Multiarchitecture Hardware Acceleration of Hyperdimensional Computing

Ian Peitzsch, Mark Ciora, Alan D. George
Department of Electrical and Computer Engineering
University of Pittsburgh
NSF Center for Space, High-performance, and Resilient Computing (SHREC)
Pittsburgh, PA, USA
{ian.peitzsch, alan.george}@nsf-shrec.org

***Abstract*—Hyperdimensional computing (HDC) is a machine-learning method that seeks to mimic the high-dimensional nature of data processing in the cerebellum. To achieve this goal, HDC represents data as large vectors, called *hypervectors*, and uses a set of well-defined operations to perform symbolic computations on these hypervectors. Using this paradigm, it is possible to create HDC models for classification tasks. These HDC models work by first transforming the input data into hypervectors, and then combining hypervectors of the same class to create a hypervector for representing that task. These HDC models can classify information by transforming new input data into hypervectors, comparing the similarity between data hypervector with each class hypervector, then classifying it based on which class has the highest similarity. Over the past few years, HDC models have greatly improved in accuracy and now compete with more common classification techniques for machine learning, such as neural networks. Additionally, manipulating hypervectors involve many repeated basic operations, making them easy to accelerate using different hardware platforms. This research seeks to exploit this ease of acceleration of HDC models and utilize oneAPI libraries with SYCL to create multiple accelerators for HDC learning tasks for CPUs, GPUs, and field-programmable gate arrays (FPGAs). The oneAPI tools are used in this research to accelerate single-pass learning, gradient-descent learning using the NeuralHD algorithm, and inference. Each of these tasks is benchmarked on the Intel Xeon Platinum 8256 CPU, Intel UHD 11th generation GPU, and Intel Stratix 10 FPGA. The GPU implementation showcased the fastest training times for single-pass training and NeuralHD training, with 0.89s and 126.55s, respectively. The FPGA implementation exhibited the lowest inference latency, with an average of 0.28ms.**

***Index Terms*—Hyperdimensional computing, machine learning, FPGA, GPU, HLS**

I. Introduction

In recent years, hyperdimensional computing (HDC) has been receiving more attention as an alternative machine-learning method to more well-known methods, such as neural networks. HDC is based on the observation that the brain uses high-dimensional representations of information to perform cognitive tasks [1]. To mimic this brain function, HDC represents data using high-dimensional vectors (hypervectors). HDC models can operate on these hypervectors to perform various learning tasks, such as activity recognition [2], language recognition [3], [4], and robot navigation [5], [6]. HDC is well suited for many applications as HDC models are highly parallel, well suited for hardware-level optimization, human interpretable, and robust to noise [1].

Recent research has increased its focus on HDC, with most of the work focusing on FPGA acceleration using hardware description languages (HDL). HDL implementations often offer the best possible FPGA-runtime performances; however, this increase in performance comes at a significant cost in development time. To ameliorate this lengthy development time, high-level synthesis (HLS) tools have been created to allow developers to design in high-level programming languages, such as C or C++. One novel HLS tool of interest is oneAPI, which uses SYCL to allow for the development of multiarchitecture accelerators. As computing systems become increasingly heterogeneous, such a multiarchitecture accelerator development tool allows for decreased development time for these heterogeneous systems.

In this paper, we investigate the acceleration of HDC learning tasks on FPGAs and GPUs using oneAPI. Unlike prior work on HDC acceleration, this work makes use of the oneAPI HLS tool instead of HDL. This research optimizes and compares general HDC implementations for both FPGAs and GPUs on training time, retraining time, latency, and throughput.

II. Background

The brain is the center of cognitive function. HDC seeks to mimic how the brain represents and manipulates information in high-dimensional space by using large vectors. The following sections gives insight into how this feat is accomplished.

A. Hyperdimensional Computing

In HDC applications, data are represented as *hypervectors*: vectors in high-dimensional space (often $>$10,000) [6]. These hypervectors benefit from the ”curse of dimensionality”, which states that in high-dimensional spaces, randomly generated vectors are nearly orthogonal [7]. This orthogonality makes random hypervectors able to represent different classifications

This research was supported by SHREC industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783.

of objects. The key aspects of hypervectors and their most common operations are covered in the following subsections.

