Designing a Programmable Accelerator for Sparse Distributed Binary Hyperdimensional Computing

Tim Deboel, Ryan Antonio, Marian Verhelst
Department of Electrical Engineering, MICAS-ESAT, KU Leuven, Belgium.

Abstract—**Sparse Distributed Binary Hyperdimensional Computing (SDB-HDC) is a framework for executing ultra-low-power AI tasks. A programmable accelerator for SDB-HDC can make use of its properties to efficiently run IoT applications. The energy efficiency and similar performance of SDB-HDC has caused it to recently gain in popularity, yet, the differences in the non-generalized application-specific encoding schemes make it difficult to design an efficient programmable accelerator. State-of-the-art accelerators focus on dense distributed binary (DDB). However, it underutilizes HDC's potential causing high switching activity due to the density of its vectors. This work starts by investigating the different sparse HDC encoding schemes. We then propose a new sorted threshold method during training that improves classification accuracy. Afterward, we develop a programmable accelerator that integrates the encoding schemes, applying temporal reuse to achieve high energy efficiency and throughput. The design is synthesized using Skywater 130nm technology and manages to achieve a competitive area and better energy efficiency compared to similar works that use DDB with a smaller process node. Access to the sorted THR method and a programmable ultra-low-power SDB-HDC accelerator that can support varying IoT applications allows for a better research space for this algorithm.**

I. Introduction

MACHINE LEARNING is used more and more in today's applications, as it allows computer systems to recognize patterns and make decisions without human intelligence. However, its downside of requiring high computational power and energy usage makes it badly suited for IoT devices. Most machine learning applications use power-hungry floating-point operations like multiply-accumulate. These often use classical Von-Neumann architectures, which have a data transfer bottleneck between the processor and memory. These downsides make them hard to integrate with IoT devices because these often require extremely low power usage. [1]

Hyperdimensional Computing (HDC) is a brain-inspired computing paradigm, which can perform machine learning tasks with amazing power efficiency. HDC represents data based on very high-dimensional vectors and simple operations, which allows for lighter computational loads compared to using conventional scalar numbers. It encodes data based on the combination of sequences of representations through bitwise manipulation, using much more energy-efficient operations. Its main use lies in classification algorithms, where a best match with the stored representations determines the output. Depending on the application, it can provide an energy-efficiency increase of over 2-10x and a throughput increase of 2-5x. [2]–[7] Other notable features include fast learning, robustness, and easy parallelism on top of cheap and simple hardware implementations.

Sparse-Distributed Binary HDC is a subset of the HDC-based algorithms. Where the applications referenced above utilize binary vectors with a density of 50%, SDB-HDC is based on high-dimensional binary vectors with a density of ones less than this, usually $< 10\%$. [8] In hardware this sparsity causes less switching of transistors, resulting in less dynamic energy usage and a further increase of HDC's benefits.

The downside of SDB-HDC is that there is a variety of non-generalized encoding schemes that are necessary to achieve good accuracy. Another drawback is that for certain applications there can be a slight accuracy drop compared to the dense distributed version (DDB). However, with the right encoding schemes this is nonexistent, or only marginal (1-3%). [8]–[12]

Currently, lots of research has been done on DDB-HDC algorithms and hardware for their extremely low energy usage. [2]–[7] Yet, very little research has been aimed at converting these DDB algorithms to an SDB-HDC representation for even higher power savings. The challenges for SDB-HDC are finding methods to achieve good accuracy on the one hand, and creating hardware for these different encoding schemes on the other hand. Existing hardware is either very application-specific or hard to generalize because of the variety of methods.

The goal of this work is to develop a programmable SDB-HDC accelerator. To this extent, we start by investigating the SDB-HDC encoding schemes and discover opportunities for improvement. The sorted threshold method is developed, modifying the regular thresholded sum encoding scheme during training for an accuracy increase of 2-20%. Afterward, a programmable hardware accelerator that supports the most common encoding schemes and their various combinations is designed. This hardware architecture combines their operations into an efficient and flexible implementation, with minimal latency. Optimizations are implemented to further decrease the area and energy usage, by reducing registers and operations, and utilizing temporal reuse.

