EmbHD: A Library for Hyperdimensional Computing Research on MCU-Class Devices

Alexander Redding
University of California, San Diego
La Jolla, USA
alredding@ucsd.edu

Xiaofan Yu
University of California, San Diego
La Jolla, USA
x1yu@ucsd.edu

Shengfan Hu
University of California, San Diego
La Jolla, USA
shh042@ucsd.edu

Pat Pannuto
University of California, San Diego
La Jolla, USA
ppannuto@ucsd.edu

Tajana Rosing
University of California, San Diego
La Jolla, USA
tajana@ucsd.edu

ABSTRACT

This paper presents EmbHD, a library for embedded Hyperdimensional Computing research on severely resource-constrained computing devices. The increasing demand for power-efficient and low-latency machine learning in mobile applications has driven the need for offloading computation onto edge devices. The library aims to enable efficient machine learning inference and training on resource-constrained microcontrollers by leveraging hardware-optimized Hyperdimensional operations.

CCS CONCEPTS

• **Computer systems organization** → **Embedded software**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

Hyperdimensional Computing, TinyML, microcontrollers

ACM Reference Format:
Alexander Redding, Xiaofan Yu, Shengfan Hu, Pat Pannuto, and Tajana Rosing. 2023. EmbHD: A Library for Hyperdimensional Computing Research on MCU-Class Devices. In *The 2nd Workshop on Networked Sensing Systems for a Sustainable Society (NET4us '23), October 6, 2023, Madrid, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3615991.3616404

1 INTRODUCTION

The growing need for power-efficient, low-latency machine learning (ML) in mobile applications has been the driving force behind the push for offloading computation onto the "edge" [2]. Many of these mobile applications fall into the category of the Internet of Things (IoT), an area dominated by smart-sensing devices which primarily perform inferences on sensor data [9]. Deployments such as these simply do not (or ideally, should not) require cloud computing resources; a service which requires a non-trivial amount of energy to access.

The timeless engineering challenge has been understanding how we can get the most out of our mobile devices. What is the greatest amount of useful computation we can do for the least amount of power? This type of performance maximization involves both hardware and software optimizations. In terms of hardware, one of the most impactful design choices one can make is the target computer.

Fully-fledged multi-core systems with application processors may offer the best raw speed, but can be power hungry and expensive at scale. Most IoT deployments opt for more energy-efficient cores that trade lesser performance for greater sustainability. Historically relegated to simple 8 and 16-bit machines, the newest generation of MCUs have seen a transition to more capable 32-bit processors, with the ARM Cortex-M family being amongst the most popular. While these single-core systems running at tens of MHz may sit towards the bottom of the computational performance ladder, they are unparalleled in power-efficiency.

Figuring out how to run modern edge computing workloads (ie. ML inference) on resource-constrained MCUs has been an active area of research in recent years. Since 2019 this concept has become known as TinyML, which seeks to open the prospect of "executing optimized ML models on ultra-low-power (<1mW) MCUs with minimal power consumption" [4]. MCU-class devices typically operate with <100KB of memory and 1-2MB of flash storage. The ability to perform the same ML task that would run on a multi-core system on an MCU instead, is quite powerful. The energy

NET4us '23, October 6, 2023, Madrid, Spain

ACM ISBN 979-8-4007-0365-2/23/10.
https://doi.org/10.1145/3615991.3616404

spent per event would be on the order of milliwatts instead of watts.

Current TinyML solutions such as TensorFlow Lite Micro (TFLM) offer a workflow in which one can train and export a quantized ML model to an MCU for inference [11]. **This paper proposes an alternative to traditional embedded neural network ML libraries by employing hardware-optimized Hyperdimensional Computing (HDC) operations in order to support efficient ML inference and training on an MCU using statically allocated memory.**

2 HYPERDIMENSIONAL COMPUTING

Hyperdimensional Computing is a "brain-inspired" form of computation that takes advantage of random, high-dimensional (D $\geq$ 10,000) binary vectors [8] known as hypervectors. Although in its infancy, HDC has been gaining attention as an alternative or replacement to traditional machine learning algorithms [5]. Compared to conventional neural networks, operations on binary hypervectors are highly efficient and hardware friendly as they only require basic bitwise operations (OR and XOR).

2.1 Operations

Hyperdimensional Computing can be imagined as a pseudo-instruction-set-architecture for a computer that operates on very large registers (hypervectors). *Binding* and *bundling* are two of the most commonly used operations, or instructions, in HDC.

