XCelHD: An Efficient GPU-Powered Hyperdimensional Computing with Parallelized Training

Jaeyoung Kang
Department of Electrical and Computer Engineering
University of California San Diego
La Jolla, CA, USA
j5kang@ucsd.edu

Behnam Khaleghi
Department of Computer Science and Engineering
University of California San Diego
La Jolla, CA, USA
bkhaleghi@ucsd.edu

Yeseong Kim
Department of Information and Communication Engineering
DGIST
Daegu, Republic of Korea
yeseongkim@dgist.ac.kr

Tajana Rosing
Department of Computer Science and Engineering
University of California San Diego
La Jolla, CA, USA
tajana@ucsd.edu

2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC) | 978-1-6654-2135-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ASP-DAC52403.2022.9712549

Abstract— Hyperdimensional Computing (HDC) is an emerging lightweight machine learning method alternative to deep learning. One of its key strengths is the ability to accelerate it in hardware, as it offers massive parallelisms. Prior work primarily focused on FPGA and ASIC, which do not provide the seamless flexibility required for HDC applications. Few studies that attempted GPU designs are inefficient, partly due to the complexity of accelerating HDC on GPUs because of the bit-level operations of HDC. Besides, HDC training exhibited low hardware utilization due to sequential operations. In this paper, we present XCelHD, a high-performance GPU-powered framework for HDC. XCelHD uses a novel training method to maximize the training speed of the HDC model while fully utilizing hardware. We propose memory optimization strategies specialized for GPU-based HDC, minimizing the access time to different memory subsystems and redundant operations. We show that the proposed training method reduces the required number of training epochs by four-fold to achieve comparable accuracy. Our evaluation results on NVIDIA Jetson TX2 show that XCelHD is up to 35× faster than the state-of-the-art TensorFlow-based HDC implementation.

I. Introduction

Machine learning on edge devices has become more prevalent as it enables real-time and in-place analysis. Techniques based on deep learning (DL) are popular for processing complex tasks by extracting high-level features. However, running such algorithms on conventional systems results in high energy consumption and slow processing speed. Brain-inspired Hyperdimensional Computing (HDC) is an alternative solution based on a long-term memory model, which emulates human cognition with operations on high-dimensional vectors, called hypervectors [1]. HDC first encodes data into hypervectors to perform the rest of the training process in a lightweight way. Many different learning applications have been shown to provide high accuracy and efficiency when implemented using HDC, e.g., language recognition [2], robotics [3], activity identification and voice recognition [4], recommendation system [5] and multimodal sensor fusion [6]. Several companies are also using HDC to implement more generalizeable learning, e.g., Google [7], IBM [8], and Numenta [9].

HDC has a number of advantages as compared to the conventional DL: (1) it is suitable for on-device learning based on hardware acceleration due to its highly parallel nature [10], (2) hidden features of information can be well-exposed, thereby empowering both training and inference with the light-weight computation and a small number of iterations, and (3) the hypervector representation inherently exhibits strong robustness against the noise and corrupted data [11]. A few hardware accelerators have been proposed for HDC, such as ASIC [2], [12], [13], and in-memory computing accelerator [14]. These works show nearly remarkable speedup and energy efficiency improvements compared to the CPU-based implementation, primarily due to their ability to leverage high parallelism of operations on hypervectors. However, ASIC and PIM have long design times and are expensive to manufacture, so they are not commonly available. More importantly, HDC applications require flexibility in terms of, e.g., hypervector lengths, encoding algorithm, and precision of operations that such platforms cannot readily offer. An excellent commercial off-the-shelf platform is GPU. Recently there has been a proliferation of GPU-enabled embedded devices, such as NVIDIA Jetson.