II. The SDB-HDC Encoding Schemes

At this moment, the two most commonly used encoding schemes for SDB-HDC are Context-Dependent Thinning (CDT) and thresholded sum. The choice and implementation of the encoding schemes are heavily application-specific.

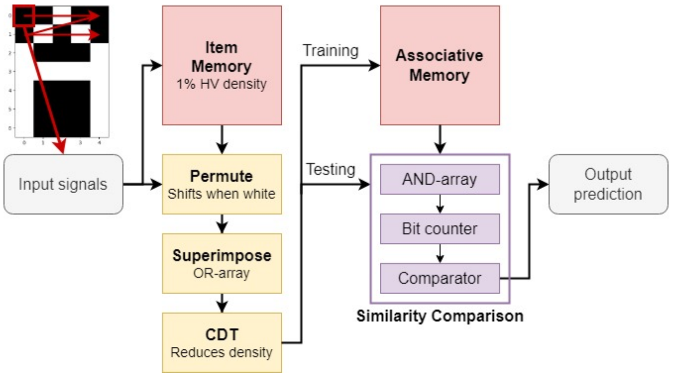


Fig. 1. Data flow for the SDB character recognition application.

A. Context-Dependent Thinning

The working of the context-dependent thinning (CDT) encoding scheme can be shown by implementing it in the example application of SDB-HDC character recognition, following [11]. Figure 1 shows the data flow for the character recognition algorithm. The input is a pixel grid displaying a character, that contains 35 black or white pixels. These pixels are iterated over from left to right and top to bottom and serve as inputs for the algorithm. The item memory (IM) acts as a storage for hypervectors (long arrays of a randomly generated distribution of 1s and 0s). In this case, it contains 35 HVs with a 1% density of 1s randomly distributed throughout each HV. Each pixel is linked to one of these HVs, thus the pixel coordinate serves as the address for the IM. These pixels are loaded sequentially into the encoder, which combines the entire input/pixel grid into a single encoded HV.

When the pixel is white, it is permuted by 1 bit to the right, and no permutation happens when it is black. The resulting (permuted) HVs are then disjuncted together by an OR-array, which we call ”bundling”. This produces a resulting bundled HV, with a much higher density than its constituents. Finally, the density is lowered again by the CDT function, to produce the final encoded HV. This encoded HV serves as a representation for the entire pixel grid and is used to predict the displayed character.

The CDT procedure is a similarity-preserving method to control the sparsity after binding or bundling multiple HVs together. [13] It is a well-studied procedure [13], [14] that has been integrated into multiple applications. [11], [15], [16] The resulting density after encoding using this procedure is determined by two factors: the starting density of the IM HVs, and the k-factor of the CDT procedure as seen in its formula [11]:

$$Z = \bigvee_{i} X_i \tag{1}$$

$$\langle Z\rangle = \bigvee_{k=1}^{K} (Z \wedge Z(k)) = Z \wedge \vee_{k=1}^{K} Z(k) \tag{2}$$

Here X_i are the input HVs, $\vee$ represents disjunction (OR) and $\wedge$ represents conjunction (AND). $\langle Z\rangle$ is the thinned vector, and $\langle Z(k)\rangle$ represents the k-th rotation of Z (k cyclic shifts). K=1 achieves the lowest density, as a higher K means more superposed vectors, which increases the density.

TABLE I
SDB-HDC CHARACTER RECOGNITION RESULTS FOR 0.98% IM DENSITY.

Distortions	Accuracy (%)	
n	D=1,024	D=2,048
0	100	100
1	98.62	99.11
2	96.58	97.69
3	94.15	95.96
4	89.54	92.38

The associative memory (AM) module handles the classification/prediction part of the algorithm. During training, each character’s encoded HV is stored on a certain address of the AM to serve as an ”object HV” for comparison. Later during the testing phase, the encoded HV based on the test data is compared against each object HV stored in the AM by measuring the overlap. This is measured by taking a conjunction (using an AND-array) and counting the active bits. These are then compared for each object HV, and the one with the highest overlap determines the output prediction. This provides a robust way of encoding and classifying input data for machine learning algorithms.

