The Hyper-Dimensional Processing Unit: Energy-Efficient Machine Learning Using Vector-Symbolic Architectures

Youbin Kim

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-232
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-232.html

December 20, 2024

The Hyper-Dimensional Processing Unit: Energy-Efficient Machine Learning Using Vector-Symbolic Architectures

By

Youbin Kim

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Jan M. Rabaey, Chair
Professor Bruno Olshausen
Professor Sayeef Salahuddin

Fall 2024

The Hyper-Dimensional Processing Unit: Energy-Efficient Machine Learning Using Vector-Symbolic Architectures

Abstract

The Hyper-Dimensional Processing Unit: Energy-Efficient Machine Learning Using Vector-Symbolic Architectures

by

Youbin Kim

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Science

University of California, Berkeley

Professor Jan M. Rabaey, Chair

Hyper-Dimensional computing (HDC) is a machine learning framework that has made inroads in low-power edge-AI applications. With simple bitwise vector operations and a small memory footprint, HDC demonstrates improved energy-efficiency for biosensing classification tasks compared to conventional machine learning methods. Recent interest in HDC has developed complex algorithms that enable the use of HDC systems for cognitive reasoning and control applications. Although previous hardware for HDC achieve impressive energy-efficiency for certain tasks, they face several issues with the growing application space. This dissertation covers the goals, design, optimization, and implementation of an energy-efficient multipurpose processor for HDC.

The first half of the dissertation overviews the recent expansion of HDC algorithms and the corresponding hardware to implement them efficiently. The shortcomings of previous hardware for HDC is analyzed and used to create goals for the Hyper-Dimensional Processing Unit (HPU), the first multipurpose HDC processor. The second half describes the physical realization and optimization of the HPU, characterized with two tape-outs. HPUv2 achieves an impressive energy-efficiency of 168 pJ per operation, making it competitive with previous application-specific hardware. The fabrication, verification, and characterization of the HPU architecture demonstrate the viability of an energy-efficient multipurpose processor that can enable intelligent HDC systems on the edge.

Contents

List of Figures

List of Tables

Acknowledgments

This dissertation and its corresponding research would not have been possible without the help and mentorship of many people. First, I would like to thank Professor Jan Rabaey who has been an incredible advisor throughout my Ph.D. His expertise and insights were instrumental in finding my research interests and shaping my academic career, and he has helped me always keep my eye on the bigger picture. I am grateful for his kindness, patience, and vibrant energy that has guided me through the challenges of this journey.

I would also like to thank Professors Sayeef Salahuddin and Bruno Olshausen. Their discussions and feedback have been invaluable in shaping this work. I have had the pleasure of collaborating with many brilliant researchers. Thank you to Mohamed Ibrahim, with whom I worked closely to plan and implement HPUv1. Thank you to Alisha Menon and Braeden Benedict for their contributions to the HD recall of reactive behavior project. Thank you to Nathania Santoso and Cecil Symes for their work in characterizing SRAM partition power for HPUv2. Thank you to Tamzid Razzaque for his assistance in the HPU testing process.

I owe many thanks to the Berkeley Wireless Research Center (BWRC) and its wonderful staff. In particular, I would like to thank Brian Richards for his guidance during the tape-out process for both HPUv1 and HPUv2, and Anita Flynn for her help in setting up the physical testing infrastructure. I also thank Columba Candy Corpus-Steudeman and Mikaela Cavizo-Briggs who have answered my countless administrative emails and questions.

Thank you to TSMC who have funded my research and manufactured both chips from my Ph.D.

Finally, thank you to my family and friends who always stood by me. This journey would not have been possible without you all.

Chapter 1

Introduction

1.1 Energy-Efficient Machine Learning

In the last decade, machine learning (ML) has brought forth a drastic and rapid advancement in computing and its applications. From artificial intelligence chatbots like OpenAI's GPT to image recognition like Apple Face ID, ML is quickly becoming a core part of our technological usage. This shift in computing paradigms is fueled by the massive network of smart sensors, devices, and actuators that exist in modern society. Dubbed the internet of things (IoT), this network generates enormous amounts of personalized data which is well-suited for ML algorithms to gain insights and make decisions or predictions from complex datasets [1]. The growth in the number of devices that record information (i.e. data producers) and use information (i.e. data consumers) will consequently increase ML usage in IoT.

Given the diverse computational complexity and requirements of IoT systems, cloud computing has emerged as a widely adopted solution due to its cost efficiency, reliability, and scalability [2], [3], [4]. In this computing model, devices request hardware resources from a central datacenter to perform algorithms and data analysis. Since such datacenters can pool requests from large numbers of different users and devices, they can achieve high resource utilization and efficiency. Cloud computing plays a major role in today's technological infrastructure. However, it faces several challenges as the data and computing requirements of IoT continues to grow [5], [6], [7]:

1. **Network bandwidth and latency**

   Connections between devices have limited bandwidth and data transfer speeds set by the communication protocol. If data is generated at a faster rate than data transmission to the cloud compute server, additional latency is needed to upload input data (in addition to the data processing time). For many applications, this response time may be too long.

2. **Transmission power**

   As the number of data producers in the network grows, so does the total amount of raw data that needs to be transmitted to the cloud. Wireless communication is energy intensive and scales with the amount of data to be transferred. Depending on the application, it may not be practical or feasible to transfer all the data required to the cloud.

3. **Privacy and security**

   For certain applications using information such as healthcare or on-body sensor data, transferring this data off the device to the cloud incurs risk of compromising data privacy. Furthermore, devices used for critical applications are reliant on wireless communication which may not always be guaranteed.

These concerns can be mitigated by performing the computations locally on or near the data producers/consumers rather than offloading the task to the cloud. This model of computing, known as edge computing[1], requires algorithms to be adapted so that they can run on devices at the edge of the network. However, edge devices (e.g. mobile phones, medical devices, smart sensors, robotics) are often energy-constrained, relying on either battery power or energy harvesting [8]. As a result, the energy efficiency of both the data-processing algorithm and the hardware platform is a key limitation.

Unfortunately, commonly used ML algorithms such as deep neural networks (DNNs) and support vector machines (SVMs) struggle to meet the constraints required for edge computing. The main issue is the vast amount of memory needed by these algorithms. For example, AlexNet, a convolutional neural network used to classify images, uses 60 million weights totaling over 200 MB of storage [9]. Even if an edge device has enough memory to fit AlexNet's weights, the large memories would contribute significant leakage power. Furthermore in energy-efficient designs, energy from data movement can exceed that of the computation itself [10]. In addition, DNN training involves multiple passes of large datasets with expensive back-propagation calculations [11] and is thus infeasible to do on the edge. Although "Big Data" has empowered ML in cloud applications, the large amounts of memory and compute make conventional ML algorithms difficult to map onto edge devices.

Edge computing has created a market for energy-efficient ML. There are many ongoing efforts to adapt conventional ML algorithms by reducing the amount of memory required. Model compression uses techniques to determine and remove unimportant weights and reduce the float precision of the weights to reduce required memory after training [12]. Recent hardware implementations have taken advantage of such techniques for DNNs [13] and SVMs [14], although with reduced accuracy and performance compared to the baseline models. As

[1]Note there are two commonly used definitions of edge computing in literature. The first refers to computing directly on endpoint devices as mentioned in the prior sentence. The second refers to offloading compute onto a nearby local edge server that serves several endpoints. In this dissertation, we will use the former definition.

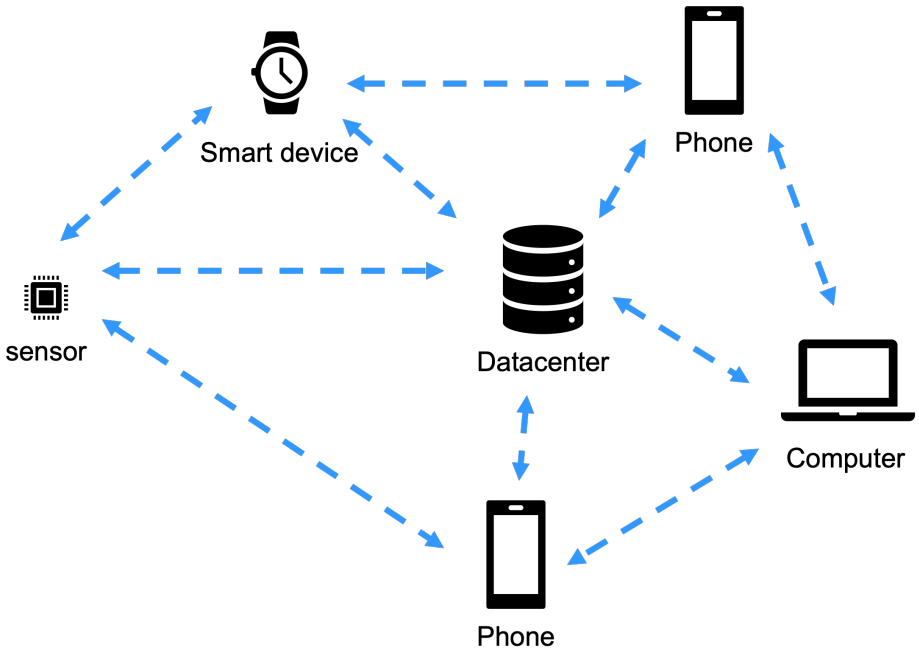

Figure 1.1: Depiction of an IoT network and with many edge devices.

such, efficient algorithms and energy-efficient hardware platforms for edge computing are still an active area of study.

1.2 Hyper-Dimensional Computing

Power-constrained circuit design is not a new concept. Since the beginning of the century, transistor speed and density has scaled to the point that performance of large processors are gated by power (and their ability to dissipate it) [15]. Digital designers soon found adding more parallelism, i.e. multiple slower cores vs. a single fast core, was more efficient given the same power budget [10]. The idea of slow but massively-parallel designs may be the key to energy-efficient computation on the edge.

Hyper-Dimensional Computing (HDC) is one such computing paradigm. Inspired by the vast neural circuits of the brain, HDC emulates neural activity patterns with vectors of very large dimension, i.e. hyper-dimensional (HD) vectors [16]. Unlike Von Neumann computing which is based on scalar arithmetic, HDC only computes using HD vectors. These vectors are combined using a simple but rich vector algebra which maps easily onto parallel and energy-efficient hardware implementations in a small memory footprint [17]. The HD representation of data also gives HDC unique robustness and fast one-shot learning capabilities useful for low power ML [18].

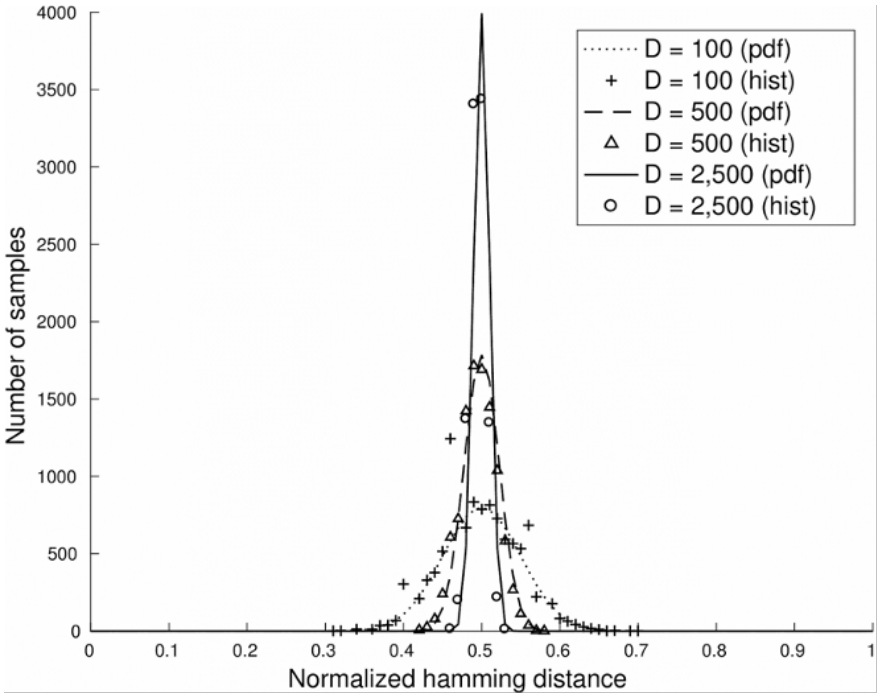

Figure 1.2: Normalized probability density function for the Hamming distance between two random binary vectors of varying dimension.

In this section, we will introduce HDC and its operating principles.

Pseudo-Orthogonality in HD Spaces

HDC algorithms are based on several important properties of randomness in HD vector spaces. In order for these properties to hold, the vector dimensions need to be large, usually in the order of 1000 or more. There are several variations of HDC collectively known as Vector-Symbolic Architectures (VSAs), which differ slightly in data type of vector elements as well as the specific vector operations used to combine and encode them [19][2]. In this dissertation, we will only consider the multiply-add-permute (MAP) variation that uses binary HD vectors whose elements are $\{0, 1\}$. For instance, a 1000 dimension binary HD vector is represented as a length 1000 bit string or word.

The key working principle of HDC is that randomly chosen HD vectors are very close to orthogonal, i.e. pseudo-orthogonal. Binary vectors are considered orthogonal if exactly

[2]As mentioned in [19], the various flavors of VSAs have advantages and disadvantages in their implementation and performance. However, since they share the same working principles, applications should be able to ported between most VSA types, although vector dimensions may need to be adjusted for performance.

half of their elements (bits) are the same and half are different. In other words, vectors $\boldsymbol{a}, \boldsymbol{b} \in \{0,1\}^d$ are defined as orthogonal if the Hamming distance $d_H(\boldsymbol{a}, \boldsymbol{b}) = d \,/\, 2$.

Now consider $\boldsymbol{a}, \boldsymbol{b} \in \{0,1\}^d$ both of whose elements are randomly chosen. For each element pair (a_i, b_i), there is a 50 % chance they match and can thus be modeled using a Bernoulli distribution $\overline{a_i \oplus b_i} \sim \text{Bern}(0.5)$. Since we can mathematically express

$$d_H(\boldsymbol{a}, \boldsymbol{b}) = \sum_{i=0}^{d} \overline{a_i \oplus b_i} \sim \sum_{i=0}^{d} \text{Bern}\left(\frac{1}{2}\right) = \text{B}\left(d, \frac{1}{2}\right) \tag{1.1}$$

as a binomial distribution. Since $\boldsymbol{a}, \boldsymbol{b}$ are HD vectors, d is large. We can use the normal approximation to the binomial distribution

$$d_H(\boldsymbol{a}, \boldsymbol{b}) \sim \mathcal{N}\left(\frac{d}{2}, \frac{d}{4}\right) . \tag{1.2}$$

Figure 1.2 plots the normalized probability density from Equation 1.2 for various values of d. In the case of $d = 10000$, the standard deviation $\sigma = 50$. This means that for 99.7 % of randomly chosen vectors $\boldsymbol{a}, \boldsymbol{b}$, their Hamming distance will fall in an extremely narrow range of $4850 < d_H(\boldsymbol{a}, \boldsymbol{b}) < 5150$ centered around the orthogonality condition of $d_H = 5000$. In this way, randomly chosen vectors are considered pseudo-orthogonal and thus can be used to represent distinct ideas, concepts, or values we wish to compute with.

HD Vector Operations

In order to leverage the pseudo-orthogonal property of HDC, its vector operations are necessarily designed around preserving or nullifying similarity. We define two vectors to be similar if their Hamming distance is not close to half the vector dimension

$$\boldsymbol{a} \sim \boldsymbol{b} \coloneqq d_H(\boldsymbol{a}, \boldsymbol{b}) \not\approx \frac{d}{2} . \tag{1.3}$$

Consequently, two vectors are dissimilar if their Hamming distance is close to half the vector dimension

$$\boldsymbol{a} \not\sim \boldsymbol{b} \coloneqq d_H(\boldsymbol{a}, \boldsymbol{b}) \approx \frac{d}{2} . \tag{1.4}$$

Only three vector transformations (multiplication, addition, permutation) and a single vector associative search operation are used to encode and decode HD vectors. Furthermore, the HD vector space is closed under all four operations.

1. **Addition**

   Also referred to as superposition or bundling, addition performs an element-wise majority function across all constituents. In practice, addition is computed into two steps: accumulation, denoted by $+$ or $\sum$, and thresholding, denoted by $[\cdot]$. Given n binary

vector inputs, the accumulation phase performs element-wise addition into a single integer vector. Thresholding then transforms the integer vector back into a binary one by mapping values greater than n / 2 to 1 and values less than n / 2 to 0. Addition preserves similarity between the output and all of its inputs, i.e. given $\boldsymbol{b} = [\sum_{i=1}^{n} \boldsymbol{a_i}]$, then $\boldsymbol{b} \sim \boldsymbol{a_i}$ for $1 \leq i \leq n$. Note that the accumulation step of addition is commutative and associative.

2. **Multiplication**

   Also known as binding, multiplication is an element-wise XOR computation of two binary HD vectors denoted with the symbol $\oplus$ or $\prod$. Given that $\boldsymbol{a} \not\sim \boldsymbol{b}$, then $\boldsymbol{a} \oplus \boldsymbol{b} \not\sim \boldsymbol{a}, \boldsymbol{b}$. Multiplication is also commutative and associative. Multiplication has some other important properties as well. First it is invertible, $\boldsymbol{a} \oplus \boldsymbol{b} \oplus \boldsymbol{b} = \boldsymbol{a} \oplus \boldsymbol{0} = \boldsymbol{a}$. Second, it distributes over addition, $\boldsymbol{b} \oplus [\sum_{i=1}^{n} \boldsymbol{a_i}] = [\sum_{i=1}^{n} \boldsymbol{b} \oplus \boldsymbol{a_i}]$. Finally, multiplication also preserves distance, $d_H(\boldsymbol{a}, \boldsymbol{b}) = d_H(\boldsymbol{a} \oplus \boldsymbol{c}, \boldsymbol{b} \oplus \boldsymbol{c})$. In this way, multiplication by a random vector can be thought of as a vector transformation function. It "randomizes" the direction of the input vectors, but does so in a way that preserves their relationships.

3. **Permutation**

   Permutation is a cyclic reordering of elements in a vector, denoted by ρ. In practice (especially in hardware), the simplest permutation to implement is a cyclic shift. In many ways, permutation is similar to multiplication by a random vector. It outputs a vector dissimilar to the original, $\boldsymbol{a} \not\sim \rho(\boldsymbol{a})$. It is also invertible $\rho^{-1}(\rho(\boldsymbol{a})) = \boldsymbol{a}$, distributes over addition $\rho([\sum_{i=1}^{n} \boldsymbol{a_i}]) = [\sum_{i=1}^{n} \rho(\boldsymbol{a_i})]$, and preserves distance $d_H(\boldsymbol{a}, \boldsymbol{b}) = d_H(\rho(\boldsymbol{a}), \rho(\boldsymbol{b}))$. Its main use is for distinguishing a vector's position in a sequence.

4. **Associative search**

   The associative search operator takes a single query vector $\boldsymbol{q}$ and a set of search vectors $\boldsymbol{a_1}, \ldots, \boldsymbol{a_n}$. The output of the search is the vector in the search set that is most similar to the query vector. In mathematical terms, the search result vector is given by $\boldsymbol{s} = \boldsymbol{a_k}$ where $k = \text{argmin}_{i=1}^{n} d_H(\boldsymbol{q}, \boldsymbol{a_i})$. The associative search is useful in cleaning up noisy versions of vectors. Often times, the vectors in the search set are collectively referred to as the associative memory.

1.3 Encoding and Computing with HD Vectors

Using the simple vector algebra defined in the previous section, HDC can compute and encode information in hierarchical structures. In this section, we will explore some commonly used encoding structures, discuss the information capacity of HD vectors, and walk through a full example of performing text language classification.

Since HDC requires random vectors to take advantage of its pseudo-orthogonality property, we start by creating random vectors to represent each input for our computation. Once

a random vector is created for an input, however, this assignment is fixed. These vectors are often called items or item vectors and consequently the entire collection of items is named the item memory. From a traditional computing view, the row address of an item can be used as a label to associate with its corresponding input. We will now demonstrate how items can be encoded into useful data structures.

1. **Key-value pairs**

   The first simple encoding scheme is the combined representation of a key and value, much like an entry in a dictionary. Assuming $\boldsymbol{k}, \boldsymbol{v}$ are items that represent a key and value respectively, we can encode a simple multiplication of the vectors $\boldsymbol{p} = \boldsymbol{k} \oplus \boldsymbol{v}$. Unlike a traditional key-value pair, the HD pair vector $\boldsymbol{p}$ can be queried by either vector to retrieve its corresponding pair. For example $\boldsymbol{p} \oplus \boldsymbol{k} = \boldsymbol{k} \oplus \boldsymbol{v} \oplus \boldsymbol{k} = \boldsymbol{v}$ and $\boldsymbol{p} \oplus \boldsymbol{v} = \boldsymbol{k} \oplus \boldsymbol{v} \oplus \boldsymbol{v} = \boldsymbol{k}$.

2. **Sequences**

   Multiplication and permute operations are used to encode sequences and patterns. The sequence of items $\{\boldsymbol{a_1}, \ldots, \boldsymbol{a_n}\}$ can be encoded as $\boldsymbol{s} = \prod_{i=1}^{n} \rho^{i-1}(\boldsymbol{a_i})$. The permutation is used to distinguish between the order of the sequence items and ensure a unique sequence vector for different orderings.

3. **Sets**

   HDC encodes sets using the addition operator[3]. For instance, we can encode a vector $\boldsymbol{s}$ that represents a set of items $\{\boldsymbol{a_1}, \ldots, \boldsymbol{a_n}\}$ by adding $\boldsymbol{s} = [\sum_{i=0}^{n} \boldsymbol{a_i}]$. HDC provides an elegant and non-brute force method to test for set membership. Assuming a query vector $\boldsymbol{q}$, we can compute $x = d_H(\boldsymbol{q}, \boldsymbol{s})$. Since $\boldsymbol{s}$ is similar to all of its constituents, if $x \not\approx d \,/\, 2$ then $\boldsymbol{q}$ is in the set. If $x \approx d \,/\, 2$, then $\boldsymbol{q}$ is not in the set.

4. **Dictionaries**

   By combining the ideas of sets and key-value pairs, we can construct dictionaries as well. A dictionary is just a set of key-value pairs constructed as $\boldsymbol{s} = [\sum_{i=0}^{n} \boldsymbol{k_i} \oplus \boldsymbol{v_i}]$. This dictionary can be queried by a key or value to get its corresponding pair. For example, assume we have the item representing the key $\boldsymbol{k_j}$ where $1 \leq j \leq n$. We can multiply the key to the dictionary vector to get

$$\boldsymbol{k_j} \oplus \boldsymbol{s} = \left[\boldsymbol{k_j} \oplus \boldsymbol{v_j} \oplus \boldsymbol{k_j} + \sum_{i=0, i \neq j}^{n} \boldsymbol{k_i} \oplus \boldsymbol{v_i} \oplus \boldsymbol{v_j} \right] \tag{1.5}$$

$$= \left[\boldsymbol{v_j} + \sum_{i=0, i \neq j}^{n} \boldsymbol{k_i} \oplus \boldsymbol{v_i} \oplus \boldsymbol{v_j} \right] . \tag{1.6}$$

[3]Note that traditional sets only contain one of each item. In our case, it is possible to add multiple of the same item, and is thus more akin to a multiset. However for simplicity, in this dissertation we will use the terms set and multiset interchangeably.

Noting that the terms $\boldsymbol{k_i} \oplus \boldsymbol{v_i} \oplus \boldsymbol{v_j}$ where $i \neq j$ are all dissimilar from $\{\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}\}$, the product vector can be seen as $\boldsymbol{v_j}$ with noise added from the other terms. We can thus perform an associative search query over the set of items $\{\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}\}$ using $\boldsymbol{k_j} \oplus \boldsymbol{s}$ as a query vector to retrieve the value vector $\boldsymbol{v_j}$.

Information Capacity

For all the data structures mentioned above, there are limits to how much data can be stored and retrieved in a single vector. For example, if too many vectors are added into a single set, the constituent vectors and set vector will start to get more and more dissimilar. For dictionaries, adding too many key-value pairs could make the product vector so noisy that associative search cannot choose the correct vector. Clearly, the number of entries in a set/dictionary (i.e. the information capacity) increases with the size of the HD vector. Both analytical [20], [21] and quantitative [22] studies observe a linear increase in the dimension of the HD vector with the number of entries to be added. As a point of reference, 1000 dimension vectors were able to achieve 99 % accuracy in both set membership and dictionary unbinding tests with the summation of 20 items/key-value pairs [22]. Depending on the complexity and size of the encoding we want to perform, the dimension of the HD vectors should be scaled accordingly for desired performance.

Case Study: Text Language Recognition

Now that we have introduced the basics of HDC, we will walk through an entire example of an HDC algorithm for language classification of texts. Language recognition was the first example of HDC applied to a supervised classification algorithm, the family of algorithms that HDC is perhaps most known for. The HDC language recognition algorithm uses Latin alphabet text to learn and classify 21 European languages [23], [24].

Using ideas from natural language processing, this HDC algorithm uses N-grams, sequences of n consecutive letters, to detect and measure language similarity. The goal is to measure the frequency of N-grams for different languages and calculate a histogram that represents each language. When given new input text, we can again compute the N-gram histogram and compare it to the histograms for each language. The closest match will be the output class.

The N-gram histogram computations and comparisons would be a memory and compute intensive task in standard computing. Given 26 letters of the alphabet and the whitespace character, there are 27 total possibilities for each element in an N-gram sequence. As a result, N-gram histograms must keep track of 27^n different entries. HDC can solve this problem efficiently. We begin by creating an item memory with 27 entries, assigning one item to each letter and whitespace. To differentiate N-grams, their representations must be unique, or pseudo-orthogonal, which can be achieved by encoding them as sequences. The N-gram frequencies are tracked by adding N-gram vectors together into a set vector. N-grams that appear multiple times will also be added multiple times into the weighted set vector, where

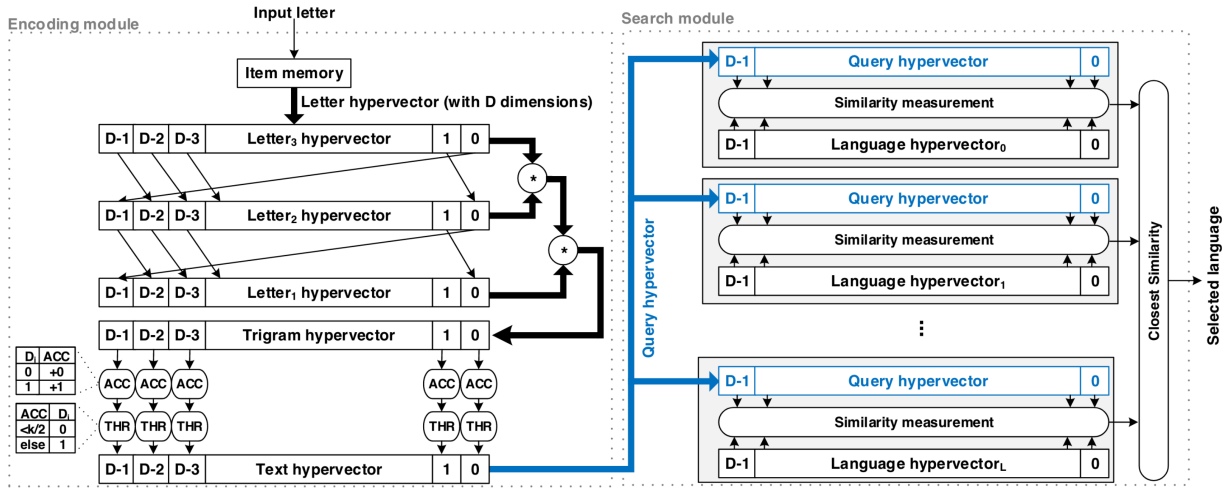

Figure 1.3: Figure 1 of [24]: Block diagram of the HDC testing algorithm used for text language recognition.

the weight reflects the frequency of each N-gram. This ensures high similarity between the set vector and frequently occurring N-gram vectors, and low similarity for rare N-grams.