Unfortunately, recent work that accelerated HDC on embedded GPU [15] showed a comparable performance over CPU-based HDC. Our recent experiments with TensorFlow XLA-based HDC also gained a marginal performance improvement (1.74×) over the CPU, showing the GPU utilization of 40% and 10% for the encoding and the training, respectively. There are several reasons why a library such as TensorFlow is not suitable for running HDC on the GPU. First, HDC training by default requires *sequential* data inputs to refine the trained model iteratively for higher accuracy. It creates a lot of HDC data transfer between CPU and GPU, which causes a memory bottleneck and impedes full exploitation of GPU parallelism. Furthermore, TensorFlow-based HDC cannot carefully leverage the mapping data to the memory hierarchy on the GPU since HDC data are large in size and have specific memory access patterns. For the off-the-shelf embedded GPUs, these characteristics lead to underutilized hardware resources, impeding the high throughput and perfor-

978-1-6654-2135-5/22/$31.00 ©2022 IEEE

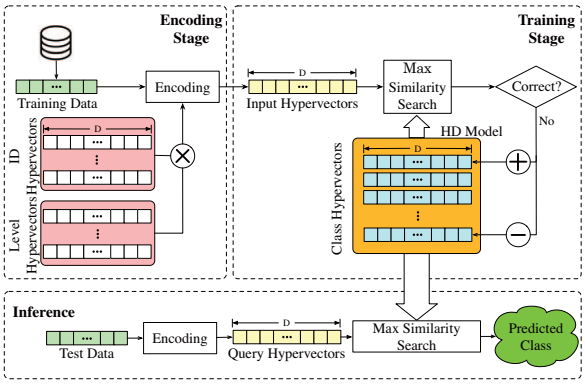


Fig. 1: Overview of HDC-based classification.

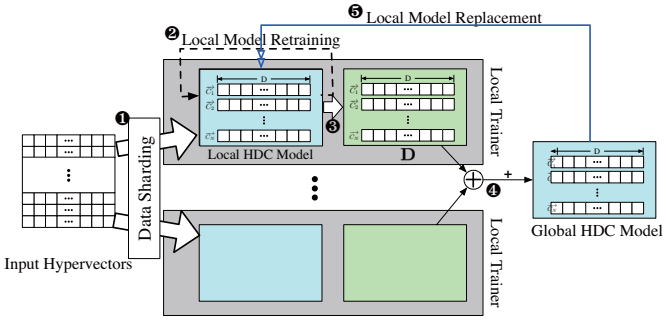


Fig. 2: Overview of the proposed parallel training method

mance of HDC applications.

In this paper, we present a GPU-powered HDC framework called XCelHD. XCelHD effectively mitigates the low parallelization and memory access issues of the existing works. 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 task 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 XCelHD for GPUs, which supports various classification tasks based on HDC. Unlike existing DL framework-based implementations, XCelHD parallelizes the HDC learning procedure in a GPU-friendly way while intelligently optimizing memory allocation/access patterns. To the best of our knowledge, this is the first work that designs GPU-centric optimization strategies for HDC.
- We propose a novel HDC training method for high parallelism. The proposed training method 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 48%) in HDC with simple parallelization, respectively.

We implemented and evaluated XCelHD with various datasets using NVIDIA Jetson TX2, which aims to low-power edge devices. Our experimental results show that the proposed parallelized training method can save the number of training iterations required to achieve the same accuracy by $4\times$. Furthermore, our design is up to $35\times$ faster than TensorFlow 2 XLA-based HDC classification implementation.

II. XCELHD OVERVIEW

Classification is a representative cognitive task that HDC can be used [4], [16]. Figure 1 illustrates the XCelHD learning procedure, which consists of the encoding, training, and inference stages.

Encoding maps (encode) real-world data into the HD space. There are various encoding methods. One of the most popular ones is the ID-level method [4], [14], [17]. First, the encoding module randomly generates orthogonal unique *ID hypervectors*, $\overrightarrow{ID}_f \in \{-1, 1\}^D$ assigned to an individual f^{th} index of feature. Next, to capture an arbitrary value m in f^{th} index of feature, we create *level hypervectors*, $\overrightarrow{LV}_m$ for using the method shown in [4]. Once ID hypervectors and level hypervectors are generated, we can encode an input data to an *input hypervector* $\overrightarrow{I}$ as follows:

$$\overrightarrow{I} = \sum_f \overrightarrow{ID}_f \odot \overrightarrow{LV}_m \quad (1)$$

where $\odot$ indicates an element-wise multiplication. The ID-level method is more complex but includes all the steps of other algorithms, such as random projection. Thus, any implementation that can carry out ID-level can also implement the random projection [4].

