case of PARTRAIN).

HDC Encoding Optimization ENC-STREAMER allocates data to the proper GPU memory hierarchy to achieve high efficiency during encoding and inference stages. The encoding is the bottleneck of the HDC application, as it takes up to 70% of the overall runtime of HDC on CPU [30]. Here we focus on one of the popular HDC encoding techniques, the ID-Level encoding, but the same strategy can be applied to a simpler RP method. A naive way to implement the ID-Level encoding on GPU is to parallelize over datapoints and each dimension of input hypervectors.

On top of the parallelization, ENC-STREAMER enhances memory transactions by leveraging GPU memory hierarchy. The GPU memory hierarchy includes several memories, e.g., global memory, shared memory, and constant memory, which have different characteristics in terms of size and latency. The execution time varies depending on a data allocation strategy.

Fig. 5 shows the HDC encoding module with ENC-STREAMER. We use two streams to overlap the computation and the memory transfer. Two streams use the same optimization techniques synchronously and runs until all datapoints are covered in an interleaved fashion. Furthermore, ENC-STREAMER applies an additional caching strategy on raw features and level hypervectors, which is a frequently used term as shown in Eq. 1. We cache raw feature values on the shared memory to enhance memory coalescing. ENC-STREAMER optimizes access level hypervectors. The access to the level hypervector is irregular as intensity values in raw data are random. Since the level hypervector is generated using flips in a regular way, we can generate $\overrightarrow{LV}$ in-place. Hence, ENC-STREAMER stores 0-th level hypervector, $\overrightarrow{LV_{base}}$, on the constant memory as it is used frequently and unchanged during the entire encoding process. We bit-pack the base-level hypervector and store it in the constant memory. Since hypervectors require a large size array, packing is essential to satisfy the constant memory constraint. Empirically, packing eight bits (`char`) showed the best performance. Each thread takes packed level hypervector (pink-colored boxes) and encodes the eight components of the input hypervector (yellow-colored boxes). We use encoded data for the rest of HDC application runtime, thus *completely eliminating the data movement costs*. For the GPUs with unified virtual addressing, we use a single stream and use managed memory since raw data requires one read transaction and does not have to be cached. This mechanism helps to alleviate the memory capacity limitation that is critical in embedded GPUs.

HDC Training Optimization Pairwise similarity computation is the next slowest component of HDC in both classification and clustering. Since OpenHD is specialized for non-binary hypervectors, we use cosine similarity metric which defined as $\delta(\vec{x}, \vec{y}) = (\vec{x} \cdot \vec{y})/(\|\vec{x}\| \cdot \|\vec{y}\|)$. To expose higher parallelism, we separately compute the numerator and the denominator, which can be done using parallel reduction technique on large dimensionality. However, it showed marginal speedup over the CPU-based implementation.

In addition to parallelization, L2-RECYCLE minimizes the memory access by identifying *invariant calculation results* during the similarity computation. It computes the L2 norm

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2022.3179226, IEEE Transactions on Computers

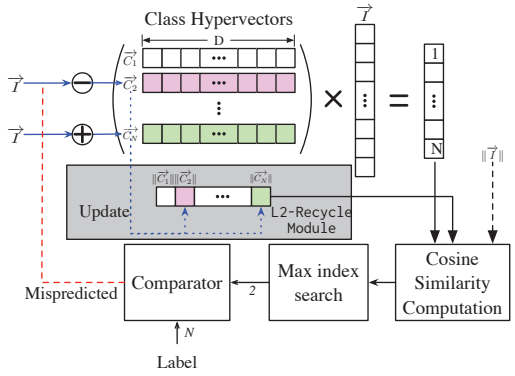


Fig. 6: Design of L2-RECYCLE-applied training module in HDC-based classification

components only when the corresponding hypervector is updated. Fig. 6 illustrates the training in HDC classification with L2-RECYCLE. Since the training procedure computes the similarity ($s[k]$) between the k^{th} class hypervector and input hypervectors, we *pre-compute* their L2 norms when updating them in the training loop. It separately stores L2 norm of class hypervectors. Our strategy removes unnecessary computation and minimizes memory access to the large-sized hypervectors.

5 EVALUATION

5.1 Experimental Setup

To verify the practical value of OpenHD, we implemented HDC-based classification [5] and clustering algorithms [27]. We evaluated the implementation of the CPU and GPU in the NVIDIA Jetson TX2 device with Jetpack 4.4.1 SDK. We set the dynamic voltage and frequency scaling governor.

Comparison to the state-of-the-art ML We first compare the HDC-based classification and clustering with the optimized DL model generated by AutoKeras [31] and K-means algorithm [32], in terms of quality, model size, and runtime.

HDC-based Classification We compare the execution time of our OpenHD-based implementation with the TensorFlow 2-based HDC and the state-of-the-art GPU-powered HDC classification, XCelHD [19]. Note that we used our TensorFlow 2-based baseline since the existing TensorFlow-based HDC [11] has higher per-inference energy than ours. For a fair comparison, we measure the speed of the second run since OpenHD caches and reuses function chunks once compiled. We discuss the details about the compilation overhead in Section 5.6. Next, to show the energy efficiency of OpenHD, we compare the energy consumption with the CPU-based HDC classification and clustering implementation which uses Python with a C++ backend to take full advantage of the parallel capabilities of SIMD. We included the communication time, e.g., memory copy time, between the GPU and the host. For the power consumption measurement, *tegrastats* utility is used. We set fixed quantization level Q to 100, and epochs to 20. The hypervector dimensionality, D, is set to 10,000 and 4,000. Also, PARTRAIN-powered OpenHD is configured with 10 local trainers.