In the training phase, input text for each language is processed and encoded into N-gram vectors which are then summed into a single vector representing that language. The language vectors are stored into the associative memory. In the testing phase, depicted in Figure 1.3, the same exact encoding and summation is performed for input text. The only difference is that rather than storing the resulting vector, we perform an associative search to find the most similar class vector.

The HDC algorithm was performed using 10000 dimension vectors and an N-gram size of $n = 4$ with a classification accuracy of 97.1 % [24]. Although the baseline method of nearest-neighbor classifier using N-gram histogram slightly outperforms the HDC method with an accuracy of 99.2 %, it requires 20× more memory.

1.4 Applications of HDC

The number of proposed and theorized HDC applications are quickly growing across a wide range of application spaces. A recent HDC survey counts over 300 applications [30]. However, only a limited number of these studies include concrete examples with well-defined problems/datasets and characterized performance. As of now, most practical applications of HDC are supervised classification tasks. Table 1.1 outlines several of such tasks that have also been compared to baseline implementations state-of-the-art conventional ML al-

Application	Dim.	Acc.	Baseline Acc.	Platform	Energy Saving/ Speedup
Language recognition [24]	10000	97.1 %	KNN 99.2 %	65 nm post-synthesis	20.0×
EMG gesture recognition [25]	10000	97.8 %	SVM 89.7 %	-	-
Emotion recognition [26]	2000	79.3 %	SVM 89.7 %	28 nm post-layout	9.5×
Voice recognition [27]	10000	88.4 %	DNN 95.9 %	CPU	11.9×
Seizure recognition [28]	10000	85.5 %	SVM 83.3 %	GPU	1.9×
DNA sequencing [29]	10000	99.7 %	SVM 94.5 %	CPU	4.3×

Table 1.1: Performance of HDC classification for edge-compute suitable applications. Accuracies and energy improvement, where available, are also reported for the best performing conventional ML alternative, as well as the platform used for energy comparison.

gorithms. In all cases, the HDC algorithms outperforms or is competitive in accuracy with the baseline, while having vastly superior energy-efficiency and computation latency.

HDC classification algorithms follow the same general structure, outlined in the language recognition case study in the previous section. Random item vectors are assigned to all possible input data values and channels. For both training and testing examples, the items representing the input are encoded into a single HD vector. The exact encoding algorithm differs between different implementations. Even for the same application, finding a smart encoding strategy can improve performance [27]. As a result, classification with HDC often requires domain knowledge to implement the encoder, e.g. N-grams for language recognition.

Outside of classification, HDC has also shown promise in applications such as robotics and control [31], [32], [33], efficient storage and search in graph and tree data structures [34], [35], and visual perception tasks when used in combination with neural networks [36], [34], [37]. Due to their recent development, there are currently limited examples and uses of these applications. However, we expect increased adaption and evaluation of these applications in the near future.

1.5 Custom Hardware for Energy-Efficient HDC

In the previous sections, we have shown how HDC algorithms compute using simple highly-parallel vector operations with a compact memory. In combination with its competitive accuracy and vastly superior efficiency, HDC is fast becoming a promising candidate for edge-compute applications (refer to Table 1.1). This section highlights several additional attractive qualities of HDC in low-power environments and argues how custom hardware design can maximize energy-efficiency.

Robustness

All HDC algorithms require real-world inputs to first be mapped onto random item vectors. Unlike scalars used for conventional computing which have most significant and least significant bits, the bits that make up the item vectors all carry the same amount of importance and information. In other words, data representation is holographic in the HD space. Additionally, uncertainty is fundamentally built-in to the structure of HDC; distances between random HD vectors are uncertain (although probabilistically narrow) and the associate search returns the most similar result. As such, random errors in HD vector computations have a low chance in corrupting the final output. For example, simulations show that the HDC language recognition algorithm can tolerate memory bit error rates 8.8× higher than the baseline SVM algorithm [24]. Several hardware implementations have shown that this robustness can be used to take advantage of new error-prone devices such as carbon nanotube FETs and RRAMs [38] and PCMs [39]. However, even standard digital CMOS designs can utilize HDC's robustness by operating at low-VDD in the near-threshold regime for higher energy-efficiency.

Efficient Training

Unlike conventional ML algorithms, training for supervised HDC tasks is fast and simple. For example, classification performs one-shot learning, accumulating encoded training examples in a single pass through the dataset. Furthermore, encoding for training and testing is exactly the same, and no additional complex calculations (e.g. backpropagation, gradient descent) are required. These qualities make online learning feasible, even on edge devices with limited data and compute resources [25], [40].

HDC in Hardware

Unfortunately, traditional compute platforms such as CPUs and GPUs cannot efficiently implement HDC algorithms. These platforms are most effective for large intensive computations such as floating point arithmetic and struggle with custom data widths and non-standard operations [41]. As a result, HDC applications aiming for energy-efficiency are often implemented on FPGAs or ASICs. FPGA designs for HDC [42], [43], [44] are common due to their quick implementation and turnaround time. However, especially if energy-efficiency is a primary concern, ASICs can often provide one order of magnitude higher efficiency as a rule of thumb [45]. In practice, ASIC implementations of HDC have shown even higher efficiency gains. [46] shows $\sim 10\,000\times$ better energy-efficiency when comparing post-synthesis simulations of HDC classification on a 28nm CMOS ASIC vs on an Nvidia Jetson TX2 CPU. [47] shows $\sim 100\times$ better energy-efficiency when comparing post-synthesis simulations of HDC classification on a 22nm CMOS ASIC vs on a FPGA. Intelligent ASIC design can enable HDC for energy-constrained computing on the edge.

1.6 Outline

This introduction has overviewed the increase in demand of energy-efficient ML and how custom hardware implementations of HDC offer a solution. Given that real-world applications of HDC have been limited (although rapidly increasing in recent years), hardware accelerators for HDC have correspondingly been designed for a single algorithm or class of algorithms. This dissertation argues for the design of an efficient general purpose processor for HDC in four steps. In the first step (Chapter 2), we investigate two promising non-classification HDC algorithms. We apply these algorithms to concrete examples and characterize their performance in order to show that HDC's edge computing capabilities are continuing to grow. The second step (Chapter 3), we discuss the limitations of previous ASICs for HDC and propose the architecture of the Hyper-Dimensional Processing Unit (HPU), a general-purpose processor for HDC. In the third step (Chapters 4-5), we implement, optimize, fabricate, and characterize the HPU design. This includes detailed performance and energy-efficiency measurements of the HPU in the context of edge computing and comparisons to existing hardware. We will conclude the dissertation with a summary of results and perspectives on the future of hardware for HDC.

Chapter 2

Investigating Non-Classification HDC Algorithms

Until recently, applications and hardware for HDC only made use of the HD classification algorithm. The classification algorithm was first introduced and applied to a European text language classification task [24], but has since found broad success in many areas as shown in Table 1.1. These applications follow the same basic HDC algorithm with differences only in the input encoding process. As a result, hardware platforms to accelerate HD classification allowed for specialized datapaths and deep pipelines optimized only for the classification algorithm. However in the last few years, new HDC algorithms have begun to emerge that have vastly different computation structures or more complex vector operations. In this chapter, we share two such emerging algorithms in which we have contributed to the algorithm development or hardware implementation.

2.1 HD Factorization

The development of the HD factorization algorithm is rooted in the disentanglement problem associated with sensory perception and cognitive reasoning. First proposed and analyzed in [34] [48], HD factorization maps the disentanglement problem into the HD space. Each quality to be unbinded (also known as a factor) is represented by a set of vectors (called items) that correspond to each possible quality value. For example, assume we are given an image with a colored digit somewhere in the picture. We wish to disentangle the image into its three constituent factors: the digit represented $\{0, \ldots, 9\}$, the color of the digit $\{\text{black}, \text{red}, \ldots, \text{green}\}$, and the position of the digit in the image $\{\text{top-left}, \ldots, \text{bottom-right}\}$.

For ease of analysis, let us assume that we have three factors each with exactly n items. We assign unique random vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ for each item of the first factor, $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n$ for the second factor, and $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n$ for the third. The input image can be represented as a unique combination of the items $\boldsymbol{q} = \boldsymbol{x}_i \oplus \boldsymbol{y}_j \oplus \boldsymbol{z}_k$ for $i, j, k \in \{1, \ldots, n\}$. The factorization problem can then be posed as follows: given a query vector $\boldsymbol{q}$ that is the multiplication of

item vectors from each factor, find the individual items (i.e. multiplicands) from each factor.

The factorization problem can be solved using an iterative HDC algorithm. In each iteration, we first assume we have guesses for each factor, $\hat{\boldsymbol{x}}[t], \hat{\boldsymbol{y}}[t], \hat{\boldsymbol{z}}[t]$ where t is the iteration number. We can then compute

$$\hat{\boldsymbol{x}}[t+1] = \left[XX^\top \boldsymbol{q} \oplus \hat{\boldsymbol{y}}[t] \oplus \hat{\boldsymbol{z}}[t]\right] \tag{2.1}$$

$$\hat{\boldsymbol{y}}[t+1] = \left[YY^\top \boldsymbol{q} \oplus \hat{\boldsymbol{x}}[t+1] \oplus \hat{\boldsymbol{z}}[t]\right] \tag{2.2}$$

$$\hat{\boldsymbol{z}}[t+1] = \left[ZZ^\top \boldsymbol{q} \oplus \hat{\boldsymbol{x}}[t+1] \oplus \hat{\boldsymbol{y}}[t+1]\right] \tag{2.3}$$

where

$$X = \begin{bmatrix} | & & | \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \\ | & & | \end{bmatrix}, \quad Y = \begin{bmatrix} | & & | \\ \boldsymbol{y}_1 & \cdots & \boldsymbol{y}_n \\ | & & | \end{bmatrix}, \quad Z = \begin{bmatrix} | & & | \\ \boldsymbol{z}_1 & \cdots & \boldsymbol{z}_n \\ | & & | \end{bmatrix}. \tag{2.4}$$

Let us break down the update process for $\hat{\boldsymbol{x}}$ shown in Equation 2.1. The innermost product $\boldsymbol{p} = \boldsymbol{q} \oplus \hat{\boldsymbol{y}}[t] \oplus \hat{\boldsymbol{z}}[t]$ multiplies the query vector with the current guesses for $\hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{z}}$. Since the all vectors in the binary HDC space is its own multiplicative inverse, $\boldsymbol{p}$ represents the guess for $\boldsymbol{x}$. The outermost operation performs the matrix multiplication $\left[XX^\top \boldsymbol{p}\right]$. We can evaluate

$$\left[XX^\top \boldsymbol{p}\right] = \left[\begin{bmatrix} | & & | \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \\ | & & | \end{bmatrix} \begin{bmatrix} — & \boldsymbol{x}_1 & — \\ & \vdots & \\ — & \boldsymbol{x}_n & — \end{bmatrix} \boldsymbol{p}\right] \tag{2.5}$$

$$= \left[\begin{bmatrix} | & & | \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \\ | & & | \end{bmatrix} \begin{bmatrix} \langle \boldsymbol{x}_1, \boldsymbol{p} \rangle \\ \vdots \\ \langle \boldsymbol{x}_n, \boldsymbol{p} \rangle \end{bmatrix}\right] \tag{2.6}$$

$$= \left[\sum_{i=1}^{n} \langle \boldsymbol{x}_i, \boldsymbol{p} \rangle \boldsymbol{x}_i\right], \tag{2.7}$$

where $\langle \cdot, \cdot \rangle$ represents the inner product assuming the binary vectors are represented in bipolar form. In other words, the matrix multiplication can be viewed as computing the vector similarities between $\boldsymbol{p}$ and all the items $\boldsymbol{x}_i$ and then performing a scaled accumulation where each item $\boldsymbol{x}_i$ is scaled by its corresponding similarity value. The end result is then thresholded back to a binary/bipolar vector and stored as the guess for the next iteration. This process is then repeated for each of the other factors to finish one iteration of the factorization algorithm.

The iterative algorithm completes when the vector guesses converge, i.e. when

$$\hat{\boldsymbol{x}}[t+1] = \hat{\boldsymbol{x}}[t], \hat{\boldsymbol{y}}[t+1] = \hat{\boldsymbol{y}}[t], \hat{\boldsymbol{z}}[t+1] = \hat{\boldsymbol{z}}[t]\,. \tag{2.8}$$

In order to start the algorithm, we must first initialize the starting guesses $\hat{\boldsymbol{x}}[0], \hat{\boldsymbol{y}}[0], \hat{\boldsymbol{z}}[0]$, which are set to the unscaled accumulation of the items for that factor. In other words

$$\hat{\boldsymbol{x}}[0] = \left[\sum_{i=0}^{n} \boldsymbol{x}\right]_{\boldsymbol{i}}, \ \hat{\boldsymbol{y}}[0] = \left[\sum_{i=0}^{n} \boldsymbol{y}\right]_{\boldsymbol{i}}, \ \hat{\boldsymbol{z}}[0] = \left[\sum_{i=0}^{n} \boldsymbol{z}\right]_{\boldsymbol{i}} . \tag{2.9}$$

The average number of iterations to converge depends on several factors. In general, the larger the search space of the factorization algorithm (defined as the total number of possible combinations of items), the algorithm requires either more iterations or larger vector dimensions to converge. If the vector dimension is too small compared to the search space, the guess iterations can get stuck in a cyclic loop and never converge.

In an effort to both make the HDC algorithm more hardware friendly and to improve the algorithm convergence speed, we investigated three methods of constraining the similarity computation and scaled accumulation portions of the factorization algorithm:

1. **Thresholding:**
   After computing the similarities in Equation 2.6, thresholding will replace all similarities less than the threshold value with zero. Since items that have very small similarities to the guess vector $\boldsymbol{p}$ are nearly orthogonal to $\boldsymbol{p}$, there is very little probability that those items are included in the query. As a result, by removing such items from the scaled accumulation, we can converge more quickly to the correct item.

2. **Quantization:**
   Quantization is similar to thresholding but with the main focus of reducing the number of bits required to hold the similarity values by reducing the dynamic range. Like thresholding, quantization will set small similarity values to zero, but it will also reduce the resolution of larger similarity values as well.

3. **Saturation:**
   Saturation affects the scaled accumulation by setting maximum and minimum values during the accumulation phase. Assuming one item is accumulated at a time, if an accumulated vector element is greater or less the saturation values, it will be clipped to the saturation values instead.

The results of the three methods are shown on an example factorization problem using 2048 dimension vectors with 3 factors and 128 items per factor in Figure 2.1. We report the accuracy and avg number of iterations to converge measured over 1000 random queries. Note that the saturation results assume the similarities have already been quantized by a threshold of 64 before the scaled accumulation phase.

Thresholding and quantization are mutually exclusive strategies that are applied to the similarity computation portion of the factorization algorithm. In terms of convergence speed and accuracy, thresholding shows superior performance, with up to a 3.5× improvement in number of iterations over the baseline and a maximum accuracy of 100 %. In general, increasing the threshold value improves factorization performance, but if the threshold is too

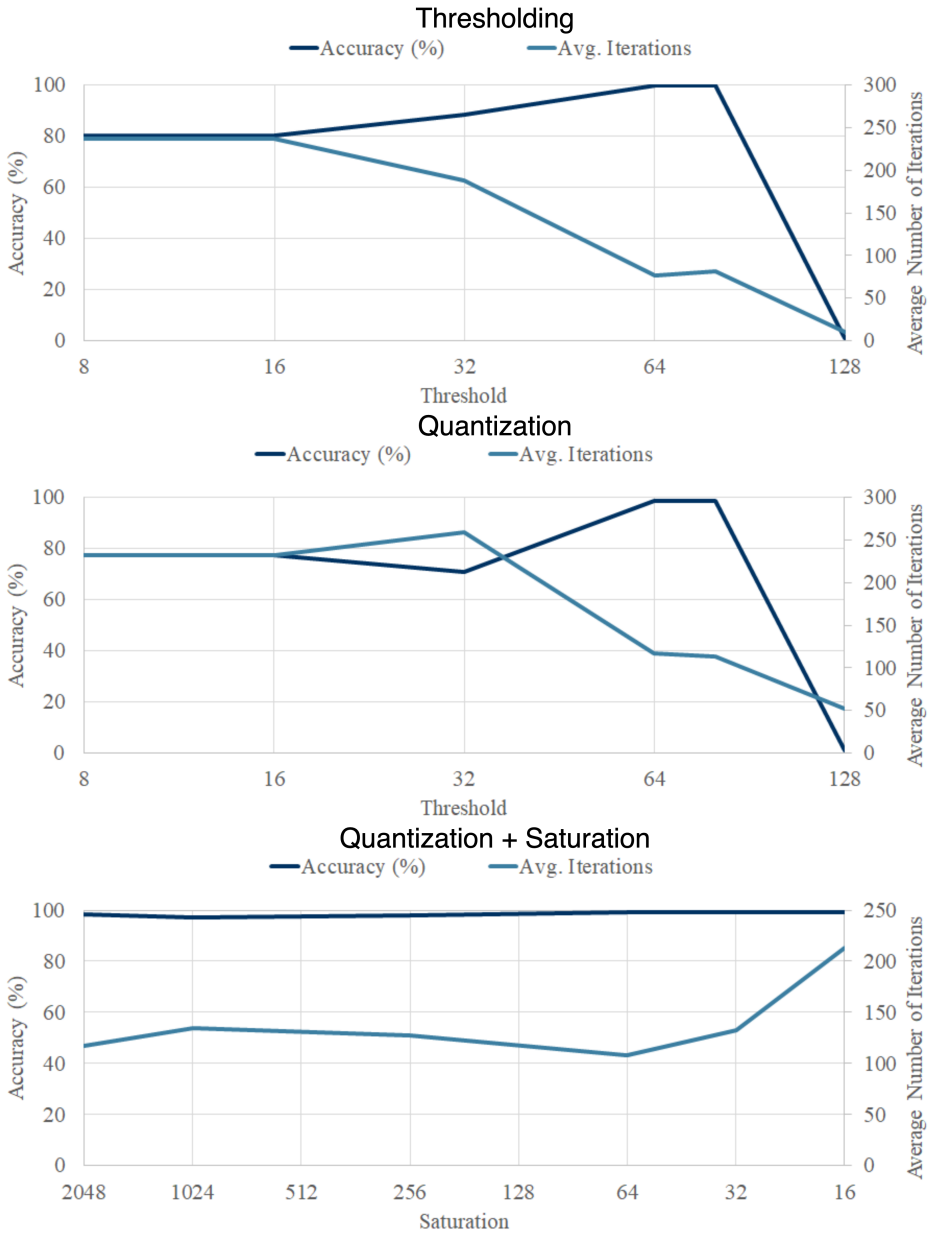

Figure 2.1: Effects of thresholding, quantization, and saturation on the accuracy and convergence of the HD factorization algorithm. The results are compiled from 1000 tests with 3 factors and 128 items per factor using 2048 dimension vectors. For the saturation results, saturation is applied after quantization with a threshold of 64.

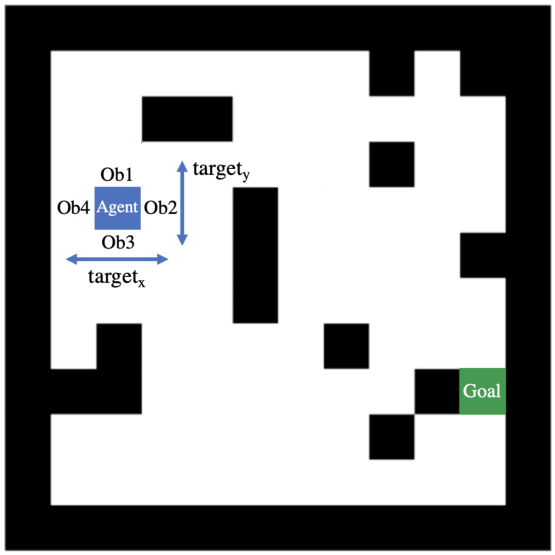

Figure 2.2: Simulated 2D environment for obstacle navigation using RORB. The agent uses sensor data inputs to avoid obstacles and reach the goal square.

large (> 64 in this case), the algorithm will suddenly collapse and fail. As a result, finding the optimal threshold value for factorization problems of different sizes may be difficult without extensive characterization. The quantization method yields similar trends to thresholding, but with less effective results. We again see an improvement in convergence speed and accuracy, although only a $2\times$ convergence speedup. However, quantization enables for compressed integer representations of the similarity values and thus reduces the necessary integer bitwidth. As a result, hardware acceleration of the HD factorization algorithm is more amenable to quantization as we gain a large portion of the performance gains while also reducing hardware cost and complexity.

After quantization, we find that saturation has little effect on the factorization performance for reasonable saturation values. Alongside quantization, this allows us to fix the integer bitwidth required in hardware for HDC while improving algorithm performance.

2.2 Recall of Reactive Behavior

One of HDC's biggest advantages over traditional ML algorithms is its ability to use domain knowledge to quickly learn over limited training data and time, which is especially useful in robotics and control applications [49]. The HDC Recall of Reactive Behavior (RORB) algorithm was initially proposed as a method to bring these advantages into a robotics application [31] [32]. Unfortunately in these works, the application details as well as algorithm performance are not presented nor measured.

In this section, we present our work on adapting the RORB algorithm to deal with heterogeneous sensory inputs as well as characterizing the algorithm performance on a well-defined application [50]. The RORB algorithm is used to navigate within a simulated 2D environment shown in Figure 2.2. The algorithm's goal is to navigate the agent (located at the blue square) to the goal (located at the green square) while avoiding all obstacles (black squares). The worlds are 10×10 grids with 15 randomly-placed obstacles. The agent has access to six sensory inputs. `Ob1 - Ob3` are boolean signals that return true or false depending on whether an obstacle is present in the tile up, right, down, and left of the agent respectively. $\texttt{target}_x$ and $\texttt{target}_y$ are ternary signals which detect whether the x and y coordinates of the agent are less than, equal to, or greater than that of the goal. In other words, these signals allow the agent to know the direction of the goal location relative to itself. Finally, the agent's last movement is also provided as an input, which is useful in training the agent not to enter a movement loop (ex. moving back and forth between two squares). The agent can move in four directions (up, down, left right), and thus after every movement, the algorithm receives the 7 new inputs from the current location and provides a directional output for the next movement.

As RORB is a supervised learning algorithm, we first manually move the agent to the goal in several random environments to create a set of sensor inputs and movement outputs to train on. The basic training algorithm is similar to that of HD classification. We first begin by assigning a random ID vector to represent each of the 7 inputs and 4 outputs, as well as vectors to represent the possible sensor values. During the RORB training phase, the sensor inputs values are multiplied to their respective ID vectors and then encoded into a single sensor vector that represents all the sensor inputs. The sensor vector is then multiplied to the output ID vector (also called the actuator vector) and accumulated into the program vector. The program vector accumulates all training instances that are sufficiently different and then is stored for use in testing.

In the testing phase, we repeat the same encoding process for the sensory inputs to produce the sensor vector. The sensor vector is then multiplied with the stored program vector to produce a noisy actuator vector. By performing an associative search over the output ID vectors, the noisy actuator vector will select the output direction towards which the agent should move. For each test, we simulate a random environment which has not been seen in the training phase. RORB uses the stored program vector to predict each consecutive movement of the agent until one of three outcomes occur. The first is when the agent reaches the goal without any collisions, which is counted as a success. The second is when the agent collides with an obstacle or wall, which is counted as a fail. Last is if the agent becomes stuck in a movement loop, which also counts as a fail.

The implementation of the sensor encoding plays a large role in the behavior of the agent. In this work, we tested three unique sensor encoding strategies which results in vastly different algorithm performance, shown in Figure 2.3.

- Modality-based encoding: Based on previous sensor-fusion classification algorithms, the sensors are grouped and multiplied together by modality. The resulting modality

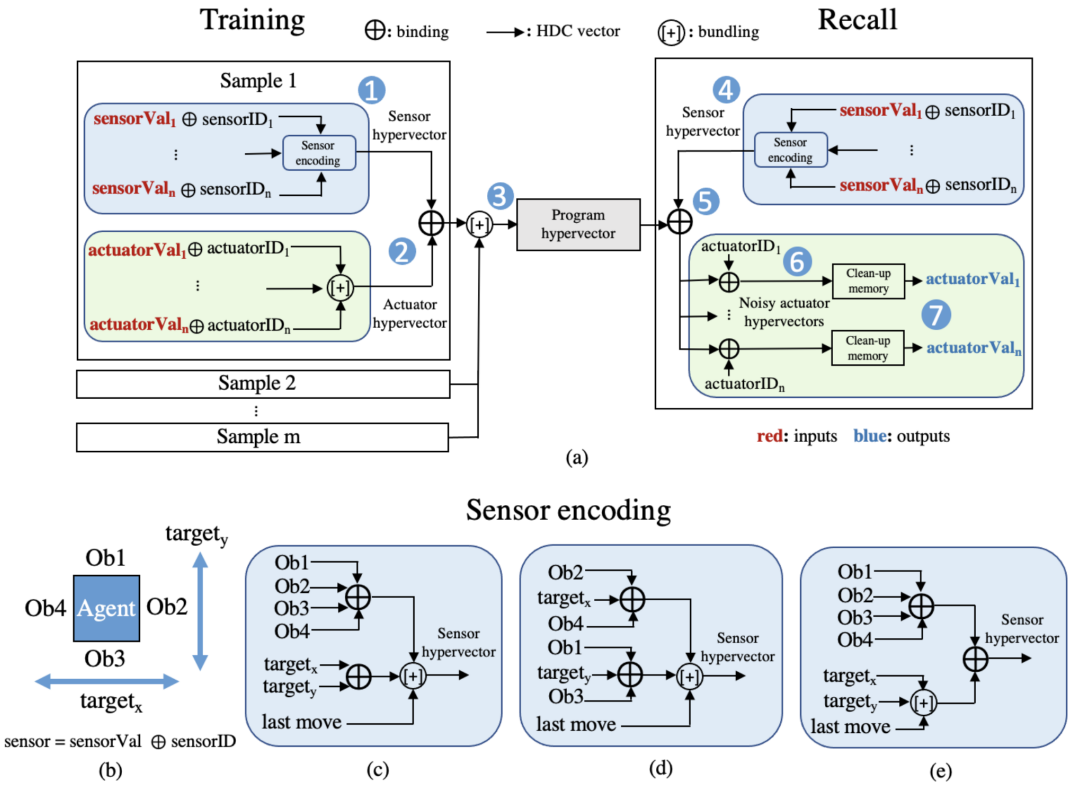

Figure 2.3: Diagram of the training and testing algorithms for RORB (a). Sensor vectors constructed from input sensor data (b). Visualization of the modality-based (c), directional (d), and constraints vs. goals (e) encoding strategies.

vectors are accumulated into the sensor vector.