Initialization and training stage builds *class hypervectors*. During the initialization, also known as single-pass training, it combines 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 (HDC model) based on the prediction using the current state of class hypervectors. First, it compares the ground truth label to the predicted class. We select a class with a maximum similarity between input hypervector and class hypervectors. For binarized hypervectors, we use the Hamming distance, while cosine similarity is utilized for another type of data. If the prediction is incorrect, the training module updates the HDC model: the input hypervector is *subtracted* from the class hypervector of the incorrectly predicted class and *added* to the class hypervector of the correct answer. We can apply the learning rate ($\eta \in (0, 1]$) for scaling the input hypervector during the update [18]. We define *epoch* as the number of training iterations.

Inference or prediction finds the most similar class hypervector to the test data. We first encode test queries (data) into hypervectors, called *query hypervectors* using the same encoding method used for training. For each query hypervector $\overrightarrow{q}$, the inference module calculates its similarity against the class hypervectors. Next, it selects a class with the highest similarity using either Hamming distance (binary hypervectors) or cosine similarity (non-binary data).

III. XCELHD OPTIMIZATION TECHNIQUES

A. PARTRAIN: *Parallel Training for HDC Classification*

The iterative training process is essential to achieve higher accuracy than the single-pass (shot) training (see Fig. 5.) However, the training stage processes data sequentially, thus it hinders from data-parallel benefits of the GPU. It can lead to the bottleneck of the GPU-based HDC. We propose a method to refine the HDC model in parallel, called PARTRAIN. It can unleash parallelism, maximize hardware utilization and achieve the target accuracy with small iterations.

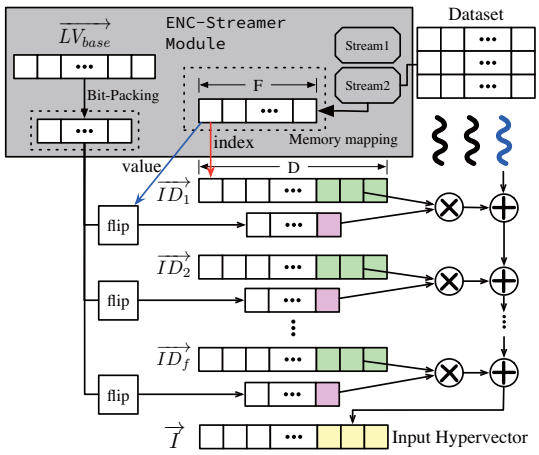


Fig. 3: Overview of ENC-STREAMER-applied HDC encoding

Fig. 2 describes the overview of the proposed training. The main principle of the proposed method uses multiple *local trainers* to generate a local HDC model and creates the global HDC model $\mathbf{G}$, which is used for the inference. A local trainer has two sets of hypervectors, each with C hypervectors, where C is the number of classes. One set stores a local HDC model, and the other accumulates the changes of the local HDC model. We initialize the latter hypervector set, $\mathbf{D}$ with zeros. The local HDC models and global HDC models are initialized with the results of HDC single-pass training.

We shard the training data by identical numbers and feed them to each local trainer. (Fig. 2-❶) It trains its local HDC model following the conventional HDC training process (Fig. 2-❷). It is updated when the predicted values differ from the ground truth. Additionally, changes of the local HDC model are aggregated to $\mathbf{D}$ (Fig. 2-❸). After training with assigned datapoints, we update the global HDC model by calculating $\mathbf{G} = \sum_i \mathbf{D}_i$, where i indicates ith local trainer (Fig. 2-❹). Finally, the global model replaces the local HDC model (Fig. 2-❺). Note that every accumulating operation is parallelized over the hypervector dimension. We define this whole process as *one epoch*. The original HDC training processes the same amount of data points in a single epoch. Using multiple local trainers in a single GPU device can enable a higher level of data parallelism. Also, the number of local trainers can be dynamically adjusted based on available hardware resources.

