Multiarchitecture Hardware Acceleration of Hyperdimensional Computing Using oneAPI

by

Ian Peitzsch

Bachelor of Science, Stony Brook University, 2021

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2023

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Ian Peitzsch

It was defended on

March 31, 2023

and approved by

Peipei Zhou, PhD, Assistant Professor, Department of Electrical and Computer Engineering

Jingtong Hu, PhD, Associate Professor, Department of Electrical and Computer Engineering

Thesis Advisor: Alan D. George, PhD, Mickle Chair Professor, Electrical and Computer Engineering

Copyright © by Ian Peitzsch
2023

Multiarchitecture Hardware Acceleration of Hyperdimensional Computing Using oneAPI

Ian Peitzsch, M.S.

University of Pittsburgh, 2023

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 class. These HDC models can classify information by transforming new input data into hyperdimensional space, comparing the similarity between transformed data with each class, 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 involves many repeated basic operations which are both easily parallelizable and pipelinable, 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 fastest inference latency, with an average of 0.28ms.

Table of Contents

Preface . . . viii

1.0 Introduction . . . 1

1.1 Background . . . 2

1.1.1 Hyperdimensional Computing . . . 2

1.1.1.1 Similarity (δ) . . . 2

1.1.1.2 Bundling (+) . . . 2

1.1.1.3 Binding ($\otimes$) . . . 3

1.1.1.4 Permutation (ρ) . . . 3

1.1.2 HDC Learning . . . 3

1.1.2.1 Encoding . . . 3

1.1.2.2 Training . . . 4

1.1.2.3 Retraining . . . 4

1.1.2.4 Inference . . . 4

1.1.3 oneAPI . . . 5

1.2 Related Work . . . 6

1.2.1 FPGA Acceleration . . . 6

1.2.2 GPU Acceleration . . . 7

2.0 System Architecture . . . 9

2.1 FPGA System Architecture . . . 9

2.2 GPU System Architecture . . . 12

3.0 Evaluation . . . 18

3.1 Inference . . . 18

3.2 Training . . . 20

4.0 Conclusions . . . 23

Bibliography . . . 25

List of Tables

Table 1: Clock frequency and area data for the FPGA designs. 14

List of Figures

Figure 1: High-level system architecture for inference with FPGA. 10

Figure 2: High-level system architecture for single-pass learning with FPGA. . . . 10

Figure 3: High-level system architecture for NeuralHD learning with FPGA. . . . 14

Figure 4: High-level system architecture for inference with GPU. 15

Figure 5: High-level system architecture for single-pass learning with GPU. 16

Figure 6: High-level system architecture for NeuralHD learning with GPU. 17

Figure 7: Comparison of throughput (images per second) with varying batch sizes. Higher values are better. 19

Figure 8: Comparison of inference latencies. Lower values are better. 20

Figure 9: Comparison of single-epoch training times. Lower values are better. . . 21

Figure 10: Comparison of training times using the NeuralHD training algorithm. Lower values are better. 22

Preface

This research was funded by industry and government members of the NSF SHREC Center and the National Science Foundation (NSF) through its IUCRC Program under grant number CNS- 1738783.

I would like to thank Intel for providing the hardware and software resources used in this research.

I would like to thank the members of the SHREC lab for guiding me throughout this research.

Special thanks to Mark Ciora for aiding in the implementation of the GPU code for this work. These contributions were essential for achieving the results within this work in a timely manner.

Finally, I would like to thank my family who have encouraged and supported me throughout this research and everything that has lead up to it. Their encouragement keeps me motivated to continue following my passions in research.

1.0 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 [10]. 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 [5], language recognition [2], and robot navigation [7]. 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 [10].

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 research 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.

1.1 Background

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

1.1.1 Hyperdimensional Computing

In HDC applications, data are represented as *hypervectors*: vectors in high-dimensional space (often >10,000). These hypervectors benefit from the "curse of dimensionality", which states that in high-dimensional spaces, randomly generated vectors are nearly orthogonal. This orthogonality makes random hypervectors able to represent different classifications of objects. The key aspects of hypervectors and their most common operations are covered in the following subsections.

1.1.1.1 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 $\delta(A, B) \approx 0$. The similarity function can be any distance measure, though commonly it is defined as the cosine similarity

$$\delta(A, B) = \frac{A \cdot B}{|A||B|} \tag{1–1}$$

Where $| \cdot |$ denotes the L2 norm.

1.1.1.2 Bundling (+)

Bundling combines hypervectors to make a hypervector that is similar to the inputs. 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.