1) Similarity (δ): Similarity measures the "relatedness" of two hypervectors. Two hypervectors A, B are considered related if $\delta(A, B) >> 0$. Similarly, A, B are considered unrelated if $\delta(A, B) \approx 0$. Oftentimes similarity is implemented as cosine similarity, so $\delta(A, B) = \frac{A \cdot B}{|A||B|}$ [8], [9].

2) Bundling ($+$): Bundling combines hypervectors to make a hypervector that is similar to the inputs [6]. So, hypervectors A, B can be bundled together to form $C = A + B$, and then $\delta(A, C) >> 0$ and $\delta(B, C) >> 0$. Commonly, bundling is implemented as the element-wise addition of the input hypervectors [10], [11].

3) Binding ($\otimes$): Binding combines hypervectors to make a hypervector that is dissimilar to the inputs [6]. So, hypervectors A, B can be binded together to form $C = A \otimes B$, and then $\delta(A, C) \approx 0$ and $\delta(B, C) \approx 0$. The binding operation has an inverse operation, named unbinding ($\oslash$), which allows for the approximate retrieval of the input hypervectors [11], [12]. So, $C \oslash A \approx B$ and $C \oslash B \approx A$. Binding is often implemented as element-wise XOR of the input hypervectors [7], [10], [12].

4) Permutation (ρ): Permutation rotates a hypervector and makes a hypervector that is dissimilar to the input [6]. So, hypervector A can be permuted and $\delta(A, \rho(A)) \approx 0$. In practice, $\rho(A)$ is doing a right rotation of the elements of A [7], [11].

B. HDC Learning

Using hypervectors and the previously mentioned operations, an HDC classification model can be constructed. The general dataflow for such a model is as follows: first, an encoding scheme is generated, then input data is encoded into hypervectors, and finally the hypervectors are operated on to either train the model or perform inference. The following sections describe in more detail the stages of this dataflow.

1) Encoding: As most real-world data is not already hyperdimensional, an encoding process is necessary to transform from the input feature space to the hyperdimensional space. Using randomly generated hyperdimensional basis vectors and the HDC operations, an input feature vector h can be encoded into its corresponding hypervector H. H can then be used by the HDC model for either inference or training.

2) Training: To train an HDC learning model, encoded hypervectors H^l of class l can be bundled into a class vector $C_l = \sum_i H_i^l$ [7], [9]. This method of training can be done in a single epoch, making it fast. While this method is fast, it does not always yield the highest accuracy, so retraining may be necessary.

3) Retraining: Simple training does not always produce the best accuracy, so the model can be fine-tuned using retraining [8]. Retraining works by comparing encoded hypervector H^l of class l to all class vectors C_j and getting the prediction $l' = \underset{j}{\operatorname{argmax}}\, \delta(H^l, C_j)$. If $l' = l$, then there is no need to change C_l or $C_{l'}$. If $l' \neq l$, then C_l and $C_{l'}$ are adjusted by $C_l = C_l + \alpha H^l$ and $C_{l'} = C_{l'} - \alpha H^l$, where α is some scalar factor [13]. Since this readjustment bundles H^l with C_l and debundles H^l from $C_{l'}$, it makes H^l more similar to C_l and less similar to $C_{l'}$.

4) Inference: Inferencing using an HDC classification model is achieved by selecting the class with the highest similarity to the encoded hypervector. So, the model predicts the encoded hypervector H is part of class l by finding $l = \underset{j}{\operatorname{argmax}}\, \delta(H, C_j)$ [9]. As the δ function on large vectors is embarrassingly parallel, this step can be efficiently computed on both GPUs and FPGAs.

C. oneAPI

Intel oneAPI is an open-source, multiarchitecture hardware acceleration interface. OneAPI uses SYCL and C++ to allow for single-source code development for CPUs, GPUs, and FPGAs [14], [15]. This single-source development process allows for code reuse for both host and accelerator code leading to faster development. OneAPI's execution model works by having accelerator code kernels submitted to an execution queue by the host. The execution of these kernels is then offloaded to the accelerator to run [14]. For memory management between the host and accelerator, oneAPI provides two methods: buffers and unified shared memory (USM) [16]. Buffers act as wrappers around data which can then be accessed on the accelerator using an accessor. Buffers abstract away data movement between the host and accelerator for ease of development. USM makes use of pointers to shared memory to facilitate data access. USM supports explicit data movement, where the developer must state where and when to move data, and implicit data movement, where the data movement is abstracted away. To aid in general development, oneAPI provides various libraries that contain already optimized code for general applications. One of these libraries is the oneAPI Math Kernel Library (oneMKL) which provides many pre-optimized math functions, such as matrix multiplication and vectorized operations [14]. Additionally, to aide in FPGA development, oneAPI provides useful reports to give area and throughput information about the design [17]. These reports give information about how much area/memory is being taken up by specific parts of code and what the maximum frequency and initiation interval of each loop are, allowing for fine-grain optimizations to the design.