Binding. A function denoted as $\otimes : \mathcal{H}_a \times \mathcal{H}_b \rightarrow \mathcal{H}_y$ which takes two points a and b in *hyperspace* (hyperdimensional space) $\mathcal{H}$ and outputs a point y which is dissimilar to both input points [12]. Binding is reversible and is used to associate two hypervectors together. For binary hypervectors, elements can be bound with XOR operations.

Bundling. A function denoted as $\oplus : \mathcal{H}_a + \mathcal{H}_b \rightarrow \mathcal{H}_y$ which takes two points a and b in hyperspace $\mathcal{H}$ and outputs a point y which is similar to both input points [12]. Bundling is used to combine a set of points. For binary hypervectors, elements can be bundled with OR operations.

2.2 Encoding

In order to use HDC for ML, a mapping from one's data to hyperspace must be created, which is known as *encoding* [8]. There are certain helpful mathematical properties of vectors in hyperspace which influence the optimal encoding method.

Random Hypervectors. Given a dimension large enough, any two randomly chosen hypervectors will be approximately orthogonal [8]. If $D = 10,000$, then each hypervector differs by about 5,000 bits from each other. These are known as *random hypervectors*. Because we are operating on binary hypervectors, one can calculate the distance between any two hypervectors by finding their *hamming distance*. Random hypervectors are useful when representing data whose values have no correlation with each other.

Level Hypervectors. If one were to take a single random hypervector N_0 and then flip a few bits to create another hypervector $N_1 \approx N_0$, then $N_1 < N_0$ but it would remain similar. Let k be the integer denoting the index in which a hypervector was created. If one were to repeat this process in which each new hypervector is a copy of the last with the addition of a few random bit flips, any hypervector N_k will be similar to other hypervectors whose index lies within a certain close range. But as a hypervector's index lies further away from k, the pair become increasingly dissimilar. Two indices far away from each other will have little to no similarity. These are known as *level hypervectors* and they are useful when representing data in a spectrum where similar values in reality should preserve similarity in hyperspace.

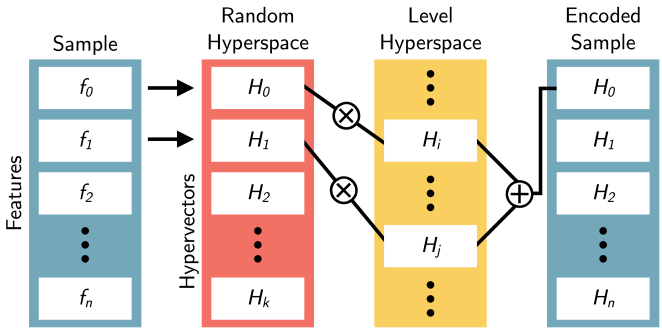

Figure 1: HDC Encoding

Although other approaches exist, mapping data to random and level hypervectors is one of the most common encoding methods for HDC. Both of these hypervector types are sufficient for encoding many 2D samples which consist of feature vectors $\{f_0, f_1, \ldots f_n\}$. A specific feature's index in this vector is mapped to a corresponding random hypervector, and a feature's value is associated with a level hypervector. An index and value hypervector are bound for each feature and then all of the products are bundled into a single hypervector representing the encoded sample. This process can be as seen in Figure 1.

Training and Inference

After encoding a sample, the resulting feature vector can then be bundled with a class hypervector corresponding to the feature's label during training. As training progresses, each class hypervector becomes increasingly similar to its constituent sample hypervectors. These class hypervectors

essentially become the weights of our ML model which can be updated with a single bundling operation. In order to classify a sample hypervector, we can simply see which class hypervector it is most similar to.

An additional benefit of operating on large hypervectors is that they are inherently robust to random bit-flips. Recent work has shown that quality loss on noisy hardware is considerably greater for deep neural networks when compared to HDC [7]. This is particularly beneficial for intermittent computing systems which do not have reliable sources of energy and may experience data corruption during failures.

3 TENSORFLOW LITE MICRO

TensorFlow Lite Micro (TFLM) is one of the most popular open-source ML frameworks for embedded systems. Developed in 2020, it aims to address the limitations of prior ML tools [3]. TFLM provides a pipeline for training and exporting statically-allocated, quantized neural network models to microcontrollers, as well as a target-optimized interpreter for running inferences on the model.

4 EMBHD

This paper proposes an alternative workflow for training and running ML models on microcontrollers by relying on Hyperdimensional Computing instead of traditional neural networks. EmbHD is essentially a hyperdimensional virtual machine written in C that enables HDC operations on microcontrollers.