1.1.1.3 Binding ($\otimes$)

Binding combines hypervectors to make a hypervector that is dissimilar to the inputs. 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. So, $C \oslash A \approx B$ and $C \oslash B \approx A$. Binding is often implemented as element-wise XOR of the input hypervectors.

1.1.1.4 Permutation (ρ)

Permutation rotates a hypervector and makes a hypervector that is dissimilar to the input. 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.

1.1.2 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.

1.1.2.1 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. There are many common ways of encoding feature vectors into hyperdimensional space, including using lookup tables and linear projection followed by non-linear operations. H can then be used by the HDC model for either inference or training.

1.1.2.2 Training

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

1.1.2.3 Retraining

Simple training does not always produce the best accuracy, so the model can be fine-tuned using retraining. Retraining works by comparing encoded hypervector H^l of class l to all class vectors C_j and calculating the prediction

$$l' = \underset{j}{\operatorname{argmax}}\, \delta(H^l, C_j) \tag{1–2}$$

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 learning factor. 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 $C_{l'}$.

1.1.2.4 Inference

Inferencing using an HDC classification model is achieved by selecting the class with the highest similarity to the encoded hypervector.The model predicts the encoded hypervector H is part of class l by finding

$$l = \underset{j}{\operatorname{argmax}}\, \delta(H, C_j) \tag{1–3}$$

As the δ function on large vectors is embarrassingly parallel, this step can be efficiently computed on both GPUs and FPGAs.

1.1.3 oneAPI

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. This single-source development process allows for reuse of both host and accelerator code leading to faster development and fewer lines of code when compared to other heterogeneous accelerator languages, such as OpenCL.

Like many other accelerators, oneAPI uses a host-accelerator model for offloading tasks in a heterogeneous system. Each accelerator device is associated with a command queue. This queue is used to send command groups from the host to the device for asynchronous execution. The command groups can either be run in a data-parallel mode or as a single task. GPUs operating in a data-parallel fashion can divide work into 1-, 2-, or 3-dimensional grids of work-groups. These work-groups are mapped to groups of threads. SIMD execution of threads in the same group is then achieved through the use of sub-groups. This execution is in contrast to data-parallel command groups for FPGAs, where instead the command group is auto-pipelined. Single-task submission for GPUs executes the command group as a single thread. For FPGAs, the single-task command group is the standard way of offloading.

OneAPI also provides methods for memory management between the host and accelerator, buffers and unified shared memory (USM). Buffers act as wrappers around data which can then be accessed on the accelerator using an accessor. Buffers abstract away the explicit data movement between the host and accelerator for ease of development. USM makes use of pointers to shared memory to facilitate data movement. 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.

In addition to methods of transferring data between host and accelerator, oneAPI provides a method of transferring data between kernels: pipes. Pipes are an FPGA-specific data movement method that uses on-device FIFO buffers to pass data between kernels. These pipes allow for kernels to operate concurrently, as data can be passed between concurrent

kernels instead of kernels needing to wait for another kernel to finish to get data. Additionally, these pipes reduce the number of reads and writes to global memory, since kernels can simply pass intermediate data between each other instead of constantly reading and writing from global memory.

To aide in general development, oneAPI provides various libraries that contain 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 [1]. Additionally, to aide in FPGA development, oneAPI provides useful reports to give area and throughput information about the design. 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.

1.2 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.

1.2.1 FPGA Acceleration

NeuralHD [10] 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, Zhou *et al.* achieved a 26.8× 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× speedup for FPGA-accelerated HDC inference over FPGA-accelerated DNN inference.

F5-HD [8] is an HDC accelerator framework. 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× faster training and 1.7× faster inference compared to an AMD R9 390 GPU. While F5-HD certainly reduces the design time to implement an optimized HDC accelerator, the framework assumes its encoding scheme is well suited for all applications. This assumptions is not necessarily true, as demonstrated in [9] where an encoding scheme better suited for neuromorphic camera data significantly improved accuracy. For such applications, the design space exploration for better encoding schemes can be explored using HLS tools, such as oneAPI.

1.2.2 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 [3] is one of the first GPU-focused frameworks for HDC. In its initial paper, XCelHD achieved 35× speedup over then state-of-the-art TensorFlow-based HDC implementation when benchmarked on an NVIDIA Jetson TX2, which is an embedded-scale GPU. 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 [4] is a more-recent GPU-focused HDC framework. OpenHD achieved upwards of 9.8× speedup for training time over XCelHD on the same device. Additionally, OpenHD achieved around 1.4× speedup for inference latency over XCelHD. OpenHD performs the same optimizations as XCelHD, but also makes use of automated data-type mutation. This automated data-type mutation tracks the value of elements in the hypervectors and only

increases the data type used to represent the elements when they exceed the range of the current type. For example, a hypervector consisting of integers where all elements are within the range $[-64, 64]$ can use the `char` data type to represent each individual element. When this hypervector is bundled with another hypervector with the same range, then the possible range of values in the result is $[-128, 128]$, which exceeds the range of `char`. So, in this case, the automated data-type mutation would convert to using the `short` data type. This feature 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.