- Directional encoding: The input sensors vectors are grouped by the x and y directions and multiplied together. The directional vectors and the last movement vector are accumulated into the sensor vector.

- Constraints vs. goals encoding: This encoding strategy attempts to convey domain knowledge by use of primary and secondary goals. Since collision with an obstacle is an automatic failure, the primary goal (i.e. constraints) of the agent should be to avoid all obstacles. As a result, the obstacle sensor vectors are directly multiplied into the sensor vector. On the other hand, the directional target vectors and the last movement vectors are used to move the agent towards the goal and thus should be considered secondary goals. Therefore, the goal vectors are accumulated together first before being multiplied into the sensor vector.

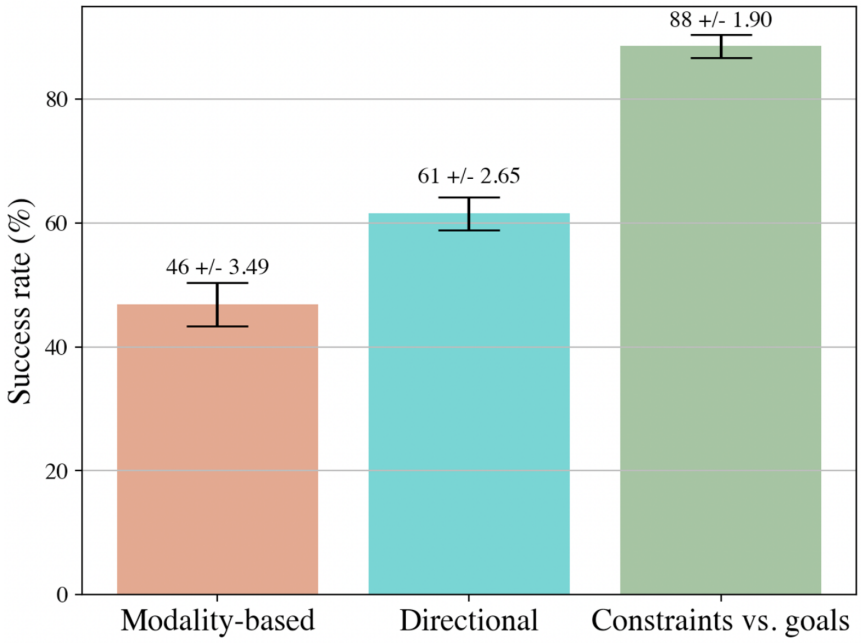

Figure 2.4: RORB success rate and standard deviation for the different sensor encoding strategies.

The success rates of each sensor encoding strategy is shown in Figure 2.4. The rates are calculated over 100 environments with 100 trials in each environment consisting of differing start and goal locations for a total of 10 000 tests. The same set of tests are used to evaluate the three strategies. Each test includes a 100-step timeout to detect movement loops.

Of the three encoding strategies, constraint vs. goals achieves by far the highest success rate at 88 % nearly doubling the worst-performing modality-based strategy of 46 %. This demonstrates that HDC can utilize intelligent encoding strategies to set behavioral priorities. In this way, domain knowledge and previous experience can be included in HDC algorithms to ensure good performance even with constraints in training.

2.3 Conclusion

As showcased in this chapter,the complexity and applicability of HDC algorithms is constantly growing. Initially HDC was only used for classification algorithms which perform perception tasks using sensory data. Now, we see that HDC can be used for other types of cognitive functions such as reasoning (eg. HD factorization) and control (eg. RORB). Dynamic and intelligent systems require the use and interconnection of all three layers of cognitive function, using each layer for tasks at different levels of cognitive abstraction [51].

Moving forwards, intelligent HDC systems and the hardware required to implement them efficiently need to be prepared to handle a wide variety of HDC algorithms.

Chapter 3

The Design of the Hyper-Dimensional Processing Unit (HPU)

3.1 Introduction

The previous chapter introduced two emerging HDC algorithms, HD factorization and recall of reactive behavior, which demonstrate the growing applicability and complexity of HDC. Traditionally, custom processors and accelerators for HDC have been designed for a specific application or class of similar algorithms (i.e. HD classification). However, with the explosion of new HDC algorithms, the need for a programmable and flexible HDC processor is becoming apparent. In this chapter, we introduce the system architecture of the Hyper-Dimensional Processing Unit (HPU), a processor capable of accelerating all-known binary HDC algorithms. We begin with a survey of existing ASICs for HDC and highlight their limitations in algorithm and hardware adaptability. These limitations help define the design goals and resulting architecture of the HPU. This chapter presents the core features of the HPU and provides a block-level description of the system design.

3.2 Previous ASICs for HDC

As mentioned in Section 1.5, the operations used for binary HDC algorithms are simple bitwise operations on vectors with thousands of dimensions and therefore struggle to map efficiently onto conventional compute platforms. Custom hardware, particularly in the form of ASICs, can provide the performance and energy-efficiency required for HDC's low-power edge applications. As a result, several custom processors have been developed for HDC in the last decade.

The first manufactured example of hardware for HDC utilized new emerging devices, namely Resistive Random Access Memory (RRAM), to take advantage of HDC's robustness to errors and randomness [38]. A similar RRAM design integrated with Carbon Nanotube Field Effect Transistors (CNFETs) became the first processor to report on-chip power and

performance [52]. These ASICs were designed specifically to perform language classification of the EUROPARL dataset and cannot be adapted to different applications or datasets. Kaunaratne et al. fabricated the first programmable HDC processor capable of performing multiple different applications [39]. This processor also utilized emerging devices in the form of Phase-Change Memory (PCM) and performed the vector operations in-memory. Due to the in-memory compute constraints of the PCM crossbar, the processor struggles to efficiently compute binary vector operations. For example, the authors chose to perform 2-min-term approximations instead of the N-gram encoding required for many HD classification algorithms. Furthermore, only the PCM modules were fabricated, and the additional digital CMOS required to program and control the memories were simulated instead. Datta et al. fabricated the first completely digital CMOS processor for HDC [53]. The processor is also programmable, but unlike Karunaratne's PCM processor, performs all vector operations in full without approximations.

Although both [39] and [53] can accelerate multiple different HDC algorithms, they are limited to only classification algorithms. The processors contain a uni-directional datapath from the item memory to the associative memory, and thus cannot perform more complex algorithms such as HD factorization which require encoded vectors to be fed back into the datapath. With the recent interest in the capabilities of HD factorization, [54] was fabricated using PCM crossbars to accelerate the iterative matrix multiplication portion of the HD factorization algorithm. Similar to [39], only the PCM modules were fabricated, and the digital logic required for encoding was simulated instead. Nevertheless, this processor marked the first manufactured ASIC to accelerate a non-classification HD algorithm.

Additionally, there are several ASIC designs that been proposed and simulated but not realized on physical hardware. Notably, [55] also proposes a fully digital CMOS processor that can be programmed to perform various classification algorithms. In accordance with many of HDC's applications, the processor attempts to push the low-power boundaries through multiple architectural optimizations such as an item memory mixer and latch-based memories. However, like [53] and [39], the processor is only able to accelerate HD classification algorithms. Also, [26] introduces a custom processor for emotion classification of the AMIGOS dataset. Although the processor is customized to perform only a single application, it introduces an effective low-power optimization technique of compressing item vectors using Cellular Automata 90 (CA90).

3.3 Goals of the HPU

As reviewed in the previous section, existing hardware for HDC is restricted in their flexibility and adaptablitity. We propose the Hyper-Dimensional Processing Unit (HPU), a multipurpose processor for HDC that can accelerate all known binary HDC algorithms. Based on the learnings and drawbacks of previous HDC hardware, highlighted in Table 3.1, the HPU has three main design goals:

	Wu et al. [52]	Karunaratne et al. [39]	Eggimann et al. [55]	Datta et al. [53]	Langenegger et al. [54]	HPU
Tech.	RRAM CNFET	90nm PCM 60nm CMOS*	22nm CMOS†	28nm CMOS	14nm PCM 14nm CMOS*	28nm CMOS
HDC Dim.	8192	10000	2048	2048	256	Variable
Application(s)	Classification (Language only)	Classification	Classification	Classification	Factorization	All

* Not fabricated (Post-Synthesis), † Not fabricated (Post-Route)

Table 3.1: Comparison of previous hardware for HDC. Existing processors can only perform a single class of HD algorithms and use fixed-dimension vectors. The Hyper-Dimensional Processing Unit (HPU) is the first processor to accelerate all known binary HDC algorithms and vary the vector dimension to fit the target application.

1. **First processor to accelerate all known binary HDC algorithms.**

   Previous ASICs for HDC have been designed for a single application or class of algorithms. With the introduction of more and more complex HDC algorithms, a multipurpose processor capable of efficiently accelerating all algorithms is critical for the employment of HDC algorithms on a general learning platform. The HPU must be easily programmable to support a multilayered HDC system which can switch between different algorithms to support various cognition tasks.

2. **First processor to change vector dimension to suit target application.**

   In addition to the programmability and flexibility requirements to implement all algorithms, HDC algorithms of different complexities will have different optimal vector sizes. For example, complex algorithms require large vector dimensions (10,000+) for acceptable accuracy. Using the same large dimensions for a simple algorithm is unnecessary when the same accuracy is achievable with much smaller dimensions. In order to save unneeded compute, and therefore energy, the vector dimensions used should be adaptable to fit the target application. Since previous hardware for HDC are designed for a specific application or applications of similar complexity (i.e. classification algorithms), the vector dimensions are fixed at design time. For the HPU, the vector dimension should be an adjustable runtime parameter.

3. **Energy-efficiency comparable to previous application-specific hardware.**

   One key advantage of HDC algorithms over conventional machine learning is their low power and energy-efficiency. Due to the resource limitations of on-the-edge computing, we consider energy per inference or task as the primary optimization metric. Despite the added flexibility and programmability to accelerate multiple algorithms, the HPU should remain competitive in energy-efficiency with the application specific hardware for HDC overviewed in the previous section.

3.4 HPU System Architecture

In order to design a processor that can accelerate all known binary HDC algorithms, we first review all the HDC vector operations and how to implement them in digital hardware. All HDC algorithms are built out of sequences of encoding and similarity operations. In an encoding sequence, vectors are combined using the three element-wise vector operations (multiplication, addition, permutation) into a single encoded vector. For the purposes of hardware implementation, we split the addition operation into its respective accumulation and thresholding operators. Notably, the multiplication, addition, and permutation operators are both commutative and associative. Furthermore, multiplication and permutation distribute over addition. Let's first consider the case where the encoding sequence contains a single addition operation. Using the distributive property, the sequence can always be represented as the addition of terms consisting of only multiplication and permutation operations.

For example

$$\begin{aligned} \boldsymbol{e} &= \boldsymbol{a} \oplus \rho\left([\boldsymbol{b} + \rho(\boldsymbol{c} \oplus \boldsymbol{d})]\right) && (3.1)\\ &= \boldsymbol{a} \oplus [\rho(\boldsymbol{b}) + \rho^2(\boldsymbol{c} \oplus \boldsymbol{d})] && (3.2)\\ &= [\boldsymbol{a} \oplus \rho(\boldsymbol{b}) + \boldsymbol{a} \oplus \rho^2(\boldsymbol{c} \oplus \boldsymbol{d})]\,. && (3.3) \end{aligned}$$

Now consider an encoding sequence with multiple additions. If the additions are nested, then we take the lowest-level addition, distribute out the terms, and evaluate the addition. This process is repeated until all nested additions are evaluated. As a result, we see that in order to evaluate any encoding sequence, we only need hardware that can perform the addition of terms made up of arbitrary multiplication and permutation sequences, and pass the addition result back into the encoding hardware.

The second type of computation required for HDC algorithms is similarity sequences. The commonly used similarity sequences include associative searches and vector construction used in factorization. Unlike the encoding operations, the similarity operations are not element-wise, relying on the Hamann similarity (equivalent to the dot-product similarity of bipolar vectors). Assuming we represent the binary vectors as bipolar ($1 \to 1, 0 \to -1$), the associative search of vector $\boldsymbol{a}$ over a set of vectors $\boldsymbol{v_1}, \cdots \boldsymbol{v_n}$ can be represented as

$$addr = \text{ArgMax}\left(V\boldsymbol{a}\right), V = \begin{bmatrix} — & \boldsymbol{v_1} & — \\ & \vdots & \\ — & \boldsymbol{v_n} & — \end{bmatrix}. \tag{3.4}$$

We note that

$$V\boldsymbol{a} = \begin{bmatrix} — & \boldsymbol{v_1} & — \\ & \vdots & \\ — & \boldsymbol{v_n} & — \end{bmatrix} \boldsymbol{a} = \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix} \tag{3.5}$$

where $s_i =< \boldsymbol{v_i}, \boldsymbol{a} >$ represent the dot-product similarities. For factorization, we extend this functionality by introducing matrix-matrix multiplication in the form of the Gram Matrix. In other words, we also need to compute

$$\boldsymbol{a_{i+1}} = \left[V^T V \boldsymbol{a_i}\right], V = \begin{bmatrix} — & \boldsymbol{v_1} & — \\ & \vdots & \\ — & \boldsymbol{v_n} & — \end{bmatrix}. \tag{3.6}$$

By first evaluating

$$\boldsymbol{a_{i+1}} = \left[V^T(V\boldsymbol{a_i})\right] = \left[\begin{bmatrix} | & & | \\ \boldsymbol{v_1} & \cdots & \boldsymbol{v_n} \\ | & & | \end{bmatrix} \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix}\right] = \left[\sum_{i=1}^{n} s_i \boldsymbol{v}_i\right], \tag{3.7}$$

we see that the factorization vector construction can be represented as an addition scaled by the vector similarities. Since the vector similarities are already computed for associative

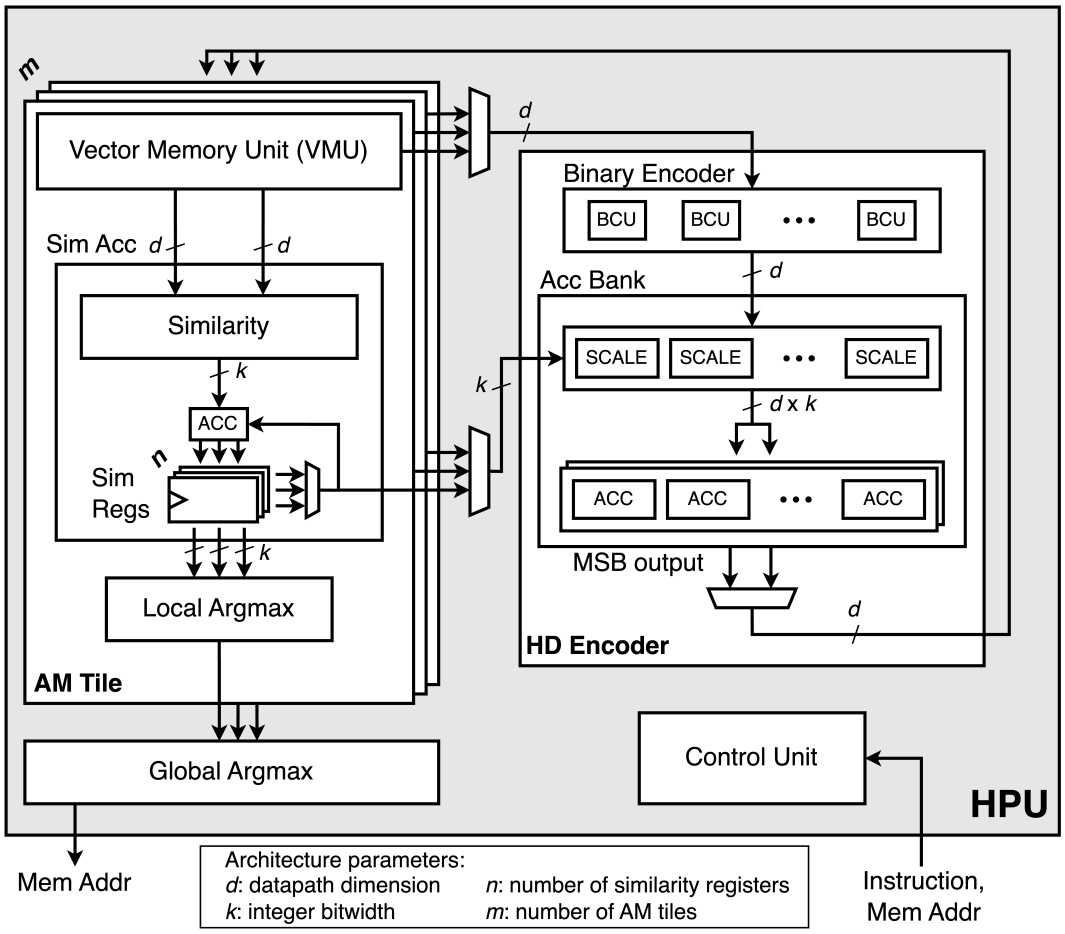

Figure 3.1: System architecture of the HPU, divided into a single HD Encoder and multiple Associative Memory (AM) Tiles. The architecture is parametrized with the design variables listed at the bottom.

search, we only need to add the ability to scale vectors before accumulating in the encoding datapath.

With the hardware requirements for a multipurpose HDC processor described above, we designed a novel system architecture for the HPU shown in Figure 3.1. The single HD Encoder handles all element-wise vector operations. The first module in the HD Encoder is the Binary Encoder. The Binary Encoder performs only permutation and multiplication operations, creating each addend necessary for the distributed and reduced addition computation for encoding. Once the addend is fully constructed, it is passed to the accumulator banks which first optionally scales the binary vector before accumulating into one of two integer vector banks. The accumulator banks output a thresholded vector to store back into the memory or pass back to the input of the HD Encoder.

The Associative Memory (AM) Tiles consist of a Vector Memory Unit (VMU) as well as a Similarity Accumulator (SA). The VMUs hold all item vectors, temporary computation vectors, as well as stored vectors such as class vectors. The VMU can pass any of these vectors to the HD Encoder or to its attached SA. The SAs are responsible for computing vector similarities for both associative search and for scaling purposes (as used in factorization). As a result, the computed similarities are temporarily held in similarity registers that can send the integer similarities to the scaling units in the HD Encoder or to an argmax unit for the associative search. The processor contains multiple AM Tiles to parallelize the similarity computation. In both associative search and factorization vector construction, the same query vector is compared against multiple other vectors which can be mapped onto multiple tiles to speed up the computation.

As such, the HD Encoder can perform the accumulation of arbitrary terms constructed from permutation and multiplication of any vectors in the VMU. The outputs from the HD Encoder can be saved back into memory or fed back to the encoder input. The AM Tiles and the HD Encoder are bidirectionally connected, which means the HPU can perform any combination of encoding and similarity sequences required for all known binary HDC algorithms. The HPU accepts instructions and memory addresses corresponding to the target item or stored vectors, which are translated by the control unit into control signals that control the vector movement and operation of each module. The HPU then outputs memory addresses that correspond to item and stored vectors as the result of an associative search.

The described architecture has several important tuneable parameters, also highlighted in Figure 3.1. First is the *datapath dimension* d. While the HPU is able to vary the vector dimension used for computation (discussed further in the next sections), the baseline datapath dimension has a large effect on the size and efficiency of the processor. If d is too large, routing and fanout of control signals and the similarity computation will become problematic for timing and power, while if d is too small, the overhead and total latency required for time-multiplexing of large algorithms may become problematic. As previously discussed, the *number of AM Tiles* m are also configurable with the optimal number depending on the amount of parallelism required for the similarity computations. Finally, the *integer bitwidth* k and *number of similarity registers* n put soft restrictions on the total size and complexity

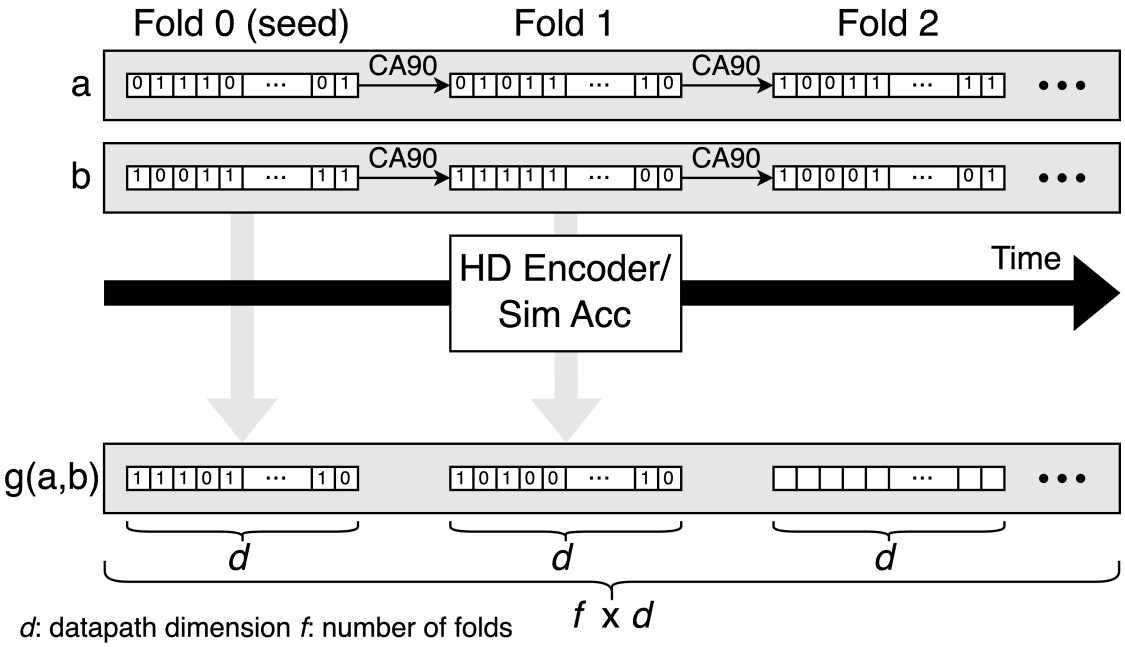

Figure 3.2: Visualization of the dimensionality scaling feature using time-multiplexing over fixed size vector folds.

of algorithms that can be run on the HPU. These architectural parameters play a large role in optimizing the HPU for energy-efficiency.

3.5 Dimensionality Scaling

The previous section highlighted the flexibility of the HPU architecture and how it can be used to accelerate all known binary HDC algorithms. This section discusses how the HPU system can adapt the dimension of the computed vectors to fit the target application, a feature we call dimensionality scaling. Since the vector multiplication, addition, and permutation operations are element-wise, the vectors can be split up into smaller sub-vectors, called folds, and computed one fold at a time. Note that permuting folds through a cyclic shift is not exactly the same as permuting the entire vector, as the values at the start and end of the folds may become corrupted. However, assuming the fold length is fairly large, this introduced bit-error is tolerated by the natural robustness of HDC. Finally, for the vector similarity computations, the bipolar dot-product computation can also be done separately on each fold and then accumulated to find the final similarity value. In other words, given a fixed datapath dimension d, we can compute using vectors of dimension $fd, f \in \mathbb{N}$ where f is the folding factor through time-multiplexing, as depicted in Figure 3.2.

Using vector folding to time-multiplex larger computations is not a novel feature. [52],

[55], and [26] all proposed a similar form of vector folding to compute with larger vectors on a fixed-size smaller datapath. However, in all three instances, the folding factor is a design time parameter used to reduce the processor's area and is unable to be changed at runtime. Supporting arbitrary folding factors for a multipurpose HDC processor adds additional challenges as well. For example, depending on the algorithm, the loop over vector folds can be the innermost loop (such as when accumulating similarities) or the outermost loop (when performing a vector encoding). When switching between the two, any temporary computation vectors need to be stored and reloaded to avoid recomputation which wastes compute time and energy. The HDC handles these challenges with the VMUs which can store and load temporary vectors as well as item and class vectors.

3.6 Pseudo-Random Generation using Cellular Automata

While vector folding enables dimensionality scaling on the HPU, using large dimension vectors would still impose a large burden on the item memories, which would need to store correspondingly large item vectors for use in the HDC algorithms. In order to compress the item memories and instead generate them on the fly, the VMUs include Cellular Automata 90 (CA90) modules. CA90 is an iterative sequence which takes in a fixed length binary sequence and return a sequence where every element is the XOR of the previous and next elements in the input sequence. In other words, given an input vector $\boldsymbol{v} \in \{0,1\}^n$,

$$\text{CA90}(\boldsymbol{v}) = \text{CA90}\left(\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix}\right) = \begin{bmatrix} v_n \oplus v_2 \\ v_1 \oplus v_3 \\ \vdots \\ v_{n-2} \oplus v_n \\ v_{n-1} \oplus v_1 \end{bmatrix}. \tag{3.8}$$

This sequence can be repeated by feeding the CA90 output back into the input which generates multiple pseudo-random sequences of bits from a randomly initiated seed. The use of CA90 to generate binary item vectors for HDC was first studied in [56] which also investigated the additional degrees of freedom introduced by each step. The study found that after a certain number of CA90 steps, very few degrees of freedom were added to the sequence. In other words, the CA90 sequence eventually repeats. The number of steps at which this occurs generally increases as the length of the initial seed grows. This phenomenon is also reflected in [26], which shows a sharp drop off in classification accuracy at large folding factors.