We can test the performance of this algorithm by introducing distortions; for n distortions, n random pixels are flipped from black to white or vice versa. Table I shows the average accuracy for all letters over 100 repetitions of random distortions, based on the HV size. We can see that a higher dimensionality lends to better results because larger HVs can hold more information.

B. Thresholded Sum

The other most commonly used encoding scheme for SDB-HDC makes use of permutation for binding and accumulation with thresholding for bundling. Its main advantage lies in the fact that a controllable threshold is used for flexible density control after the accumulation. The working of this encoding scheme will be examined based on the example application of language recognition.

Figure 2 shows an SDB-HDC implementation of the encoding section for this application. This algorithm is based on an n-gram sliding window, that stores the n previous characters. To achieve good accuracy with the SDB-version of this algorithm, [12] proposes a method where each letter in the n-gram receives a ”unique” permutation, to reduce the bias from similar n-letter permutations on the encoded HV. Each letter is given a unique signature between 0-31, which can for example be their IM address. The permutation received by each letter HV in the n-gram window is then calculated as the XOR of the signatures of all other $n-1$ letters. A majority sum is used to combine the n-gram HVs, which is followed by another accumulator to bundle them over the entire length of the input text. This result is then binarized using a controllable threshold (THR) based on the number of input characters, to produce the encoded text HV.

During training, 21 European languages are encoded with about a million bytes of training data for each, retrieved from

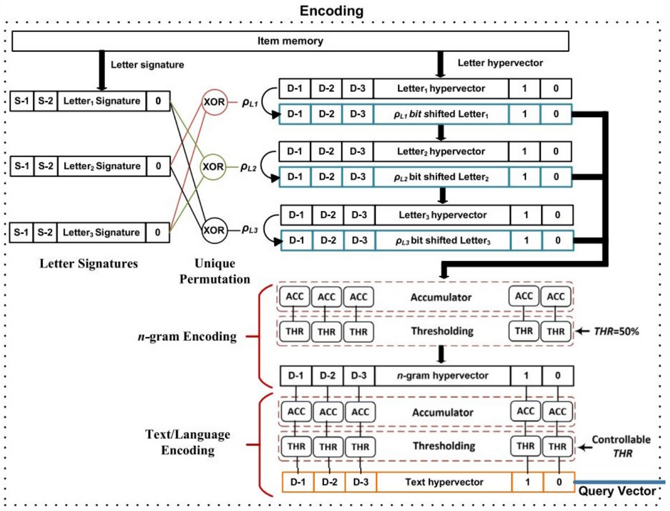


Fig. 2. Architecture for SDB-HDC language recognition. (retrieved from [12])

[17]. These ”language HVs” are then stored in the associative memory, to be compared with the encoded HV from the test data in the same way as described above for character recognition. During testing, the classification accuracy is determined over 1,000 samples per language from [18] and averaged over all 21 languages. However, the results showed inconsistencies in the accuracies, as some languages had a far lower average accuracy than most others. This was linked to those languages having a lower language HV density, which can be attributed to the fact that lower densities will inherently lead to lower overlap scores. To combat this, we implement a different method of thresholding the accumulated HVs during training, which results in nearly the same density for all language HVs.

C. Sorted THR

We call this method *Sorted THR*, as it is based on a sorting step to calculate the threshold. The training step for these algorithms only has to happen once and is usually performed in software, which means we can make use of more complex algorithms here. We modify the second bundling step of the SDB algorithm, but only during the training phase. It performs an accumulation as normal and creates a vector of length D containing the accumulated values for each dimension, but the thresholding step is replaced by the new method. Sorted THR sorts the accumulated vector from large to small values, and a cutoff index is defined as follows:

$$\mathrm{idx} = \mathrm{round}(\mathrm{thrval} * D) - 1 \tag{3}$$

Here *thrval* is a value between 0 and 1, representing the desired encoded HV density. The value of the sorted array at this index is then taken as the threshold value.

This new threshold value is applied in a regular thresholding step to the (unsorted) accumulated vector. Like the regular thresholded sum, all values greater than or equal to the threshold are set to 1 and all others are kept 0 in the resulting encoded HV. Listing 1 shows a Python implementation of this algorithm. Here *v* is the hypervector containing the accumulated values after training and *final_density* is a value between 0 and 1 that determines the resulting encoded HV density.