2.0 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 [10] for the encoding method. This kernel relies on using D randomly generated basis vectors $\vec{B}_1, \vec{B}_2, ..., \vec{B}_D \in \mathbb{R}^n$. 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:

$$h_i = \cos(\vec{B}_i \cdot \vec{F} + b) \times \sin(\vec{B}_i \cdot \vec{F}) \tag{2–1}$$

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.

2.1 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. This number of compute units was determined through testing with varying numbers of compute units to have the best performance for inference latency. 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.

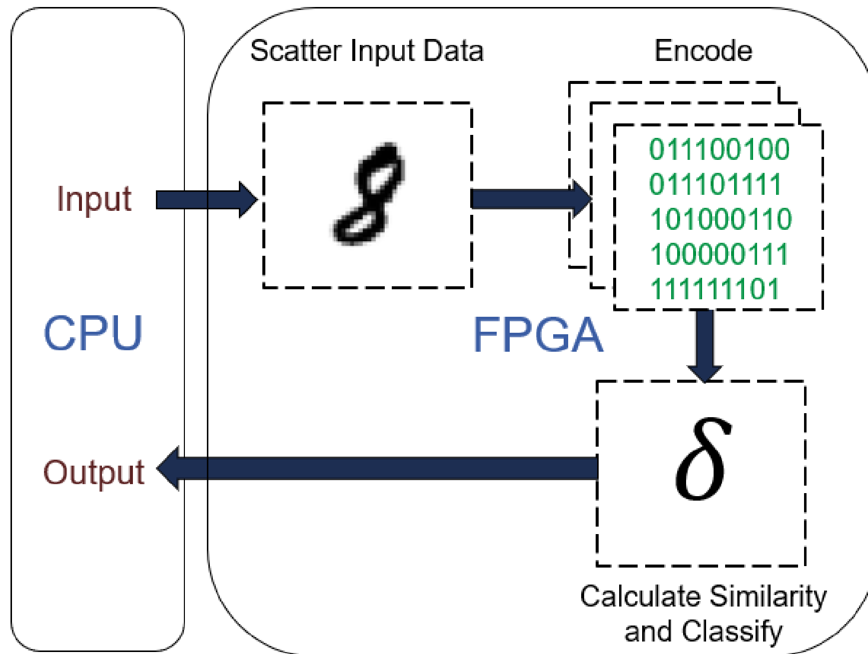

Figure 1: High-level system architecture for inference with FPGA.

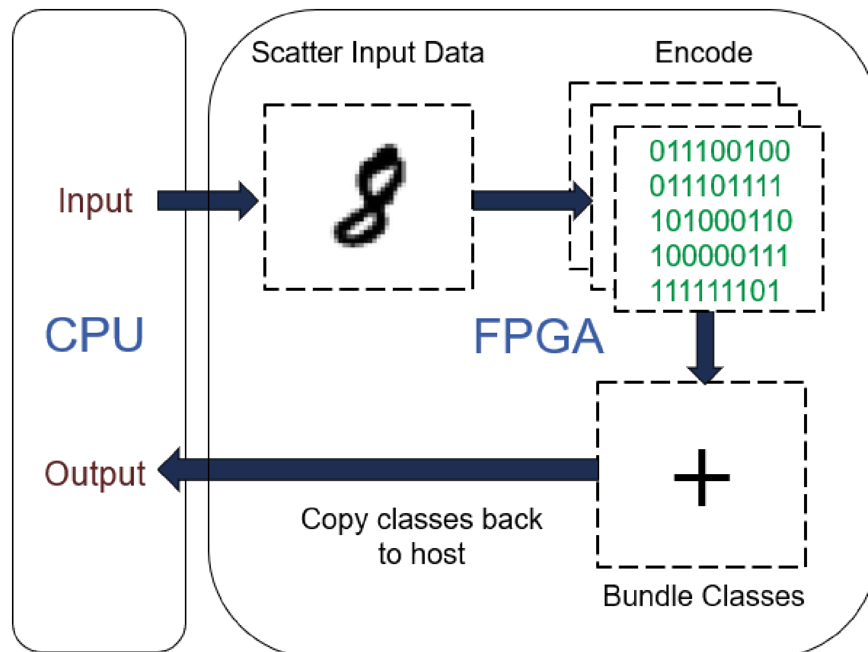

Figure 2: High-level system architecture for single-pass learning with FPGA.

