GenieHD: Efficient DNA Pattern Matching Accelerator Using Hyperdimensional Computing

Yeseong Kim, Mohsen Imani, Niema Moshiri, and Tajana Rosing
University of California San Diego
{yek048, moimani, a1moshir, tajana}@ucsd.edu

***Abstract*—DNA pattern matching is widely applied in many bioinformatics applications. The increasing volume of the DNA data exacerbates the runtime and power consumption to discover DNA patterns. In this paper, we propose a hardware-software co-design, called GenieHD, which efficiently parallelizes the DNA pattern matching task. We exploit brain-inspired hyperdimensional (HD) computing which mimics pattern-based computations in human memory. We transform inherent sequential processes of the DNA pattern matching to highly-parallelizable computation tasks using HD computing. The proposed technique first encodes the whole genome sequence and target DNA pattern to high-dimensional vectors. Once encoded, a light-weight operation on the high-dimensional vectors can identify if the target pattern exists in the whole sequence. We also design an accelerator architecture which effectively parallelizes the HD-based DNA pattern matching while significantly reducing the number of memory accesses. The architecture can be implemented on various parallel computing platforms to meet target system requirements, e.g., FPGA for low-power devices and ASIC for high-performance systems. We evaluate GenieHD on practical large-size DNA datasets such as human and Escherichia Coli genomes. Our evaluation shows that GenieHD significantly accelerates the DNA matching procedure, e.g., 44.4× speedup and 54.1× higher energy efficiency as compared to a state-of-the-art FPGA-based design.**

***Index Terms*—DNA sequencing, Hyperdimensional computing, Pattern matching**

I. Introduction

DNA pattern matching is an essential technique in many applications of bioinformatics. In general, a DNA sequence is represented by a string consisting of four nucleotide characters, *A, C, G,* and *T*. The pattern matching problem is to examine the occurrence of a given *query* string in a *reference* string. For example, the technique can discover possible diseases by identifying which reads (short strings) match a reference human genome consisting of 100 millions of DNA bases [1]. The pattern matching is also an important ingredient of many DNA alignment techniques. BLAST, one of the best DNA local alignment search tools [2], uses the pattern matching as a key step of their processing pipeline to find representative k-mers before running subsequent alignment steps.

Despite of the importance, the efficient acceleration of the DNA pattern matching is still an open question. Although prior researchers have developed acceleration systems on parallel computing platforms, e.g., GPU [3] and FPGA [4], they offer only limited improvements. The primary reason is that existing pattern matching algorithms they relied on, e.g., Boyer-Moore (BM) and Knuth-Morris-Pratt (KMP) algorithms [1], are at heart sequential processes. Their acceleration strategies parallelize the workloads by either scheduling multiple DNA searching tasks or streaming long-length DNA sequences, consequently resulting in high memory requirements and runtime. In this context, the pattern matching problem should be revisited not only to accelerate the existing algorithms on the parallel computing platforms, but also to redesign a hardware-friendly algorithm itself.

In this paper, we propose a novel hardware-software codesign of GenieHD (Genome identity extractor using hyperdimensional computing), which includes a new pattern matching algorithm and the accelerator design. The proposed design is based on brain-inspired hyperdimensional (HD) computing [5]. HD computing is a computing method which mimics the human memory efficient in pattern-oriented computations. In HD computing, we first encode raw data to patterns in a high-dimensional space, i.e., high-dimensional vectors, also called *hypervectors*. HD computing can then imitate essential functionalities of the human memory with hypervector operations. For example, with the hypervector addition, a single hypervector can effectively combine multiple patterns. We can also check the similarity of different patterns efficiently by computing the vector distances. Since the HD operations are expressed with simple arithmetic computations which are often dimension-independent, parallel computing platforms can significantly accelerate HD-based algorithms in a scalable way.

Based on HD computing, GenieHD transforms the inherent sequential processes of the pattern matching task to highly-parallelizable computations. The followings summarize the contributions shown in this paper:

1) We propose a novel hardware-friendly pattern matching algorithm based on HD computing. GenieHD encodes DNA sequences to hypervectors and discover multiple patterns with a light-weight HD operation. Besides, we can reuse the encoded hypervectors to query many DNA sequences newly sampled which are common in practice.

2) We show an acceleration architecture to execute the proposed algorithm efficiently on general parallel computing platforms. The proposed design significantly reduces the number of memory accesses to process the HD operations, while fully utilizing the available parallel computing resources. We also present how to implement the proposed acceleration architecture on the three parallel computing platforms, GPGPU, FPGA, and ASIC.