III. Related Work

One of the advantages of HDC is how amenable it is to hardware acceleration, especially on FPGAs and GPUs. Due to this trait, many works have explored accelerating HDC on these platforms. The following sections describe work in the areas of both FPGA and GPU acceleration for HDC.

A. FPGA Acceleration

NeuralHD [1] is a training method for HDC models that increases model accuracy and reduces the necessary hyperdimensions to achieve that accuracy. This feat is achieved by using a dynamic encoder during training and altering this encoder to have the best accuracy. Using a Kintex-7 FPGA, the

researchers achieved a $26.8\times$ speedup for NeuralHD training time over an FPGA accelerated dense neural network (DNN) training time with little reduction in accuracy between the NeuralHD trained model and the DNN. This work also demonstrated $12.6\times$ speedup for FPGA accelerated HDC inference over FPGA accelerated DNN inference.

F5-HD [18] is an HDC accelerator generator. F5-HD generates an FPGA accelerator using specifications and constraints given by the user. This abstraction removes the need for knowledge of HDL for designing custom FPGA accelerators for many HDC applications which greatly reduces development time. An F5-HD FPGA accelerator running on a Kintex-7 FPGA achieved $7.8\times$ faster training and $1.7\times$ faster inference compared to an AMD R9 390 GPU.

B. GPU Acceleration

While GPUs have been targeted for HDC acceleration, often they are being used as a baseline for comparison with FPGA acceleration. However, there are fewer publications optimizing HDC specifically for GPUs. Though, recently there have been efforts on this front.

XCelHD [19] is one of the first GPU-focused frameworks for HDC. In its initial paper, XCelHD achieved $35\times$ speedup over then state-of-the-art TensorFlow-based HDC implementation when benchmarked on an NVIDIA Jetson TX2. XCelHD introduced a parallel training method, called *ParTrain*. ParTrain works by training multiple local models in parallel and then combines these local models to form a single global model. Additionally, XCelHD made use of various memory optimizations, such as a streaming module for encoding and compute recycling for similarity calculations.

OpenHD [20] is a more recent GPU-focused HDC framework. OpenHD achieved upwards of $9.8\times$ speedup for training time over XCelHD on the same device. Additionally, OpenHD achieved around $1.4\times$ speedup for inference latency over XCelHD. OpenHD performs the same optimizations as XCelHD, but also makes use of data type mutation. Data type mutation reduces the data type used for the hypervector elements to its minimal necessary size, which allows for the smallest memory footprint and more efficient use of local memory.

IV. System Architecture

This section describes the system architectures for both the FPGA- and GPU-accelerated HDC models. Both models rely on a modified Radial Basis Function (RBF) kernel as described in [1] for the encoding method. This kernel relies on using D randomly generated basis vectors $\vec{B}_1, \vec{B}_2, ..., \vec{B}_D$ all of length n. Then, an input data vector $\vec{F} = \{f_1, f_2, ..., f_n\}$ can be encoded into its corresponding hypervector $\vec{H} = \{h_1, h_2, ..., h_D\}$ by performing:

$$h_i = \cos(\vec{B}_i \cdot \vec{F} + b) \times \sin(\vec{B}_i \cdot \vec{F}) \tag{1}$$

where b is randomly sampled uniformly from $[0, 2\pi)$. $\vec{H}$ can then be passed onto the classification stage for inference or the fitting stage for training. The following sections go into more detail about the various optimizations specific to the FPGA- and GPU-optimized designs.

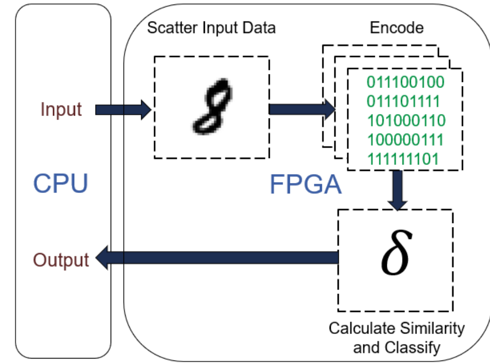

Fig. 1. High-level system architecture for inference with FPGA.