TABLE II
LANGUAGE RECOGNITION ACCURACY COMPARISON FOR VARYING HV SIZE.

Algorithm	D=2000	D=4000	D=6000	D=8000	D=10000
Dense	69.4%	86.4%	90.5%	92.1%	96.1%
Sparse	74.2%	86.5%	91.5%	93.6%	95.4%
Sorted THR	**95.1%**	**97.6%**	**98.0%**	**98.5%**	**98.7%**

Listing 1. Python implementation of the sorted THR function

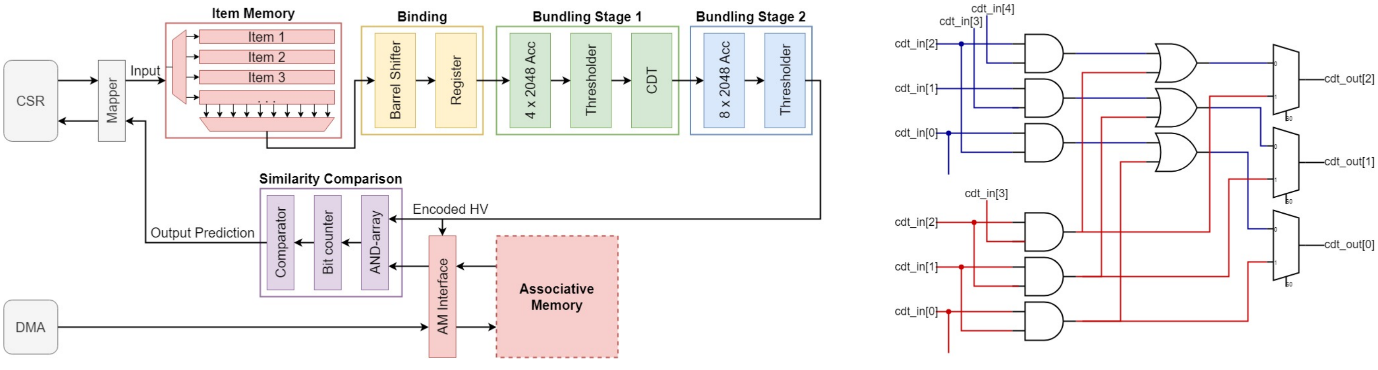
```
sort_array = np.copy(v)
sort_array[::-1].sort()
cutoff = round(len(v)*final_density)
threshold = sort_array[cutoff-1]

for i in range(len(v)):
    if v[i] >= threshold:
        v[i] = 1
    else:
        v[i] = 0
```

Using this method, the resulting density can be precisely controlled during training, removing the accuracy loss from a reduced overlap due to density disparities. Table II shows the performance increase by using this algorithm on the sparse language recognition from [12]. It is also compared with the DDB version from [2]. The dense and sorted THR versions have a window size of 3, and the sparse algorithm uses a 4-gram, where the parameters giving the highest accuracy are selected. Note that the sorted THR method gives the best results when using a 3-gram with 2% IM density. The sparse version from [12] performed best using 4% IM density and a 4-gram, thus our modification also decreases the latency and energy usage during testing on top of the accuracy boost. This method can potentially boost the viability and adoption of SDB-HDC algorithms, as it solves its key problem of slight accuracy loss compared to DDB.

III. ACCELERATOR DESIGN

To solve the problem of the integration of the SDB-HDC encoding schemes with each other in hardware, a programmable accelerator design is proposed. This architecture integrates the most commonly used encoding schemes of CDT and thresholded sum, in a flexible, yet efficient accelerator. The full design can be seen in Figure 3a.

The communication between this accelerator and the processor or core that controls it happens through a Control and Status Register (CSR) and the associative memory interface. The CSR is a stack of 32-wide registers that provide the functionality to program the accelerator, input data, and retrieve the output prediction. A status register can be read to retrieve the current information and predicted output, while all other registers can be written to program the desired functionality. The ”input” register is continuously written during execution when the accelerator is ready to receive a new input. This is used to keep providing inputs until *input_done* is set high, after which the similarity comparison will begin.