To observe the performance focusing on the actual use-case of HDC-based classification, we evaluated the implementation on a wide range of following benchmark datasets [33], [34]. **CARDIO**: a medical diagnosis based on patients' information, **UCIHAR**: detecting human activity based on 3-axial linear acceleration and angular velocity data, **ISOLET**: recognizing audio of the English alphabet from different people, **FACE**: classifying images with faces/non-faces, and **PAMAP2**: classifying five human activities based on a heart rate and inertial measurements. Attribute information for these datasets is listed in Table 2.

TABLE 2: Statistics of the datasets

Dataset	Features	Classes	Training set	Test set
CARDIO	21	2	1913	213
UCIHAR	561	12	6213	1554
ISOLET	617	26	6238	1559
FACE	608	2	21441	2494
PAMAP2[2]	75	5	22500	22500

HDC-based Clustering We compare the execution time of our OpenHD-based implementation and the TensorFlow-based HDC clustering running on the GPU. We used ten epochs in our HDC-based clustering algorithm. The clustering phase can be early-terminated as the HDC model generally converges before ten epochs. However, we fixed the number of iterations for a fair comparison. We evaluate the HDC-based clustering using the fundamental clustering problem suite (FCPS) [35] which addresses general challenges for clustering. We experiment with a subset of FCPS, including **Hepta** (consists of six classes with different inner class variances), **Tetra** (dataset with small inter class distances), **TwoDiamonds** (two classes with touching classes) and **WingNut** (two classes with inter class density variation). We also experimented with the **Iris** [33] pattern recognition dataset.

5.2 Comparison of HDC to State-of-the-art ML

To compare HDC-based and DNN-based classification, we run the neural architecture search (NAS) for the DNN model with the maximum accuracy using AutoKeras [31]. The comparison between HDC and the most accurate DNN model found is done on the NVIDIA Jetson TX2. The HD model is trained for ten epochs, and the execution time for the NAS is excluded. For the comparison of HDC-based clustering, we compare it with the K-means algorithm running on GPU.

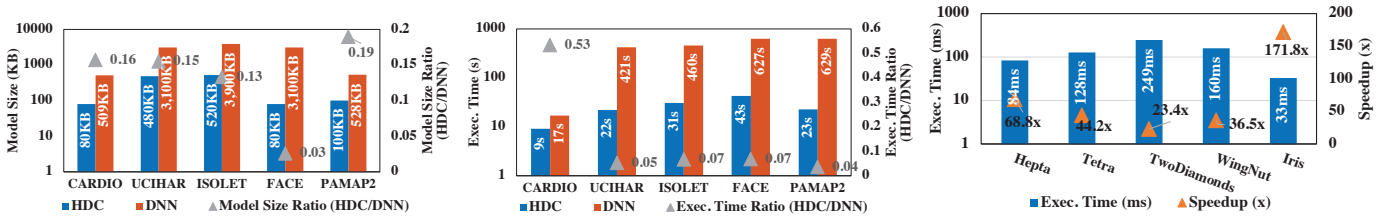
Classification accuracy HDC-based classification has a comparable accuracy DNN-based classification. The former offers 95.5% of accuracy on average, while the latter shows 94.7% on average. HDC-based classification accuracy remains stable once sufficient dimension size is reached [36]. Additional dimensionality can be used for better noise resilience [20].

Classification model size Fig. 7(a) illustrates the HDC-based and the DNN-based model size. HDC is on average 9× smaller than the DNN-based models. By decreasing D, The HDC model can reduce the model size by 20% with an accuracy drop of less than 1%.

Classification training and inference time As depicted in Fig. 7(b), HDC-based classification runs 11.7× faster

2. We randomly sampled from the original dataset as it cannot be fitted to the Jetson TX2 memory.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2022.3179226, IEEE Transactions on Computers



(a) (b) (c)

Fig. 7: Comparison between OpenHD-based HDC and the state-of-the-art ML method. (a) Model size of DNN-based vs. HDC-based classification (b) Runtime of DNN-based vs. HDC-based classification (c) K-means vs. HDC-based clustering

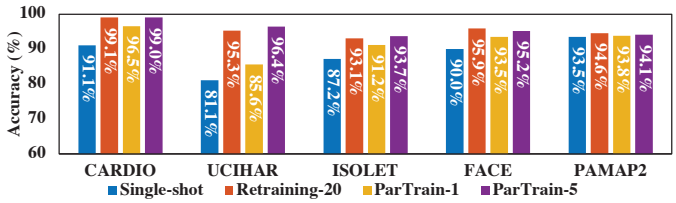


Fig. 8: Accuracy comparison between PARTRAIN and the conventional HDC training method.

than the DNN-based method on average. While existing DNN-based applications in embedded devices focus only on optimization of inference through TensorRT [37] and TVM [38] libraries, HDC-based solutions enable both lightweight training and inference in embedded devices.

Clustering accuracy, speed, and efficiency To evaluate HD Clustering, we set D to 1024, as it gives a comparable normalized mutual information score to K-means. This is a significantly smaller hypervector dimensionality as compared to classification, resulting in much smaller model sizes. The score difference was within 3.1% on average. Fig. 7(b) illustrates the execution time ratio between K-means and HDC-based clustering running on GPU. HDC-based clustering is 53× faster on average compared to K-means, at nearly the same power consumption.