While [26] uses CA90 to both create and extend item vectors, the HPU stores an individual seed for each item vector. This ensures that the CA90 steps are used purely to create vector folds, allowing the HPU to compute with extremely large vector dimensions if desirable. As a result the HPU seed lengths are set to match the datapath dimension d such

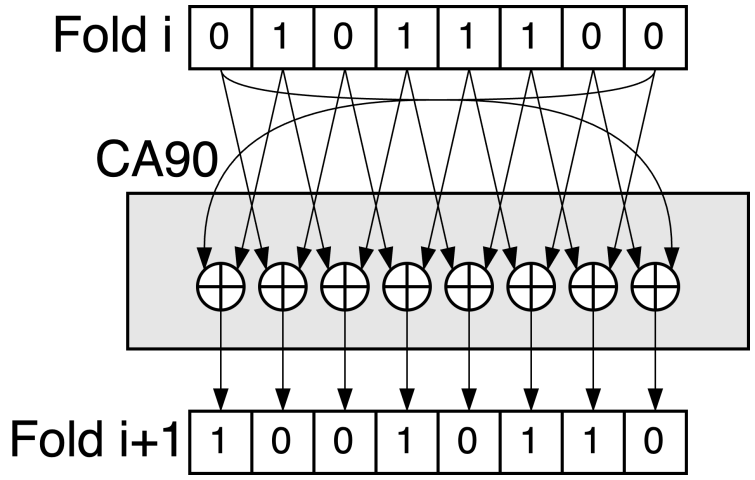

Figure 3.3: Block diagram of the CA90 unit and how item vector folds are updated.

that each CA90 step represents a single vector fold used for dimensionality scaling, show in Figure 3.3.

3.7 Conclusion

In this chapter, we covered the limitations of previous hardware for HDC. The proposed HPU addresses these limitations by being flexible and adaptable in both vector operations and vector size. We have shown that the HPU can implement any combination of binary encoding and similarity computation operations. As a result, the HPU can accelerate all known binary HDC algorithms while remaining energy-efficient. As a strictly binary HDC processor, the HPU cannot implement HDC algorithms with certain non-binary vector operations. For example, HPU does not have the capability to store and load integer vectors as required for the classification retraining method described in [40] to increase classification accuracy. However, such non-binary operations have limited use in HDC applications, and often the end goal of such methods can be implemented in other ways. For example, as shown in Chapter 2, application accuracy can be boosted with intelligent vector encoding schemes.

Finally, the architectural parameters of the HPU impose constraints on the binary HDC algorithms that can be effectively accelerated. Firstly, the memory rows in the VMUs must be sufficient to store all necessary item vectors, stored vectors, and temporary computation vectors. The folding factor determines the number of rows required per stored vector, limiting the maximum folding factor based on memory capacity. Secondly, the integer bitwidth and the number of similarity registers significantly influence application performance when mapped to the HPU. A small integer bitwidth, relative to the effective vector dimension, can degrade the accuracy of similarity metrics. In scenarios requiring scaled accumulation, if the

number of addends exceeds the available similarity registers, recomputation of similarities for each vector fold is necessary, reducing performance. Consequently, selecting appropriate architectural parameters is critical for ensuring the HPU's applicability and energy efficiency, as will be further discussed in the following chapters.

Chapter 4

HPUv1: Design and Implementation

4.1 Introduction

With the goals, system architecture, and primary features of the HPU in place, this chapter discusses the design, implementation, analysis, and results of HPUv1. As the first version of the HPU, HPUv1 is meant to be a small-scale proof of concept of the proposed system architecture. The main goals are to showcase that HPUv1 can implement all known binary HDC algorithms and can adjust the vector dimension using a combination of dimensionality scaling and CA90 vector generation. HPUv1 is also an important source of data and analysis for the further optimization of the HPU architecture. This chapter gives a detailed breakdown of each module in HPUv1 as well as the physical design process. We share the on-chip verification and measurements of HPUv1, and finally analyze and discuss the issues of HPUv1 and opportunities for improvement.

4.2 Processor Design

Although the previous chapter overviewed the system architecture and modules of the HPU, the exact hardware implementations of the modules were not discussed. This section provides details on the hardware and design choices for each module.

Vector Memory Units (VMUs)

We begin with the VMUs which store, load, and generate vectors as required by the processor. As shown in Figure 4.1, each VMU contains an SRAM and an output register attached to a CA90 module. The SRAMs hold all random item vectors in the top rows while the other rows are reserved for stored and/or encoded vectors. The exact number of item and stored vectors vary depending on the needs of each algorithm. The SRAM width is sized to the datapath dimension d such that all vector fold read and writes can occur in a single cycle. In practice, the VMU contains several SRAMs in parallel due to SRAM width restrictions.

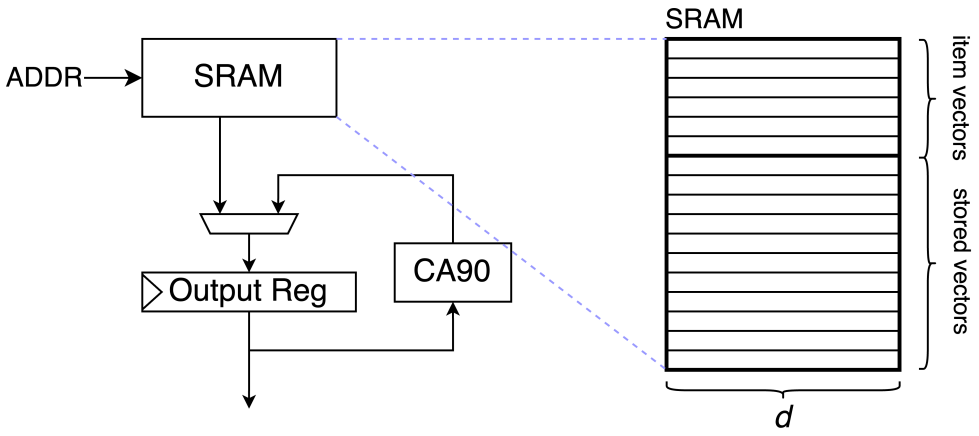

Figure 4.1: Block diagram of the VMU and depiction of the address space of the contained SRAM.

In this case, the SRAM addresses and control signals are driven together so that they act as a single large-bandwidth memory.

Note that for item vectors, each SRAM row corresponds to a different item vector seed. However, when using dimensionality scaling, all encoded vectors must be stored in full. For example with a folding factor of four, all encoded vectors should be stored as four lines in SRAM with each line representing a single fold. As such, the number of stored vector rows is generally large in order to support high folding factors.

When an SRAM read occurs, the output values are first passed to an output register. From there, stored vectors are directly sent to the HD Encoder or Similarity Unit. However, with dimensionality scaling, item vectors need further processing before leaving the VMU. Since the SRAM only contains the item vector seeds, the CA90 module updates the vector currently in the output reg and stores the update back into the register. A single update step is enacted each cycle, and thus the total number of cycles to prepare a vector fold is equal to the current fold number n. In other words, vector reads from the VMU take one cycle for stored vectors, and a variable $1 + n$ cycles for item vectors.

The HPU address space is designed such that each VMU contains unique item vectors, while stored vectors may or may not be unique. In this way, each VMU has individual enables set by custom instructions, which allows vector reads, writes, and CA90 updates to enact on any subset of the AM Tiles. During the encoding process, only a single vector enters the HD Encoder at a time, and thus only the VMU which contains the desired vector is activated. However, if we wish to make use of the AM Tiles for associative search, all VMUs should output a vector for its corresponding Similarity Unit. As a result, all memory instructions contain both a row address and one-hot tile enable signal.

HD Encoder

The HD Encoder consists of three main components: the Binary Encoder, the Scale Unit, and the Accumulator Banks as shown in Figure 4.2. Vector inputs from the VMUs are first passed to the Binary Encoder, which is made of up of d parallel Binary Compute Units (BCUs) where d is the datapath dimension. Each BCU is a single-bit datapath that operates on one element of the input binary vector. The BCUs are responsible for the creation of the expanded multiply and permute terms that will eventually be accumulated. Each BCU contains a single bit register fed by a three input multiplexer. The first option simply loads the register from the input bit, essentially just loading the entire vector from the VMU into the Binary Encoder. The second option loads the value from the previous BCU and adapts the BCU registers into a shift register. This performs the cyclic shift operation that is used for vector permutation. Finally, the last option performs an XOR between the currently stored bit and the input bit, storing the result back into the register. This represents the multiplication between the vector currently in the Binary Encoder and the input vector. Each of these three operations occur in a single cycle and is controlled by an operation select signal which is shared by all the BCUs.

Once the Binary Encoder finishes all required permute and multiply operations, the resulting vector is passed to the Scale Units, which converts each element from a binary element to a two's complement integer element. It also optionally scales the entire vector by an input similarity value as required for scaled addition. When converting from binary to integer, we map

$$\{0, 1\} \rightarrow \{-1, +1\} \tag{4.1}$$

when not using scaling, and

$$\{0, 1\} \rightarrow \{-s, +s\} \tag{4.2}$$

when using scaling by similarity value s. With this mapping, thresholding after accumulation can simply be done by taking the most significant bit (MSB) and allows us to avoid computing an exact threshold value.

After converting the binary vector to an integer vector, the vector is accumulated into one of two accumulator banks. Two accumulator banks are required to handle nested addition loops in an encoding sequence, as we must hold the outer loop value as integers while the inner loop is being computed. Each bank consists of a row of integer Accumulate Units, which performs an element-wise accumulation of the integer vectors from the Scale Unit. One accumulation is performed per cycle, and when the summation is finished, it is thresholded back to a binary vector. In accordance with the mapping in equation 4.1, negative integers map to binary 0 while positive integers map to binary 1. From a hardware perspective, this is achieved by taking the complement of the MSB of each element. As a result, the vectors output by the HD Encoder are always binary and correspondingly written into one or more of the VMUs.

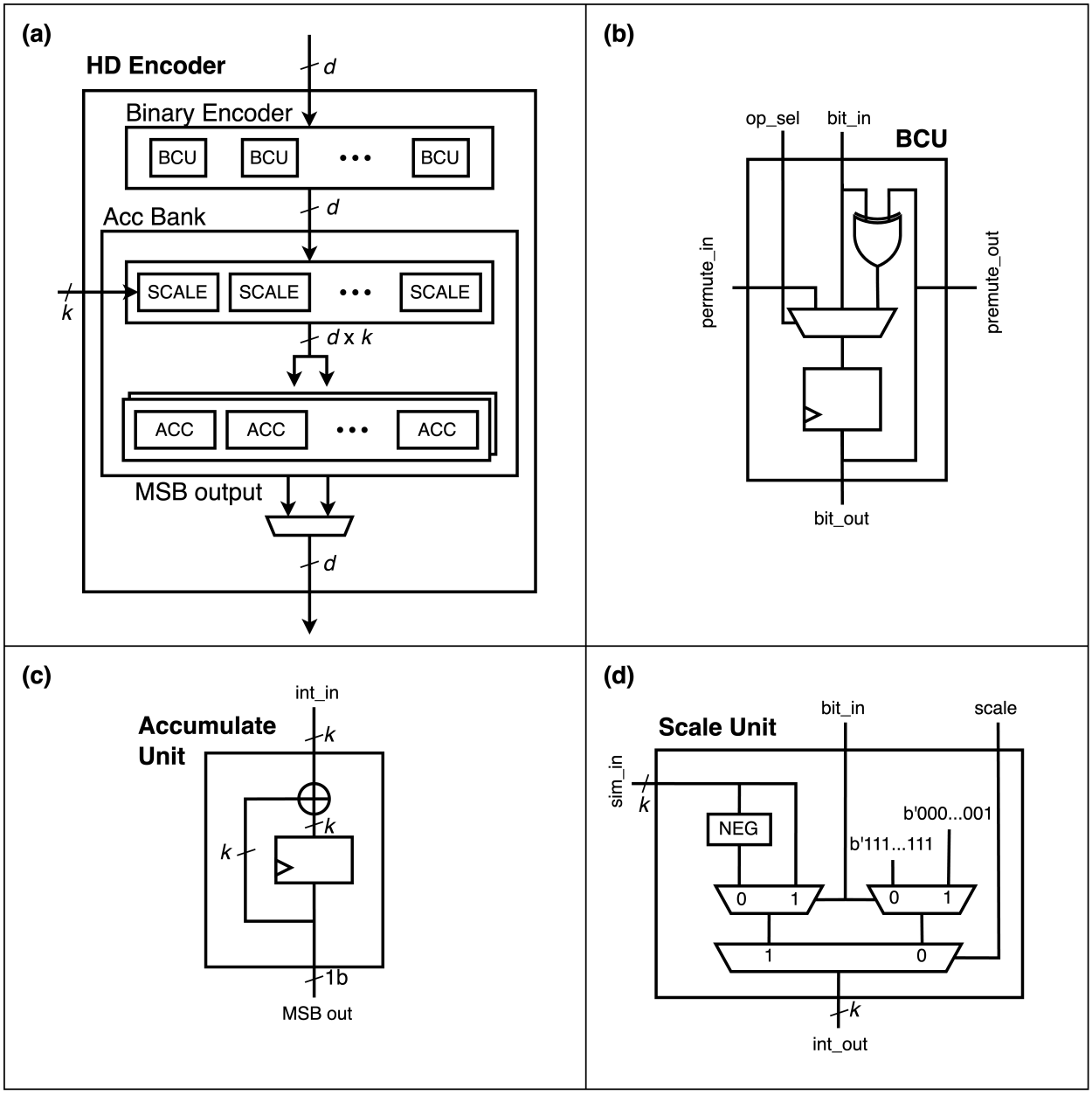

Figure 4.2: Block diagram from the HD Encoder (a) with detailed hardware diagrams for the BCU (b), Accumulate Unit (c), and Scale Unit (d). d represents the datapath dimension and k is the integer bitwidth.

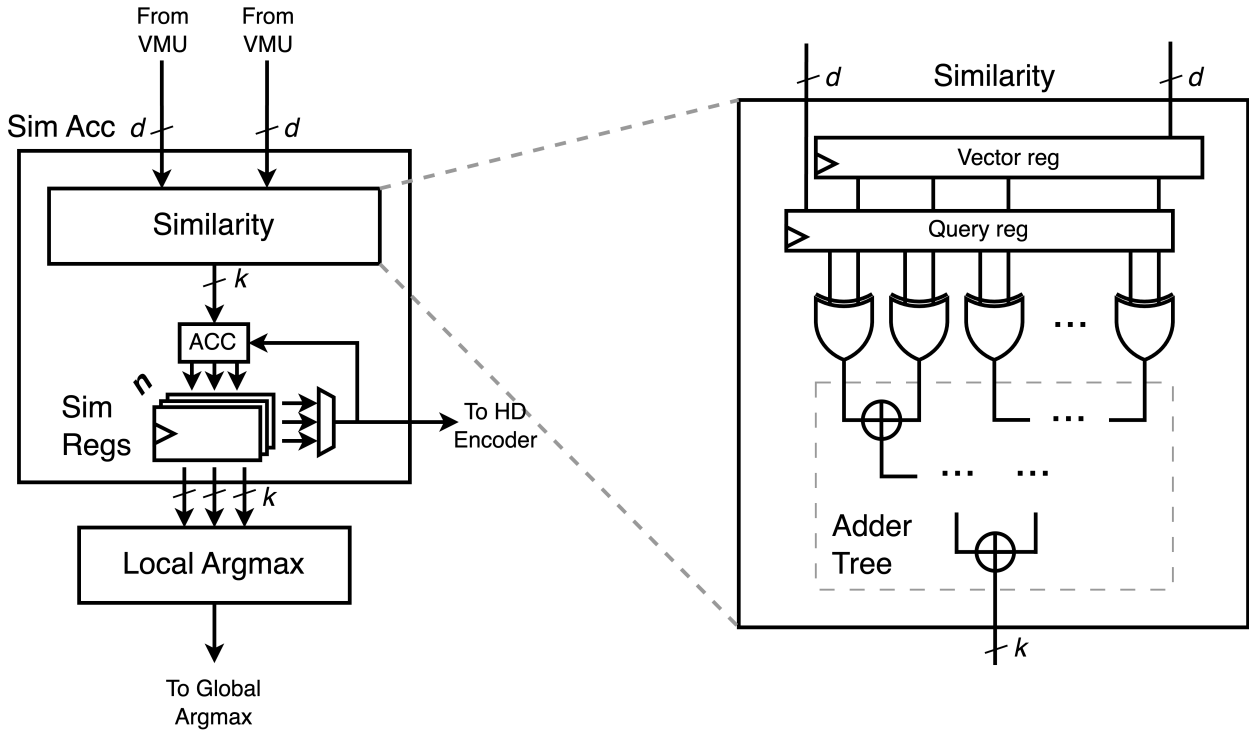

Figure 4.3: Block diagram of the Similarity Accumulator and Local Argmax Unit within each AM Tile. A hardware diagram of the Similarity Unit is also shown.

Similarity Accumulator and Argmax

The similarity and argmax computations occur in a separate datapath which exists in each AM Tile. Within in each tile, the Similarity Accumulator accepts a vector from the VMU and stores it in either the query register or the vector register. Although both registers feed into the Similarity module, when the vector register is loaded, the SRAM row address of the vector is also stored in a separate address register. The Similarity module computes the Hamann similarity, which corresponds to the dot-product if the binary vectors were transformed into their bipolar counterparts. In hardware, this is computed by taking the elementwise XOR between the two vectors, shown in Figure 4.3. The resulting d bits are mapped from binary to integer as

$$\{0, 1\} \rightarrow \{+1, -1\} \tag{4.3}$$

and then fed into an adder tree. The mapping transforms the similarity metric from Hamming distance to Hamann similarity. The integer output is then quantized and passed to the Similarity Accumulator.

The Similarity Accumulator accumulates the Hamann similarity across multiple vector folds when using dimensionality scaling. Each accumulator includes n similarity registers

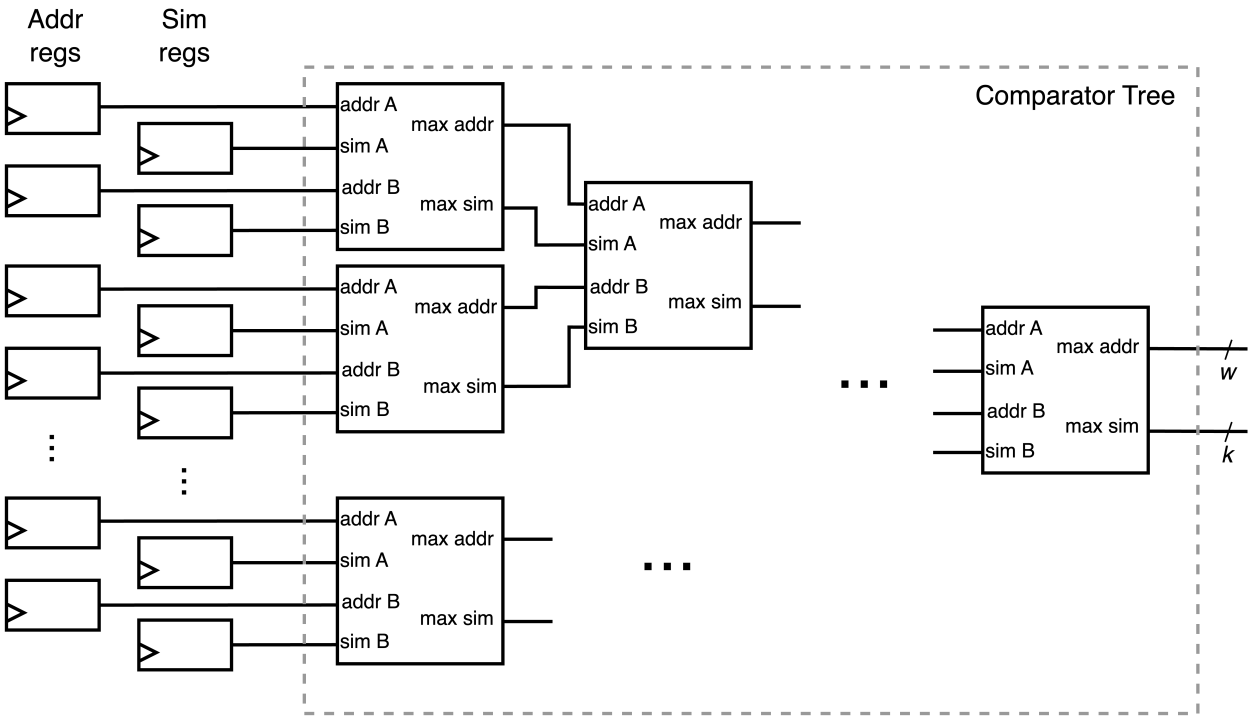

Figure 4.4: Block diagram of the Argmax Units. The same comparator tree structure is used for both the Local Argmax and Global Argmax Units.

which hold the similarities for different vectors that are output either to the HD Encoder for scaled accumulation or to the Local Argmax unit for associative searches. The number of similarity registers n limits the number of argmax comparisons that can be done in a single cycle on a single tile. Moreover, for certain algorithms that use scaled accumulation using the computed similarity values, nm limits the number of addends in the scaled accumulation, where m is the number of AM Tiles. Although performing scaled accumulation on more than nm similarity values is possible, the similarities will have to be recomputed if they are overwritten, which greatly reduces algorithm efficiency. As such, n should be chosen to be large enough to cover most practical scaled accumulation use cases.

For associative searches, the HPU must return the address of the vector most similar to the query. Correspondingly, for every similarity calculated, the corresponding address of each vector sent to the vector register is saved in n address registers. Since control unit sends the same row address to every AM Tile, a single set of address registers can be shared across all the tiles. Once the similarities are all accumulated, the similarity registers and corresponding address registers are fed into the Local Argmax Unit within each tile. Here, a comparator tree is used to output the maximum similarity value alongside its corresponding vector address in a single cycle, as shown in Figure 4.4. The resulting output from each tile's Local Argmax Unit is then output into a single Global Argmax Unit, which also is

implemented as a comparator tree. The Global Argmax Unit also has the ability to compare the set of inputs with the last computed maximum similarity and argmax address so that the associative search can be performed over a larger number of vectors than the total number of similarity registers.

Control Unit and Instruction Set

The HPU is an instruction-based processor which performs one instruction per clock cycle. The system architecture does not include a program memory and therefore requires instructions to be input to the processor every cycle. These instructions are then processed by the control unit which decodes the instructions into module-level control signals. HPUv1's instruction set consists of 33 different instructions with a 6 bit opcode and 16 bit address field, overviewed in Tables 4.1 and 4.2, where each instruction targets a specific HPU module.

The instructions can be categorized as follows:

- **NOP**
  As implied by the name, the `NOP` instruction stalls the processor and holds all register values.

- **Control registers**
  Instructions 2-5 set control registers that configure HPUv1's active datapaths. `tile_en_set` configures the active AM Tiles, allowing the programmer to turn off AM Tiles if the additional similarity computation parallelism and memory size are not necessary. `mem_en_set` controls the write enables for the VMUs, allowing the user to specify which memories should get written to. `accbank_set` selects which accumulator bank is targeted by the accumulate instructions. Finally, `lcomp_set` selects which similarity register values are input into the Local Comparator.

- **IO**
  Instructions 6-13 control HPUv1's input and output buffers. These buffers are used to load both integers and vectors in and out of the HPU.

- **VMU**
  Instructions 14-19 control the HPUv1's VMUs. These include SRAM read/writes as well as CA90 updates. Only VMUs that are activated (according to the control registers) are affected by these instructions.

- **Datapath**
  Instructions 20-33 manipulate the HD Encoder, Similarity Accumulator, and argmax units which implement the vector operations required by HDC computations. These instructions can be flexibly combined into programs for all known binary HDC algorithms.

	Instruction	Opcode	Description
1	`nop`	100000	Do nothing, stall all datapaths, hold all registers.
2	`tile_en_set`	010000	Set active tiles using one-hot active arg.
3	`mem_en_set`	010001	Set active VMUs to be written to using one-hot active arg.
4	`accbank_set`	000000	Select accumulator register to be written to/read from using bank address arg.
5	`lcomp_set`	010010	Set active similarity registers for local argmax using one-hot active arg.
6	`ibuff_clear`	100001	Clear vector and integer input buffers.
7	`obuff_clear`	100010	Clear vector and integer output buffers.
8	`hdenc_clear`	100011	Clear all registers in the HD Encoder.
9	`sim_clear`	100100	Clear all registers in the Similarity Units.
10	`ibuff_vec_load`	100101	Load vector input into input vector buffer.
11	`ibuff_int_load`	100111	Load integer input into input integer buffer.
12	`obuff_vec_load`	000010	Load vector from memory at tile address arg, VMU row address arg or from the Global Argmax Unit into output vector buffer.
13	`obuff_int_load`	100111	Load similarity value from Global Argmax Unit to the output integer buffer.
14	`memreg_load`	001001	Load the memory out registers from the VMU row address arg for all active VMU tiles.
15	`mem_store_ibuff`	000001	Store vector from input vector buffer to VMU row address arg of active memories.
16	`mem_store_be`	000100	Store vector from Binary Encoder to VMU row address arg of active memories.
17	`mem_store_acc`	000011	Store vector from selected accumulator bank to VMU row address arg of active memories.
18	`ca90_load`	000101	Load vector from memory out registers to CA90 registers of all active tiles.
19	`ca90_update`	101011	Update vector in CA90 register by one CA90 step for all active tiles.

Table 4.1: Table of opcodes and descriptions of instructions 1-19 of the HPUv1 ISA. Instructions 20-33 described in Table 4.2.

	Instruction	Opcode	Description
20	`be_load`	000110	Load vector from output of VMU from AM Tile address arg to the Binary Encoder.
21	`be_bind`	000111	Bind vector from output of VMU from AM Tile address arg to the Binary Encoder.
22	`be_perm`	101000	Permute vector in the Binary Encoder.
23	`accbank_load`	110001	Load vector from Binary Encoder to selected accumulator bank. The vector is optionally scaled by the similarity register given by AM Tile address arg and similarity register address arg.
24	`accbank_add`	110010	Accumulate vector from Binary Encoder to selected accumulator bank. The vector is optionally scaled by the similarity register given by AM Tile address arg and similarity register address arg.
25	`accbank_retrieve`	101001	Retrieve vector from selected accumulator bank back to accumulator.
26	`accbank_save`	101010	Save vector from accumulator to selected accumulator bank.
27	`query_load`	001000	Load vector from VMU row address arg to query registers for all active tiles.
28	`sim_compute`	101100	Compute similarity between vector in the query register and VMU output for all active tiles.
29	`simreg_load`	010011	Load similarity value from Similarity Unit to register determined by similarity register address arg for all active tiles.
30	`simreg_add`	010100	Add similarity value from Similarity Unit to register determined by similarity register address arg for all active tiles.
31	`lcomp_load`	101101	Compute argmax similarity of active similarity registers for all active tiles.
32	`gcomp_load`	101110	Compute argmax similarity stored in the Local Argmax Units of active tiles.
33	`gcomp_update`	101111	Compute argmax similarity stored in the Local Argmax Units of active tiles in addition to previously computed global argmax.

Table 4.2: Table of opcodes and descriptions of instructions 20-33 of the HPUv1 ISA. Instructions 1-19 described in Table 4.1.

	HPUv1
Process	TSMC 28nm CMOS
Size	1.6 mm × 1.6 mm
Timing	200 MHz at 0.9 V
Datapath Dimension	512
Integer Bitwidth	12
Similarity Registers	16
AM Tiles	4
Total SRAM	262KB

Table 4.3: Summary of HPUv1 chip specifications and architectural parameters.