B. HDC-BASED MEMORY OPTIMIZATION

XCelHD encompasses two key optimization techniques to reduce the memory access times due to frequent accesses to large size hypervectors: ENC-STREAMER and L2-RECYCLE. They are used during the encoding and the training of HDC.

1) ENC-Streamer: ENC-STREAMER is designed to allocate data to the appropriate memory hierarchy to achieve high efficiency during data-intensive encoding and inference stages. These two stages map training and testing data to the HD space, respectively. Encoding often takes as much as 70% of the overall runtime of HDC on CPU [19], which is the main bottleneck of HDC application. Here we focus on one of the popular HDC encoding techniques, the ID-Level encoding. Same strategy work for the simpler random projection method as well. A naive way to implement the ID-Level encoding on GPU is to parallelize over training data and each dimension of input hypervectors. A thread accumulates $\overrightarrow{ID}^i \odot \overrightarrow{LV}^i$ over features where i is the index of an element in a hypervector that a thread is computing.

ENC-STREAMER enhances memory transactions on top of the parallelization. Fig. 3 shows the design of the encoding module with ENC-STREAMER. The GPGPU memory hierarchy includes several memories, e.g., global memory, shared memory, and constant memory, each having different characteristics such as size and latency. The execution time diverges depending on a data allocation strategy of an algorithm. For instance, global memory offers the largest capacity with long latency. Constant memory has a small size and is read-only but can be accessed with a few clock cycles.

We use two streams to hide the memory copy time between the host and the GPU. The streamed data is encoded to the hypervector using the aforementioned method and stored in the global memory. On top of that, ENC-STREAMER applies an additional caching strategy for ID hypervectors and level hypervectors, which is a frequently used term as shown in Eq. 1. We cache $\overrightarrow{ID}$ in the shared memory. This encoded data is used for the rest of the runtime, thus *completely eliminating the data movement costs*. At the same time, the same strategy is applied to the other stream synchronously. It runs until all datapoints are covered in an interleaved fashion. For the GPUs with unified virtual addressing, we use a single stream and managed memory. Raw data requires one read transaction and does not have to be cached. This mechanism helps to alleviate the memory capacity limitation that is especially critical in embedded GPUs.

Furthermore, ENC-STREAMER optimizes access to level hypervectors. Since the feature values are random, the access to their associated level hypervector is irregular. In other words, it entails non-coalesced access to memory, leading to increased latency. Based on the fact that the level hypervector generation method uses flips according to quantized levels, we generate only one level hypervector to use it as a base, called *base-level hypervector*, $\overrightarrow{LV_{base}}$. Then, according to the value of each index, the corresponding level hypervector is generated in-situ. Nevertheless, this approach can still cause contention during memory access if it is placed on the global memory. We bit-pack the base-level hypervector and store it in constant memory. Since hypervectors require a large size array, packing is essential to satisfy the constant memory constraint. Empirically, packing eight bits showed the best performance. Each thread encodes the eight components of the input hypervector.

2) L2-Recycle: Pairwise similarity computation is the core component of HDC training and inference. The cosine similarity $\delta(x, y)$ between vector $\vec{x}$ and $\vec{y}$ is defined by

$$\delta(x, y) = \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\| \|\vec{y}\|}. \quad (2)$$

To accommodate higher parallelism, we compute the numerator and the denominator separately. Then, operations, including the L2 norm, can leverage the parallel reduction technique because of the large dimensionality. For HDC vectors, decomposing product-and-sum computations into multiple threads can lead to higher performance. However, this strategy showed a marginal improvement compared to the CPU-based HDC in our experiments due to a lack of data-parallel benefits.