5.3 Accuracy of PARTRAIN

PARTRAIN enhances data parallelism of HDC-based training using multiple trainers. Thus, it can look up more data than the traditional HDC training technique in the same period. We compared PARTRAIN-enabled HDC classification to the existing training method. In particular, two baselines are used: single-pass training (Single Shot) and iterative training with 20 epochs (Retraining-20). We also observed PARTRAIN with different epoch settings, with single (PARTRAIN-1) and five (PARTRAIN-5) iterations. Fig. 8 shows the achieved peak accuracy with different training settings. PARTRAIN-5 case achieves a similar accuracy compared to the original HDC with 20 epochs. We can reduce required epochs by 4×. It implies that PARTRAIN can achieve the target accuracy with fewer epochs. Furthermore, the overhead of gathering information from local trainers was negligible, which is less than 5% of the runtime.

5.4 OpenHD Efficiency

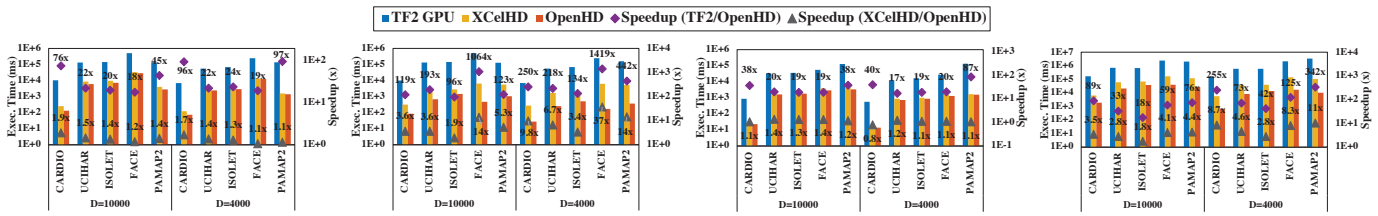
5.4.1 Speed Improvements

Classification: Fig. 9 compares the execution time of the HDC classification implemented with OpenHD, XCelHD and TensorFlow 2. Compared to the TensorFlow-based HDC, the encoding speed of XCelHD and OpenHD is improved by ENC-STREAMER which flexibly maps the data on the proper CUDA memory hierarchy. OpenHD adds optimization using data type mutation on top of XCelHD. Assigning data type reduces the memory footprint of HDC applications and allows storing more data in the local memory. Moreover, OpenHD enables efficient use of memory bandwidth which is significant for large-sized hypervector. Note that OpenHD efficiently leverages GPGPU memory hierarchy as encoding API of OpenHD adopts the optimization in XCelHD. As shown in Fig. 9(a), OpenHD-powered HDC encoding is on average 1.37× and 35× faster than the XCelHD and the TensorFlow 2-based HDC, respectively.

The iterative training stage mainly benefits from the memory access fusion technique and L2-RECYCLE. In addition to XCelHD with L2-RECYCLE, OpenHD further optimizes the TensorFlow-based HDC training with memory access fusion. The prediction is based on `hd.search` being AOT compilation, which is primarily benefitted by L2-RECYCLE. Also, during the update of class hypervectors, operations need to be written line-by-line in code due to the Python syntax. However, memory access fusion reduces global memory access overhead. The kernel is generated automatically according to the maximum number of epochs. The encoding stage includes the writing operation of the hypervectors only once. In contrast, we apply operations to the large-sized hypervector repeatedly. Memory access fusion greatly impacts the speedup compared to XCelHD, leading to a larger speedup. Note that more local trainers can be added on the OpenHD-based HDC since we assign the proper data type to the hypervector and reduce the memory footprint. As shown in Fig. 9(b), OpenHD provides on average 6.8× speedup over XCelHD and 258× speedup over TensorFlow-based HDC in the HDC training module.

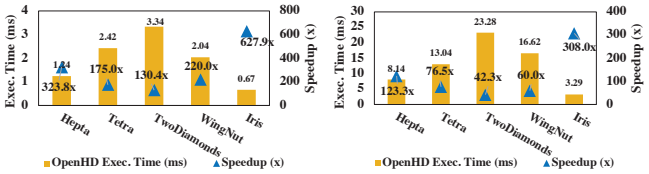
The inference stage consists of encoding the test set and similarity computation. Since OpenHD adaptively generates code, the memory mapping policy is changed according to the test set configuration. Then the similarity search operation for hypermatrix is applied by the `hd.sim` function. Since the encoding takes up a large portion of the inference stage, more than 99%, the trend was similar to the encoding stage (see Fig. 9(a) and (c)). In the case of the CARDIO dataset, the execution time of OpenHD was slower than the XCelHD. It is due to the communication overhead between the Python front-end and CUDA backend, while both implementations have similar optimizations. For other cases, OpenHD consistently showed speedup over base-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2022.3179226, IEEE Transactions on Computers



(a) Encoding (b) Training (per epoch) (c) Inference (d) End-to-end

Fig. 9: Speed comparison of OpenHD vs. the baseline HDC implementation



(a) Encoding (b) Clustering (per epoch)

Fig. 10: Speed comparison of OpenHD vs. Tensorflow GPU-based HDC clustering

lines. OpenHD-powered inference stage yields on average 1.17× and 27× of speedup compared to XCelHD-based and TensorFlow-based HDC, respectively.

Overall, on the HDC classification, OpenHD runs faster than baselines. In particular, OpenHD-based implementation shows the best improvement on the training stage. PARTRAIN helps to overcome limited parallelism and underutilization during the training process in GPU-based HDC. By adaptively generating CUDA code with regard to the characteristics of the dataset and the encoding strategy, the training and inference stage benefitted. For the end-to-end execution of the HDC process, OpenHD-powered implementation gains on average 4.5× and 78× over the state-of-the-art GPU-based HDC and TensorFlow-based HDC classification, respectively, as shown in Fig. 9(d).