(a) (b)

Fig. 3. (a) The programmable SDB-HDC accelerator architecture. (b) Diagram for a CDT implementation with a maximum k-factor of 2.

The mapper module implements the signature encoding scheme described in [12] for a maximum sliding window size of 12. This is used by [3] and is sufficient for nearly all applications. Besides this, it also sets the input-ready signal high when the accelerator can receive a new input. Next up, the item memory is made of a lookup table that stores 35 hypervectors, as this is enough to support most applications. [3], [5], [11], [12] Keeping a small amount of sparse HVs in the IM has the additional benefit that during synthesis, the bit locations where all HVs contain a 0 will be optimized and removed from the LUT. This reduces its size even further and makes a LUT a good choice for SDB-HDC. Additionally, if a larger IM would be necessary, this design can easily be scaled up as all other hardware besides the input size doesn't need to change.

Afterward, the HVs get sent through the binding module, which implements the permutation function. This is done with a fully combinational barrel shifter made of 6 layers of muxes, that can apply anywhere from 0 to 63 rotations in a single cycle. A maximum of 63 is chosen, because the window size of 12 for signature encoding can create a maximum shift of $35 + 12 = 47$ and no other applications would require more.

The permuted result is then sent to the bundling module, where multiple HVs get bundled together into one. This module is made up of 2 stages that each have an accumulator and thresholding, because 2 stages is sufficient for most applications. [3]–[5], [11], [12] Both stages make use of saturating accumulators, where the accumulator saturates when the MSB turns to 1 and stops counting afterward. This can save on area and energy usage because counting above the thresholding value doesn't have any benefit. The first accumulator is 2048 by 4 bits, meaning a minimum of 8 ones are necessary before saturation happens. Usually, the first stage has a maximum threshold value of 50%, so at least 16 HVs can be added before any accuracy loss is possible. Furthermore, because we are using sparse HVs, the chance of there being all ones in the same bit location for all 16 is extremely slim. Coupled with the redundancy of HDC, there is a lot more margin before accuracy loss becomes a problem in reality. The thresholder sets its output to 1 when the accumulated value is greater than or equal to the threshold and to 0 otherwise. This combination doubles as a disjunction (for the CDT encoding scheme) when the threshold value is chosen to be 1.

In the first stage, this is followed by a fully combinational implementation of the CDT function with a maximum k-factor of 3. Increasing the k-factor also increases the resulting density, so a k-factor that is too high would bring the output density close to the input. This would mitigate the effect of the CDT operation, hence why a small maximum value was chosen. Figure 3b shows the CDT implementation for a maximum k-factor of 2, where the bottom output would be selected for K=1, and the top one for K=2.

The second bundling module implements another accumulation and thresholding function, where the saturating accumulator uses a 2048-by-8-bit register. It functions in the same way as the one from stage 1, but this accumulator can handle 128 ones before saturating. This means that e.g. if THR=0.30%, 426 ones can be added before saturation has an effect, which should be plenty given that we work with sparse HVs. Additionally, the threshold for stage 2 is not static, but calculated by multiplying a threshold value with the number of inputs. This uses a counter that can handle 2048 different inputs, as the most encountered during language recognition is 1139 characters and other applications usually don't need that many.

The associative memory itself is not part of the accelerator design, as we are focusing on the data path. We created an AM interface to interact with the larger (shared) memory of the processor that controls this accelerator. It is designed to be fully compatible with the PULP platform [19], so its AM interface can be used to read data from the shared memory. The base address and max AM address entered into the CSR describe the address range where the AM hypervectors are stored in this memory. The accelerator uses these to determine the read addresses. The AM interface can then read 2048 bits at a time, to perform one similarity comparison each clock cycle. For testing purposes, a memory of 32 64x32 register banks is used in simulation to store up to 32 trained HVs, but this is not included in the synthesis.

A 2048-wide AND array is used to compute the overlap with the HV read from the memory, followed by a bit counter to count all active bits and determine the similarity score.

TABLE III
SYNTHESIZED AREA AND AVERAGE POWER USAGE BASED ON HV SIZE.