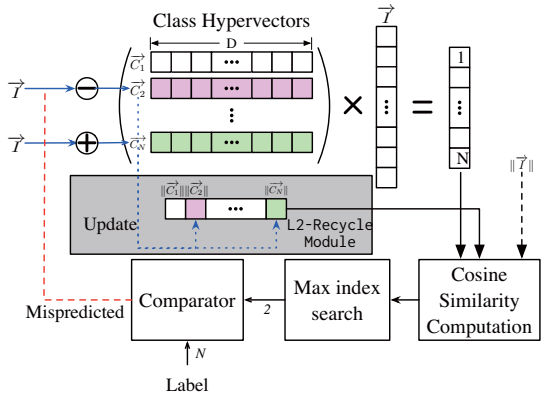


Fig. 4: Design of L2-RECYCLE-applied training module in HDC-based classification

Instead, we devise the L2-RECYCLE module, which optimizes the memory access time in HDC by identifying *invariant calculation results* in the similarity computation. It computes the two L2 norm components only when required. Fig. 4 shows the design of L2-RECYCLE-applied training module in HDC-based classification. Since the training procedure calculates 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. Also, L2-RECYCLE separately manages the array of the $\|\overrightarrow{C_k}\|$. This strategy significantly minimizes the memory access time to the large size of hypervectors. In addition, it removes unnecessary computation during the HDC application execution.

IV. EVALUATION

A. Experimental Setup

In this section, we test XCelHD on HDC-based classification [4]. We evaluated the implementation on the CPU and GPU in the NVIDIA Jetson TX2 device with Jetpack 4.4.1. We set the dynamic voltage and frequency scaling governor.

We compare the performance of our XCelHD-based implementation with the state-of-the-art TensorFlow 2 XLA-based HDC, which implements the design in [15]. Note that we did not adopt TensorRT/TFLite-based inference implementation, as end-to-end HDC execution includes the model training. Next, to show the energy efficiency of XCelHD, we compare the energy consumption with the state-of-the-art CPU-based HDC classification [18] 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 XCelHD is configured with 10 local trainers.

Datasets To verify the actual use-case of HDC-based classification, we evaluated the implementation on a wide range of following benchmark datasets [20], [21] including **UCIHAR**: detecting human activity, **FACE**: classifying images with faces and non-faces, each sampled from Caltech's $10,000$ web faces dataset, CIFAR-100, and Pascal VOS 2012 datasets, **ISOLET**: recognizing audio of the English alphabet, and **PAMAP2**: classifying human activities based on data from a heart rate monitor and inertial measurement units.

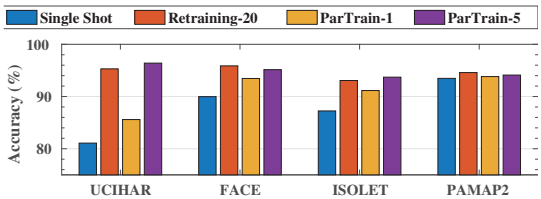


Fig. 5: Accuracy comparison between PARTRAIN and the conventional HDC training method.