Clustering: Fig. 10 shows the clustering speed improvements of OpenHD as compared to TensorFlow-based implementation for the encoding and the clustering stages. For these two stages we get 252× and 94× speedup on average. In the encoding stage of OpenHD, each thread loops over the features. The clustering dataset has a smaller number of features than the dataset used in the classification evaluation. Hence, while it shows a similar trend to Fig. 9(a), the execution time reduction is substantial. Every single clustering epoch, the algorithm loops over the data. Comparing the absolute value of the dataset size to the feature size, the former is significantly larger than the latter, leading to a longer execution time during the clustering. Therefore, it shows a more limited speedup than the encoding stage. For the end-to-end execution of HDC-based clustering, OpenHD-based implementation is on average 96× (up to 146×) faster than TensorFlow-based HDC.

5.4.2 *Energy Efficiency Improvements*

We compared the energy consumption of OpenHD to the state-of-the-art HDC classification running on the low-powered CPU. CPU-based HDC uses SIMD, which supports a smaller degree of parallelism than the hypervector dimensionality. Since it has limited parallel computing resources, we disabled PARTRAIN. Fig. 11 shows the energy efficiency improvement of OpenHD over the baseline. The encoding stage enables high parallelism and consumes more power, but reduced execution time improves the energy efficiency. The training module uses low power in nature due to limited parallelism. However, compared to the CPU-based HDC, L2-RECYCLE significantly reduces the execution time by removing redundant operations. It showed 125× energy consumption reduction on average. We disabled PARTRAIN in this experiment, but enabling it can enhance energy efficiency further. Even if it has more parallelism and consumes more power, the reduced number of iterations compensates for the increased power consumption. Ultimately, OpenHD achieves up to 172× energy efficiency improvement over the CPU-powered HDC classification.

5.5 Effectiveness of ENC-STREAMER and L2-RECYCLE

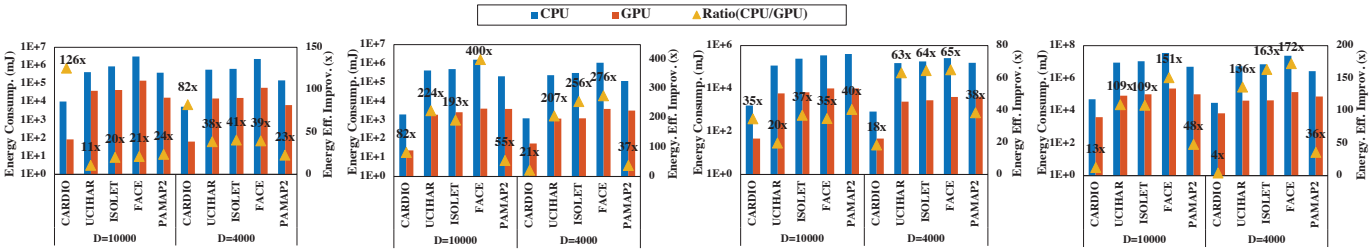
In this section, we observe the effectiveness of ENC-STREAMER and L2-RECYCLE by comparing the execution time with and without applying strategies. We set several scenarios to measure general effectiveness and fixed the dataset size to $10,000$. ENC-STREAMER mainly optimizes the encoding stage, which is affected by dimension size D and the number of features F. Our experiment conducted on four scenarios, (F, D): Case A) $(500, 10000)$, case B) $(500, 4000)$, case C) $(250, 10000)$ and case D) $(250, 4000)$. As shown in Fig. 12(a), ENC-STREAMER is most effective on large hypervector dimensionality, achieving $1.94\times$ in Case A. Moreover, ENC-STREAMER improves energy efficiency since the power consumption of the two variants is similar.

L2-RECYCLE is effective on the iterative training of HDC classification. We measured execution time by varying the number of classes C and the dimensionality D. We evaluated on four scenarios: Case A) $(10, 10000)$, case B) $(10, 4000)$, case C) $(20, 10000)$ and case D) $(20, 4000)$. On the large D, L2-RECYCLE showed significant speedup up to $12\times$ (see Fig. 12(b)). Without L2-RECYCLE, the execution time increases dramatically on large dimensionality. Our optimization strategy offers similar training time regardless of D. The overhead is minimal; in the worst case, case C requires a small additional memory cost of 80 Byte.

5.6 Code Generation Overhead

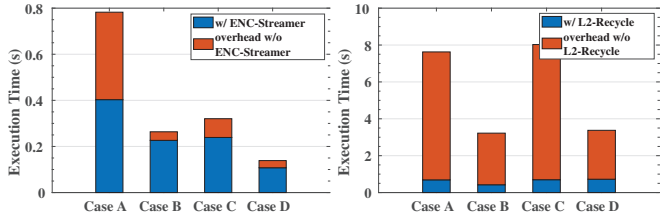
As OpenHD performs JIT compilation, there is an overhead of code analysis. Nevertheless, since OpenHD caches the compiled code chunk once, there is no overhead when rerun. We compared the execution time for the first run and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2022.3179226, IEEE Transactions on Computers



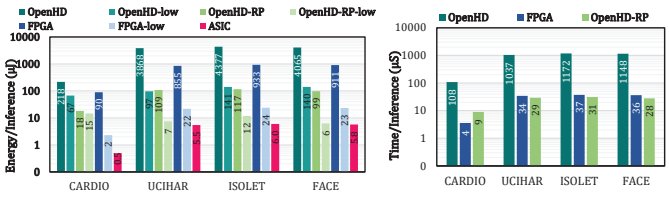
(a) Encoding (b) Training (per epoch) (c) Inference (d) End-to-End