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 from the inference implementation because of memory constraints, as can be seen in Table 1. These 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 the norm of the vector is forced to be 1.

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 1.1.2.3 using $\alpha = 0.037$. This process is repeated for an arbitrary number of iterations. The number of iterations used in this work is based on the number of iterations used in the sample implementation from [10]. 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. These exit conditions were determined by following the sample Python implementation provided by [10]. In all test cases, the training ended due to the convergence exit case.

2.2 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 matrix multiply as shown in Equation 2–2.

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

Then using oneMKL's element-wise cosine and sine kernels, 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 maximum similarity is sent back to the host.

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 identical to 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. This number of groups was used as it is the maximum number of working groups that the GPU this implementation was tested on allowed. 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. The first two exit conditions come from the sample implementation of Neural, as provided by [10]. The third exit condition was added after observing the model was converging to some value that was not 100%. In all test cases, the training finished due to this third exit case.

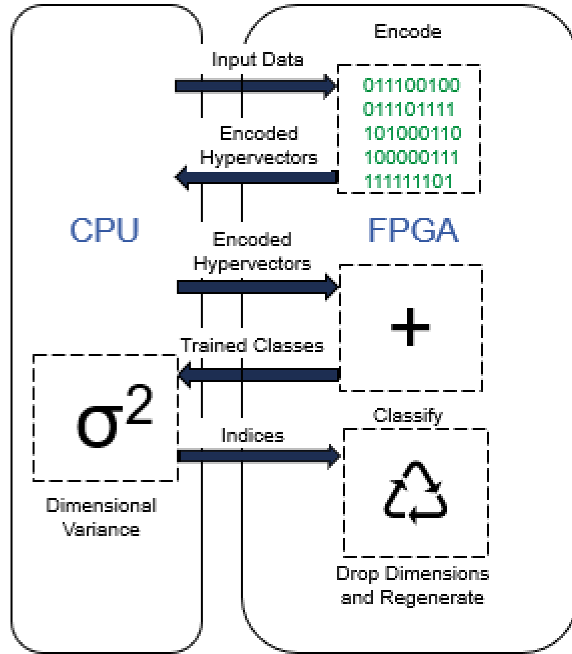

Figure 3: High-level system architecture for NeuralHD learning with FPGA.

Table 1: Clock frequency and area data for the FPGA designs.

Operation	Clock Freq. (MHz)	Initiation Interval (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

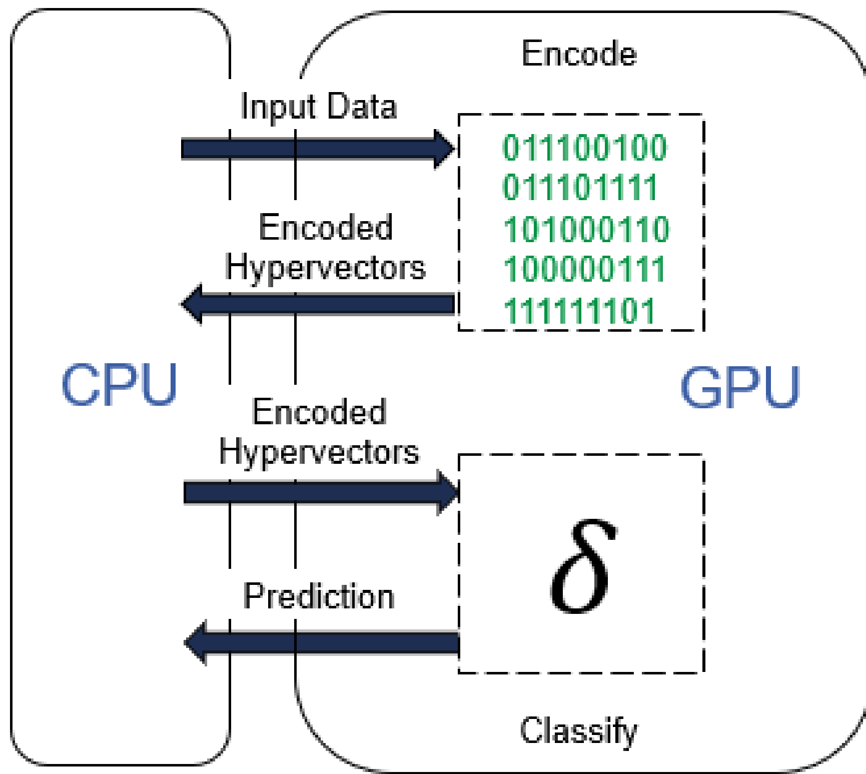

Figure 4: High-level system architecture for inference with GPU.