B. Accuracy of PARTRAIN

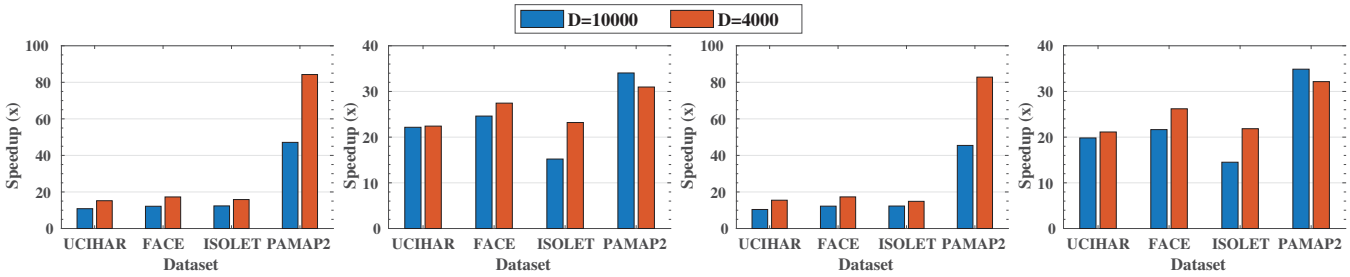
If XCelHD with PARTRAIN refines an HDC model effectively, it should show comparable accuracies to the XCelHD without PARTRAIN, i.e., conventional HDC training method. As a baseline, we compared the HDC model, which trained with a single epoch (Single Shot), 20 epochs of training (Retraining-20). Also, we measure the accuracy of the first (PARTRAIN-1) and the fifth epoch (PARTRAIN-5), for PARTRAIN-powered XCelHD. As shown in Fig. 5, XCelHD shows comparable accuracies compared to the baseline. The peak performance of the baseline was challenging to obtain with a single epoch of PARTRAIN. Nevertheless, after five iterations, the PARTRAIN-powered HDC model offered a similar accuracy compared to the original HDC. Considering that the conventional training method requires 20 epochs to reach the target accuracy, the proposed strategy can quickly reach the target accuracy. It reduces required epochs by $4\times$ while enhancing the data-parallel benefits and hardware utilization on GPU. Note that the overhead of gathering local trainer information and distributing the global HDC model was less than 5% of the total execution time.

C. Performance Improvements

Fig. 6 describes the performance improvement of the XCelHD-powered HDC as compared to the TensorFlow GPU-based implementation. We measured the execution time per stage since the optimization technique that dominates each stage is different.

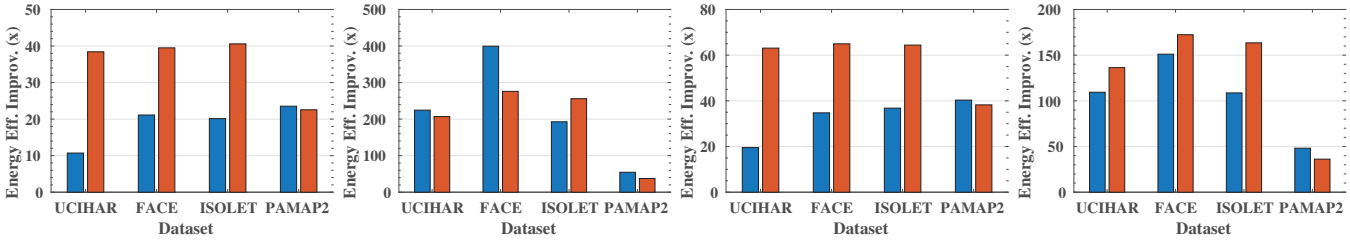
The encoding module would be mainly optimized by ENC-STREAMER. Multiple accesses to the dataset, $\overrightarrow{LV}_m$, $\overrightarrow{ID}_f$ occurs, so caching on constant memory improves the performance. The ENC-STREAMER module flexibly maps the data on the proper CUDA memory hierarchy. Also, accumulating through for loop is used due to the reduction operation. In the GPGPU architecture, the size of the register assigned to each thread is limited. If the program exceeds the constraint, it utilizes global memory. As shown in Fig. 6(a), the scenarios with relatively small features are accelerated more compared to the others. When data is fittable on the on-chip memory constraint, ENC-STREAMER in XCelHD automatically utilizes it. The result shows that XCelHD enables the encoding module on average $20\times$ faster, with a max speedup of $84\times$, compared to the TensorFlow-based HDC.