Property	D=512	D=1024	D=1536	D=2048
Area [mm^2]	0.77	1.49	2.26	3.03
Power [mW]	13.26	25.05	37.04	40.37

This counter is implemented in a fully combinational way; it uses groups of adder trees that are optimized by the synthesis tool to compute the overlap in a single cycle. The address corresponding to the highest overlap is stored and (after subtraction by the base address) returned as the predicted output to the CSR.

The latency of the accelerator is made as low as possible, by minimizing the time between inputs. This means that for character recognition, it can take one input every cycle, and for n-gram language recognition, it can take one input every n cycles. Additionally, the lowest number of registers that avoid timing errors is used and the switching in every module has been minimized. Everything combined leads to this final architecture, which can support a clock period of $t_{CLK} = 8.5$ ns.

IV. PERFORMANCE RESULTS

The functionality of the design is tested and the latency is retrieved using simulations in Questasim. The accelerator is synthesized using the Hammer flow [20], and the open source Skywater 130nm PDK [21]. For this, we also use commercial Cadence tools like Genus and Innovus, and simulation tool VCS Synopsys. As mentioned earlier, we focus on the accelerator data path, so the memory part of the AM is not synthesized.

Table III shows the synthesized area and average power usage for different HV sizes. The power metric is the average power usage for a mixture of tests consisting of all 26 character recognition letters, a language recognition sample of 26 characters, and 9 mixed tests based on a 20-character sample. These mixed tests explore different accelerator settings and edge cases such as a sliding window size up to 12, a CDT k-factor up to 3, only CDT in stage 1 bundling, only stage 2 bundling active, etc. A detailed breakdown of the area usage for each module when D=2048 can be seen in Figure 4. The bundling module takes up most of the area due to its massive accumulators, which require a lot of register and logic size. Note that *AM* refers to the ”AM module”, meaning the memory interface and similarity comparison, but not the memory itself.

Table IV shows how many clock cycles are taken by the encoding and classification steps for the character and language recognition tests. The latency numbers are computed on the right, by multiplying these with the clock period. This and all subsequent results show the language recognition test when using a sample of 100 characters, as we will use this to compare it to similar DDB accelerators.

The energy usage can then be computed by multiplying the latency and average power usage. This is displayed in Figure 5, where the average latency for the mixed tests is used (894 ms).

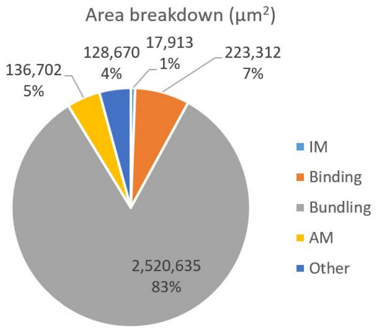


Fig. 4. Area breakdown for the accelerator with D=2048.

TABLE IV
LATENCY BREAKDOWN FOR CHARACTER AND LANGUAGE RECOGNITION.

Test	Char [cc]	Lang [cc]	Char [ns]	Lang [ns]
Input data	35	300	297.5	2550
Classification	26	22	221	187
Other	7	12	59.5	102
Total	68	334	578	2839

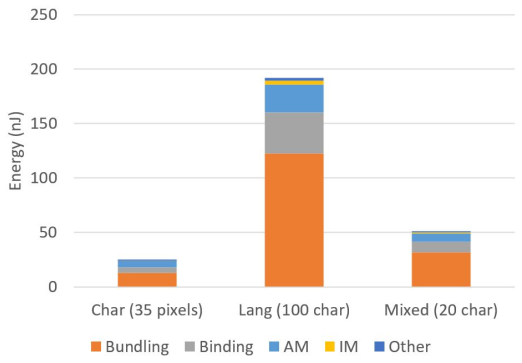


Fig. 5. Energy usage for different applications with D=2048.

TABLE V
PERFORMANCE COMPARISON WITH [22].