3) We evaluate GenieHD with practical datasets, human and Escherichia Coli (E. coli) genome sequences. The experimental results show that GenieHD significantly accelerates the DNA matching algorithm, e.g., 44.4× speedup and 54.1× higher energy efficiency when comparing our FPGA-based design to a state-of-the-art FPGA-based design. As compared to an existing GPU-based implementation, our ASIC design which has the similar die size outperforms the performance and energy efficiency by 122× and 707×. We also show that the power consumption can be further saved by 50% by allowing minimal accuracy loss of 1%.

II. Related Work

Hyperdimensional Computing HD computing is originated from a human memory model, called sparse distributed memory developed in neuroscience [5]. Recently, computer scientists recapped the memory model as a cognitive, pattern-oriented computing method. For example, prior researchers showed that the HD computing-based classifier is effective for diverse applications, e.g., text classification, multimodal sensor fusion, speech recognition, and human activity classification [6]–[10]. The work in [11]

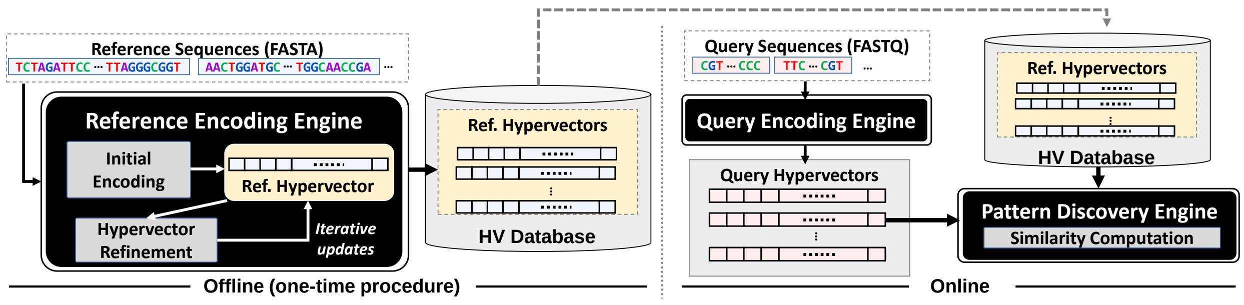


Fig. 1. Overview of GenieHD

recently uses HD computing for DNA sequence classification. Prior work also show application-specific accelerators on different platforms, e.g., FPGA [12]–[15] and ASIC [16]. Processing in-memory chips were also fabricated based on 3D VRRAM technology [17]. The previous works mostly utilize HD computing as a solution for classification problems. In this paper, we show that HD computing is an effective method for other pattern-centric problems, and propose a novel DNA pattern matching algorithm.

DNA Pattern Matching Acceleration The efficient pattern matching is an important task in many bioinformatics applications, e.g., single nucleotide polymorphism (SNP) identification, onsite disease detection and precision medicine development [1]. Many acceleration systems have been proposed on diverse platforms, e.g., multiprocessor [18] and FPGA [19]. For example, the work in [4] proposes an FPGA accelerator that parallelizes partial matches for a long DNA sequence based on KMP algorithm. The work in [3] proposed a parallel pattern matching method that streams the long-length reference into different CUDA cores. Our work is different in that we accelerate a new HD computing-based algorithm which is specialized for parallel systems and also effectively scales for the number of queries to process.

III. GenieHD Overview

Figure 1 illustrates the overview of the proposed GenieHD design. GenieHD exploits HD computing to design an efficient DNA pattern matching solution (Section IV.) During the offline stage, we convert the *reference* genome sequence into hypervectors and store into the *HV database*. In the online stage, we also encode the *query* sequence given as an input. GenieHD in turn identifies if the query exists in the reference or not, using a light-weight HD operation that computes hypervector similarities between the query and reference hypervectors. All the three processing engines perform the computations with highly-parallelizable HD operations. Thus, many parallel computing platforms can accelerate the proposed algorithm. We present the implementation on GPGPU, FPGA, and ASIC based on a general acceleration architecture (Section V.)

Nowadays, raw DNA sequences are publicly downloadable in standard formats, e.g., FASTA for references [20]. Likewise, the HV databases can provide the reference hypervectors encoded in advance, so that users can efficiently examine different queries without performing the offline encoding procedure repeatedly. For example, it is typical to perform the pattern matching for billions of queries streamed by a DNA sequencing machine. In this context, we also evaluate how GenieHD scales better than state-of-the-art methods when handling multiple queries (Section VI.)

IV. DNA Pattern Matching Using HD Computing

The major difference between HD and conventional computing is the computed data elements. Instead of booleans and numbers, HD computing performs the computations on ultra-wide words, i.e., hypervectors, where all words are responsible to represent a datum in a distributed manner. HD computing mimics important functionalities of the human memory [5]. For example, the brain efficiently aggregates/associates different data and understands similarity between data. The HD computing implements the aggregation and association using the hypervector addition and multiplication, while measuring the similarity based on a distance metric between hypervectors. The HD operations can be effectively parallelized in the granularity of the dimension level.