Iterative training makes predictions based on the current class hypervectors and updates class hypervectors as a result. In particular, the prediction is primarily benefitted by L2-RECYCLE. Fig. 6(b) illustrates the speedup of the training stage. Although the training module does not offer substantial data-level parallelism compared to the encoding module, XCelHD improves the speed by reusing computation results. Furthermore, since PARTRAIN can reduce the number of



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

Fig. 6: Speed comparison of XCelHD versus TensorFlow GPU-based HDC classification



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

Fig. 7: Energy consumption comparison of XCelHD versus HDC running on the embedded CPU

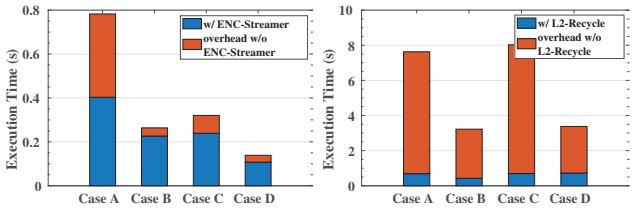
required epochs to achieve the target accuracy, XCelHD can lead to significant speedup in the actual deployment of GPU-powered HDC applications. XCelHD provides up to 34× speedup, with 24× speedup on average in the inference module compared to the TensorFlow GPU-based HDC.

The inference stage improvement originates from the combination of optimization for the encoding module and the prediction, i.e., from ENC-STREAMER and L2-RECYCLE. Unlike the training stage, which processes data sequentially to update the class hypervectors, we can make a prediction on multiple test data in parallel. The inference module benefits from both simultaneous data processing and parallel reduction. As shown in Fig. 6(c), XCelHD yields performance gain up to 83×, and 20× on average.

Ultimately, XCelHD offers significant speedup in all stages of the HDC classification compared to the existing TensorFlow-based solution. XCelHD most efficiently handles the encoding stage, which is the bottleneck of CPU-based HDC. PARTRAIN helps to overcome limited parallelism and underutilization during the training process in GPU-based HDC. For the end-to-end execution of HDC process, XCelHD-powered implementation gains up to 35×, with an average speedup of 23× as illustrated in Fig. 6(d).

D. Energy Efficiency Improvements

Fig. 7 shows a comparison of the energy consumption of XCelHD against the low-power CPU. Here, we disabled PARTRAIN for a fair comparison. In terms of power consumption, XCelHD showed different power consumption trends for each stage. Since XCelHD maximizes parallelism in the encoding module, it showed the highest power consumption. Nevertheless, the execution time reduced significantly, leading to 41× (25×) energy improvement at maximum (on average). The training stage without PARTRAIN involves the serialized data feed, causing limited thread utilization. Even if this can impede energy efficiency improvement, we successfully reduced the computation and the execution time using L2-RECYCLE, leading to a significant energy efficiency improvement: up to 400× and 163× on average. Enabling PARTRAIN increases power consumption up to 7× due to higher hardware utilization. Nevertheless, the proposed framework offers a significant amount of energy improvement in the case of XCelHD without PARTRAIN. Therefore, increased power consumption can also be compensated by reduced epochs.



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

Fig. 8: Execution time with and without proposed optimization techniques

Overall, XCelHD results in up to 172× improved energy efficiency over CPU. The energy efficiency improvements of XCelHD versus TensorFlow GPU are directly proportional to the performance improvements shown in Fig. 6, as both run on the same hardware platform.

E. Effectiveness of ENC-STREAMER and L2-RECYCLE

To show the effectiveness of our memory access optimization strategies, ENC-STREAMER and L2-RECYCLE, we compare XCelHD with and without these modules. The effectiveness of our optimization strategies varies according to the dataset characteristics. The performance of the encoding module is affected by the dimension size D and the feature size F. The speed (per epoch) of the training stage is affected by D and the number of classes C. While fixing the dataset size to $10,000$, we set representative scenario combinations.

For ENC-STREAMER, we evaluated on four scenarios, (F, D): Case A) $(500, 10000)$, case B) $(500, 4000)$, case C) $(250, 10000)$ and case D) $(250, 4000)$, with fixed $C = 10$. As shown in Fig. 8(a), ENC-STREAMER improves the speed by 48%, 14%, 25% and 22% in case A, case B, case C, and case D, respectively. As the dimensionality increases, the benefit of

ENC-STREAMER increases. Also, we observed that the power consumption between the two variants is similar, leading to energy efficiency improvement.