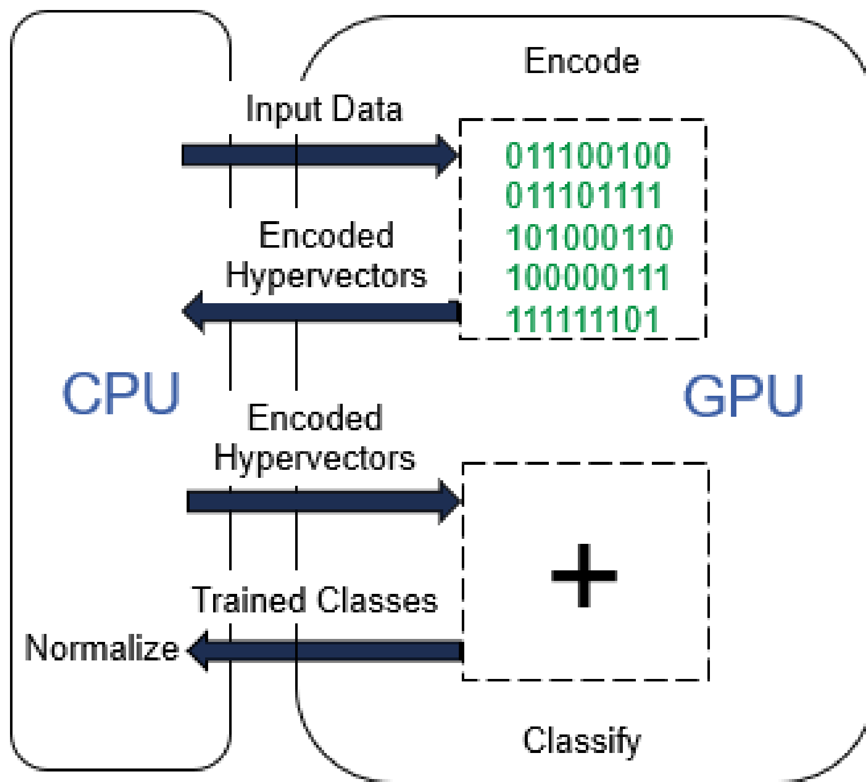

Figure 5: High-level system architecture for single-pass learning with GPU.