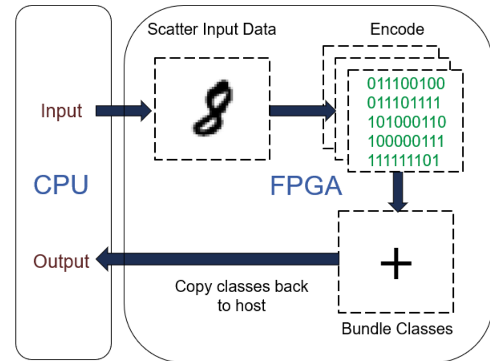

Fig. 2. High-level system architecture for single-pass learning with FPGA.

A. FPGA System Architecture

The general dataflow for the FPGA inference design is shown in Fig. 1. It begins with streaming in feature vectors from the host using unified shared memory (USM) with explicit data movement. The input vectors are then scattered to 25 compute units for encoding. As each dimension of a hypervector can be encoded independently, each compute unit can run in parallel. This parallel execution significantly reduces the inference time compared to using a single compute unit, as the encoding stage is the bottleneck of the data pipeline. From the encoders, the encoded hypervectors are then piped to a single classification kernel. This classification kernel pieces the parts from each encoding compute unit together to form a single hypervector. Then, this hypervector is compared to each class hypervector and the class with the highest similarity is selected as the prediction. The prediction is then streamed out using USM with explicit data movement.

The dataflow for the FPGA single-pass training design is similar to the inference dataflow and is shown in Fig. 2. Again, training feature data is first streamed onto the FPGA from the host using USM with explicit data movement. This data is then scattered to 8 compute units for the encoding stage. The number of compute units is reduced because of memory

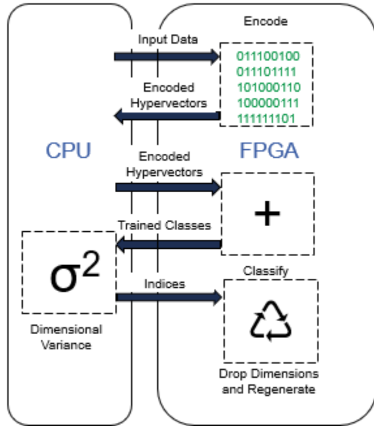

Fig. 3. High-level system architecture for NeuralHD learning with FPGA.

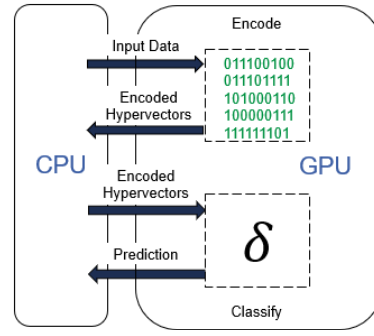

Fig. 4. High-level system architecture for inference with GPU.

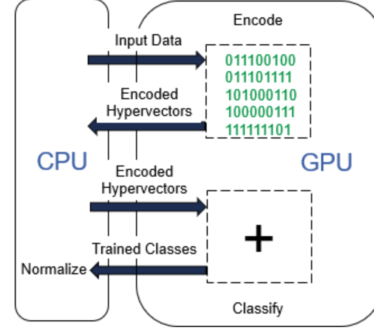

Fig. 5. High-level system architecture for single-pass learning with GPU.

constraints, as can be seen in Table I. The output encoded partial hypervectors are sent to the fitting kernel. This kernel combines the partial hypervectors to form a single hypervector and reads in the corresponding label. Then the kernel bundles the hypervector into the class corresponding to that label. After all the training data has gone through the encoding and fitting stages, the class hypervectors are streamed from the FPGA to the host using USM with explicit data movement. Finally, the host normalizes each class hypervector. This normalization reduces the complexity of the similarity function when doing inferences, as $\delta(\vec{H},\vec{C}) = \frac{\vec{H}\cdot\vec{C}}{|\vec{H}||\vec{C}|}$ simplifies to just $\delta(\vec{H},\vec{C}) = \frac{\vec{H}\cdot\vec{C}}{|\vec{H}|}$.