Fig. 11: Energy consumption comparison of GPU (OpenHD) vs. HDC running on the embedded CPU



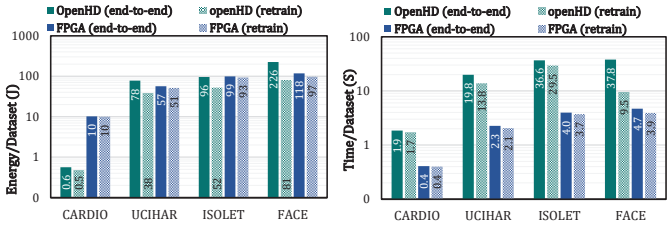
(a) ENC-STREAMER (b) L2-RECYCLE

Fig. 12: Execution time reduction by proposed optimizations



(a) Inference energy (b) Inference time

Fig. 13: OpenHD inference compared to FPGA and ASIC.



(a) Train energy (b) Train time

Fig. 14: OpenHD training compared to FPGA.

the second run to measure the overhead of JIT compilation. The overhead of code generation and compilation was at most 88%. Besides, the profiling results show that the compute utilization and throughput of GPU are improved by 1.72× and 1.82×, respectively. Since pre-compiled binaries are used in actual deployment, the performance efficiency of the HDC application can be maximized in the GPU environment without an effort for kernel optimization.

5.7 Comparison with Other Hardware Implementation

We compare OpenHD with an efficient FPGA design implemented and verified on Xilinx Alveo U200 accelerator card, and with the ASIC design of [11]. The FPGA implementation leverages RP encoding which relies on matrix-vector multiplication and is inherently more efficient, but less accurate in certain applications such as time-series. Therefore in Fig. 13 we also present a variant of OpenHD that uses RP, OpenHD-RP. Since the ASIC implementation achieves lower accuracy, for a fair comparison, we also compare OpenHD-low and FPGA-low, reduced-dimension GPU and FPGA that yield the same accuracy of ASIC.

Using 10,000 dimensions, the average per-inference energy consumption of OpenHD, which uses ID-level encoding, is 3.9× of FPGA. However, OpenHD-RP, that uses the same encoding as FPGA, consumes 7.3× less energy than FPGA. OpenHD-RP-low (reduced dimension with similar accuracy of ASIC) improves the energy by 1.35× with respect to FPGA-low, and only consumes 3.05× more energy than the full-custom ASIC. In higher dimensions, OpenHD-RP shows better improvement over FPGA (7.3× vs 1.35×). It is because FPGA uses on-chip SRAM memory with single-cycle access regardless of the dimensionality, so its execution and energy improves linearly in lower dimensions, while for GPU the data movement overhead manifests more significantly in lower dimensions (wherein computation cost is smaller).

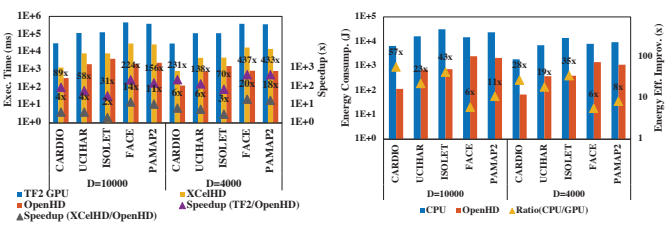
Fig. 13(b) compares the per-inference execution time of OpenHD and FPGA. Using the more-complicated ID-level encoding, OpenHD inference time is 30.8× higher than the FPGA, while OpenHD-RP is only 1.08× slower than the FPGA (with 7.3× better energy as alluded above). Note that the ASIC [11] has not provided execution time numbers, so we omitted it from comparisons.

Fig. 14 compares the training energy and execution time of OpenHD and FPGA for 20 epochs. OpenHD improves the end-to-end and iterative training (retrain)-only energy by 1.63× and 2.78×. The retrain-only energy does not include the encoding for which OpenHD consumes more energy due to using ID-level encoding. Therefore, the energy improvement without considering encoding is higher (with RP encoding, the end-to-end energy improvement would be even higher). Finally, the end-to-end and iterative retrain of OpenHD is, respectively, 7.4× and 4.9× slower than FPGA. Using RP encoding can improve the encoding hence the end-to-end time, but retrain time will improve intact as it deals with already encoded hypervectors. Note that Xilinx Alveo U200 is a high-performance FPGA and offers massive parallelism. Comparing with low-end or mid-range FPGA, we expect OpenHD could excel in performance, as well.

5.8 Evaluation on High-performance Hardware

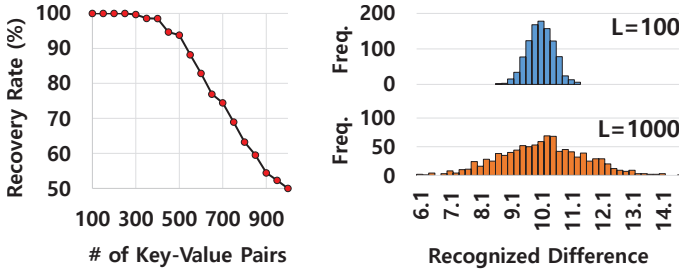
High-performance hardware can also be utilized to process HDC in the real-world; the workload can be offloaded to the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2022.3179226, IEEE Transactions on Computers



(a) Speedup (b) Energy Efficiency

Fig. 15: Improvement of OpenHD compared to baselines running on high-end hardware



(a) Key-Value Storage (b) Structure Comparison

Fig. 16: Stochastic nature of HDC for other cognitive tasks

cloud, or the user may develop a new HDC algorithm on the high-end GPUs. OpenHD is runnable on the desktop-class GPU since it uses CUDA as a backend. As such, we measured the execution time and energy consumption of OpenHD and baselines for HDC-based classification task on the high-performance system, Intel i7-8700K with 16GB RAM and NVIDIA Geforce GTX 1080Ti. We measured the power consumption using Intel PowerLog and `nvidia-smi` command.