Currently, MCUs have been constrained to only running ML inferences locally. Training algorithms such as gradient descent are too resource heavy and slow to efficiently run on MCUs during deployment. With HDC, model updates can happen with a single addition which greatly simplifies the training process. Enabling on-device training helps to make MCUs more capable edge computing machines and can open the door for smarter IoT applications.

4.1 System Design

EmbHD is built upon a generalized matrix operation-based framework. Although this library is primarily for Hyperdimensional Computing, matrix representations are preferred for maximum code re-usability for other applications and for future additions to the library that may extend beyond HDC.

Hypervector Representations. EmbHD uses a human-readable format for storing hypervector parameters and data in the form of matrices. Matrices are structs which contain an array of `MData` structs that provide a simple way to access a wide range of 32-bit data representations without type-casting. Consequently, all matrices are 32-bit-aligned. A single matrix is sufficient to represent a hypervector, or an entire hyperspace. Here is an example of how to create a $D = 10,000$ binary hypervector using an `MData` array and a `Matrix` struct:

```
MData  binary_hv_data[313];
Matrix binary_hv = {
    .dtype  = MBin,
    .height = 1,
    .width  = 10000,
    .size   = 313,
    .data   = binary_hv_data;
};
```

Matrices can exist in both volatile and non-volatile memory. EmbHD also has a Python library to export PyTorch Tensors to a `Matrix` H file which can be included in one's application source code.

HDC Operations. Because EmbHD is built upon a general matrix data representation, it contains functions for adding and multiplying matrices which are effectively the same as bundling and binding for Hyperdimensional Computing. These functions are called `MAdd` and `MMult` respectively, and they (as well as other matrix functions in EmbHD) operate on single matrix rows at a time. Here's an example of how to multiply two rows:

```
MMult(&dest, 0, &src0, 0, &src1, 0);
```

In this example, we are multiplying the first row of the `src0` matrix with the first row of the `src1` matrix and storing the result in the first row of the `dest` matrix. In terms of HDC, we can consider both `src0` and `src1` to be hyperspaces and we are selecting two hypervectors from each of them to bind.

Although EmbHD can be compiled for many different architectures, it was designed with the popular ARM Cortex-M4 in mind. This processor's DSP instructions make it an ideal target for low-power HDC ML applications. The energy savings from these DSP instuctions can be seen in Figure 2 which compares the Cortex-M4's DSP instructions versus the equivalent emulated code to perform the same task.

Because HDC operations involve adding and multiplying large hypervectors, DSP instructions can help to parralelize this when using integer data representations. Using binary hypervectors offer even more parralelization as one can operate on 32 dimensions per each processor instruction.

4.2 Training

Training an HDC model can be done both offline or on-device. EmbHD was designed to pair well with the Python Hyperdimensional Computing library TorchHD [6]. Similarly to the Tensorflow to TensorFlow Lite Micro pipeline, one can train

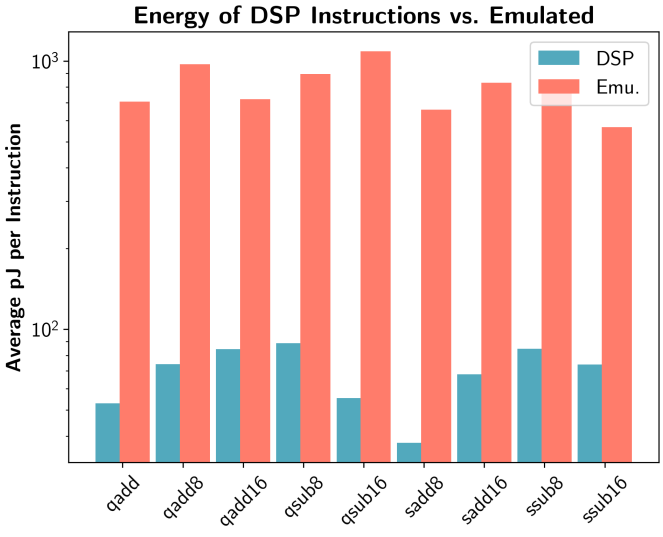

Figure 2: Comparing Cortex-M4 DSP instructions with manual loops

an HDC ML model in TorchHD and export it to a `Matrix` representation for use with EmbHD.