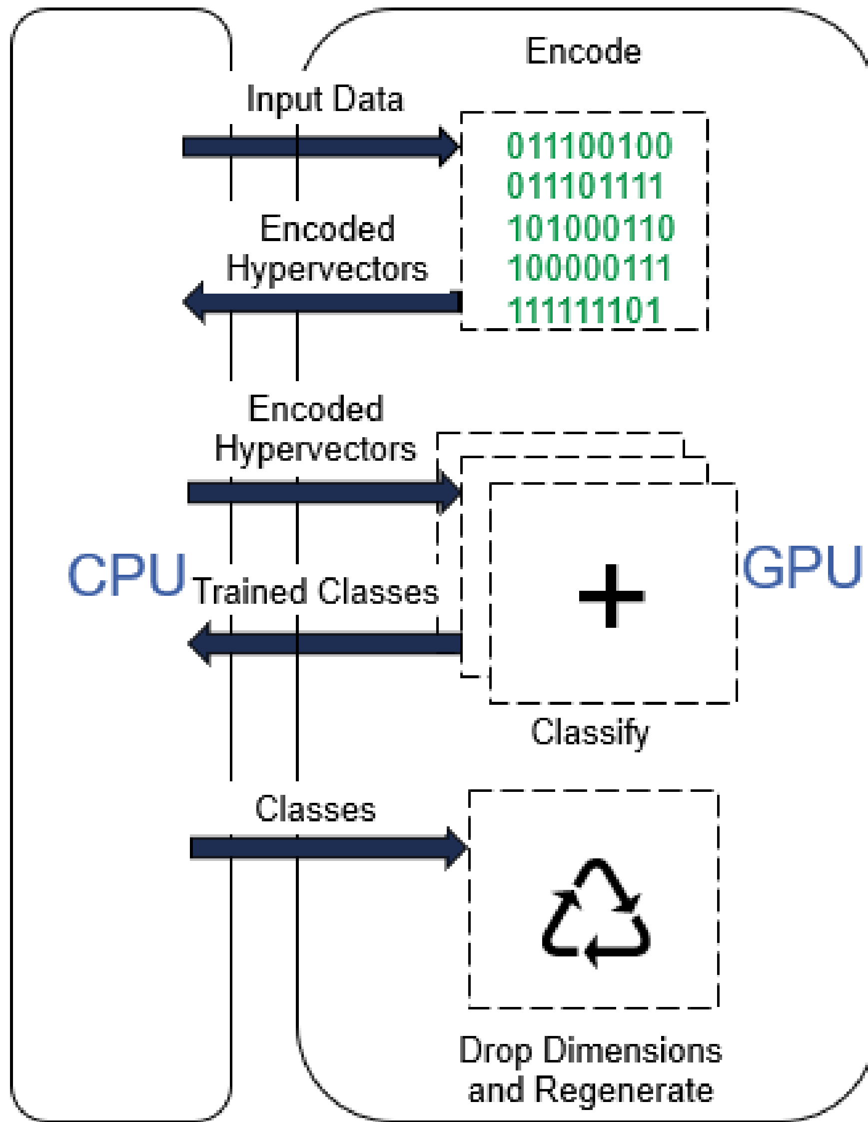

Figure 6: High-level system architecture for NeuralHD learning with GPU.

3.0 Evaluation

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 [6]. This MNIST dataset contains 60000 training images and 10000 test images of handwritten numbers 0 through 9. Each of these images is 28×28 pixels in size. MNIST was selected for this research as it was used in [10] as one of their benchmarking datasets. All metrics are compared to an 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. 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 model optimization is outside the scope of this work.

3.1 Inference

The inference latency benchmarks are shown in Figure 8. The FPGA design achieved the lowest latency, with a speedup of $3\times$ 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. This slowdown is due to the GPU's classification stage being implemented in a data-parallel fashion where each work group receives its own hypervector. Thus, in a single inference this implementation degenerates to a serial implementation. The FPGA implementation, on the other hand, does not use this style of classification, so it does not experience this degradation. Additionally, the UHD is an integrated GPU, so 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

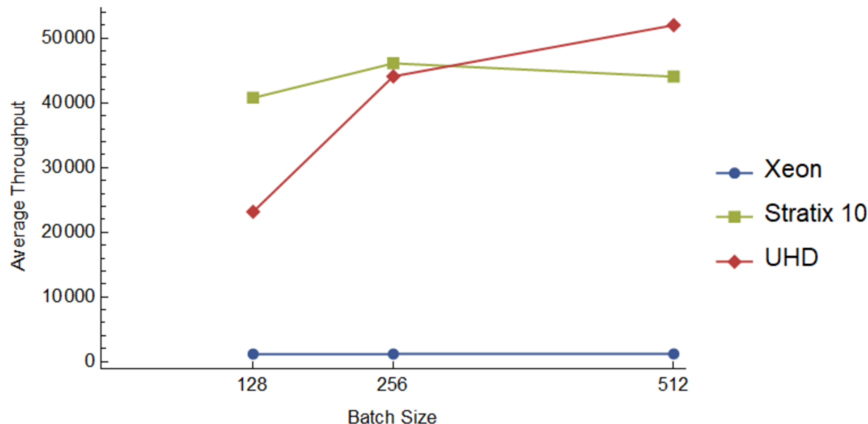

Figure 7: Comparison of throughput (images per second) with varying batch sizes. Higher values are better.

latency and give further speedup. Quantization of the HDC model would likely give further latency speedup, at the sacrifice of some accuracy. Using FP16 or BFloat instead of FP32 would reduce the necessary on-chip memory used by the model. Even further quantization to integer or fixed-point representations could additionally reduce latency for the FPGA implementation, as multiply and accumulate operations on these data types require fewer cycles to process than floating point representations.

Figure 7 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 and that the GPU can handle much more parallelism than the other devices. Increasing the batch size allows for the GPU's data-parallel classification stage to shine and make use of all available threads. The FPGA design, however experiences relatively little change in performance in comparison as its inference pipeline reaches full capacity with fewer input data. There is a small decrease with increasing the batch size