4.3 Physical Implementation

HPUv1 serves as a small-scale proof of concept of the HPU architecture and thus architecture parameters were selected according to ease of implementation. Table 4.3 outlines key architectural parameters and chip specifications for HPUv1. The processor's base datapath dimension is 512 bits wide, with 12 bit integer width in order to cover the full dynamic range of the Hamann similarity. HPUv1 includes 4 AM Tiles each with 1024 row VMUs and 4 similarity registers, yielding an aggregate storage capability of 4096 vector folds of 512 bits each (262KB total). We fabricated HPUv1 using TSMC28nm CMOS using only standard digital cells and SRAMs within a total die area of 1.6 mm × 1.6 mm. Physical design of the chip was performed using Cadence Genus and Innovus, starting from Register Transfer Logic (RTL) written in SystemVerilog. HPUv1 does not have any internal clock generation hardware and relies on an off-chip clock input. Since we are targeting optimal energy-efficiency, which usually occurs at reduced clock frequency and supply voltage, HPUv1 timing is closed at a theoretical max frequency of 200 MHz at 0.9 V.

IO and Communication

The fabricated HPUv1 processor contains a simple In Out (IO) controller in addition to the HPU core. The IO controller handles reading and writing from the input and output buffers of the HPU. The HPU contains two input buffers and two output buffers, one for vectors and one for integer values. The input vector buffer allows the programmer to save a vector into the HPU VMUs, and is required to initialize the item memories before starting an HDC algorithm. The input integer buffer is used to insert custom similarity values for use in scaled accumulation. The output vector buffer can load either a vector from the VMU or the argmax address held in the global comparator unit. The former can be used for verification purposes, while the latter reads out associative search results. Finally, the output integer buffer can output any integer stored in the similarity registers.

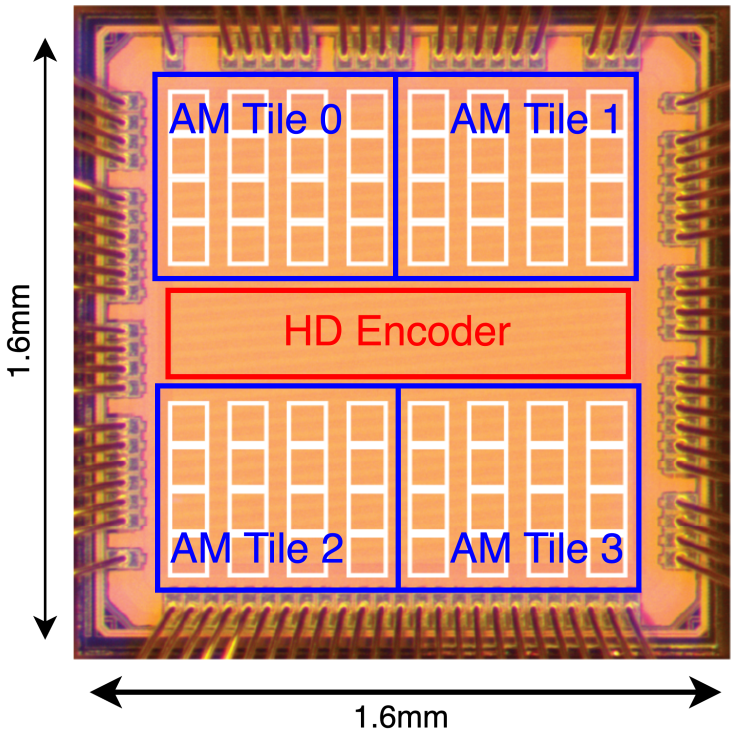

Figure 4.5: Die micrograph of HPUv1 with an overlay of the chip floorplan. The white squares represent individual SRAM blocks.

The input and output vector buffers contain 512 bits (64B) of data which would require too many data IO ports. As a result, the IO controller implements shift registers with custom shift control pins to fill or output the buffers one byte per cycle. Since the integer buffers are 12 bits, HPUv1 has 12 input and output data ports, where the full 12 bits are used for integer load/reads and the bottom 8 are used for vector load/reads.

HPUv1 uses a wirebond pad ring to get signals on and off the die. Each pad is connected to an IO cell that requires a higher 1.3 V supply (VDDPST, VSSPST) in addition to the core supply (VDD, VSS). The HPUv1 die contains a total of 86 pads: 8 VDD, 8 VSS, 8 VDDPST, 12 VSSPST, 1 clock, 1 reset, 12 data in, 12 data out, 22 instruction bits, and 2 IO control signals. Pads are arranged around the square perimeter with the intermixing of supply and ground pads to abide by the electrostatic discharge rules. HPUv1 sits in a Ceramic Pin Grid Array (CPGA) package which wirebonds signals from the die pads to the package pins.

Floorplan

Figure 4.5 shows the die micrograph of HPUv1 with the superimposed floorplan. Since the HPU requires flexibility in vector movement between the various VMUs and compute units, it is critical to intelligently floorplan the processor to minimize congestion and timing issues due to routing. As the HD Encoder needs to connect to every VMU, it is placed in the center of the floorplan with the 4 AM Tiles in each corner. Each tile contains 16 SRAMs of size 32x1024 in parallel which are equally spaced throughout the tile. The similarity and argmax compute units are placed in between the SRAMs within each tile.

4.4 Chip Verification and Measurement

The python emulator, program compiler, and test setup used to verify and measure HPUv1 is covered in detail in Appendix A. In this section, we discuss the verification process as well as power and performance measurements for HPUv1.

Kernel Tests

Most processors report performance and energy-efficiency in terms of operations per second (OPS) and operations per second per Watt (OPS/W). For the HPU, there is no single representative operation that can be used to summarize expected power and performance. As a result, we define the four following kernel operations for the HPU. The kernel operations are small instruction macros that are common among almost all HDC algorithms on the HPU.

1. **CA90:** The CA90 operation represents a single update step of a vector fold using CA90.

2. **Multiply-add:** Given n pairs of vectors $(\boldsymbol{a_i}, \boldsymbol{b_i}), i = 1, \ldots, n$, the multiply-add operation encodes $[\sum_{i=0}^{n} \boldsymbol{a_i} \oplus \boldsymbol{b_i}]$. Reported kernel power and performance is normalized by n.

3. **Ngram encoding:** Given n vectors $\boldsymbol{a_1}, \ldots, \boldsymbol{a_n}$, the Ngram encoding computes $\boldsymbol{a_n} \oplus \rho(\boldsymbol{a_{n-1}} \oplus (\ldots \oplus \rho(\boldsymbol{a_1}))$. Reported kernel power and performance is normalized by n.

4. **Associative search:** Given n vectors and a single query vector, the associative search computes the similarity between the query vector and all n vectors and returns the address of the most similar vector. Reported kernel power and performance is normalized by n.

These kernel operations are a good representation of the HPU operations that occur during an actual benchmark. As a result, they are suitable proxy to report power and performance to compare between HDC platforms. In order to ensure a fair comparison, the kernel operations computed with an effective vector dimension of 1024, and thus HPUv1

uses a folding factor of 2 in the kernel measurements. The kernel operations cover nearly all compute instructions for the HPU and are valuable for HPU verification. In the verification tests, encoded vectors are stored into the VMUs and then shifted out to compare against the emulator results.

Verification and Functional Issues

When verifying the kernels on HPUv1, we discovered several implementation issues summarized below:

- **Triple memory read**
  The VMUs in HPUv1 were mistakenly fabricated with a mismatch in the control signals paths. The SRAM chip enable signal is delayed compared to the read address bits the signal to update the output register. As a result, the `memreg_load` instruction does not properly load the memory output register. Fortunately, we find that three consecutive `memreg_load` instruction properly synchronizes the VMU control signals and loads the desired vector into the memory output register. As a result, this issue does not hamper the functionality of HPUv1, although it reduces performance due to the increase memory read latency.

- **Similarity metric**
  HPUv1 was fabricated with a bug in the Similarity Unit. Although we designed the Similarity Unit to compute the Hamann similarity, HPUv1 was synthesized with an old version of the Similarity Unit RTL which computes the Hamming distance, which introduce two key functional issues. First, the associative search uses the similarity values to return the most similar vector to the query. In terms of the Hamann similarity, more positive values mean the vectors are more similar and vice versa. As a result, we use argmax to determine the address of the most similar vector. However for Hamming distance, large values mean the vectors are more dissimilar. Thus using argmax over the Hamming distance returns the *least* similar vector instead of the most similar. Unfortunately, for algorithms end with an associative search such as classification, HPUv1 will be unable to verify functionality on-chip.

  Second, compared to the Hamann similarity which is zero-centered, the Hamming distance is not. This becomes an issue for scaled accumulation in which we want the final sum vector to be more representative of the most similar vector. In the case of the Hamming distance, the most similar vector will have a distance close to zero and thus will be the least represented vector in the sum. As a result, algorithms with scaled accumulation also cannot be verified on-chip.

  Luckily, these issues can be easily remedied by properly computing the Hamann similarity instead of the Hamming distance. Future versions of the HPU contain the correct Similarity Unit module with updated RTL.

While the VMU issue only affects processor performance, the similarity metric bug prevents us from verifying any full benchmarks as all benchmark outputs are results of an associative search. However, algorithms without similarity computations can be fully verified. As a result, we find that HPUv1 correctly computes the CA90, Mulitply-Add, and Ngram Encode kernels. We also mapped the language classification algorithm onto HPUv1 and verified that the encoded class vectors indeed match our emulator.

Despite being unable to verify HPUv1's similarity computation, we can assume that the power measured during the associative search will be extremely similar to a version of HPUv1 that correctly computes the Hamann similarity, as the Hamann similarity and Hamming distance only differ by a binary to integer mapping layer in the adder tree.

Kernel Measurements

Since the HPU aims to optimize energy-efficiency, we first aim to find the optimal operating voltage and frequency to operate HPUv1. For each kernel, we sweep the core VDD and clock frequency and record the kernel accuracy. The kernel accuracy is defined as the percentage of bits that match between the encoded vector output by HPUv1 and the one computed in the emulator. Figure 4.6 shows these accuracies in the form of heatmaps for each kernel. The figure also plots the energy-efficient operating points, defined as the lowest VDD that still achieves 100 % accuracy at each clock frequency. Note that the energy-efficient operating points are different for different kernels. For example, The CA90 and Ngram kernels fail at lower VDDs compared to the multiply-add kernel, which signifies that the accumulator banks have less timing slack when compared to the CA90 module and Binary Encoder.

For each kernel, we measure the HPUv1 power and latency at its energy-efficient operating points and compute the corresponding kernel energy, shown in Table 4.4. From the energy-efficient operating points, we then report the lowest energy data point, highlighted in green in the table.

4.5 Power and Performance Bottlenecks

The on-chip measurements from the previous section are difficult to analyze as the reported data cannot be split into its individual constituents. In order to optimize the HPU, we need simulated system-level and module-level power and performance to determine the areas that are bottleneck for the HPU. To get use-case accurate data, we mapped the emotion recognition algorithm [57] into HPUv1 instructions and simulated the program on the post-synthesis netlist. We used PrimeTime PX to run power analysis on the post-synthesis netlist with the cycle-accurate switching data. The power breakdown for the emotion recognition benchmark with a folding factor of 8 (4096 dimension vectors) is shown in Table 4.5. Of the total average power of 15.16 mW dissipated by HPUv1, the SRAMs are the dominant source of processor power at nearly 70 %, followed by the AM Tiles (without the SRAMs) at 17 %, and the HD Encoder at 12.6 %.

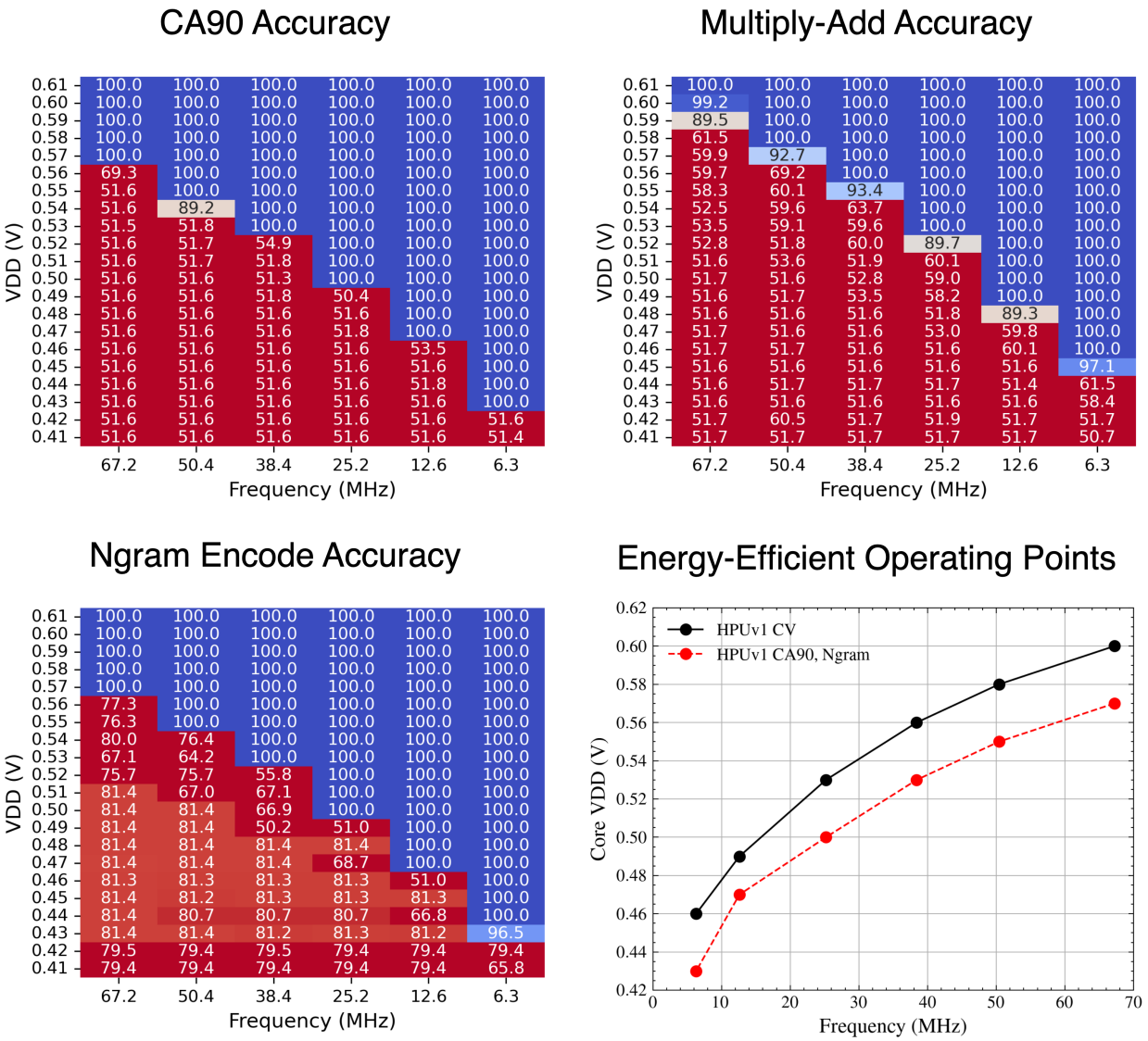

Figure 4.6: Kernel accuracy for three kernels measured on HPUv1 as a function of core voltage (VDD) and clock frequency. Blue signifies higher accuracy while red signifies lower accuracy. The resulting energy-efficient operating points are also shown.

CA90

Freq (MHz)	VDD (V)	Avg Power (mW)	Latency (µs)	Energy/Op (nJ)
67.2	0.57	2.58	0.04	0.09
50.4	0.55	2.10	0.05	0.10
38.4	0.53	1.63	0.06	0.10
25.2	0.50	1.07	0.09	0.10
12.6	0.47	0.58	0.19	0.11
6.3	0.43	0.35	0.38	0.13

Multiply-Add

Freq (MHz)	VDD (V)	Avg Power (mW)	Latency (µs)	Energy/Op (nJ)
67.2	0.60	3.03	0.40	1.20
50.4	0.58	2.36	0.53	1.24
38.4	0.56	1.82	0.69	1.26
25.2	0.53	1.21	1.06	1.27
12.6	0.49	0.60	2.11	1.27
6.3	0.46	0.41	4.22	1.71

Ngram Encode

Freq (MHz)	VDD (V)	Avg Power (mW)	Latency (µs)	Energy/Op (nJ)
67.2	0.57	2.45	0.23	0.56
50.4	0.55	1.99	0.31	0.61
38.4	0.53	1.49	0.40	0.60
25.2	0.50	0.99	0.61	0.60
12.6	0.47	0.56	1.22	0.69
6.3	0.43	0.33	2.44	0.82

Table 4.4: Measured kernel power, latency, and energy at the energy-efficient operating points. Lowest-energy data point highlighted in green.

Category	Dynamic power (mW)	Leakage power (mW)	Total power (mW)	%
Total	13.18	1.98	15.16	
SRAMs	10.08	0.33	10.48	69.2%
AM Tiles	2.12	0.48	2.6	17.3%
HD Encoder	0.92	0.99	1.91	12.6%
Other	0.06	0.07	0.17	0.9%

Table 4.5: Post-synthesis delay-annotated average power simulation of the Emotion Recognition algorithm [57] on HPUv1 with a folding factor of 8. Power analyzed at the TT0p9V25C corner with a 200 MHz clock.

Instruction	Cycles	%
Total	655572	
`sram_read`	273204	41.7%
`ca90_update`	149800	22.9%
`ca90_load`	86752	13.2%
Other	145816	22.2%

Table 4.6: Instruction breakdown for the Emotion Recognition algorithm [57] run on HPUv1 with a folding factor of 8.

We also investigated the instruction frequency of the benchmark program by taking a histogram of the instructions, with the top three shown in Table 4.6. Again we find that the VMU instructions make up nearly 78 % of all operations performed by the HPU. The simulated benchmark data yields two main conclusions:

- **SRAM power is problematic.**
  In our benchmark test, we found that the power dissipated by compute datapaths (AM Tiles and HD Encoder) account for only $\sim$ 30 % of the total processor's power, and the remaining 70 % is spent reading and writing to the SRAM. Reducing SRAM area and activity should be a key factor in optimizing HPU's energy-efficiency.

- **Reading vectors out of the VMU is inefficient.**
  The benchmark test also revealed that HPUv1 spends a vast majority (78 %) of cycles executing VMU instructions. Even if we account for the fixing the triple read issue, the VMU instruction percentage only drops to $\sim$ 70 %. Furthermore, at higher folding factors, the VMU instruction percentage will grow due to the additional cycles required to update the vector folds.

  Since HPUv1 instructions only target a single module, this means that the VMU heavily bottlenecks the HPU architecture. As a result, the HD Encoder or Similarity Accumulator must stall and wait until the VMU finishes preparing the vectors. Since the algorithm runtime linearly correlates with the total test energy, improving the process of preparing and moving vectors from the VMU to the datapaths is critical.

As a baseline first version of the proposed architecture, HPUv1 has much room to grow in terms of power and performance. With the findings from this section and the experience from the physical design process, we have identified key weaknesses which will be improved for future versions of the processor.

Chapter 5

HPUv2: Optimization and Characterization

5.1 Introduction

The previous chapter overviewed our first attempt to design and build the HPU. As a prototype for the HPU architecture, HPUv1 focused on the first two goals for the HPU outline in Chapter 3: the ability to accelerate all known binary HDC algorithms and the flexibility to adapt the computed vector dimensions. In order to achieve the third HPU goal, our second processor, HPUv2, aims to optimize energy-efficiency in accordance with HDC's many low-power use cases.

This chapter covers the extensive changes made both to the system architecture and individual hardware modules compared to HPUv1. These optimizations are analyzed through post-synthesis power simulations and a python emulator. HPUv2 shares a similar physical design process to HPUv1 and is fabricated in the same 28nm CMOS process which allows for direct comparison of power and performance results. We perform rigorous low-power characterization of HPUv2 and provide comparisons against HPUv1 as well as previous hardware.

5.2 Energy Optimization and Updates

In Chapter 4, we discovered that VMUs are the limiting factor for both power and performance for HPUv1. As a result, the bulk of the energy optimizations performed for HPUv2 focus on the VMUs. The following three sections outline the main hardware changes performed for HPUv2.

CA90 Cache & Memory-Datapath Pipeline

In our test benchmark for HPUv1, we found that nearly 70 % of the benchmark instructions were VMU operations. The majority of these instructions (50 % of the total benchmark) are

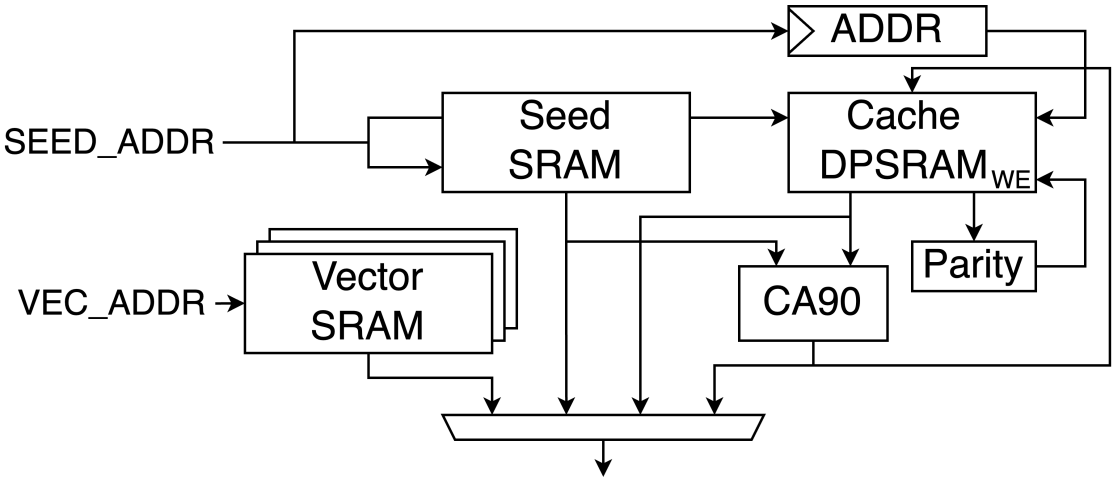

Figure 5.1: Block diagram of updated VMU with added automatic write-back CA90 cache. The cache stores an additional parity bit used to determine if write-back is necessary.

spent loading and updating the CA90 register within the VMU. Furthermore, for higher folding factors, CA90 instructions will make up an even higher percentage of total instructions. Since vector folding time-multiplexes the vector operations, we expect the number of cycles performing encoding or similarity computations to scale linearly with the folding factor. For the VMUs, the number of cycles to update an item vector seed to the output vector depends on the current fold number. For example, if an algorithm uses a folding factor of f, then the total number of cycles spend updating the CA90 register is

$$1 + 2 + 3 + \cdots + f = \frac{f(f+1)}{2} \sim \mathcal{O}(f^2) \tag{5.1}$$

quadratic with f whereas all other operations are linear.

To remove the quadratic dependency, HPUv2 implements a CA90 cache which holds the most recent CA90 update of each seed in a separate SRAM, shown in Figure 5.1. While HPUv1 had a single SRAM that held both item vectors and stored vectors, HPUv2 has separate SRAMs for each vector type. The Seed SRAM only holds the random seeds for item vectors, the Cache SRAM holds CA90 updates of the item vector seeds, and the Vector SRAM hold all encoded and stored vectors. Each SRAM is width d to ensure that one full vector fold can be read/written to the VMU every cycle. Furthermore, the number of rows in the Cache SRAM matches that of the Seed SRAM to guarantee updates of all seeds can be stored in the cache.

The updated VMU has 4 distinct operations:

- **Vector write**
  A vector is input to the VMU and stored in the Vector SRAMs. In most cases this

will be an encoded vector (such as a class vector) or a temporary computation vector. This operation takes one cycle.

- **Vector read**
  A vector from the Vector SRAMs is read and output from the VMU. This operation takes one cycle.

- **Item write**
  An item vector seed is input to the VMU and stored in the Seed SRAM. This usually only occurs during processor setup to initialize the random item seeds. This operation takes one cycle.

- **Item read**
  An item vector fold is requested from the VMU, which behaves differently based on the current fold number. If the current fold number is non-zero, the Cache vector at the requested address is read. Cache SRAM also contains one extra bit which holds the parity corresponding to the CA90 update number of the stored vector. If the parity matches the current fold number, then the cached vector is output. If not, the cached vector is first passed through the CA90 module and then the updated vector is output. The updated vector is then written back to the Cache SRAM in the next cycle. The cache is implemented as dual port SRAM (DPSRAM) so that the write-back happens automatically in the next cycle, and the SRAM can continue read operations as normal. Finally, if the current fold number is zero, then the seed vector is output, while the CA90 updated seed is written to initialize the cache in the next cycle. In order for the write-back to occur at the correct address, the address for all item reads are registered into an address register which connects to the write address port of the cache. All item reads take one cycle, while the automatic cache write-back occurs on the next cycle if necessary.

The addition of the CA90 cache greatly reduces runtime of HDC algorithms by lowering the number of CA90 updates required, especially at higher fold numbers. All CA90 operations now happen in the background by means of the automatic cache and thus no explicit CA90 load and update instructions are necessary. Moreover, all VMU read operations for HPUv2 have a fixed one cycle latency. In other words, the CA90 cache now guarantees that a vector fold needed by the HD Encoder or Similarity Accumulator is available after one cycle.

To gain a more representative understanding of the resulting instruction speedup, we analyze the effects on three kernels (Multipy-Add, Ngram encoding, Associative Search) described in Section 4.4. Mapping the kernels to both HPUv1 and HPUv2 instructions, we compute the total number of instructions required as a function of the folding factor f and the number of vectors/vector pairs n, shown in Table 5.1. The CA90 cache removes the quadratic dependency on f and the guaranteed one cycle latency also reduces the coefficient of the linear term as well.

Kernel	HPUv1 Instructions	HPUv2 Instructions
Multiply-Add	$5nf + \frac{nf(f-1)}{2}$	$3nf$
Ngram encoding	$3nf + \frac{nf(f-1)}{2}$	$2nf$
Associative search	$2 + 2f + \frac{f(f-1)}{2}$	$2 + f$

Table 5.1: Comparison of number of instructions needed to implement for kernel operations in HPUv1 vs in HPUv2. f is the folding factor while n is the number of channel value pairs or the Ngram number.

Furthermore, the added CA90 cache also enables the optimization of the HPUv2 control pipeline. Since reading a vector from the VMU in HPUv1 has a variable latency depending on the type of vector read and the fold number, HPUv1's instructions only activates one module at a time. This ensures that all required VMU instructions finish before the VMU output is read into the vector datapaths. However, since HPUv2 VMUs have a fixed read latency, we can prefetch vectors that we wish to compute with in the next cycle from the VMU. In this way, both the VMU and the HD Encoder or Similarity Accumulator can be active in the same cycle. The vector prefetch is implemented by adding a second pipeline stage for the control signals of the vector datapaths, while the VMU keeps only a single pipeline stage as depicted in Figure 5.2. We also demonstrate how the CA90 cache and vector prefetch reduces the number of instructions to perform the same HDC algorithm. The charts on the right indicate when modules are active, indicated by a filled-in blue cell. Each row in the table represents one clock cycle.