Property	DDB design	This design
Technology	TSMC 28 HPM	Skywater 130
Total Cell Area	1.27 sq. mm.	3.03 sq. mm.
t_{CLK}	2.4 ns	8.5 ns
Total estimated power	267 mW	67.7 mW
Total number of Cells	4.73×10^5	1.93×10^5
Number of Buf/Inv Cells	37501	15377

Finally, we can compare the performance to similar programmable DDB-HDC accelerators. We can best compare it to the programmable DDB-HDC accelerator from Datta et al. [22], as this work uses the same dimensionality D=2,048. However, its design is built on the TSMC 28nm High-K/Metal-gate (HKMG) node, which is a much smaller process than the Skywater 130nm we use. A comparison of the main specifications can be seen in Table V. The total estimated power refers to the average power used during the language recognition test with a 100-character input sentence. Note that the smaller process node allows for a significantly smaller cell

TABLE VI
Performance comparison for language recognition with [22] and [23].

Design	Technology	Area [mm^2]	Power [mW]	Clock Frequency	Energy/test [nJ]
Eggimann et al.	65 nm	1.43	86.5×10^{-3}	100 kHz	1205
Eggimann et al.	GF22	0.29	23.7×10^{-3}	100 kHz	332
Datta et al.	TSMC28	1.27	267	416.67 MHz	250
This work	**Skywater130**	**3.03**	**67.7**	**117.65 MHz**	**192**

area and clock period. Further, the item memory used in this design is significantly larger, taking up 62% of its total area, while it only takes up about 6% in our design. A larger item memory makes the design more flexible but is unnecessary for most applications. [3], [5], [11], [12] Do note that this is a synthesis result and the memory storing the AM data is not included, slightly reducing our numbers.

Another DDB-HDC programmable accelerator we can compare the design to is created by Eggimann et al. [23]. What's unique about this design, is that it uses HV rematerialization for their item memory. This means that instead of using a ROM or LUT to store pre-generated HVs, they are generated on the fly through mixing and permutations to reduce the area at the cost of higher latency.

The DDB language recognition application from [2] has been tested on both accelerators, so we can use it to compare the performance. The results comparing this design [23] and Datta et al.'s work [22] with our design are shown in Table VI. Note that the SDB version of this algorithm using sorted THR achieves the best results with a 3-gram sliding window. However, the DDB implementation maximizes accuracy using a 4-gram, which makes this version inherently have a higher latency. Eggimann et al.'s work has significantly lower average power, but their low clock frequency increases the latency and energy usage. We can see that our work has the lowest energy per prediction, highlighting the benefit of using SDB-HDC. Additionally, our design only needs 334 clock cycles (2839 ns) to perform the prediction for a 100-character sentence, which is a lot better than the 1400 cycles used by [23] due to their rematerialization.

V. Conclusion

This work[1] presents two contributions to the field of Sparse Distributed Binary Hyperdimensional Computing (SDB-HDC), known for its high energy efficiency in executing machine learning tasks. Firstly, the sorted threshold algorithm was introduced during training with the thresholded sum encoding scheme. This boosts classification accuracy by removing density disparities in the encoded hypervectors. It was applied to the language recognition application and managed an accuracy increase of 2.5% for large HV sizes, and up to 20% for smaller sizes. This addresses the typical accuracy limitations of SDB-HDC and could increase its viability and adoption.

The second contribution is the design of a programmable hardware accelerator for SDB-HDC IoT applications. This accelerator integrates the most common encoding schemes while achieving high speed and energy efficiency. When comparing our design to other state-of-the-art programmable accelerators, it manages to outperform similar designs in lower technology nodes on energy per prediction and maintain a competitive area usage. These advances enhance the implementation and integration of SDB-HDC algorithms, leading to a better research space for these algorithms and their integration into IoT devices.

References

[1] S. Basu, R. E. Bryant, G. De Micheli, T. Theis, and L. Whitman, "Nonsilicon, non-von neumann computing-part i [scanning the issue]," *Proceedings of the IEEE*, vol. 107, no. 1, pp. 11–18, 2019.

[2] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:9812826

[3] M. Imani, T. Nassar, A. Rahimi, and T. Rosing, "Hdna: Energy-efficient dna sequencing using hyperdimensional computing," in *2018 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI)*, 2018, pp. 271–274.

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

[5] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, 2017, pp. 1–8.

[6] A. Moin, A. Zhou, A. Rahimi, S. Benatti, A. Menon, S. Tamakloe, J. Ting, N. Yamamoto, Y. Khan, F. Burghardt, L. Benini, A. C. Arias, and J. M. Rabaey, "An emg gesture recognition system with flexible high-density sensors and brain-inspired high-dimensional classifier," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5.