The dataflow for the NeuralHD training using an FPGA is shown in Fig. 3. The encoding implementation is different for the NeuralHD training than the other FPGA implementations because sufficient FPGA resources to accommodate the number of encoding compute units to see higher performance were not available. Instead, the input training data is streamed onto the FPGA in batches where then each feature vector is individually encoded through a matrix-vector multiplication and cosine/sine function. After the encoding stage, the encoded hypervectors are sent to the fitting stage. The fitting stage takes the hypervectors in batches and trains the class vectors as described in II-B3 using $\alpha = 0.037$. This process is repeated for an arbitrary number of iterations. Once the number of iterations has been reached, the class hypervectors are streamed back to the host. The host then calculates the dimension-wise variance between the class hypervectors. The 200 dimensions with the lowest variances are dropped and regenerated by generating a new basis hypervector for each of these dimensions and zeroing that entry in each class. This process of dropping and regeneration is performed on the FPGA. The training data is then re-encoded and the process continues. Training ends when either a maximum number of regeneration iterations has been reached or training has converged to 100% accuracy on the training set.

B. GPU System Architecture

Unlike the FPGA designs, the GPU designs made use of buffers and accessors for data movement between the host and accelerator. This choice was made as there was a negligible difference in runtime between using buffers and accessors instead of USM. Additionally, buffers and accessors allowed for easier programming since they abstract away data movement.

The GPU dataflow for inference is shown in Fig. 4. Input feature vectors are first passed to the encoder. Encoding is achieved by using oneMKL's general matrix multiply to compute

$$\begin{bmatrix} \vec{B}_1 \\ \vec{B}_2 \\ \vdots \\ \vec{B}_D \end{bmatrix} \vec{F} = \begin{bmatrix} \vec{B}_1 \cdot \vec{F} \\ \vec{B}_2 \cdot \vec{F} \\ \vdots \\ \vec{B}_D \cdot \vec{F} \end{bmatrix} \tag{2}$$

. Then using oneMKL's element-wise cosine and sine, this intermediate vector is fully encoded into its corresponding hypervector. This encoding implementation is easily batched, as batching it transforms it from a matrix-vector multiplication to a true matrix-matrix multiplication. The encoded hypervector is then sent to the classification kernel which calculates similarities between the hypervector and each class in separate parallel work items. Finally, the maximum similarity is calculated using a reduction, and the class value associated with the max similarity is sent back to the host.

Operation	Clock Freq. (MHz)	II	% DSP	% LUT	% FF	% BRAM
Inference	225	1	10	0	11	48
Single-pass Training	263	1	1	0	7	82
NeuralHD	198	1	5	0	7	64

TABLE I
CLOCK FREQUENCY AND AREA DATA FOR THE FPGA DESIGNS.

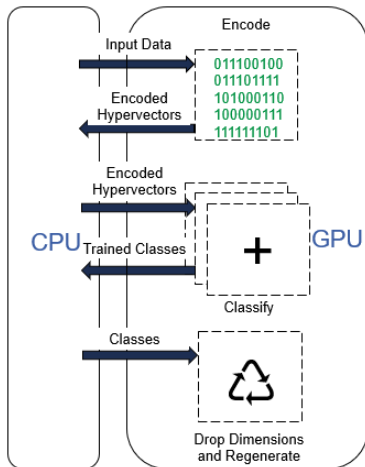

Fig. 6. High-level system architecture for NeuralHD learning with GPU.

The dataflow for the single-pass training using a GPU is shown in Fig. 5. The encoding stage for the single-pass training with the GPU is exactly the same as the encoding stage for inference. After the encoding stage, the encoded hypervectors are sent to the fitting stage. The fitting stage creates a separate work item for each class. Each work item goes through the entire training set of hypervectors and bundles hypervectors with labels matching their designated class value into that class hypervector. Finally, each work item normalizes its class hypervector. The dataflow for the NeuralHD training using a GPU is shown in Fig. 6. The encoding stage used for this implementation is identical to the encoding stages used in inference and single-pass learning. After the encoding stage, the encoded hypervectors are sent to the fitting stage. In the fitting stage, the encoded hypervectors are split into 512 groups. Each group has its own copy of the classes and retrains this local copy against its set of hypervectors with $\alpha = 0.037$. After each group has finished, all of the local classes are averaged together to form the new global classes. This process is repeated for an arbitrary number of iterations. The number of iterations is determined through trial-and-error for maximizing accuracy and training time. Once the number of iterations has been reached, the dimension-wise variance between the class hypervectors is calculated. The 200 dimensions with the lowest variances are dropped and regenerated by generating a new basis hypervector for each of these dimensions and zeroing that entry in each class. The training data is then re-encoded and the process continues. Training ends when either a maximum number of regeneration iterations has been reached, training has converged to 100% on the training set, or training accuracy has remained stagnant for more than 3 iterations.