With improvements in instruction efficiency, i.e. the number of instructions to implement the same HDC operations, in both the VMU and the Control Unit, we compiled a full factorization benchmark on both HPUv1 and HPUv2 to compare the instruction speedup, shown in Figure 5.3. The benchmark was compiled with 1, 2, 4, 8, and 16 folds corresponding to a speedup from $\sim 2\times$ to $> 30\times$. As predicted, the number of instructions for HPUv2 is linear with the number of folds, leading to a quadratic speedup when compared to HPUv1. The instruction speedup improves both algorithm latency and throughput, and as a result lowers benchmark energy.

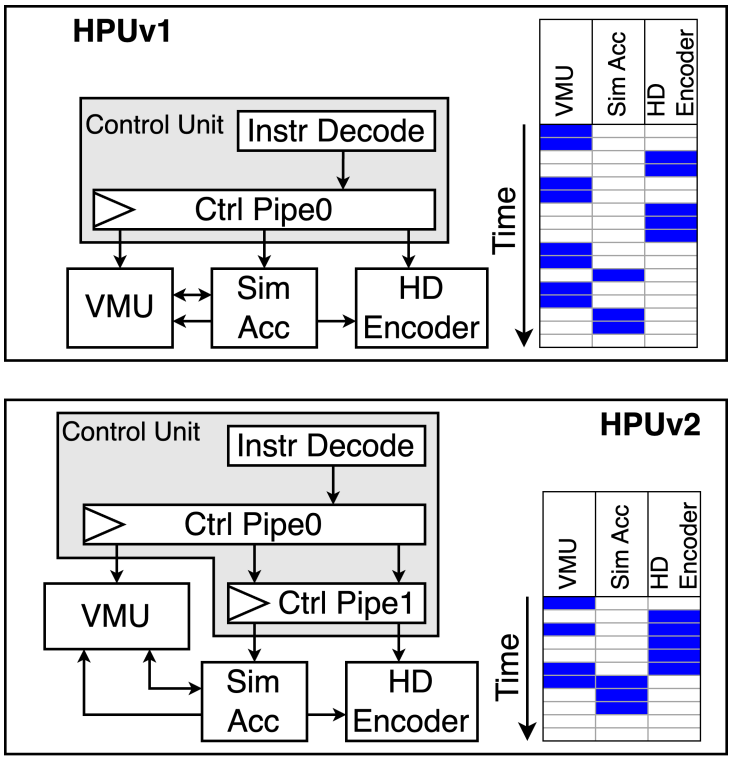

Figure 5.2: Comparison of the Control Unit pipelines between HPUv1 and HPUv2. The right-hand tables show the resulting speedup where each row represents one cycle and a blue cell indicates that the module is active.

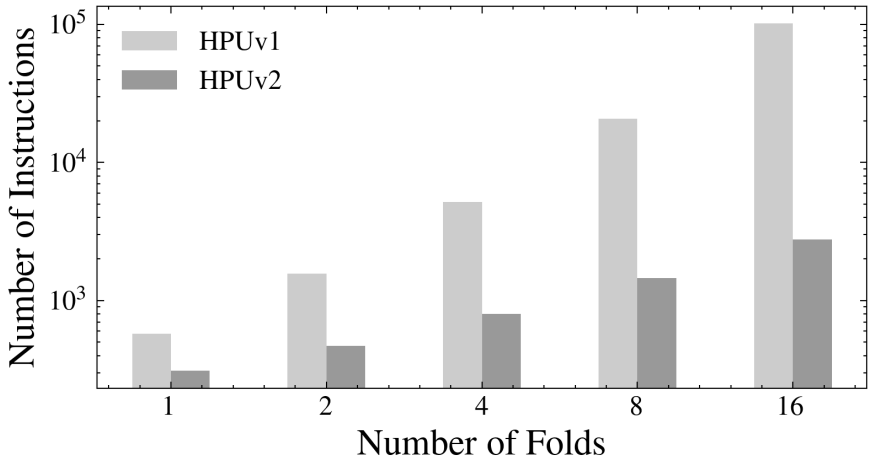

Figure 5.3: Number of instructions for the HD factorization benchmark on HPUv1 and HPUv2 vs. number of vector folds used. Note that both axes are log-scale.

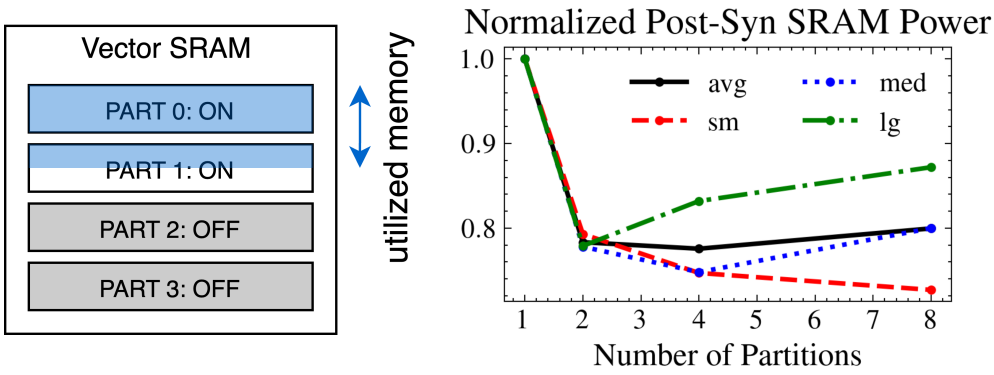

Figure 5.4: Depiction of the SRAM partitioning scheme for general vector storage. Normalized post-synthesis power vs. number of partitions for benchmarks with small, medium, and large vector usage. Power analyzed at the TT0p9V25C corner with a 200 MHz clock.

Memory Reduction and Partitioning

HPUv1 revealed that SRAM power was the dominant portion of total processor power. For HPUv2, we employ several optimizations reduce SRAM power. The simplest process is to cut down the amount of SRAM on chip. The number of rows of SRAM limits the number of vectors and the folding factor that can be used for an algorithm on the HPU. We surveyed the HPU vector usage across a broad assortment of HDC algorithms with different folding factors, where the vector usage represents the number of SRAM rows required, defined as the number of item vectors added to the number of stored vectors multiplied by the folding factor. At a folding factor of one, the recall of reactive behavior has the highest vector usage at 274, while at a folding factor of 16, language classification overtakes it with a vector usage of 401. We also simulated an extremely large synthetic benchmark using HD factorization with 9 factors and 70 items per factor, which uses 648 to 933 vectors at 1 and 16 folding factors respectively.

The total vector storage capacity of HPUv1 is 4096 rows, far greater than the vector usage we find across existing HDC algorithms. In order to reduce SRAM area and power, we reduce the vector storage capacity to 2048 for HPUv2, half of the original design. Note that even 2048 is much greater than the maximum usage of 933 vectors we found. However, in order to ensure that future algorithms that require more vectors or a higher folding factor can fit on HPUv2, we decided for a conservative limit of 2048. The 2048 rows are split into 512 rows for the Seed SRAM, 512 rows for the Cache SRAM, and 1024 rows of general vector storage.

In most cases, especially at low folding factors, many algorithms use fewer than 200 vector storage rows, much fewer than the 2048 capacity of HPUv2. In particular, the number of rows needed for general vector storage can be quite low at low folding factors. As a result, HPUv2 partitions the general vector storage into separate SRAMs so that smaller algorithms

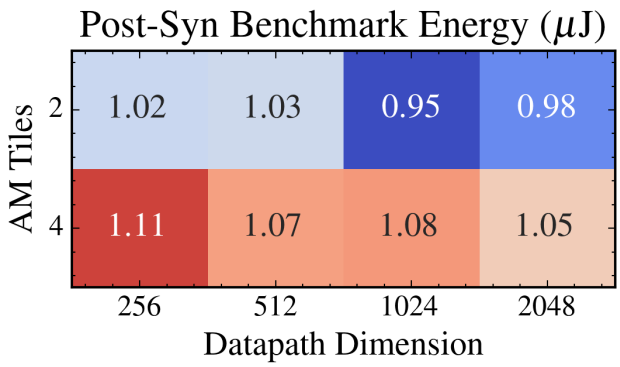

Figure 5.5: Post-synthesis energy of a HD factorization benchmark on HPUv2 synthesized with different number of AM Tiles and datapath dimension. Lower energy is blue while higher energy is red. Power analyzed at the TT0p9V25C corner with a 200 MHz clock.

can turn off unneeded SRAMs to store power, as shown in Figure 5.4.

We computed the total SRAM power in addition to the post-synthesis partition control logic overhead across different number of partitions. For each number of partitions, we looked at three HDC algorithms with small, medium, and large amounts of vector usage. The power shown is normalized to see the percent power savings for each benchmark. For small algorithms, all required vectors can likely fit into the first partition, and thus a larger number of partitions means smaller partition sizes and consequently less total power. On the other hand, large algorithms would likely use most of the available partitions, so increasing the number of partitions only introduces additional overhead without significantly changing the amount of active/inactive SRAM. Since most known algorithms fall under the small and medium categories, HPUv2 uses 4 vector partitions which provides an average of 22 % power savings across all benchmark sizes.

Architecture Parameter Optimization

With large improvements to both the memory output efficiency and power, the final front of optimization focuses on the architectural parameters of HPUv2. While HPUv1 selected its architectural parameters based on ease of implementation, we analyzed and chose architectural parameters based on energy-efficiency for HPUv2.

The base datapath dimension of the processor d and the number of AM tiles m are used to trade off processor size and performance. Unlike the other architectural parameters, these parameters do not affect the size or complexity of algorithms that can fit on the HPU. In other words, any combination of d and m will be functionally equivalent. For example, if we wish to compute s similarities using t dimension vectors, we need to use a folding factor of $\frac{t}{d}$ with $\frac{s}{m}$ similarity compute instructions per fold. Choosing higher d or m lowers the amount of time-multiplexing required by increasing the parallelism. In order to find

the optimal values of d, m, we ran post-synthesis power simulations of an HD factorization benchmark on different combinations of d and m, shown in Figure 5.5. In general, we see that more AM Tiles is energy-inefficient. The optimal datapath dimension size depends on the number of AM Tiles, with the optimal value increasing with a larger number of tiles. For HPUv2, we chose the lowest energy combination of 2 AM Tiles with a datapath dimension of 1024. Although we only simulated a few combinations of parameters since post-synthesis simulations are expensive, we are confident about the choice based on the trends in the data. The 2048 rows of vector storage split over the 2 AM Tiles so that each tile contains 256 rows for the Seed SRAM, 256 rows for the Cache SRAM, and 4 partitions of 128 rows each for general vector storage. In total, the total SRAM capacity of HPUv2 is 252KB, the same as for HPUv1. However, HPUv2 is approximately double the size of HPUv1 due to its larger datapath dimension of 1024, which means that the proportional power from SRAM will be lower.

HPUv2 also implements two techniques to reduce integer bitwidth to 8 from 11 in HPUv1. Integers are used in the HD Encoder accumulators as well as the Similarity Compute and Accumulate units. In HPUv2, the similarity computation output is quantized according to the instruction argument. The quantization right shifts the output integer by a set number of bits so that at large folding factors, the similarity values can be appropriately quantized so that the similarity accumulators do not overflow. Second, all accumulators are equipped with overflow protection so that they saturate at the high and low limits ($[-128, 127]$ for 8 bit integers). For these limits, our simulations show that once the accumulator in the HD Encoder saturates, the accumulated value is very unlikely to flip its sign. As a result, 8 bit integers show no reduction in algorithm accuracy if the quantization value is carefully chosen.

Finally, the total number of similarity registers is increased to 32 for HPUv2, with 16 registers per tile. This choice ensure that algorithms with a large similarity register requirement such as HD factorization can be efficiently accelerated on HPUv2. We find that 16 registers per tile is close to the upper limit in terms of timing and routing congestion for the similarity accumulators without a complete redesign of the hardware, and in practice 32 registers is plenty for nearly all HDC algorithms.

Bugfixes

HPUv2 fixes the two functional flaws found in the design and analysis of HPUv1. The newly designed VMU no longer requires consecutive read operations to output a vector to memory. Furthermore, as HPUv2 uses the VMU as a pipeline stage, all memory read instruction are combined with a data movement instruction (see next section for more detail). HPUv2 also properly computes the Hamann similarity instead of the Hamming distance.

	Instruction	Opcode	Description
1	`nop`	00000	Do nothing, input datapath and register stall commands into instruction pipeline.
2	`fold_rst`	00001	Set current fold number to 0.
3	`fold_incr`	00010	Increment current fold number by one.
4	`part_set`	00110	Set active memory partitions with one-hot active arg for tile specified using tile address arg.
5	`lcomp_set`	00111	Set active similarity registers for local argmax using one-hot active arg for tile specified using tile address arg.
6	`tile_en_set`	01000	Set active AM Tiles using one-hot active arg.
7	`mem_en_set`	01001	Set active VMUs using one-hot active arg.
8	`ibuff_vec_load`	00011	Load vector input into input vector buffer.
9	`ibuff_int_load`	00100	Load integer input into input integer buffer.
10	`obuff_vec_load`	10010	Load vector from memory at tile address arg, VMU row address arg or from the Global Argmax Unit into output vector buffer.
11	`obuff_int_load`	11000	Load similarity value from similarity register at tile address arg, buffer address arg, or from Global Argmax Unit to the output integer buffer.
12	`mem_store_acc`	10011	Store vector from accumulator bank given by bank address arg to VMU row address arg of VMUs set by one-hot active arg.
13	`mem_store_ibuff`	10100	Store vector from input buffer to VMU row address arg of VMUs set by one-hot active arg.
14	`mem_store_be`	10101	Store vector from Binary Encoder to VMU row address arg of VMUs set by one-hot active arg.
15	`be_perm`	00101	Permute vector in the Binary Encoder.
16	`be_load`	10000	Load vector from VMU at row address arg, tile address arg or from the accumulator bank to the Binary Encoder.
17	`be_mult`	10001	Multiply vector from VMU at row address arg, tile address arg or from the accumulator bank to the Binary Encoder.

Table 5.2: Table of opcodes and descriptions of instructions 1-17 of the HPUv2 ISA. Instructions 18-26 described in Table 5.3.

	Instruction	Opcode	Description
18	`accbank_load`	11001	Load vector from Binary Encoder to accumulator bank at bank address arg. The vector is optionally scaled by the similarity register given by AM Tile address arg and similarity register address arg.
19	`accbank_add`	11010	Add vector from Binary Encoder to accumulator bank at bank address arg. The vector is optionally scaled by the similarity register given by AM Tile address arg and similarity register address arg.
20	`query_load`	10110	Load vector from VMU row address arg to query registers for all active tiles.
21	`sim_compute`	10111	Compute similarity between vector in the query register and vector from VMU row address arg to query registers for all active tiles.
22	`simreg_load`	11011	Load similarity value from Similarity Unit to register determined by similarity register address arg for all active tiles.
23	`simreg_add`	11100	Add similarity value from Similarity Unit to register determined by similarity register address arg for all active tiles.
24	`lcomp_load`	01100	Compute argmax similarity of active similarity registers for all active tiles.
25	`gcomp_load`	01010	Compute argmax similarity stored in the Local Argmax Units of active tiles.
26	`gcomp_update`	01011	Compute argmax similarity stored in the Local Argmax Units of active tiles in addition to previously computed global argmax.

Table 5.3: Table of opcodes and descriptions of instructions 18-26 of the HPUv2 ISA. Instructions 1-17 described in Table 5.2.

Modified Instruction Set

With the changes in the HPUv2 control unit and instruction pipeline, HPUv2 has a more streamlined instruction set that combines several HPUv1 instructions into a single instruction. Similar to HPUv1, HPUv2 accepts and performs one instruction per cycle, requiring an instruction input to the processor every cycle. HPUv2 has a slightly smaller instruction set of 26 instructions with a 5 bit opcode and 20 bit address field, overviewed in Tables 5.2 and 5.3. Compared to HPUv1, there area few key differences:

- The new `fold_rst` and `fold_incr` instructions set the current fold number, which is required for the CA90 Caches.
- There are no longer any individual memory read commands. Since the VMU acts as a pipeline stage, HPUv2 commands including `be_load`, `be_mult`, and `sim_compute` now take in an additional vector and tile address argument. The vector is read on the first cycle and computed in the datapath during the second.
- Certain control registers which were often changed during an algorithm (such as the write memory select and accumulator bank select) were removed for HPUv2. Instead, the `mem_store` and `accbank` instructions take one-hot VMU active and accumulator bank address arguments respectively.

5.3 Physical Implementation

With the optimization and analysis described in the previous section, HPUv2 maximizes the energy-efficiency of the HPU architecture so that it can be competitive with existing application specific processors. Table 5.4 summarizes the processor specifications and architectural parameters, and compares them to HPUv1.

The fabrication process for HPUv2 is the same as HPUv1: TSMC28nm CMOS using only standard digital cells and SRAMs. Physical design of the chip was also performed using Cadence Genus and Innovus, starting from RTL written in SystemVerilog. Like HPUv1, relies on an off-chip clock input, with timing closed at the same frequency of 200 MHz at 0.9 V. HPUv2 is a significantly larger chip than HPUv1 with a die area of 2.1 mm $\times$ 2.1 mm that is nearly twice the area of HPUv1. The larger area accommodates the 2$\times$ larger datapath, which means that for a given compute vector size, HPUv2 can operate with half the folding factor of HPUv1.

IO and Communication

HPUv2 has a slightly modified a IO controller in comparison to HPUv1. It still contains two input buffers and two output buffers, one for vectors and one for integer values. The vector buffers are again implemented as 8 bit shift registers. However now that the integer

	HPUv1	HPUv2
Process	TSMC 28nm CMOS	TSMC 28nm CMOS
Size	1.6 mm × 1.6 mm	2.1 mm × 2.1 mm
Timing	200 MHz at 0.9 V	200 MHz at 0.9 V
Datapath Dimension	512	1024
Integer Bitwidth	12	8
Similarity Registers	16	32
AM Tiles	4	2
Total SRAM	262KB	262KB
SRAM partitions/tile	-	4

Table 5.4: Summary of HPUv2 chip specifications and architectural parameters and comparison to HPUv1.

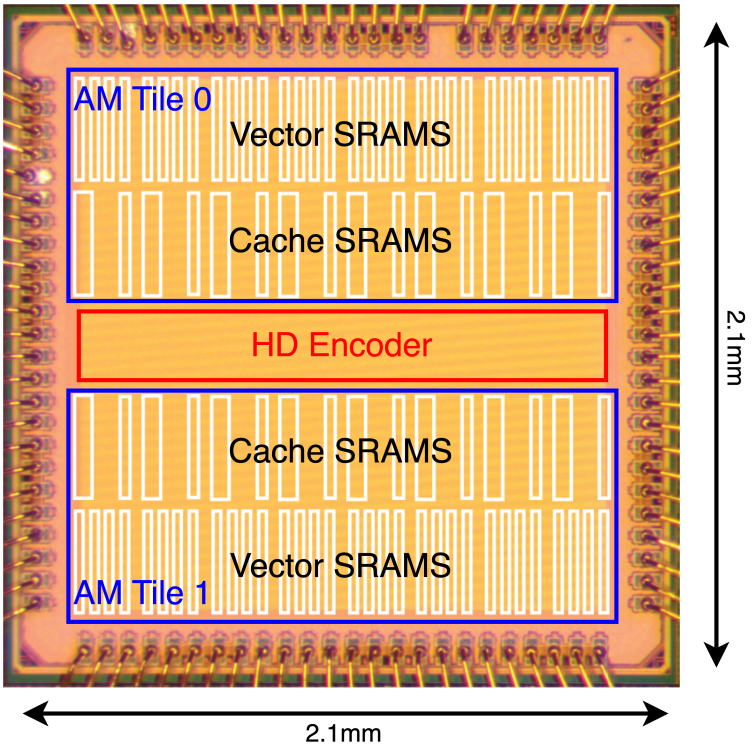

Figure 5.6: Die micrograph of HPUv2 with an overlay of the chip floorplan. The white squares represent individual SRAM blocks.

bitwidth is also 8, HPUv2 only requires 8 input and 8 output data ports. The shift register controls are slightly modified so that the user must assert the shift signal to shift the register rather than as a trigger signal in HPUv1. This change allows for slightly better flexibility when switching between integer and vector inputs or outputs.

HPUv2 also uses a wirebond pad ring to get signals on and off the die powered by the same IO cells. The HPUv2 die contains a total of 96 pads: 12 VDD, 12 VSS, 12 VDDPST, 12 VSSPST, 1 clock, 1 reset, 8 data in, 8 data out, 26 instruction bits, 2 IO control signals. These pads are arranged around the perimeter of the chip which are wirebonded to a CPGA package.

Floorplan

Figure 5.6 shows the die micrograph of HPUv2 with the superimposed floorplan. The floorplan uses a similar methodology with the HD Encoder in the center as it can receive a vector from any SRAM on the chip. Since there are only two AM Tiles, each tile is placed on the top and bottom sides of the die. Within each tile, the Seed SRAM and Cache are placed closest to the HD Encoder, as that the most common source of vectors loaded into the encoder. As the VMUs require an extremely wide bandwidth of 1024 bits, each memory is split into 8 parallel SRAMs each with a width of 128 bits (129 bits for the cache). The similarity and argmax units are again placed in between the SRAMs within each tile.

5.4 Chip Measurements

For information about the python emulator, program compiler, and test setup used to verify and measure HPUv2, please refer to Appendix A. In this section, we discuss characterization of HPU kernel power tests and full benchmarks.

Kernel Characterization

We begin by running and verifying the kernel tests described in Section 4.4. The kernel test validity is first checked by reading out the stored encoded vectors or search results and compared against the emulated outputs. With the updated Similarity Unit, HPUv2 passes all kernels including the associative search. We measure the accuracy of each kernel across a range of voltages and frequencies to find the energy-efficient operating points of HPUv2. For the CA90, multiply-add, and Ngram encode kernels, accuracy is determined by the percentage of bits that are correctly read out of the VMUs after encoding. For the associative search, accuracy is computed as the percentage of correct search results. The resulting accuracy heatmaps are presented in Figure 5.7. Note that for the associative search, the maximum accuracy does not reach 100 % due unexpected behavior if all the similarity values being compared are negative. However, other than in our random kernel,

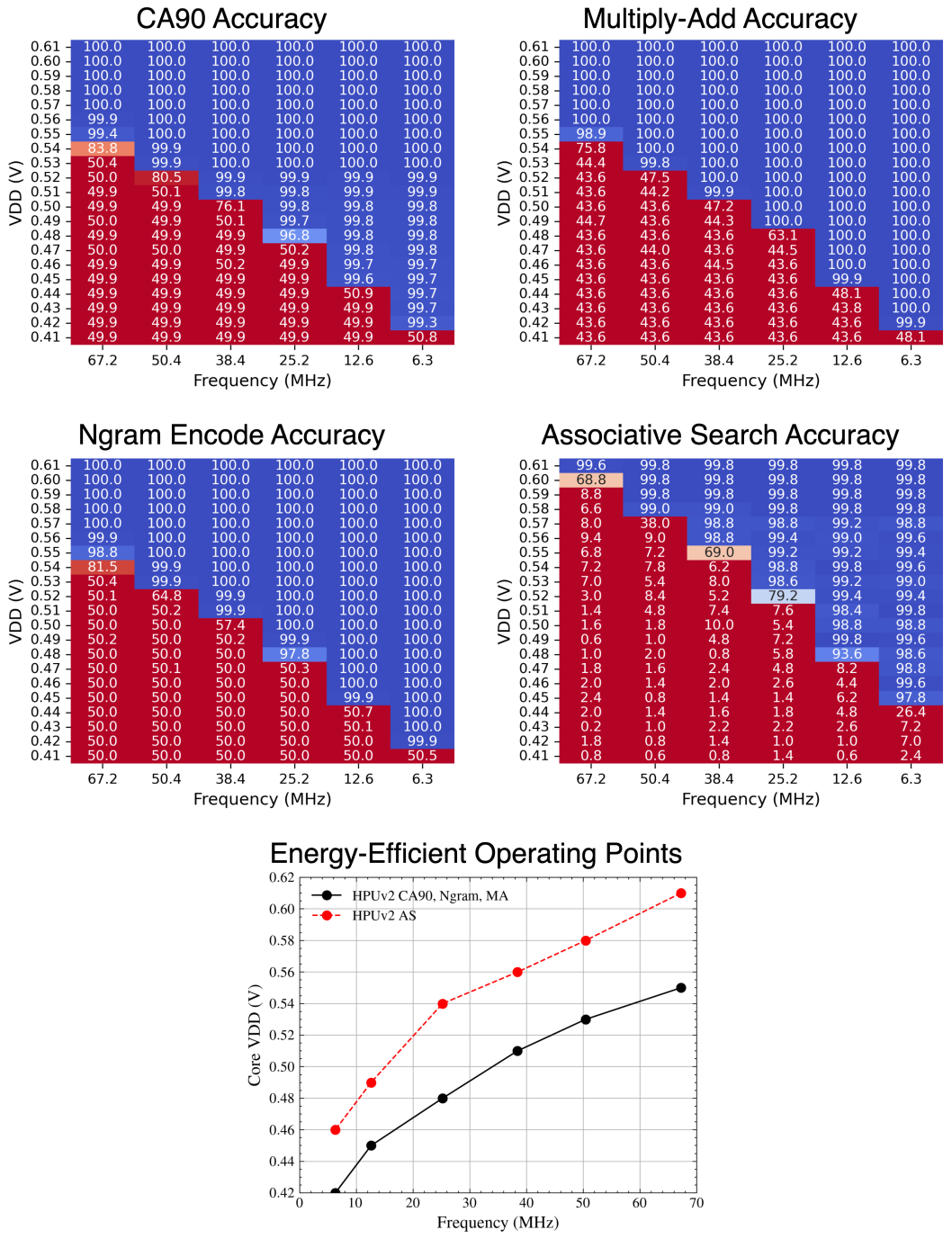

Figure 5.7: Kernel accuracy for three kernels measured on HPUv2 as a function of core voltage (VDD) and clock frequency. Blue signifies higher accuracy while red signifies lower accuracy. The resulting energy-efficient operating points are also shown.

this will never occur practically and does not pose a functional issue for HPUv2. The energy-efficient operating points, defined as the lowest voltage that provides $> 99\,\%$ accuracy for each frequency, are significantly lower for the associative search kernel compared to the HD Encoder kernels. This is understandable because although HDC is robust to bit-errors in the vector datapaths, any errors induced in the argmax computation would cause a completely different vector address to be output.