[7] D. Kleyko, E. Osipov, A. Senior, A. I. Khan, and Y. A. Şekerciogğlu, "Holographic graph neuron: A bioinspired architecture for pattern processing," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 6, pp. 1250–1262, 2017.

[8] D. Rachkovskij, "Representation and processing of structures with binary sparse distributed codes," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 2, pp. 261–276, 2001.

[9] K. Schlegel, P. Neubert, and P. Protzel, "A comparison of vector symbolic architectures," *Artificial Intelligence Review*, vol. 55, no. 6, pp. 4523–4555, Dec. 2021. [Online]. Available: http://dx.doi.org/10.1007/s10462-021-10110-3

[10] M. Laiho, J. H. Poikonen, P. Kanerva, and E. Lehtonen, "High-dimensional computing with sparse vectors," in *2015 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, 2015, pp. 1–4.

[11] D. Kleyko, E. Osipov, and D. A. Rachkovskij, "Modification of holographic graph neuron using sparse distributed representations," *Procedia Computer Science*, vol. 88, pp. 39–45, 2016, 7th Annual International Conference on Biologically Inspired Cognitive Architectures, BICA 2016, held July 16 to July 19, 2016 in New York City, NY, USA. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S187705091631660X

[12] M. Imani, J. Hwang, T. Rosing, A. Rahimi, and J. M. Rabaey, "Low-power sparse hyperdimensional encoder for language recognition," *IEEE Design & Test*, vol. 34, no. 6, pp. 94–101, 2017.

[1] See the codebase at https://github.com/TimDeboel/SDB-HDC-accelerator

[13] D. Kleyko, A. Rahimi, D. A. Rachkovskij, E. Osipov, and J. M. Rabaey, "Classification and recall with binary hyperdimensional computing: Tradeoffs in choice of density and mapping characteristics," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 12, pp. 5880–5898, 2018.

[14] D. A. Rachkovskij and E. M. Kussul, "Binding and normalization of binary sparse distributed representations by context-dependent thinning," *Neural Computation*, vol. 13, no. 2, p. 411–452, Feb. 2001. [Online]. Available: https://doi.org/10.1162/089976601300014592

[15] M. Hersche, E. M. Rella, A. Di Mauro, L. Benini, and A. Rahimi, "Integrating event-based dynamic vision sensors with sparse hyperdimensional computing: a low-power accelerator with online learning capability," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 169–174. [Online]. Available: https://doi.org/10.1145/3370748.3406560

[16] 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," *CoRR*, vol. abs/2111.06077, 2021. [Online]. Available: https://arxiv.org/abs/2111.06077

[17] U. Quasthoff, M. Richter, and C. Biemann, "Corpus portal for search in monolingual corpora," in *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, N. Calzolari, K. Choukri, A. Gangemi, B. Maegaard, J. Mariani, J. Odijk, and D. Tapias, Eds. Genoa, Italy: European Language Resources Association (ELRA), May 2006. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2006/pdf/641_pdf.pdf

[18] P. Koehn, "Europarl: A parallel corpus for statistical machine translation," https://www.statmt.org/europarl/, 2005.

[19] P. Platform, "Pulp faqs," https://pulp-platform.org/. [Online]. Available: https://pulp-platform.org/

[20] "Introduction to hammer," https://hammer-vlsi.readthedocs.io/. [Online]. Available: https://hammer-vlsi.readthedocs.io/en/stable/index.html

[21] google, "Github - google/skywater-pdk: Open source process design kit for usage with skywater technology foundry's 130nm node." [Online]. Available: https://github.com/google/skywater-pdk?tab=readme-ov-file

[22] S. Datta, R. A. G. Antonio, A. R. S. Ison, and J. M. Rabaey, "A programmable hyper-dimensional processor architecture for human-centric iot," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 439–452, 2019.

[23] M. Eggimann, A. Rahimi, and L. Benini, "A 5 uw standard cell memory-based configurable hyperdimensional computing accelerator for always-on smart sensing," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 10, pp. 4116–4128, 2021.