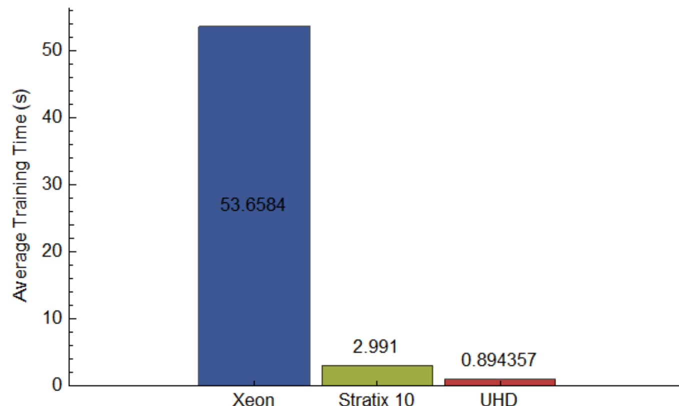

Fig. 7. Comparison of single-epoch training times. Lower values are better.

V. RESULTS

This research benchmarked single-epoch training, NeuralHD retraining, inference latency, and inference throughput using an Intel Stratix 10 FPGA and an Intel UHD 630 GPU as accelerators. For these benchmarks, the MNIST handwritten numbers dataset was used to train and test. Additionally, all metrics are compared to an optimized serial implementation that was executed on an Intel Xeon Platinum 8256 (Cascade Lake) CPU (3.8GHz, 4 Cores). All development, benchmarking, and data collection was conducted using Intel's DevCloud [21]. All implementations use an HDC classification model that uses 2000 hyperdimensions of 32-bit floating-point values. The FPGA implementation would benefit from quantization, but this is outside the scope of this work. The results from the benchmarks of the single-epoch training are shown in Fig. 7. All three of the implementations achieved similar accuracy of approximately 85%. Of the three architectures benchmarked, the GPU achieved the fastest training time with speedup of almost 60$\times$ over the CPU baseline. The FPGA design only achieved a speedup of 17.9$\times$ over the CPU baseline. The likely limiting factor with the FPGA design is onboard memory, as the read/write memory required for the class vectors takes up 18% of the BRAM, while the read-only memory used in the inference design only takes up 2% of the BRAM (see Table I). Due to this limiting factor, the encoding stage in the single-epoch training design could not be optimized as much as the encoding stage used in the inference designs.

Figure 8 shows the training times on all three architectures using the NeuralHD retraining algorithm. Similar to the single-epoch training results, the GPU design was faster than the other designs, achieving a speedup of 13.9$\times$ over the

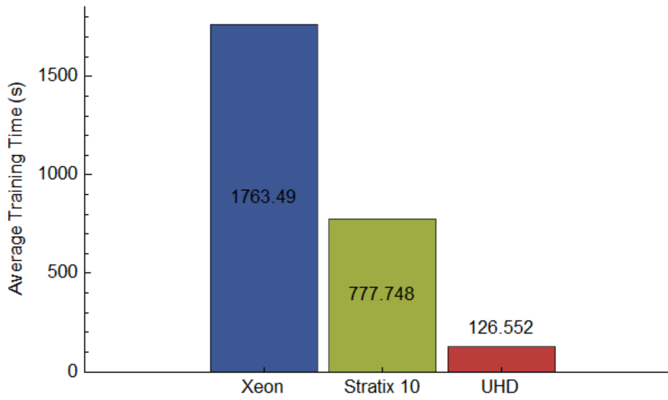

Fig. 8. Comparison of training times using the NeuralHD training algorithm. Lower values are better.