Regardless of whether one chooses to train offline or on-device, one must still export the hypervectors necessary for encoding sample data. This can also be done with TorchHD like so:

```
import torch, torchhd
import export_matrix_lib

DIMENSION = 10000
NUM_HV = 100

hv = torchhd.random(NUM_HV, DIMENSION)
convert_mdata(hv, "randhv", static=True)
```

In this example we create a random hyperspace of $D = 10,000$ which contains 100 hypervectors using TorchHD. We then call the EmbHD function `convert_mdata` which will convert the Tensor returned by TorchHD into `Matrix` representation. The second string argument sets the variable name of the `Matrix` struct for reference in the application source code. The final argument `static` is a Boolean which tells the function to add additional C attributes to place the generated `Matrix` into flash memory if set to True.

For the purposes of encoding, one can pre-generate and export random and level hyperspaces and store them in flash memory as they will never change. If training on-device, the weights (or class) `Matrix` can be stored in RAM to allow for model updates.

Table 1: Details of the microcontroller testing platform, a SparkFun RedBoard Artemis.

CPU	RAM	Flash	Clock
ARM Cortex-M4F	1M	384K	48/96MHz

Table 2: Datasets Used for Comparison. MNIST is images of handwritten numbers, and ISOLET is recordings of spoken letters.

Name	|Features|	|Classes|	|Training|	|Testing|
MNIST	768	10	60,000	10,000
ISOLET	617	26	6,238	1,559

4.3 Inference

After a sample datum has been encoded to hypervector form, inference can be performed by finding which class hypervector the encoded sample is closest to. When using binary hypervectors, one can find this by using hamming distance. With EmbHD this looks like the following:

```
unsigned int distance;
MHamDist(&distance, &src0, 0, &src1, 0);
```

`MHamDist` gives us the number of differrent bits between the first row of the matrix `src0` and the first row of the matrix `src1`, or in terms of HDC it gives us the distance between the first hypervectors in the `src0` and `src1` hyperspaces. This distance is stored by reference to an integer variable.

5 EVALUATION

To evaluate the performance of EmbHD and the trade-offs between using HDC versus traditional neural network libraries, we compare EmbHD with one of the most popular embedded neural network ML libraries: TensorFlow Lite Micro. Both EmbHD and TFLM were compared on a SparkFun RedBoard Artemis development board which contains an Ambiq Apollo3 microcontroller. The specifications of this MCU can be seen in Table 1. EmbHD was compiled with ARM DSP instructions enabled and the TFLM cortex_m_generic target was compiled with the ARM CMSIS-NN optimized kernel.

5.1 Experimental Setup

The performance of EmbHD and TensorFlow Lite Micro were compared over the MNIST and ISOLET datasets. Characteristics for these datasets can be found in Table 2. TorchHD was used to train models offline for EmbHD using baseline single-pass HDC. The standard TensorFlow to TensorFlow Lite Micro pipeline was used to train neural network models offline for TFLM. EmbHD was compared with TFLM using

Table 3: Performance Results of EmbHD and TFLM

Dataset	Library	Parameters	Accuracy	µJ per Inference
MNIST	TFLM Float	1 hidden layer of 64 nodes	96%	20.6
	TFLM Quant		96%	6.05
	EmbHD	$D = 7,000$	80%	2036.68
ISOLET	TFLM Float	1 hidden layer of 128 nodes	95%	32.82
	TFLM Quant		95%	11.17
	EmbHD	$D = 10,000$	81%	1999.86

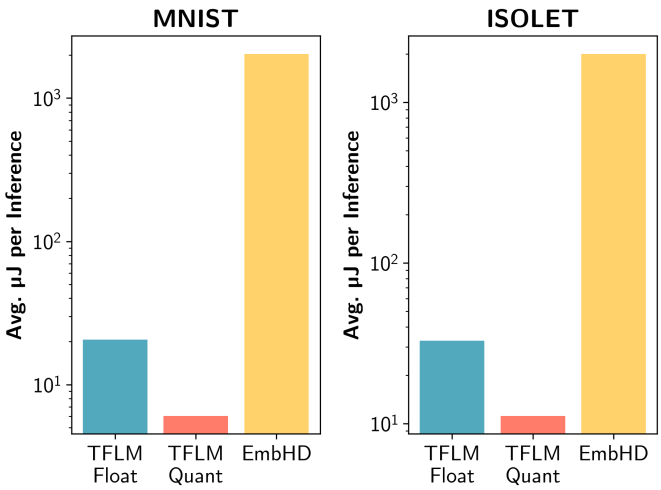

Figure 3: Energy comparison of TensorFlow Lite Micro versus EmbHDper ML inference

both floating point and quantized integer weights. All three models had their performance averaged over 10 inferences and had their power consumption measured using a Rocket-Logger.