Fig. 15 shows the speedup and energy efficiency improvement results for datasets tested on the high-performance hardware. On high-end hardware, OpenHD shows similar speedup trend to the embedded environment. For large tasks, OpenHD also shows a high degree of acceleration. Even if the high-end GPU consumes more power, due to the short execution time, the energy efficiency improvement is larger than that of the embedded device. Overall, our design achieves an average speedup of $6.7\times$ over XCelHD and energy efficiency improvement of $18\times$ on average over CPU.

5.9 Case Study: OpenHD on Other HDC Tasks

Since OpenHD is a framework to implement various applications using HDC paradigm, we can also implement non-ML applications. Here, we introduce two additional HDC application examples implemented with OpenHD to discuss potentials of HDC.

Key-value storage: With the element-wise addition and multiplication, we can effectively implement a hash table-like data structure (also accelerated on GPU for high efficiency): $\overrightarrow{H} = \sum_i \overrightarrow{K}_i \times \overrightarrow{V}_i$ where $\overrightarrow{K}_i$ and $\overrightarrow{V}_i$ are the encoded hypervectors for the key and value. The single hypervector includes multiple key-value relations. We can check if a target value $\overrightarrow{V}$ is associated with a key $\overrightarrow{K}$: $\delta(\overrightarrow{H} \times \overrightarrow{K}, \overrightarrow{V})/D \simeq 1$ when using the dot product for the similarity metric. Unlike the traditional hash table, the key-value structure can *categorize* information effectively. For example, let us give an analogous example – "which fruits are red?." To memorize the relationship between the fruit and color, we can encode each fruit and color to $\overrightarrow{K}_i$ and $\overrightarrow{V}_i$. If $\overrightarrow{Y}$ is the hypervector that represents the yellow color, $\overrightarrow{H} \times \overrightarrow{Y}$ is similar to all hypervectors associated with $\overrightarrow{R}$; near-orthogonal for other fruits. An interesting property of HDC is that the dimension size determines the amount of information that can be stored in a single hypervector.

Fig. 16 (a) shows the recovery rate according to the number of pairs in HDC-based key-value storage. Because the number of relations that can be distinguished by a single hypervector is limited, the recovery rate decreases as the number of key-value pairs increases. However, as the dimension size of the hypervector increases, more relations can be contained.

Structure comparison: The human memory easily recognizes differences in structured objects, e.g., chronological changes of train station connections due to constructions. Let us consider a problem that compares two lists where a list has different items. We may encode the structure of the list using adjacent items, say $\overrightarrow{L}_i$ and $\overrightarrow{R}_i$: $\overrightarrow{M} = \sum_i \overrightarrow{L}_i \times \overrightarrow{R}_i$. When we have two hypervectors encoded for the different lists, $\overrightarrow{M}_1$ and $\overrightarrow{M}_2$, we can identify the number of indels using a single similarity computation: $\delta(\overrightarrow{M}_2 - \overrightarrow{M}_1, \overrightarrow{M}_2)/D$.

Fig. 16 (b) shows the distribution of recognized differences according to the list length L when data is encoded and stored in a single hypervector. When a single hypervector contains more information, i.e., when L is large, it showed a lower resolution for a relation between data. In contrast, for small L, the HDC-based solution captures the difference well. However, as in the example of key-value storage, using a single hypervector is efficient regarding the data size. Also, the resolution issue can be mitigated by increasing the dimensionality of the hypervector.

6 Related Work

6.1 Automated GPU Acceleration

There are previous studies that accelerate algebraic operations on the GPU without users' awareness of the GPU. CuPy [39] offers GPU-accelerated application with NumPy-compatible functions. Their work leverages existing CUDA-based libraries such as cuDNN, cuSPARSE, and cuBLAS. In [40], the authors present the system that generates CUDA code from the C syntax. Their work mainly enhances data access patterns using the existing polyhedral compiler optimization technique. Numba [41] accelerates a Python code using the LLVM compiler library, generating machine code at runtime. Their work enables the parallelization of the application using multithreading, SIMD vectorization, and GPU. Libraries such as Kokkos [42] and RAJA [43] have been released to increase the portability of CUDA code with the abstraction for parallel execution and memory management. The library uses modern C++-like syntax and provides an abstraction for operations not only on loop execution, but loop partitioning, reorder, and tile. Furthermore, for simple operations, execution configurations are automatically set. Our work is different from these general-purpose

libraries in that OpenHD intelligently utilizes the characteristics of HDC, adapting various optimization schemes specialized to HDC for higher efficiency.

6.2 Hardware Acceleration of HDC

HDC consists of many bit-level arithmetic operations, including addition and multiplication. It can effectively be parallelized and handled on various parallel hardware platforms. The ASIC designs are proposed, e.g., similarity computation circuits [12] and text classification ASIC design [15] based on advanced memory technology. The work in [14] also shows how to accelerate the HD encoding scheme using PIM techniques. In [44], the authors present a hybrid ASIC architecture that runs both DL and HDC. In [25], the authors propose the HDC-based classification running on the FPGA.