In this work, we represent DNA sequences with hypervectors, and perform the pattern matching procedure using the similarity computation. To encode a DNA sequence to hypervectors, GenieHD uses four hypervectors corresponding to each base alphabet in $\Sigma = \{A, C, G, T\}$. We call the four hypervectors as *base hypervectors*, and denote with $\Sigma_{HV} = \{\mathbf{A}, \mathbf{C}, \mathbf{G}, \mathbf{T}\}$[1]. Each of the hypervectors has D dimensions where a component is either -1 or +1 (biopolar), i.e., $\{-1, +1\}^D$. The four hypervectors should be uncorrelated to represent their differences in sequences. For example, $\delta(\mathbf{A}, \mathbf{C})$ should be nearly zero, where δ is the dot-product similarity. The base hypervectors can be easily created, since any two hypervectors whose components are randomly selected in $\{-1, 1\}$ have almost zero similarity, i.e., *nearly orthogonal*.

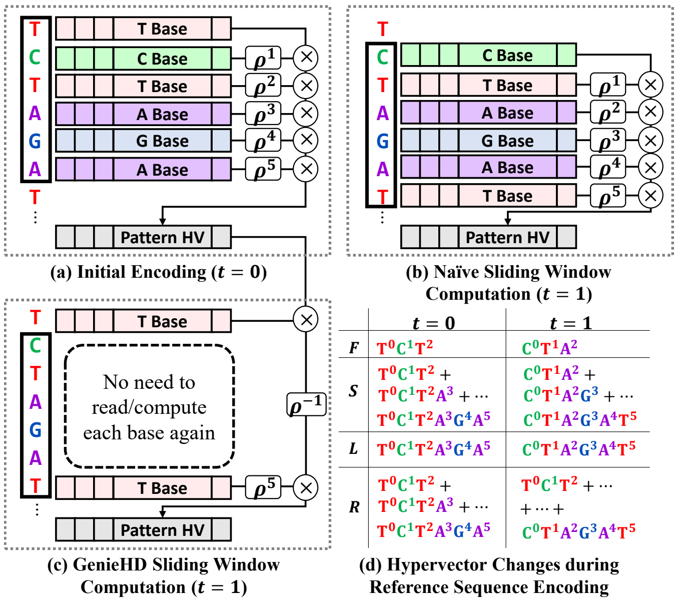
A. DNA Sequence Encoding

DNA pattern encoding: GenieHD maps a DNA pattern by combining the base hypervectors. Let us consider a short query string, 'GTACG'. We represent the string with $\mathbf{G} \times \rho^{\mathbf{1}}(\mathbf{T}) \times \rho^{\mathbf{2}}(\mathbf{A}) \times \rho^{\mathbf{3}}(\mathbf{C}) \times \rho^{\mathbf{4}}(\mathbf{G})$, where $\rho^n(\mathbf{H})$ is a *permutation* function that shuffles components of $\mathbf{H}$ ($\in \Sigma_{HV}$) with n-bit(s) rotation. For the sake of simplicity, we denote $\rho^n(\mathbf{H})$ as $\mathbf{H}^n$. $\mathbf{H}^n$ is nearly orthogonal to $\mathbf{H} = \mathbf{H}^0$ if $n \neq 0$, since the components of a base hypervector are randomly selected and independent of each other. Hence, the hypervector representations for any two different strings, $\mathbf{H}_\alpha$ and $\mathbf{H}_\beta$, are also nearly orthogonal, i.e., $\delta(\mathbf{H}_\alpha, \mathbf{H}_\beta) \simeq 0$. The hyperspace of D dimensions can represent 2^D possibilities. The enormous representations are sufficient to map different DNA patterns to near-orthogonal hypervectors.

Since the query sequence is typically short, e.g., 100 to 200 characters, the cost for the online query encoding step is negligible. In the followings, we discuss how GenieHD can efficiently encode the long-length reference sequence.

Reference encoding: The goal of the reference encoding is to create hypervectors that include all combinations of patterns. In practice, the approximate lengths of the query sequences are known, e.g., the DNA read length of the sequencing technology. Let us defined that the lengths of the queries are in a range of $[\perp, \top]$. The length of the reference sequence, $\mathcal{R}$, is denoted by N. We also use following notations: (i) $\mathbf{B_t}$ denotes the base hypervector for the t-th character in $\mathcal{R}$ (0-base indexing), and (ii) $\mathbf{H}_{(a,b)}$ denotes the hypervector for a subsequence, $\mathbf{B}_a^0 \times \mathbf{B}_{a+1}^1 \times \cdots \times \mathbf{B}_{a+b-1}^{b-1}$.

[1]In this paper, we use bold Latin symbols to represent hypervectors.