5.2 Results

The results of the library comparisons over the MNIST and ISOLET datasets can be found in Table 3. The "Parameters" column in this table describes the how the models were created. The "Accuracy" column lists each model's respective accuracy over the entire testing data for each dataset. The last column "µJ per Inference" is the average energy expended per each model inference on the Apollo3 microcontroller used for testing.

At the surface level, EmbHD performed signifcantly worse than TFLM using floats and quantized weights. On both datasets EmbHD saw about 15% lower accuracy than TFLM and saw two orders of magnitude more energy expended per inference, as can be seen in Figure 3. These results aren't too surprising as Hyperdimensional Computing is still a young area of machine learning and we are still trying to understand the best methods for using it. There already exist many considerable improvements [1, 7, 10] over baseline HDC, however many of these use HDC in addition to a neural network front-end. This paper is primarily focused on comparing the pure spirit of HDC to traditional embedded neural network libraries.

Instead, it makes more sense to reason about these results as trade-offs. Unlike traditional ML libraries like TensorFlow Lite Micro, EmbHD allows one to actually train a model on a deployed microcontroller. This comes at a cost of accuracy and energy. Although Hyperdimensional Computing is inherently highly hardware friendly and highly parallel, it still requires significantly more mathematical operations per encoding and inference when compared to a small single-layer neural network.

6 CONCLUSION

EmbHD is not a direct replacement for libraries such as TensorFlow Lite Micro, but it serves as an alternative with unique characteristics distinct from neural network libraries. For edge computing applications that require devices to be able to autonomously update their models, Hyperdimensional Computing can offer a promising future. The ultimate goal of EmbHD is to make future HDC research on the edge easier and to provide a starting point for others looking to move their HDC work onto an embedded system.

7 ACKNOWLEDGEMENTS

This work was funded in part by a National Science Foundation (NSF) grant (CNS-2233894). We thank the anonymous reviewers for their aid in improving the paper.

REFERENCES

[1] Toygun Basaklar, Yigit Tuncel, Shruti Yadav Narayana, Suat Gumussoy, and Ümit Y. Ogras. 2021. Hypervector Design for Efficient Hyperdimensional Computing on Edge Devices. *TinyML Research Symposium* (2021).

[2] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. 2020. An Overview on Edge Computing Research. *IEEE Access* 8 (2020), 85714 – 85728.

[3] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, Rocky Rhodes, Tiezhen Wang, and Pete Warden. 2021. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. *4th MLSys Conference* (2021).

[4] Dr. Lachit Dutta and Swapna Bharali. 2021. TinyML Meets IoT: A Comprehensive Survey. *Internet of Things* 16 (2021), 100461.

[5] Lulu Ge and Keshab K. Parhi. 2020. Classification using Hyperdimensional Computing: A Review. *IEEE Circuits and Systems Magazine* 20, 2 (2020), 30–47.

[6] Mike Heddes, Igor Nunes, Pere Vergés, Dheyay Desai, Tony Givargis, and Alexandru Nicolau. 2022. Torchhd: An Open-Source Python Library to Support Hyperdimensional Computing Research. (2022).

[7] Alejandro Hernandez-Cane, Namiko Matsumoto, Eric Ping, and Mohsen Imani. 2021. OnlineHD: Robust, Efficient, and Single-Pass Online Learning Using Hyperdimensional System. *Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2021), 56–61.

[8] Pentti Kanerva. 2009. Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors. *Cognitive Computation* 2, 1 (2009), 139–159.

[9] Asif Ali Laghari, Kaishan Wu, Rashid Ali Laghari, Mureed Ali, and Abdullah Ayub Khan. 2021. A Review and State of Art of Internet of Things (IoT). *Archives of Computational Methods in Engineering* 29 (2021), 1395–1413.

[10] Yejia Liu, Shijin Duan, Xiaolin Xu, and Shaolei Ren. 2023. MetaLDC: Meta Learning of Low-Dimensional Computing Classifiers for Fast On-Device Adaption. *TinyML Research Symposium* (2023).

[11] Swapnil Sayan Saha, Sandeep Singh Sandha, and Mani Srivastava. 2022. Machine Learning for Microcontroller-Class Hardware: A Review. *IEEE Sensors Journal* 22, 22 (2022), 21362–21390.

[12] Anthony Thomas, Sanjoy Dasgupta, and Tajana Rosing. 2021. A Theoretical Perspective on Hyperdimensional Computing. *Journal of Artificial Intelligence Research* 72 (2021), 215–249.