For each kernel and its energy-efficient operating points, we measure the on-chip power and latency to compute the kernel energy per operation, shown in Table 5.5 with the lowest-energy point highlighted in green. Across all kernels, we see that lowest-points all occur at the same frequency of 12.6 MHz.

Benchmark Characterization

HPUv2 is characterized over three benchmarks: European language classification [24], EMG gesture classification [58], and HD factorization [48]. The benchmarks were chosen they have been characterized on prior hardware for HDC, which allows for a fair comparison point for the HPU. Furthermore, the applications span over different HDC algorithms, other than the HPU, no single existing ASIC can accelerate all three.

In order to find the energy-efficient operating points for each benchmark, they are first characterized on a wide range of chip voltage and frequencies. The benchmark accuracy at each point is shown in the heatmaps in Figure 5.8. The language classification benchmark takes in random input text of different 21 different European languages and outputs a language prediction. 100 tests for each language are run at each operating point for a total of 2100 tests, and accuracy is reported as the percentage of correct predictions. HPUv2 uses a folding factor of 2 for an effective vector dimension of 2048 in order to match the benchmark accuracy reported in literature. The EMG gesture recognition benchmark predicts hand gestures from 64 channels of input EMG signals. For the accuracy characterization, we run 1300 tests on each operating point and again report the percentage of correct predictions. A vector folding factor of one is chosen as the smallest factor that meets the accuracy reported in literature. For the factorization benchmark, we implemented several networks with different numbers of factors and items per factor. The accuracy heatmap is shared for 3 factors with 64 items per factor, which is computed using a folding factor of 2. 200 factorization tests are computed at each operating point, with the accuracy being the percentage of correct factorizations.

When choosing energy-efficient operating points for the benchmarks, we first find the baseline benchmark accuracy, which is defined as the benchmark accuracy at the highest voltage and lowest frequency operating point (i.e. 0.62 V, 6.3 MHz) since we expect no failed timing paths and thus no injected error. We then define a threshold accuracy as 99 % of the baseline accuracy. The energy-efficient operating points are chosen as the lowest VDD that achieves at least threshold accuracy for each frequency point, as shown in Figure 5.9. For the operating frequency shown (25.2 MHz), the lowest VDD that maintains greater than threshold accuracy is 0.53 V. The figure also the low-VDD robustness of the HPU. At

CA90

Freq (MHz)	VDD (V)	Avg Power (mW)	Latency (µs)	Energy/Op (nJ)
67.2	0.55	5.47	0.02	0.09
50.4	0.53	4.00	0.02	0.08
38.4	0.51	2.68	0.03	0.07
25.2	0.48	1.69	0.04	0.07
12.6	0.45	0.73	0.08	0.06
6.3	0.42	0.41	0.17	0.07

Multiply-Add

Freq (MHz)	VDD (V)	Avg Power (mW)	Latency (µs)	Energy/Op (nJ)
67.2	0.55	5.77	0.05	0.27
50.4	0.53	4.15	0.06	0.26
38.4	0.51	2.95	0.08	0.24
25.2	0.48	1.77	0.12	0.22
12.6	0.45	0.83	0.25	0.20
6.3	0.42	0.45	0.49	0.22

Ngram Encode

Freq (MHz)	VDD (V)	Avg Power (mW)	Latency (µs)	Energy/Op (nJ)
67.2	0.55	5.43	0.03	0.18
50.4	0.53	3.99	0.04	0.17
38.4	0.51	2.74	0.06	0.16
25.2	0.48	1.71	0.09	0.15
12.6	0.45	0.76	0.17	0.13
6.3	0.42	0.43	0.35	0.15

Associative Search

Freq (MHz)	VDD (V)	Avg Power (mW)	Latency (µs)	Energy/Op (nJ)
67.2	0.55	5.88	0.04	0.23
50.4	0.53	4.15	0.05	0.22
38.4	0.51	2.82	0.07	0.19
25.2	0.48	1.89	0.10	0.20
12.6	0.45	0.80	0.21	0.17
6.3	0.42	0.47	0.42	0.20

Table 5.5: Measured kernel power, latency, and energy at the energy-efficient operating points on HPUv2. Lowest-energy data point highlighted in green.

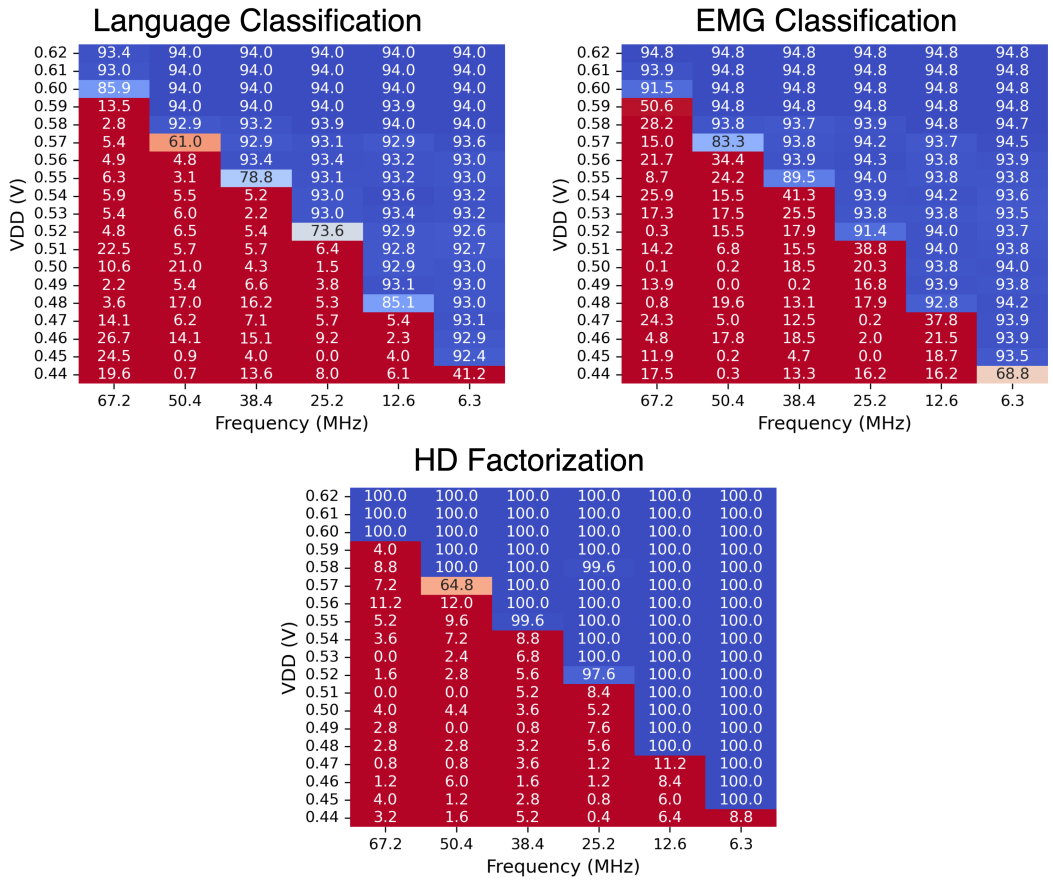

Figure 5.8: Benchmark accuracy for three European language classification [24], EMG gesture classification [58], and HD factorization [48] measured on HPUv2 as a function of core voltage (VDD) and clock frequency. Blue signifies higher accuracy while red signifies lower accuracy.

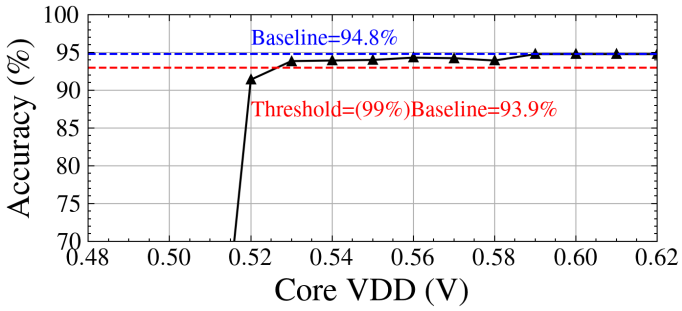

Figure 5.9: Example of choosing an energy-efficient operating point (0.53 V) by threshold accuracy for the EMG classification benchmark at 25.2 MHz.

VDDs between 0.53 V − 0.58 V, we see a small drop in accuracy compared to the baseline. However, since the HPU can tolerate some datapath errors, the algorithm still performs acceptably. The computed energy-efficient operating points for the different benchmarks are nearly identical and match up with the energy-efficient operating points of the associative search kernel. It appears that the associative search kernel limits the low-VDD robustness of the HDC algorithms

Using same process to compute minimum kernel operation energy, we find the minimum energy operating point for each benchmark by measuring the power and latency at the energy-efficient operating points, shown in Figure 5.10. For all benchmarks, we find that the minimum energy operating is at 0.49 V and 12.6 MHz. The resulting minimum energy per query is measured and summarized for all benchmarks in Figure 5.11.

5.5 Discussion

In this section, we compare and analyze the measured data from HPUv1, HPUv2, and existing hardware for HDC. We first focus on the HPU the power and performance of the HPU kernels, before moving on to the three benchmarks characterized on HPUv2.

HPU Kernel Comparison

The HPU kernels, defined in Section 4.4, represent common HDC computations used in most algorithms. In order to evaluate the energy optimizations of the HPUv2, Table 5.6 compares the kernel power, latency, and resulting energy per operation for 1024 dimension vectors between HPUv1 and HPUv2. We also include a comparison to the same HPU kernels run on an NVIDIA GTX 1080 GPU platform. The GPU implementation uses Torchd [59], an optimized package for python based off of PyTorch, and uses the `nvidia-smi` command line interface to monitor and record GPU power. Note that the CA90 kernel is not included

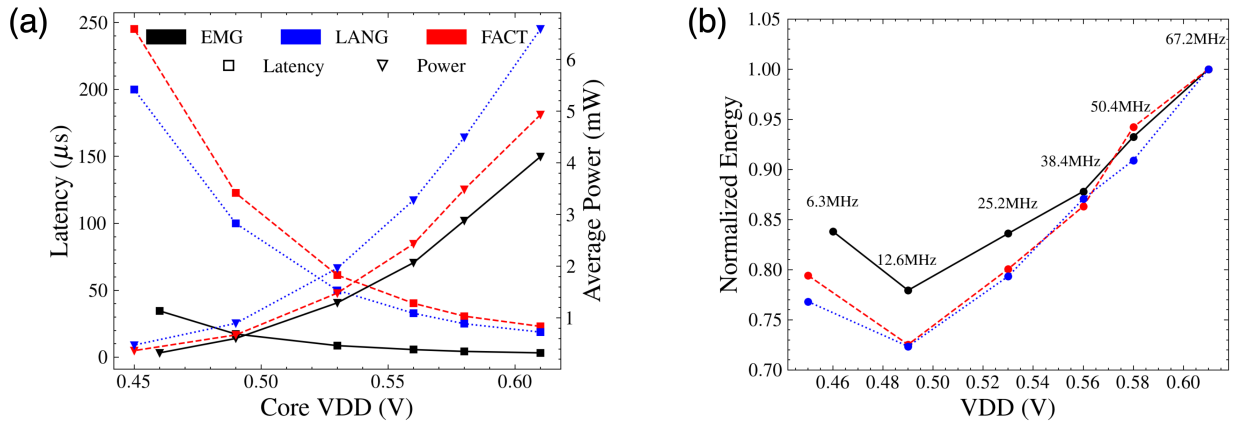

Figure 5.10: Latency and average power of the benchmarks over their respective energy-efficient operating points (a) and resulting normalized energy as a function of core voltage (b). The lowest energy operating point is 0.49 V and 12.6 MHz for all benchmarks.

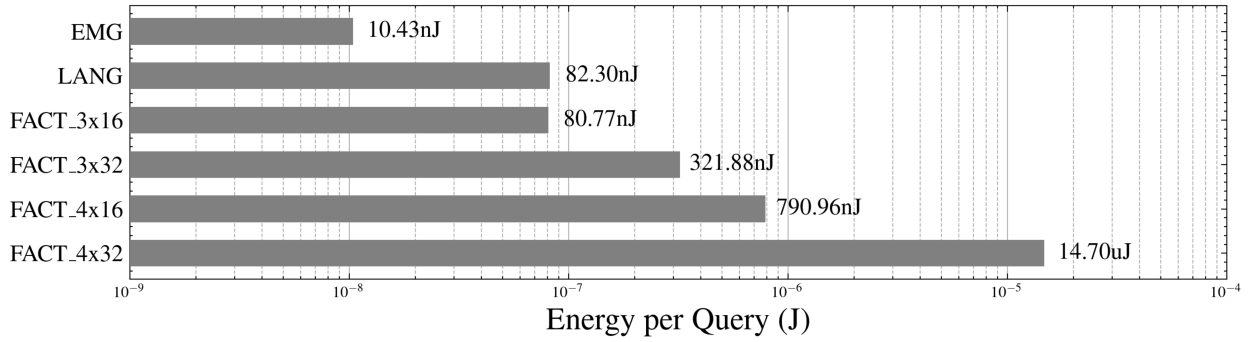

Figure 5.11: Energy per query of all measured HPUv2 benchmarks at the minimum energy operating point of 0.49 V and 12.6 MHz. The factorization benchmarks use a naming convention of $f \times i$ where f is the number of factors and i is the number of items per factor.

as the CA90 operation is built into other computation instructions and thus is not a useful point of comparison.

We find that with the optimized design, HPUv2 is 4.5× more energy efficient than HPUv1 across all kernels with an average energy per operation of 168 pJ compared to 750 pJ. The kernel operations are conducted on a relatively small 1024 dimension, and thus we can expect HPUv2 to present even greater energy-efficiency gains vs. HPUv1 for larger and more complex algorithms. Note that the average power between HPUv1 and HPUv2 are fairly similar, which is surprising as HPUv2 has double the datapath dimension and chip area of HPUv1. However, the kernels run on HPUv2 have a lower minimum-energy operating VDD of 0.45 V, 12.6 MHz compared to 0.57 V, 67.2 MHz for HPUv1. On the other hand, the slower clock frequency should correspondingly increase HPUv2's kernel latency, but with the vastly improved instruction efficiency of HPUv2, we see much lower latency on HPUv2.

When compared to the GPU implementation, both HPUv1 and HPUv2 have orders of magnitude greater energy-efficiency. For instance, HPUv2 has 650000× lower average energy per operation. This is mainly due to the massive difference in processor power, with GPU on the order of 100 W and the HPU on the order of 1 mW. Note that the reported GPU latencies are the best case scenario. Torchhd uses batching to load multiple different vector operation onto the GPU. For the GPU kernels we assume that each test is independent and thus may be computed in parallel. During an actual HDC algorithms, there may be temporal dependence on certain operations in which case we expect the GPU latency to be higher, leading to even worse energy per operation.

Comparison with Previous Hardware

With HPUv2 demonstrating large improvements in energy over a GPU, we now compare against previous hardware for HDC on real benchmarks. The three benchmarks compared are language classification (LANG), EMG gesture classification (EMG), and HD factorization (FACT) as they are the most commonly characterized benchmarks for existing HDC ASICs. The benchmark energy comparisons are found in Table 5.7, where the columns represent different existing processors and the three tested benchmarks are grouped into rows. We first note that the HPU is the first and only processor that can accelerate all three benchmarks shown. Additionally, several of the listed processors are not fully fabricated (marked on the table), and thus part or all of the power data is simulated.

We also note that the HPU is the only processor that uses different HDC vector dimensions for different benchmarks. The dimension for HPUv2 was selected as the smallest folding factor (i.e. multiple of 1024) that achieves the accuracy and/or performance reported in literature. For example, the EMG benchmark only requires 1024 dimensions to achieve 93.9 % accuracy while the FACT benchmark requires 16384 dimensions to reach 98.2 %.

For both classification benchmarks, HPUv2 outperforms the previous classification-only processors. For LANG, HPUv2 operates with 82.3 nJ per query, nearly 4× lower than the next lowest energy per query. The query latency is also much faster at just 122.6 µs compared to 14 ms, a 114× improvement over the next best performer. We see a similar story for EMG,

	GPU			HPUv1			HPUv2		
Kernel	Power	Latency	Energy/op	Power	Latency	Energy/op	Power	Latency	Energy/op
Multiply-Add	106.1 W	747 ns	79 µJ	0.60 mW	2111 ns	1274 pJ	0.83 mW	246 ns	203 pJ
Ngram	63.5 W	2216 ns	141 µJ	0.58 mW	1222 ns	561 pJ	0.76 mW	175 ns	133 pJ
Assoc Search	164.7 W	538 ns	105 µJ	1.07 mW	387 ns	416 pJ	0.80 mW	208 ns	167 pJ
Average			108 µJ			750 pJ			168 pJ

Table 5.6: Comparison of HDC kernel power, latency, and energy between GPU (NVIDIA GTX 1080), HPUv1, and HPUv2. The kernel operations assume 1024 dimension vectors.

		Nature'20 [39]	TCAS-I'21 [55]	ESSCIRC'23 [53]	Nature'23 [54]	HPUv2
	Platform	90nm RRAM 65nm CMOS*	22nm CMOS†	28nm CMOS	14nm PCM 14nm CMOS*	28nm CMOS
	Applications	HD classification	HD classification	HD classification	HD factorization	All binary HDC
LANG [24]	HDC Dim	10000	2048	—	—	2048
	Accuracy	92.8 %	86 %	—	—	93.1 %
	Latency	—	14 ms	—	—	122.6 µs
	Energy/query	430 nJ	322 nJ	—	—	82.3 nJ
EMG [58]	HDC Dim	—	2048	2048	—	1024
	Accuracy	—	95 %	92.9 %	—	93.9 %
	Latency	—	6.78 ms	118 µs	—	17.3 µs
	Energy/query	—	191 nJ	25.6 nJ	—	10.4 nJ
FACT [34]	HDC Dim	—	—	—	256	16384
	Search Space	—	—	—	16 777 216	1 048 576
	Accuracy	—	—	—	99.7 %	98.2 %
	Latency	—	—	—	2.2 ms	16.1 ms
	Energy/query	—	—	—	33.1 µJ	13.7 µJ

* Not fabricated (energy from post-synthesis), † Not fabricated (energy from post-route)

Table 5.7: Comparison of HPUv2 with previous hardware for HDC across three benchmarks. Note many of the previous hardware include simulated power and energy data (marked).

where HPUv2's 10.4 nJ per query and latency of 17.3 µs improve that of previous hardware by 2.4× and 16× respectively.

For the factorization benchmark, HPUv2 does not perform as well as the factorization-only processor [54], at least for large factorization problems. The number of factors f and items per factor i define the factorization search space of i^f, which represents the total possible combinations of items. In general, larger search space computations require correspondingly larger vector dimensions as the algorithm grows larger and more complex. However, [54] uses the naturally-occurring variance in the PCMs to inject randomness into the iterations to improve convergence. As a result, they are able to factorization very large search spaces (16 777 216) using only 256 dimension vectors. Without the variance injection, the factorization algorithm can struggle at larger search spaces. For example, the HPU struggles to reasonably factorize search spaces of more than 1 048 576, even using a vector dimension of 16 384. Even though the search space of [54] is 16× larger, their reported power is only 2.5× larger than HPUv2's reported 13.7 µJ energy per query. Furthermore, their processor is able to compute the factorization 8× faster than HPUv2's 16.1 ms latency.

Nearly all the performance and energy difference can be attributed to the fact that they can achieve the better accuracy with a much smaller vector dimension. It may be possible that even without PCM variance, we can modify the factorization algorithm implemented on the HPU to achieve the same purpose. For example, we can add a few randomly selected unused vector seeds to the scaled accumulation step, which may help the algorithm break out of a convergence loop. Although not yet explored, if such algorithmic adjustments can be made and the vector dimension is correspondingly lowered, HPUv2 can achieve much better performance on the factorization benchmark. We should also note that [54] uses a much more advance process of 14nm compared to 28nm in HPUv2 which contributes to their better performance and energy measurements.

Finally, we expect HPUv2 to be more competitive with factorization-only processor for small search spaces. Currently, practical applications of HD factorization only operate over a fairly small search space. For example, the visual decomposition example tested in [54] only uses a search space of 1200. On the HPUv2, this can easily fit into the `FACT_3_16` benchmark from Figure 5.11 which only requires 80.8 nJ per query. Unfortunately, the factorization-only processor does not separately report energy for this use-case.

Overall, HPUv2 has successfully achieved the three main goals of that it aimed to solve. First, it has demonstrated its ability to accelerate all known binary HDC algorithms through three distinct benchmarks. Second, Each benchmark uses a different HDC vector dimension, which is adapted to the size and complexity of the target application. Third, HPUv2 has shown promising energy-efficiency. For the kernel tests, HPUv2 outperforms GPU energy per operation by 6 orders of magnitude with an average of 168 pJ per operation. HPUv2 outperforms previous HD classification processors in both energy and performance, and remains competitive with existing HD factorization processors.

Chapter 6

Conclusions

6.1 Overview

The proliferation of smart devices and sensors is pushing computation and decision-making closer to endpoint devices. This is especially problematic for machine learning and artificial intelligence applications which are traditionally power-hungry and most often offload complicated and expensive algorithms onto a centralized compute platform or datacenter. As a result, there is an increasing demand for energy-efficient algorithms and corresponding hardware platforms to fit this shift in computing paradigms.

This dissertation explores the design of a general-purpose processor to accelerate Hyper-Dimensional Computing (HDC) algorithms for energy-efficient learning on the edge. The first two chapters introduce HDC and why it is a good fit for certain low-power AI applications. The third chapter outlines the goals and overall system architecture of the Hyper-Dimensional Processing Unit (HPU) and how it efficiently accelerates all binary HDC algorithms. The fourth and fifth chapters describe the physical design, silicon implementation, and power and performance results of HPUv1 and HPUv2. This final chapter summarizes the key results and contributions of the dissertation, provides insight on possible future directions of the HPU project, and concludes with some final thoughts.

6.2 Summary of Results

1. **Robot Navigation Algorithm using Recall of Reactive Behavior**

   Although an HDC algorithm for robot learning through the Recall of Reactive Behavior (RORB) has been proposed, there had not yet been a detailed implementation or review of the performance and limitations of the algorithm. We defined an environment with a robot in a two-dimensional grid with randomly placed obstacles and a target location. We then applied the RORB algorithm with a modified encoding scheme to train the robot to move to the target while avoiding obstacles. The modified RORB algorithm achieves up to 88.57 % success rate in new previously-unseen random environments.

2. **Development of the HPU Architecture**

   The HPU is the first general-purpose processor capable of accelerating all binary HDC algorithms. Previous hardware developed for HDC are limited to a specific application or type of application (e.g. classification, factorization) and many can only accelerate certain operations/kernels and rely on off-chip CPU/FPGA for some HDC operations. To overcome these limitations, we designed the HPU ISA so that it can perform any HD operation on any vector in any order. Furthermore, the HPU has built in dimensionality scaling using a combination of vector folding and CA90. As a result, the HPU is the first accelerator for HDC that can change the dimension of the HD vectors at runtime to fit the application at hand. The HPU is a completely digital design written in SystemVerilog and only using standard cells and SRAMs, which allows it to be easily portable over different technologies.

3. **HPUv1: Silicon Implementation and Verification of the HPU architecture**

   While several accelerators for HDC have been proposed, only a few have been implemented, verified, and tested in silicon. HPUv1 was fabricated in TSMC 28nm CMOS as a small-scale proof of concept of the HPU architecture. We set up four kernel tests which cover all operations used in HDC algorithms. HPUv1 passes all three encoding kernels, for which power and latency were measured at the lowest energy operation points. Due to an error in the implementation of the vector similarity calculator, HPUv1 cannot be verified over full-scale benchmarks. However, measurements from HPUv1 can still serve as a reliable baseline for the unoptimized HPU architecture.

4. **HPUv2: Silicon Implementation and Characterization of an optimized HPU architecture**

   The previous testing and analysis of HPUv1 highlighted the memory-datapath interface as the main bottleneck in both power and performance. With HPUv2, we made several architectural optimizations aimed at improving memory read latency and standby memory power. HPUv2 was also fabricated in TSMC 28nm CMOS and successfully verified across all four kernel tests with an average energy per kernel operation of 168 pJ. We also tested and measured HPUv2 for three different applications: Language recognition (82.3 nJ per query), EMG gesture recognition (10.4 nJ per query), and factorization problems over various size search spaces (80.8 nJ − 14.7 µJ per query). HPUv2 is the first ASIC implementation to accelerate HDC algorithms of different types (classification and factorization). Furthermore, HPUv2 is also the first processor to compute the entire factorization algorithm on-chip. For each application, HPUv2 remains competitive in both benchmark accuracy and energy to previous application-specific hardware.

6.3 Future Work

The design and implementation of HPUv2 focused primarily around improving the memory-datapath interface. There are still many areas to explore in both architecture optimization and silicon implementation in order to further improve the capabilities and performance of the HPU. Furthermore, with the constant evolution and progress of HDC algorithms as well as emerging device and memory technologies, there will always be adjustments and optimizations to suit a general-purpose and energy-efficient processor for HDC. Based on our experience designing and optimizing the HPU, we present several directions for further research:

1. **Multi-VDD HPU implementation**

   Our kernel tests for both HPUv1 and HPUv2 show that for the same operating frequency, different kernels can fail at different VDDs. In particular, the HDC encoding operations are capable of operating at $\sim 50\,\text{mV}$ lower than the distance compute and search operations. HDC algorithms are robust to random bit-level errors in computed vectors, but not to errors in the processor control pipeline, distance compute, or search. Thus, we propose an HPU implementation with three power rails: one for the SRAMs, one for error-resilient logic (encoding datapath, pseudo-random generators), and one for error-vulnerable logic (control, distance compute, search). As a result, we can isolate the error-resilient portions of the HPU to study and take advantage of HDC's properties of robustness. Since these areas are expected to have the highest switching activity for most HDC algorithms, the multi-VDD implementation can save significant power.

2. **On-chip program memory**

   HPUv1 and HPUv2 require an instruction input every cycle. Consequently, programs are currently stored off-chip on the testing FPGA. However, with small changes to the HPU ISA and a small dedicated program memory, all instructions can be stored on-chip. This change primarily allows conditional instructions to be executed on the HPU, which may be useful in certain HDC applications. For example, conditional instructions can be used to check for early convergence in factorization algorithms, or used for adaptive training or retraining [40] in supervised learning tasks. Programs for the HPU are also extremely cyclical and, with the addition of jump/loop instructions, can be mapped onto a small amount of program lines.

3. **Scratchpad memory**

   HPUv2 features separate SRAMs for the item vector seeds, CA90 cache, and partitioned SRAMs for general vector storage. Currently, the general vector storage only stores binary vectors, either intermediate computation vectors or fully-encoded vectors. Transitioning the general vector storage into a scratchpad memory with variable output port width would increase the flexibility and efficiency of the HPU in certain

cases. The scratchpad would have the ability to store and retrieve integers from the distance registers as well as integer vectors from the accumulators. For example in training a classification task, HPUv2 must finish training a single class before moving on to the next. The new scratchpad configuration would support training of multiple classes at the same time, which is useful in applications requiring online learning.