Although FPGA, ASIC, and PIM implementations offer superior efficiency, design and synthesis time impede rapid development. Prior works explored the GPU-powered HDC implementation. The work in [11] shows TensorFlow for HDC classification acceleration on GPU. Hypervectors can be treated as a tensor data type. Hence, HDC operations can be implemented and accelerated with the GPU with tensor operations in TensorFlow. The results show that TensorFlow-based HDC running on the NVIDIA Jetson Nano consumes more energy and runs just as slow as HDC on CPU. Thus, we need a tool such as OpenHD, which automatically creates an efficient mapping of HDC applications to GPUs. OpenHD maximizes parallelism specific to HDC applications, while effectively leveraging the memory hierarchy and minimizing the memory accesses.

7 Conclusion

In this paper, we presented a GPU-based HDC acceleration framework called OpenHD. OpenHD allows users to implement various HDC applications in Python while automatically accelerating them on GPU with highly optimized techniques. We address the memory access challenges in the current HDC application running on the GPU, with efficient cache utilization and data reuse. The proposed optimization module benefits the encoding and the prediction of the HDC, which affects the end-to-end pipeline of HDC applications. Furthermore, we propose a parallelized training method, called ParTrain, to enhance hardware utilization and data-parallel benefits. Users can use proposed techniques with NumPy-style Python syntax and automatically get fully optimized CUDA code with the best possible performance for commonly used HDC applications such as classification and clustering. For the standardized HDC operations, our framework offers API to compile in AOT as a runtime library. In our experiments, OpenHD-based classification offers comparable accuracy to the DNN-based solution with 9× smaller model and 11.7× faster execution time. OpenHD-based clustering is 53× faster compared to the K-means algorithm at comparable accuracy. Moreover, the proposed ParTrain reduces the required epochs to obtain comparable accuracy by $4\times$. Evaluation results with the low-powered embedded GPU show that OpenHD is 4.5× and 146× faster execution time on average for classification and clustering applications, respectively, compared to the state-of-the-art GPU-based HDC implementation.

Acknowledgments

This work was supported in part by CRISP, one of six centers in JUMP (an SRC program sponsored by DARPA), SRC Global Research Collaboration grant, DARPA HyDDENN grant, and NSF grants #1911095, #2003279, #2100237, and #2120019. This work was also supported by the National Research Foundation (NRF) of Korea (NRF-2018R1A5A1060031).

References

[1] P. Kanerva, *Sparse distributed memory*. MIT press, 1988.
[2] Y. Kim, M. Imani, N. Moshiri, and T. Rosing, "Geniehd: Efficient dna pattern matching accelerator using hyperdimensional computing," in *Proceedings of the 23rd Conference on Design, Automation and Test in Europe*, ser. DATE '20. San Jose, CA, USA: EDA Consortium, 2020, p. 115–120.
[3] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *International Symposium on Low Power Electronics and Design*. Association for Computing Machinery, 2016, p. 64–69.
[4] A. Mitrokhin, P. Sutor, C. Fermüller, and Y. Aloimonos, "Learning sensorimotor control with neuromorphic sensors: Toward hyperdimensional active perception," *Science Robotics*, vol. 4, no. 30, May 2019.
[5] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *International Conference on Rebooting Computing (ICRC)*. IEEE, 2017, pp. 1–8.
[6] O. Räsänen and S. Kakouros, "Modeling dependencies in multiple parallel data streams with hyperdimensional computing," *IEEE Signal Processing Letters*, vol. 21, no. 7, pp. 899–903, 2014.
[7] F. Asgarinejad, A. Thomas, and T. Rosing, "Detection of epileptic seizures from surface eeg using hyperdimensional computing," in *2020 42nd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*. IEEE, 2020, pp. 536–540.
[8] Y. Wu, G. Wayne, A. Graves, and T. Lillicrap, "The kanerva machine: A generative distributed memory," *arXiv preprint arXiv:1804.01756*, 2018.
[9] G. Karunaratne, M. L. Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian, "In-memory hyperdimensional computing," *arXiv preprint arXiv:1906.01548*, 2019.
[10] S. Ahmad and J. Hawkins, "Properties of sparse distributed representations and their application to hierarchical temporal memory," *arXiv preprint arXiv:1503.07469*, 2015.
[11] S. Datta, R. Antonio *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.
[12] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, "Exploring hyperdimensional associative memory," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2017.
[13] M. Imani, J. Messerly, F. Wu, W. Pi, and T. Rosing, "A binary learning framework for hyperdimensional computing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 126–131.
[14] S. Gupta, M. Imani, and T. Rosing, "Felix: Fast and energy-efficient logic in memory," in *Proceedings of the International Conference on Computer-Aided Design*, 2018.
[15] H. Li, T. F. Wu, A. Rahimi *et al.*, "Hyperdimensional computing with 3d vrram in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition," in *IEEE IEDM*, 2016, pp. 16.1.1–16.1.4.
[16] M. Imani, S. Pampana, S. Gupta, M. Zhou, Y. Kim, and T. Rosing, "DUAL: Acceleration of clustering algorithms using digital-based processing in-memory," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Oct. 2020.
[17] J. Kim, H. Lee, M. Imani, and Y. Kim, "Efficient brain-inspired hyperdimensional learning with spatiotemporal structured data," in *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2021, pp. 1–8.

[18] D. Kleyko, A. Rahimi, D. A. Rachkovskij, E. Osipov, and J. M. Rabaey, "Classification and recall with binary hyperdimensional computing: Tradeoffs in choice of density and mapping characteristics," *IEEE transactions on neural networks and learning systems*, vol. 29, no. 12, pp. 5880–5898, 2018.