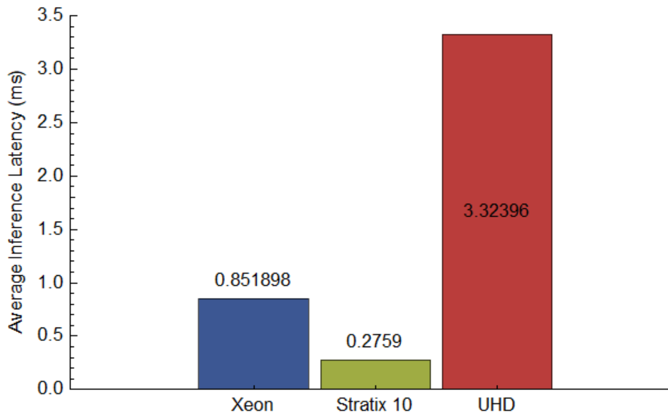

Fig. 9. Comparison of inference latencies. Lower values are better.

CPU baseline. However, the CPU and FPGA implementations achieved an accuracy of approximately 97% while the GPU implementation only achieved an accuracy of approximately 94%. This reduction in accuracy is due to fitting implementation for GPU as described in Section IV-B. The FPGA implementation only achieves a 2.3× speedup over the CPU baseline which is lower than both the GPU implementation of NeuralHD and the speedup for single-epoch training on FPGA. There are two main limiting factors as to why this is the case. First, as NeuralHD is a more complex algorithm, more FPGA memory is required, which significantly reduced the optimizations that were possible. Secondly, fewer stages of the NeuralHD algorithm could be effectively accelerated by the FPGA. The main stage that could not be effectively offloaded was the dimension dropping stage because it requires calculating variance and normalizing the class hypervectors, which are operations that require thousands of divisions and are not well suited for FPGAs.

The inference latency benchmarks are shown in Figure 9. The FPGA design achieved the lowest latency, with a speedup of 3× over the CPU baseline. This latency was achieved by spreading the encoding stage across many compute units. The GPU design exhibited higher latency than the CPU baseline.

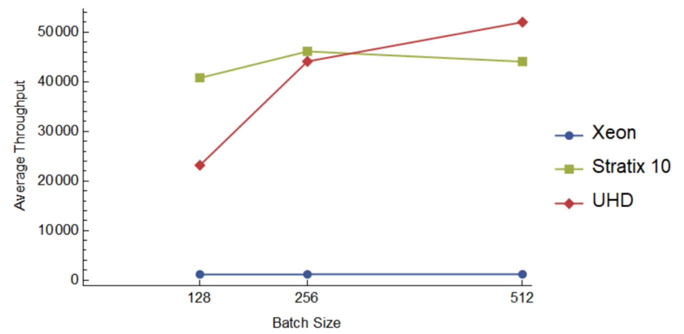

Fig. 10. Comparison of throughput (images per second) with varying batch sizes. Higher values are better.

As the UHD is an embedded GPU, its memory is shared with the host CPU. This shared memory should lead to lower latency compared to the FPGA, which has its own memory separate from the CPU and requires transfer time. However, the FPGA design offers enough speedup to fully overcome this added latency and give further speedup.

Figure 10 shows the average throughputs of each design with batch sizes of 128, 256, and 512. For batch sizes of 256 and 128, the FPGA design achieves the highest throughput. For the batch size of 512, the GPU design surpasses the FPGA design in throughput. These results indicate the GPU design has the highest throughput out of the three designs with larger batch sizes.

VI. Conclusions

This research benchmarks and compares GPU and FPGA accelerators written in oneAPI for HDC learning tasks. The GPU achieved upwards of 44× higher throughput than the CPU, 53× faster single-pass learning, and 13.9× faster NeuralHD training, making it the best performing in these metrics. However, the GPU was the worst performing device for inference latency with a 3.9× slowdown compared to the CPU. The FPGA design showcases better inference latency at 3× speedup over the CPU. The FPGA design also performs competitively for training times and throughput at upwards of 39× higher throughput, 17.9× speedup for single-pass learning, and 2.3× speedup for NeuralHD training. By leveraging oneAPI to create these designs, these performances were achieved without the need for multiple different libraries and compilers. Furthermore, using oneAPI allowed for a quick development time and accelerators that can be easily integrated into larger C++-based systems. Unfortunately, comparisons between the designs in this work and designs from related works are not able to be fairly made. This lack of comparable work is due to many factors including a lack of easily interpretable latency, training time, and throughput values for similar hardware and similar datasets. Additionally, many of these related works do not make their source code available for lines of code or other ease-of-use comparisons.