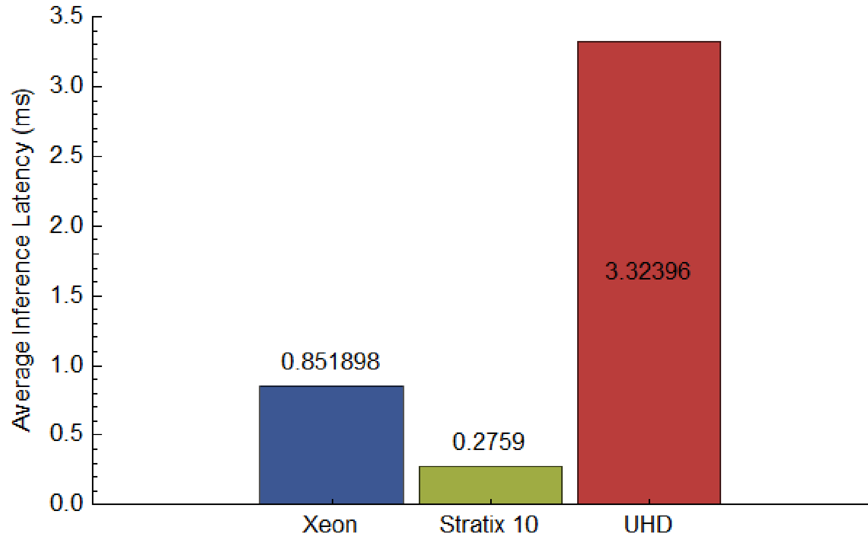

Figure 8: Comparison of inference latencies. Lower values are better.

from 256 to 512, but this can be attributed to pipeline stalls due to needing to read in data from DDR RAM. Similar to the inference latency, the FPGA design's inference throughout likely can be improved through quantization, as is demonstrated in [8] with a comparison of throughput using fixed-point, power-of-two, and binary representations of hypervectors.

3.2 Training

The results from the benchmarks of the single-epoch training are shown in Fig. 9. 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× over the CPU baseline. The FPGA design only achieved a speedup of 17.9× over the CPU baseline. The likely limiting factor with the FPGA design is the reduced number of compute units. This reduced number of compute units is limited by the onboard memory, as the read/write memory required for the class vectors takes up 18% of the BRAM,

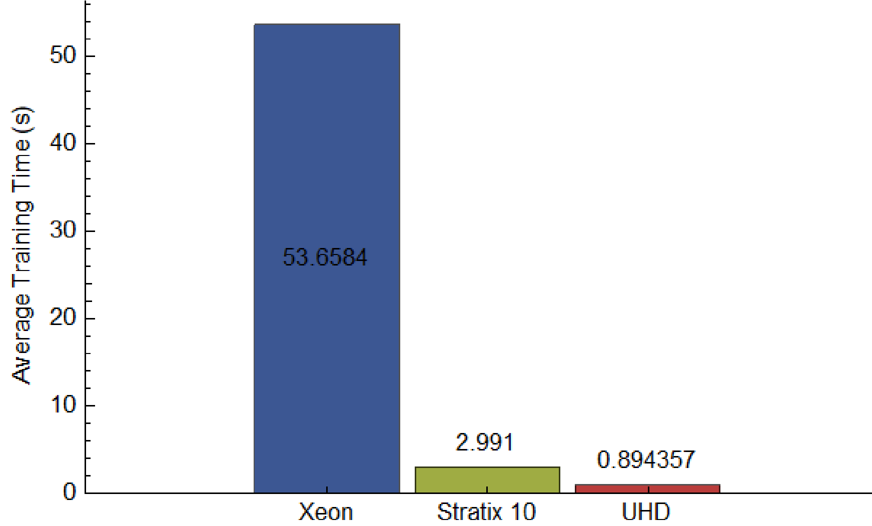

Figure 9: Comparison of single-epoch training times. Lower values are better.

while the read-only memory used in the inference design only takes up 2% of the BRAM. 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 due to not having as many parallel compute units.

Figure 10 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× over the 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 2.2. 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

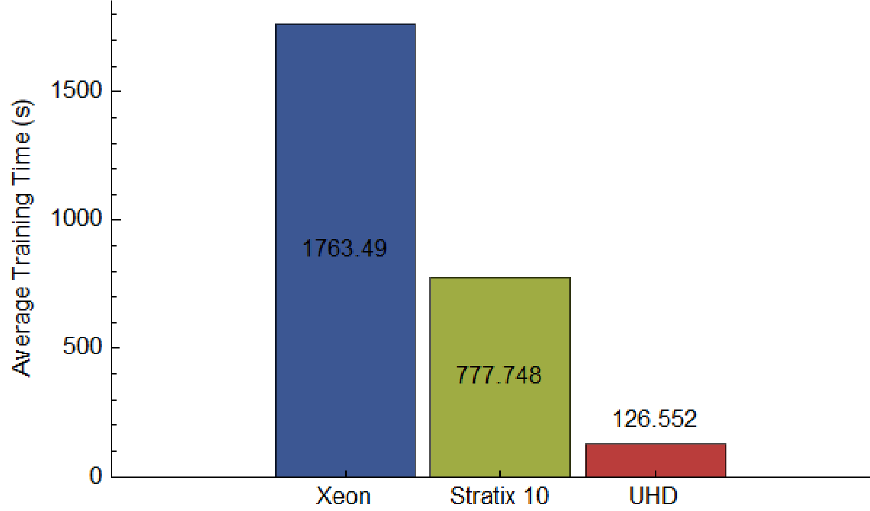

Figure 10: Comparison of training times using the NeuralHD training algorithm. Lower values are better.