[19] J. Kang, B. Khaleghi, Y. Kim, and T. Rosing, "Xcelhd: An efficient gpu-powered hyperdimensional computing with parallelized training," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022, pp. 220–225.

[20] 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, Jan. 2009.

[21] A. Rahimi, S. Datta, D. Kleyko, E. P. Frady, B. Olshausen, P. Kanerva, and J. M. Rabaey, "High-dimensional computing as a nanoscalable paradigm," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 9, pp. 2508–2521, 2017.

[22] Y. Kim, M. Imani, and T. Rosing, "Efficient human activity recognition using hyperdimensional computing," in *Proceedings of the 8th International Conference on the Internet of Things*, 2018, pp. 1–6.

[23] F. R. Najafabadi, A. Rahimi, P. Kanerva, and J. M. Rabaey, "Hyperdimensional computing for text classification," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

[24] A. Joshi, J. Halseth, and P. Kanerva, "Language recognition using random indexing," 2014.

[25] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, "F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing," in *International Symposium on FPGAs*. ACM, 2019, pp. 53–62.

[26] M. Imani, J. Morris, S. Bosch, H. Shu, G. De Micheli, and T. Rosing, "Adapthd: Adaptive efficient training for brain-inspired hyperdimensional computing," in *IEEE BioCAS*, 2019, pp. 1–4.

[27] M. Imani, Y. Kim, T. Worley, S. Gupta, and T. Rosing, "HDCluster: An accurate clustering using brain-inspired high-dimensional computing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Mar. 2019.

[28] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[29] B. Peksag, "astor," https://github.com/berkerpeksag/astor, 2020.

[30] M. Imani, J. Morris *et al.*, "Bric: Locality-based encoding for energy-efficient brain-inspired hyperdimensional computing," in *56th Annual Design Automation Conference*, 2019, pp. 1–6.

[31] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2019, pp. 1946–1956.

[32] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07. USA: Society for Industrial and Applied Mathematics, 2007, p. 1027–1035.

[33] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[34] A. Reiss and D. Stricker, "Introducing a new benchmarked dataset for activity monitoring," in *2012 16th International Symposium on Wearable Computers*, 2012, pp. 108–109.

[35] A. Ultsch and J. Lötsch, "The fundamental clustering and projection suite (FCPS): A dataset collection to test the performance of clustering and data projection algorithms," *Data*, vol. 5, no. 1, p. 13, Jan. 2020.

[36] J. Morris, M. Imani, S. Bosch, A. Thomas, H. Shu, and T. Rosing, "Comphd: Efficient hyperdimensional computing using model compression," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2019, pp. 1–6.

[37] NVIDIA, "TensorRT," https://developer.nvidia.com/tensorrt, 2020.

[38] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 579–594.

[39] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[40] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic c-to-cuda code generation for affine programs," in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, ser. CC'10/ETAPS'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 244–263.

[41] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: Association for Computing Machinery, 2015.

[42] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[43] D. A. Beckingsale, T. R. Scogland, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, and B. S. Ryujin, "RAJA: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, Nov. 2019.

[44] M. Nazemi, A. Esmaili, A. Fayyazi, and M. Pedram, "Synergiclearning: neural network-based feature extraction for highly-accurate hyperdimensional learning," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

Jaeyoung Kang Jaeyoung Kang received a B.E. degree in electrical engineering from Korea University, Seoul, South Korea, in 2019. Currently, he is a third-year Ph.D. student in Electrical and Computer Engineering at the University of California San Diego, La Jolla, CA, USA. His research interests include deep learning-based algorithm acceleration on heterogeneous system architecture, GPU-based acceleration for big data analysis in bioinformatics, and hyperdimensional computing.

Behnam Khaleghi Behnam Khaleghi is a Ph.D. candidate in the Department of Computer Science and Engineering at the University of California San Diego, CA, USA. He received M.S. and B.S. degrees from the Department of Computer Engineering at the Sharif University of Technology in 2013 and 2016, respectively. His research interests include brain-inspired computing, ML acceleration, reconfigurable computing, and VLSI design automation.

Tajana Rosing Tajana Šimunić Rosing (Fellow, IEEE) received the M.S. degree in engineering management concurrently and the Ph.D. degree from Stanford University, Stanford, CA, USA, in 2001. She is a Professor, a Holder of the Fratamico Endowed Chair, and the Director of System Energy Efficiency Laboratory, University of California at San Diego, La Jolla, CA, USA. From 1998 to 2005, she was a full-time Research Scientist with HP Labs, Palo Alto, CA, USA, while also leading research efforts with Stanford University, Stanford, CA, USA. She was a Senior Design Engineer with Altera Corporation, San Jose, CA, USA. She is leading a number of projects, including efforts funded by DARPA/SRC JUMP CRISP program with focus on design of accelerators for analysis of big data, DARPA and NSF funded projects on hyperdimensional computing, and SRC funded project on IoT system reliability and maintainability. Her current research interests include energy-efficient computing, cyber–physical, and distributed systems.

Yeseong Kim Yeseong Kim is an Assistant Professor in the Department of Information and Communication Engineering at Daegu Gyeongbuk Institute of Science and Technology, Daegu, South Korea, where he directs the Computation Efficient Learning Lab. His research interests include alternative computing, computer architecture, and embedded systems. He received a B.S. degree in computer science and engineering from Seoul National University, Seoul, South Korea, and an M.S. and Ph.D. degrees in computer science from the University of California San Diego, La Jolla, CA, USA.