When evaluating L2-RECYCLE, we fixed $F = 500$ and changed the C. We experimented on four scenarios, (C, D): Case A) $(10, 10000)$, case B) $(10, 4000)$, case C) $(20, 10000)$ and case D) $(20, 4000)$. In the case of $D = 10000$, it leads to around $12\times$ performance improvement, while case B and case D (D=4000) shows $8\times$ and $5\times$ speedup, respectively. As shown in Fig. 8(b), the training without L2-RECYCLE is heavily affected by the dimensionality. In contrast, our optimization strategy guarantees almost consistent execution time regardless of the D and C, with only a minimal amount of additional memory usage (e.g., 80 Byte in case C). Reducing the redundant computation plays a significant role in accelerating the training with minimal overhead.

V. RELATED WORK

HDC consists of many bit-level arithmetic operations that can be effectively parallelized. The ASIC designs are proposed, e.g., similarity computation circuits [11] and text classification ASIC design [22] based on advanced memory technology. The work in [14] also shows how to accelerate the HD encoding scheme using PIM techniques. In [23], the authors present a hybrid ASIC architecture that runs both DL and HDC. In [17], the authors propose the HDC-based classification running on the FPGA. Although FPGA, ASIC, and PIM implementations can offer superior efficiency, their design and synthesis time impede rapid development. The work in [15] 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 illustrating the need for a tool such as XCelHD that can automatically create an efficient mapping of HDC applications to GPUs. XCelHD maximizes parallelism specific to HDC applications while effectively leveraging the memory hierarchy and minimizing the memory accesses.

VI. CONCLUSION

In this paper, we presented a GPU-based HDC acceleration framework called XCelHD. 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 on the GPU. The proposed PARTRAIN reduces the required epochs to obtain comparable accuracy by $4\times$. Also, our evaluation results with the low-powered embedded GPU show that XCelHD is $23\times$ faster execution time on average as compared to the existing TensorFlow GPU-based HDC classification implementation.

ACKNOWLEDGMENTS

This work was supported in part by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, in part by SRC-Global Research Collaboration grants, and also NSF grants # 2100237, #1730158, #1826967, and #1911095.

REFERENCES

[1] P. Kanerva, *Sparse distributed memory*. MIT press, 1988.
[2] 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.
[3] 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.
[4] 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.
[5] Y. Guo, M. Imani, J. Kang, S. Salamat, J. Morris, B. Aksanli, Y. Kim, and T. Rosing, "Hyperrec: Efficient recommender systems with hyperdimensional computing," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 384–389. [Online]. Available: https://doi.org/10.1145/3394885.3431553
[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] Y. Wu, G. Wayne, A. Graves, and T. Lillicrap, "The kanerva machine: A generative distributed memory," *arXiv preprint arXiv:1804.01756*, 2018.
[8] G. Karunaratne, M. L. Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian, "In-memory hyperdimensional computing," *arXiv preprint arXiv:1906.01548*, 2019.
[9] S. Ahmad and J. Hawkins, "Properties of sparse distributed representations and their application to hierarchical temporal memory," *arXiv preprint arXiv:1503.07469*, 2015.
[10] 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.
[11] 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.
[12] T. F. Wu, H. Li, P.-C. Huang, A. Rahimi, J. M. Rabaey *et al.*, "Brain-inspired computing exploiting carbon nanotube fets and resistive ram: Hyperdimensional computing case study," in *IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2018, pp. 492–494.
[13] M. Imani, C. Huang, D. Kong, and T. Rosing, "Hierarchical hyperdimensional computing for energy efficient classification," in *Proceedings of the 55th Annual Design Automation Conference*, 2018.
[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] S. Datta, R. A. G. Antonio, A. R. S. Ison, and J. M. Rabaey, "A programmable hyper-dimensional processor architecture for human-centric IoT," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 439–452, Sep. 2019.
[16] A. Joshi, J. Halseth, and P. Kanerva, "Language recognition using random indexing," 2014.
[17] 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.
[18] 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.
[19] 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.
[20] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml
[21] 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.
[22] 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*, pp. 16.1.1–16.1.4.
[23] 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.