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.

For both the GPU and FPGA designs, the speedups for NeuralHD is far less than the speedups for the single-epoch training. This phenomenon is due to HDC retraining only encoding once and then repeatedly retraining using those encoded hypervectors, since the encoding stage is the most computationally intensive part of the algorithm. So, with NeuralHD, encoding only occurs at the beginning and then after every regeneration. Most of the speedup observed in the FPGA and GPU implementations come from the encoding stage. With retraining reducing the amount of computation time spent in encoding the impact of the encoding speedup is lessened, and the impact of classification stage speedup is increased. However, the classification stage is not nearly as computationally intensive for the CPU since there a far fewer classes than input features, so there is not as much room for improvement over the CPU baseline.

4.0 Conclusions

This research benchmarks and compares GPU and FPGA accelerators using the novel heterogeneous design tool 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 compared to the CPU.

The GPU's better performance in the learning tasks and in throughput metrics is due to both the UHD having unified memory with the host and the UHD having more execution units than the FPGA design had compute units for the encoding stage. The unified memory for both device and host allows for the device the utilize all of the host's memory with no memory transfer costs. Thus, large amounts of input data can be sent to the GPU for processing with no overhead, unlike the FPGA as it is discrete and requires overhead to transfer memory from the host to the device. Additionally, the GPU's faster processing was achieved by its 32 execution units, each of which can execute eight 32-bit floating point operations per clock cycle. As the GPU implementation made use of optimized kernel libraries from oneMKL, these parts of the GPU implementation had many man-hours spent optimizing the design for the GPU architecture. Compared to the FPGA, there do not exist any oneAPI libraries to assist in the implementation process, so less time could be spent on design-space exploration. Furthermore, the GPU's 32 execution units, each capable of eight 32-bit floating point operations per clock cycle, allows for more floating point operations per second (FLOPS) than any of the FPGA implementations. This difference in FLOPS especially effects the encoding stage, which is the bottleneck of all of the designs, as it requires many FLOPS to perform. However, these properties of the GPU were not enough to lower its inference latency. The main contributing factor for the GPU's high inference

latency is the classification stage was implemented in a data-parallel method that puts each encoded hypervector into a workgroup for processing. This implementation allows for high throughput with batches of inputs, but for inferring a single input this stage is reduced to serial execution, causing a loss in performance. Since the FPGA implementation does not do this data-parallel style of classification stage, it does not see this performance degradation, which limits its throughput when scaled up to more data.

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.

Bibliography

[1] Intel. *Intel® oneAPI Math Kernel Library (oneMKL) - Data Parallel C++ Developer Reference*, 2022.

[2] Aditya Joshi, Johan T Halseth, and Pentti Kanerva. Language geometry using random indexing. In *Quantum Interaction: 10th International Conference, QI 2016, San Francisco, CA, USA, July 20-22, 2016, Revised Selected Papers 10*, pages 265–274. Springer, 2017.

[3] Jaeyoung Kang, Behnam Khaleghi, Yeseong Kim, and Tajana Rosing. Xcelhd: An efficient gpu-powered hyperdimensional computing with parallelized training. In *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 220–225, 2022.

[4] Jaeyoung Kang, Behnam Khaleghi, Tajana Rosing, and Yeseong Kim. Openhd: A gpu-powered framework for hyperdimensional computing. *IEEE Transactions on Computers*, 71(11):2753–2765, 2022.

[5] Yeseong Kim, Mohsen Imani, and Tajana S. Rosing. Efficient human activity recognition using hyperdimensional computing. In *Proceedings of the 8th International Conference on the Internet of Things*, IOT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[7] Alisha Menon, Anirudh Natarajan, Laura I. Galindez Olascoaga, Youbin Kim, Braeden Benedict, and Jan 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)*, pages 7335–7341, 2022.

[8] Sahand Salamat, Mohsen Imani, Behnam Khaleghi, and Tajana 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*, FPGA '19, page 53–62, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Zhuowen Zou, Haleh Alimohamadi, Yeseong Kim, M. Hassan Najafi, Narayan Srinivasa, and Mohsen Imani. Eventhd: Robust and efficient hyperdimensional learning with neuromorphic sensor. *Frontiers in Neuroscience*, 16, 2022.

[10] Zhuowen Zou, Yeseong Kim, Farhad Imani, Haleh Alimohamadi, Rosario Cammarota, and Mohsen 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*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.