VII. Acknowledgment

This research was supported by SHREC industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783.

References

[1] Z. Zou, Y. Kim, F. Imani, H. Alimohamadi, R. Cammarota, and M. Imani, "Scalable edge-based hyperdimensional learning system with brain-like neural adaptation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3480958

[2] Y. Kim, M. Imani, and T. S. Rosing, "Efficient human activity recognition using hyperdimensional computing," in *Proceedings of the 8th International Conference on the Internet of Things*, ser. IOT '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3277593.3277617

[3] A. Joshi, J. Halseth, and P. Kanerva, "Language Recognition using Random Indexing," *arXiv e-prints*, p. arXiv:1412.7026, Dec. 2014.

[4] G. Karunaratne, A. Rahimi, M. L. Gallo, G. Cherubini, and A. Sebastian, "Real-time language recognition using hyperdimensional computing on phase-change memory array," in *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2021, pp. 1–1.

[5] A. Menon, A. Natarajan, L. I. G. Olascoaga, Y. Kim, B. Benedict, and J. M. Rabaey, "On the role of hyperdimensional computing for behavioral prioritization in reactive robot navigation tasks," in *2022 International Conference on Robotics and Automation (ICRA)*, 2022, pp. 7335–7341.

[6] P. Neubert, S. Schubert, and P. Protzel, "An introduction to hyperdimensional computing for robotics," *KI - Künstliche Intelligenz*, vol. 33, no. 4, pp. 319–330, Sep. 2019. [Online]. Available: https://doi.org/10.1007/s13218-019-00623-z

[7] T. Yu, Y. Zhang, Z. Zhang, and C. D. Sa, "Understanding hyperdimensional computing for parallel single-pass learning," in *Advances in Neural Information Processing Systems*, A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, Eds., 2022. [Online]. Available: https://openreview.net/forum?id=8ON84BdnSn

[8] L. Ge and K. K. Parhi, "Classification using hyperdimensional computing: A review," *IEEE Circuits and Systems Magazine*, vol. 20, no. 2, pp. 30–47, 2020.

[9] A. Thomas, S. Dasgupta, and T. Rosing, "A theoretical perspective on hyperdimensional computing," *J. Artif. Int. Res.*, vol. 72, p. 215–249, jan 2022. [Online]. Available: https://doi.org/10.1613/jair.1.12664

[10] R. W. Gayler, "Multiplicative binding, representation operators & analogy (workshop poster)," 1998. [Online]. Available: http://cogprints.org/502/

[11] D. Kleyko, D. A. Rachkovskij, E. Osipov, and A. Rahimi, "A survey on hyperdimensional computing aka vector symbolic architectures, part i: Models and data transformations," *ACM Comput. Surv.*, vol. 55, no. 6, dec 2022. [Online]. Available: https://doi.org/10.1145/3538531

[12] 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. [Online]. Available: https://doi.org/10.1007/s12559-009-9009-8

[13] M. Imani, S. Bosch, S. Datta, S. Ramakrishna, S. Salamat, J. M. Rabaey, and T. Rosing, "Quanthd: A quantization framework for hyperdimensional computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2268–2278, 2020.

[14] *Intel® oneAPI Programming Guide*, Intel, 2022, release 2023.0. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/oneapi-programming-guide.pdf

[15] *SYCL™ Specification*, Khronos, 2020. [Online]. Available: https://registry.khronos.org/SYCL/specs/sycl-2020-provisional.pdf

[16] *oneAPI GPU Optimization Guide*, Intel, 2022, release 2022.3. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/oneapi-gpu-optimization-guide.pdf

[17] *FPGA Optimization Guide for Intel® oneAPI Toolkits*, Intel, 2022, rev. 13. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/oneapi-dpcpp-fpga-optimization-guide.pdf

[18] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, "F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 53–62. [Online]. Available: https://doi.org/10.1145/3289602.3293913

[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] J. Kang, B. Khaleghi, T. Rosing, and Y. Kim, "Openhd: A gpu-powered framework for hyperdimensional computing," *IEEE Transactions on Computers*, vol. 71, no. 11, pp. 2753–2765, 2022.

[21] "Intel® devcloud for oneapi." [Online]. Available: https://devcloud.intel.com/oneapi/