4. **Architecture design space exploration**

   Although we performed a rudimentary analysis of varying architecture parameters for HPUv2, there is still a large design space yet to be explored. In addition to the base datapath dimension and number of distance compute lanes, the HPU features many configurable parameters such as integer size, memory size, and number of memory partitions. The exploration should also consider the target operating point, which will likely be much lower VDD compared to typical library characterization.

5. **HPU compiler**

   Appendix A touches on the python emulator and compiler we used to generate programs for the HPU. These programs were built using python functions and require large amounts of manual coding to keep track of vector to symbol mappings and addresses. Furthermore, there are multiple ways to implement the same computation on the HPU and it is possible that the benchmark programs can be further optimized and shortened. Moving forward, the development of a proper HPU compiler is critical for users to efficiently create new programs for the HPU.

6.4 Looking Forward

When energy-efficiency is critical, well-designed application specific processors can outperform a general-purpose processor like the HPU. However, a versatile and programmable processor for HDC may play a significant role in a generalized learning framework.

With the recent explosion of AI, ML is asked to solve problems of growing complexity. Different ML frameworks (CNNs, transformers, HDC, etc.) have complementary strengths, weakness, and applications they excel at. Consequently, complex tasks may benefit from a multi-layered algorithm that involves several ML algorithms on a shared platform. Such a platform requires general-purpose compute units that can work together to solve diverse problems.

For example, one of the largest challenges for HDC is finding intelligent ways to map input features of an algorithm onto HD vectors [19]. This is especially true for 2D images, where HDC performs MNIST digit classification with fairly low accuracy even compared to small CNNs [60]. However, recent studies have successfully fused shallow CNNs trained to distinguish image features which are then mapped onto HD item vectors for cognitive reasoning in HDC [36], [61], [62].

These hybrid applications point to the advantage of a generalized low-power AI SoC composed of an HPU, neural network accelerator, and CPU to arbitrate data and compute between the coprocessors. While many low-power general-purpose neural network accelerators have been proposed, implemented, and improved upon [63], [64], [65], [66], [67], low-power general-purpose HDC accelerators have yet to be studied and optimized. The HPU is the first step towards that direction.

Bibliography

[1] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," en, *Science*, vol. 349, no. 6245, pp. 255–260, Jul. 2015.

[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services & Cloud Computing and Scientific Applications — Big Data, Scalable Analytics, and Beyond, vol. 29, no. 7, pp. 1645–1660, Sep. 2013.

[3] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: State-of-the-art and research challenges," en, *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, May 2010.

[4] T. L. Duc, R. G. Leiva, P. Casari, and P.-O. Östberg, "Machine Learning Methods for Reliable Resource Provisioning in Edge-Cloud Computing: A Survey," *ACM Comput. Surv.*, vol. 52, no. 5, 94:1–94:39, Sep. 2019.

[5] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[6] H. Li, K. Ota, and M. Dong, "Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing," *IEEE Network*, vol. 32, no. 1, pp. 96–101, Jan. 2018.

[7] A. V. Dastjerdi and R. Buyya, "Fog Computing: Helping the Internet of Things Realize Its Potential," *Computer*, vol. 49, no. 8, pp. 112–116, Aug. 2016.

[8] M. Merenda, C. Porcaro, and D. Iero, "Edge Machine Learning for AI-Enabled IoT Devices: A Review," en, *Sensors*, vol. 20, no. 9, p. 2533, Jan. 2020.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, vol. 25, Curran Associates, Inc., 2012.

[10] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb. 2014, pp. 10–14.

[11] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," in *2017 IEEE Custom Integrated Circuits Conference (CICC)*, Apr. 2017, pp. 1–8.

[12] S. Han, H. Mao, and W. J. Dally, *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*, en, Feb. 2016.

[13] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: An Energy-Efficient Deep Neural Network Accelerator With Fully Variable Weight Bit Precision," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, Jan. 2019.

[14] V. Jain, S. Giraldo, J. D. Roose, L. Mei, B. Boons, and M. Verhelst, "TinyVers: A Tiny Versatile System-on-Chip With State-Retentive eMRAM for ML Inference at the Extreme Edge," *IEEE Journal of Solid-State Circuits*, vol. 58, no. 8, pp. 2360–2371, Aug. 2023.

[15] J. M. Rabaey, *Low Power Design Essentials* (Series on Integrated Circuits and Systems), eng, 1. Ed. Berlin: Springer US, 2009, ISBN: 978-0-387-71713-5.

[16] P. Kanerva, "Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors," en, *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, Jun. 2009.

[17] A. Rahimi, P. Kanerva, L. Benini, and J. M. Rabaey, "Efficient Biosignal Processing Using Hyperdimensional Computing: Network Templates for Combined Learning and Classification of ExG Signals," *Proceedings of the IEEE*, vol. 107, no. 1, pp. 123–143, Jan. 2019.

[18] A. Rahimi, S. Datta, D. Kleyko, *et al.*, "High-Dimensional Computing as a Nanoscalable Paradigm," en, *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 9, pp. 2508–2521, Sep. 2017.

[19] D. Kleyko, D. A. Rachkovskij, E. Osipov, and A. Rahimi, "A Survey on Hyperdimensional Computing aka Vector Symbolic Architectures, Part I: Models and Data Transformations," *ACM Comput. Surv.*, vol. 55, no. 6, 130:1–130:40, Dec. 2022.

[20] E. P. Frady, D. Kleyko, and F. T. Sommer, "A Theory of Sequence Indexing and Working Memory in Recurrent Neural Networks," en, *Neural Computation*, vol. 30, no. 6, pp. 1449–1513, Jun. 2018.

[21] D. Kleyko, A. Rosato, E. P. Frady, M. Panella, and F. T. Sommer, "Perceptron Theory Can Predict the Accuracy of Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 7, pp. 9885–9899, Jul. 2024.

[22] K. Schlegel, P. Neubert, and P. Protzel, "A comparison of vector symbolic architectures," en, *Artificial Intelligence Review*, vol. 55, no. 6, pp. 4523–4555, Aug. 2022.

[23] A. Joshi, J. T. Halseth, and P. Kanerva, "Language Geometry Using Random Indexing," en, in *Quantum Interaction*, J. A. De Barros, B. Coecke, and E. Pothos, Eds., vol. 10106, Cham: Springer International Publishing, 2017, pp. 265–274, ISBN: 978-3-319-52288-3 978-3-319-52289-0.

[24] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A Robust and Energy-Efficient Classifier Using Brain-Inspired Hyperdimensional Computing," en, in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, San Francisco Airport CA USA: ACM, Aug. 2016, pp. 64–69, ISBN: 978-1-4503-4185-1.

[25] S. Benatti, F. Montagna, V. Kartsch, A. Rahimi, D. Rossi, and L. Benini, "Online Learning and Classification of EMG-Based Gestures on a Parallel Ultra-Low Power Platform Using Hyperdimensional Computing," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 3, pp. 516–528, Jun. 2019.

[26] A. Menon, D. Sun, S. Sabouri, *et al.*, "A Highly Energy-Efficient Hyperdimensional Computing Processor for Biosignal Classification," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 16, no. 4, pp. 524–534, Aug. 2022.

[27] 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)*, Nov. 2017, pp. 1–8.

[28] A. Burrello, L. Cavigelli, K. Schindler, L. Benini, and A. Rahimi, "Laelaps: An Energy-Efficient Seizure Detection Algorithm from Long-term Human iEEG Recordings without False Alarms," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2019, pp. 752–757.

[29] 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)*, Mar. 2018, pp. 271–274.

[30] D. Kleyko, D. Rachkovskij, E. Osipov, and A. Rahimi, "A Survey on Hyperdimensional Computing aka Vector Symbolic Architectures, Part II: Applications, Cognitive Models, and Challenges," *ACM Comput. Surv.*, vol. 55, no. 9, 175:1–175:52, Jan. 2023.

[31] P. Neubert, S. Schubert, and P. Protzel, "An Introduction to Hyperdimensional Computing for Robotics," en, *KI - Künstliche Intelligenz*, vol. 33, no. 4, pp. 319–330, Dec. 2019.

[32] P. Neubert, S. Schubert, and P. Protzel, "Learning Vector Symbolic Architectures for Reactive Robot Behaviours," en,

[33] E. Osipov, D. Kleyko, and A. Legalov, "Associative synthesis of finite state automata model of a controlled object with hyperdimensional computing," in *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, Oct. 2017, pp. 3276–3281.

[34] E. P. Frady, S. J. Kent, B. A. Olshausen, and F. T. Sommer, "Resonator Networks, 1: An Efficient Solution for Factoring High-Dimensional, Distributed Representations of Data Structures," en, *Neural Computation*, vol. 32, no. 12, pp. 2311–2331, Dec. 2020.

[35] I. Nunes, M. Heddes, T. Givargis, A. Nicolau, and A. Veidenbaum, "GraphHD: Efficient graph classification using hyperdimensional computing," en, in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Antwerp, Belgium: IEEE, Mar. 2022, pp. 1485–1490, ISBN: 978-3-9819263-6-1.

[36] P. Neubert, S. Schubert, K. Schlegel, and P. Protzel, "Vector Semantic Representations as Descriptors for Visual Place Recognition," en, in *Robotics: Science and Systems XVII*, Robotics: Science and Systems Foundation, Jul. 2021, ISBN: 978-0-9923747-7-8.

[37] G. Karunaratne, M. Schmuck, M. Le Gallo, *et al.*, "Robust high-dimensional memory-augmented neural networks," en, *Nature Communications*, vol. 12, no. 1, p. 2468, Apr. 2021.

[38] H. Li, T. F. Wu, S. Mitra, and H.-S. P. Wong, "Device-architecture co-design for hyperdimensional computing with 3d vertical resistive switching random access memory (3D VRRAM)," in *2017 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*, Apr. 2017, pp. 1–2.

[39] G. Karunaratne, M. Le Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian, "In-memory hyperdimensional computing," en, *Nature Electronics*, vol. 3, no. 6, pp. 327–337, Jun. 2020.

[40] A. Hernández-Cano, N. Matsumoto, E. Ping, and M. Imani, "OnlineHD: Robust, Efficient, and Single-Pass Online Learning Using Hyperdimensional System," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Feb. 2021, pp. 56–61.

[41] M. Véstias and H. Neto, "Trends of CPU, GPU and FPGA for high-performance computing," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2014, pp. 1–6.

[42] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, "F5-HD: Fast Flexible FPGA-based Framework for Refreshing Hyperdimensional Computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, New York, NY, USA: Association for Computing Machinery, Feb. 2019, pp. 53–62, ISBN: 978-1-4503-6137-8.

[43] M. Imani, Z. Zou, S. Bosch, *et al.*, "Revisiting HyperDimensional Learning for FPGA and Low-Power Architectures," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2021, pp. 221–234.

[44] M. Imani, S. Bosch, S. Datta, *et al.*, "QuantHD: A Quantization Framework for Hyperdimensional Computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2268–2278, Oct. 2020.

[45] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec. 2016, pp. 77–84.

[46] S. Datta, R. A. G. Antonio, A. R. S. Ison, and J. M. Rabaey, "A Programmable Hyper-Dimensional Processor Architecture for Human-Centric IoT," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 439–452, Sep. 2019.

[47] *Tiny-HD: Ultra-Efficient Hyperdimensional Computing Engine for IoT Applications — IEEE Conference Publication — IEEE Xplore*.

[48] S. J. Kent, E. P. Frady, F. T. Sommer, and B. A. Olshausen, "Resonator Networks, 2: Factorization Performance and Capacity Compared to Optimization-Based Methods," *Neural Computation*, vol. 32, no. 12, pp. 2332–2388, Dec. 2020.

[49] N. Sünderhauf, O. Brock, W. Scheirer, *et al.*, "The limits and potentials of deep learning for robotics," en, *The International Journal of Robotics Research*, Apr. 2018.

[50] A. Menon, A. Natarajan, L. I. G. Olascoaga, Y. Kim, B. Benedict, and J. M. Rabaey, "On the Role of Hyperdimensional Computing for Behavioral Prioritization in Reactive Robot Navigation Tasks," in *2022 International Conference on Robotics and Automation (ICRA)*, May 2022, pp. 7335–7341.

[51] M. Ibrahim, Y. Kim, and J. M. Rabaey, "Efficient Design of a Hyperdimensional Processing Unit for Multi-Layer Cognition," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2024, pp. 1–6.

[52] T. F. Wu, H. Li, P.-C. Huang, *et al.*, "Brain-inspired computing exploiting carbon nanotube FETs and resistive RAM: Hyperdimensional computing case study," in *2018 IEEE International Solid-State Circuits Conference - (ISSCC)*, Feb. 2018, pp. 492–494.

[53] S. Datta, B. Richards, H. Liew, Y. Kim, D. Sun, and J. M. Rabaey, "HDBinaryCore: A 28nm 2048-bit Hyper-Dimensional biosignal classifier achieving 25 nJ/prediction for EMG hand-gesture recognition," in *ESSCIRC 2023- IEEE 49th European Solid State Circuits Conference (ESSCIRC)*, Sep. 2023, pp. 229–232.

[54] J. Langenegger, G. Karunaratne, M. Hersche, L. Benini, A. Sebastian, and A. Rahimi, "In-memory factorization of holographic perceptual representations," en, *Nature Nanotechnology*, vol. 18, no. 5, pp. 479–485, May 2023.

[55] 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, Oct. 2021.

[56] D. Kleyko, E. P. Frady, and F. T. Sommer, "Cellular Automata Can Reduce Memory Requirements of Collective-State Computing," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 6, pp. 2701–2713, Jun. 2022.

[57] E.-J. Chang, A. Rahimi, L. Benini, and A.-Y. A. Wu, "Hyperdimensional Computing-based Multimodality Emotion Recognition with Physiological Signals," in *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, Mar. 2019, pp. 137–141.

[58] A. Moin, A. Zhou, A. Rahimi, *et al.*, "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)*, May 2018, pp. 1–5.

[59] M. Heddes, I. Nunes, P. Vergés, *et al.*, "Torchhd: An Open Source Python Library to Support Research on Hyperdimensional Computing and Vector Symbolic Architectures," *Journal of Machine Learning Research*, vol. 24, no. 255, pp. 1–10, 2023.

[60] E. Hassan, Y. Halawani, B. Mohammad, and H. Saleh, "Hyper-Dimensional Computing Challenges and Opportunities for AI Applications," *IEEE Access*, vol. 10, pp. 97 651–97 664, 2022.

[61] M. Hersche, M. Zeqiri, L. Benini, A. Sebastian, and A. Rahimi, "A neuro-vector-symbolic architecture for solving Raven's progressive matrices," en, *Nature Machine Intelligence*, vol. 5, no. 4, pp. 363–375, Apr. 2023.

[62] P. Sutor, D. Yuan, D. Summers-Stay, C. Fermuller, and Y. Aloimonos, *Gluing Neural Networks Symbolically Through Hyperdimensional Computing*, en, May 2022.

[63] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[64] A. Parashar, M. Rhu, A. Mukkara, *et al.*, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17, New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 27–40, ISBN: 978-1-4503-4892-8.

[65] S. Yin, P. Ouyang, S. Tang, *et al.*, "A High Energy Efficient Reconfigurable Hybrid Neural Network Processor for Deep Learning Applications," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 4, pp. 968–982, Apr. 2018.

[66] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, "SNAP: An Efficient Sparse Neural Acceleration Processor for Unstructured Sparse Deep Neural Network Inference," *IEEE Journal of Solid-State Circuits*, vol. 56, no. 2, pp. 636–647, Feb. 2021.

[67] S. Ryu, H. Kim, W. Yi, *et al.*, "BitBlade: Energy-Efficient Variable Bit-Precision Hardware Accelerator for Quantized Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 6, pp. 1924–1935, Jun. 2022.

Appendix A

Testing and Measurement Methodology

A.1 Python Emulator and Compiler

We have constructed python emulators for both HPUv1 and HPUv2 which emulate the memory contents and datapath and control registers of the HPU. Although not cycle accurate, the emulators can read input HPU instructions and update its registers and memories correspondingly. The python emulators have two main purposes. First, the emulator outputs are used for verification of the physical ASICs. For a given test program, it is run on the emulator and all emulator outputs are recorded and compared to the outputs from the ASIC.

Second, the python emulator is highly useful in developing programs for the HPU. The HPU benchmark applications are developed by hand by first assigning and keeping track of memory addresses that map to the various vectors required by the application. We also defined several HPU macros, i.e. groupings of commonly used HPU instructions, as python functions that call individual instructions in the emulator. Although the process is slow and tedious, the emulator allows us to verify the accuracy of the benchmark application and tune certain settings such as the folding factor before testing on the physical hardware.

A.2 Test Setup

The measurement and validation of HPUv1 and HPUv2 is performed on a custom designed Printed Circuit Board (PCB) shown in Figure A.1. The PCB houses and connects three primary components:

1. **The HPU package**
   Both HPUv1 and HPUv2 are wirebonded to a 13×13 Ceramic Pin Grid Array (CPGA) package. The test PCB contains sockets that interface with the package pins.

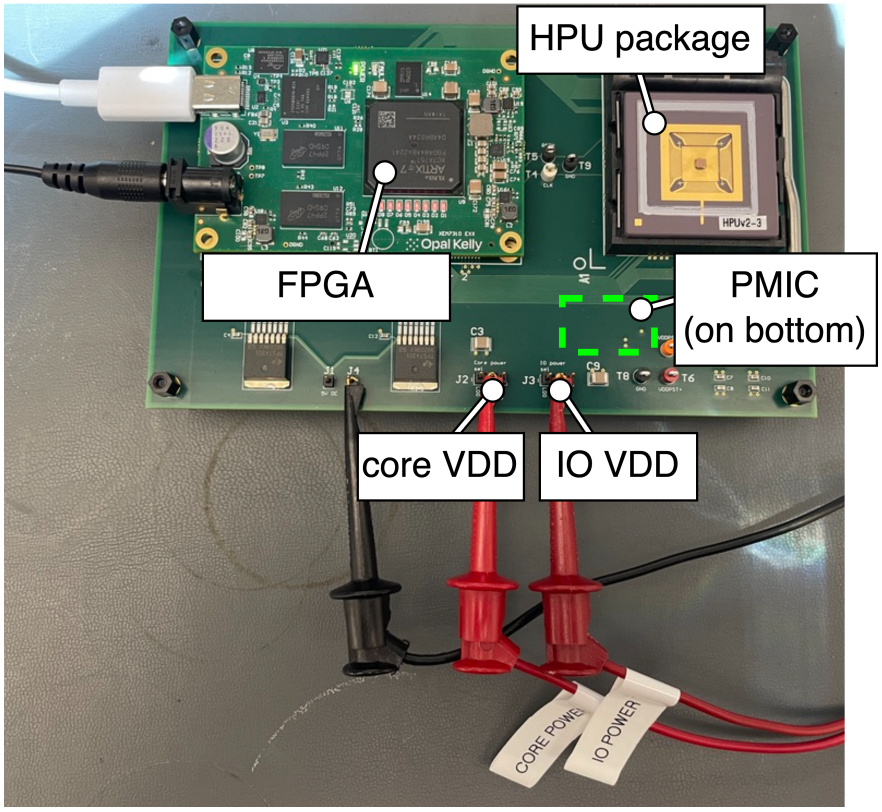

Figure A.1: Custom PCB with labeled components.

2. **Opal Kelly XEM7310 FPGA development board**
   The programming and measurement of the HPU is handled by a Field-Programmable Gate Array (FPGA). The Opal Kelly XEM7310 houses a Xilinx Artix-7 XC7A75T FPGA as well as necessary peripheral circuitry such as a USB interface, clock crystal, and voltage regulators.

3. **TI INA229 PMIC**
   The HPU power is measured by a Texas Instrument INA229 Power Management Integrated Circuit (PMIC) which can measure power with up to 0.5 % accuracy by detecting the voltage drop over a shunt resistor. The INA229 can be configured and sampled using a 5 MHz SPI interface.

The test PCBs for HPUv1 and HPUv2 are nearly identical, with the only differences being the connection traces between the HPU data and control pins and the FPGA. The FPGA handles four main testing responsibilities. First, the FPGA clock generators are used to supply the HPU clock so that the two chips can synchronize data communication. Second, the FPGA provides instruction to the HPU every cycle. Before program execution, all program

instructions are loaded from a computer to the large Block Random Access Memory (BRAM) on the FPGA. While idle, the FPGA asserts the `nop` instruction to the HPU. When triggered by the computer, the FPGA begins program operation, reading out the stored instructions line by line and outputting them to the HPU instruction pins at every clock cycle. Once the program finished, the FPGA again asserts the `nop` instruction. The program BRAM is sized at 524KB which can hold a maximum of 65 536 instructions.

Third, the FPGA handles all data movement required to initialize the HPU memories as well as to read out vectors, integers, and associative search results from the HPU. When the FPGA reads a data load or store instruction, it temporarily asserts a `nop` instruction while it prepares to load or read the data. Data read and writes from the FPGA to the HPU are interfaced to a separate BRAM just for IO. The FPGA input and output include shift registers to deal with vector movement. While a vector is being shifted in or out, the FPGA will stall the HPU accordingly. In other words, stall instructions required for data reading or loading does not need to be built in to the program.

Finally, when the FPGA is running a program, it will query the PMIC on the test PCB and store the measured power data in a separate power BRAM. Once the program is complete, data from the power BRAM and IO data BRAM can be transferred back to the computer. All communication with the computer is handled using a python front end.

The HPU has two necessary power rails to set the core VDD and IO VDD respectively. The power rails are connected an external DC power supply through the power pins available on the PCB. The IO VDD is set to 1.3 V while the core VDD can be varied. Although each rail has a PMIC, all reported HPU power and energy numbers are from the core VDD rail. The entire test system diagram is shown in Figure A.2

Although the PCB was designed to minimize the lengths of the traces that connect the HPU clock, data, and control pins to the FPGA, the trace parasitics limit the maximum frequency of the HPU. While both HPUv1 and HPUv2 were designed for a maximum frequency of 200 MHz, in practice we were only able to achieve a maximum frequency of 108 MHz using the test PCB. Fortunately, the main goal of the power characterization is to minimize measured benchmark energy, which as shown by our measured data, occurs at relatively low-frequency operating points.

A.3 Measurement Methodology

All programs to be run on the HPU are split into three subprograms. The first subprogram initializes the item memories and sets the proper configuration registers of the benchmark. The second subprogram handles all benchmark computation. The third subprogram is optional and used for verification purposes, where HPU memory contents and search results can be read back to the FPGA. The subprograms are separated so that they can be run at different core VDDs. Notably, at low core VDD, even though the core logic, memory, etc. are all operating normally, the HPU cannot properly drive the IO cells and thus output any signals to the FPGA. As a result, the first and third subprograms are always run with core

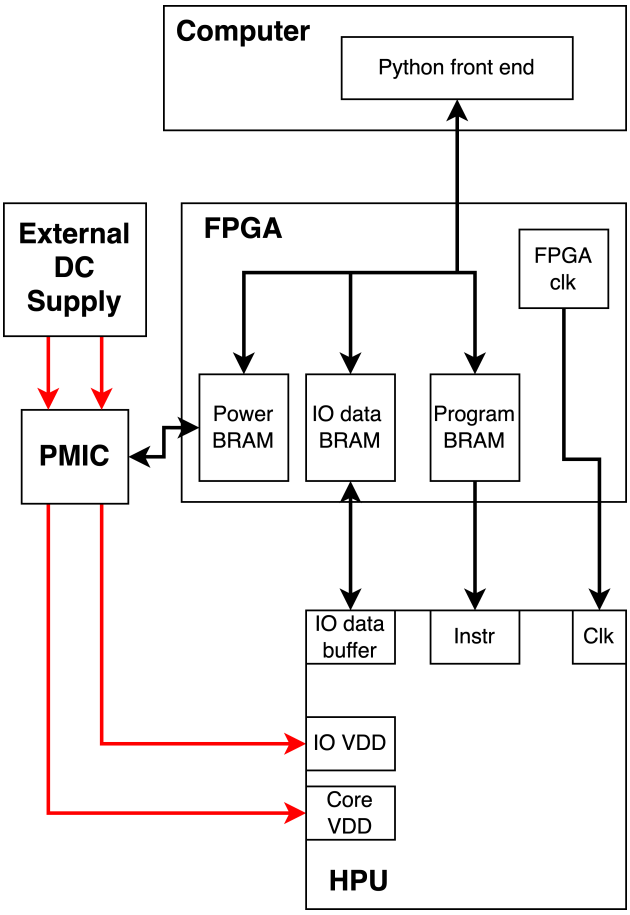

Figure A.2: System diagram of the testing setup.

VDD set to the nominal 0.9 V and only lowered to the energy-efficient operating points for the second main subprogram. All reported benchmark power is measured only during this second subprogram.

The PMIC computes and updates its power measurement at a frequency of 20 kHz. During program operation, the FPGA will then sample the PMIC output at 49 kHz. For the power measurement of each benchmark, we prepare a maximum length program consisting of multiple benchmark queries, called a batch. The reported power is an average over 100 batches. An example PMIC readout is shown for the EMG benchmark run on HPUv2 in Figure A.3.

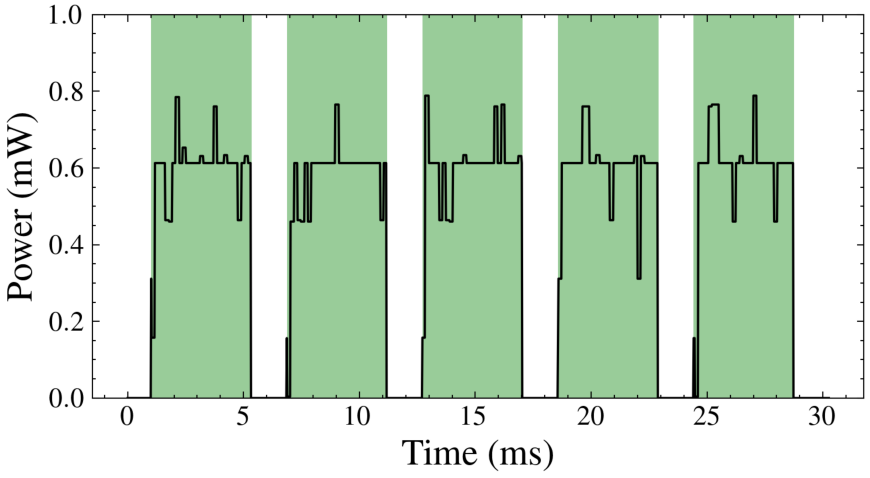

Figure A.3: Power trace measured by PMIC for 5 batches of the EMG benchmark with each batch containing 250 queries. Power measured on HPUv2 operating at 0.49 V, 12.6 MHz.