(a) Initial Encoding ($t = 0$)

(b) Naïve Sliding Window Computation ($t = 1$)

(c) GenieHD Sliding Window Computation ($t = 1$)

	$t = 0$	$t = 1$
F	$T^0C^1T^2$	$C^0T^1A^2$
S	$T^0C^1T^2 +$ $T^0C^1T^2A^3 + \cdots$ $T^0C^1T^2A^3G^4A^5$	$C^0T^1A^2 +$ $C^0T^1A^2G^3 + \cdots$ $C^0T^1A^2G^3A^4T^5$
L	$T^0C^1T^2A^3G^4A^5$	$C^0T^1A^2G^3A^4T^5$
R	$T^0C^1T^2 +$ $T^0C^1T^2A^3 + \cdots$ $T^0C^1T^2A^3G^4A^5$	$T^0C^1T^2 + \cdots$ $+ \cdots +$ $C^0T^1A^2G^3A^4T^5$

(d) Hypervector Changes during Reference Sequence Encoding

Fig. 2. Illustration of Encoding. For (a), (b), and (c), the window size is 6. (d) illustrates the reference encoding steps described in Algorithm 1.

Algorithm 1: Reference Encoding Algorithm

```
S ← $\mathbf{H}_{(0,\perp)} + \mathbf{H}_{(0,\perp+1)} + \cdots + \mathbf{H}_{(0,\top)}$
F ← $\mathbf{H}_{(0,\perp)}$; L ← $\mathbf{H}_{(0,\top)}$
R ← S
for $t \leftarrow 0$ to $N - \top$ do
    L ← $\mathbf{B}_t^{-1} \times \mathbf{L}^{-1} \times \mathbf{B}_{t+\top}^{\top}$
    S ← $\mathbf{B}_t^{-1} \times (\mathbf{S} - \mathbf{F})^{-1} + \mathbf{L}$; R ← $\mathbf{R} + \mathbf{S}$
    F ← $\mathbf{B}_t^{-1} \times \mathbf{F}^{-1} \times \mathbf{B}_{t+\perp}^{\perp}$
end
```

Let us first consider a special case that encodes every substring of the size n from the reference sequence, i.e., $n = \perp = \top$. The substring can be extracted using a sliding window of the size n to encode $\mathbf{H}_{(t,n)}$. Figure 2(a) illustrates the encoding method for the first substring, i.e., $t = 0$, when $n = 6$. A naive way to encode the next substring, $\mathbf{H}_{(1,n)}$, is to run the permutations and multiplications again for each base, as shown in Figure 2(b). Figure 2(c) shows how GenieHD optimizes it based on HD computing specialized to remove and insert new information. We first multiply $\mathbf{T}^0$ with the previously encoded hypervector, $\mathbf{T}^0\mathbf{C}^1\mathbf{T}^2\mathbf{A}^3\mathbf{G}^4\mathbf{A}^5$. The multiplication of two identical base hypervectors yields the hypervector whose elements are all 1s. Thus, it removes the first base from $\mathbf{H}_{0,n}$, producing $\mathbf{C}^1\mathbf{T}^2\mathbf{A}^3\mathbf{G}^4\mathbf{A}^5$. After performing the rotational shift (ρ^{-1}) and element-wise multiplication for the new base of the sliding window ($\mathbf{T}^5$), we obtain the desired hypervector, $\mathbf{C}^0\mathbf{T}^1\mathbf{A}^2\mathbf{G}^3\mathbf{A}^4\mathbf{T}^5$. This scheme only needs two permutations and multiplications regardless of the substring size n.

Algorithm 1 describes how GenieHD encode the reference sequence in the optimized fashion; Figure 2(d) shows how the algorithm runs for the first two iterations when $\perp = 3$ and $\top = 6$. The outcome is $\mathbf{R}$, i.e., the reference hypervector, which combines all substrings whose sizes are in $[\perp, \top]$. The algorithm starts with creating three hypervectors, $\mathbf{S}$, $\mathbf{F}$, and $\mathbf{L}$, (Line 1∼3). $\mathbf{S}$ includes all patterns of $[\perp, \top]$ in each sliding window; $\mathbf{F}$ and $\mathbf{L}$ keep tracks of the first and last hypervectors for the $\perp$-length and $\top$-length patterns, respectively. Intuitively, this initialization needs $O(\top)$ hypervector operations. The main loop implements the sliding window scheme for multiple lengths in $[\perp, \top]$. It computes the next $\mathbf{L}$ using the optimized scheme (Line 5). In Line 6, it subtracts $\mathbf{F}$, i.e., the shortest pattern in the previous iteration, and multiply $\mathbf{B}_t^{-1}$ to remove the first base from all patterns combined in $\mathbf{S}$. Then, $\mathbf{S}$ includes the patterns in the range of $[\perp, \top - 1]$ for the current window. After adding $\mathbf{L}$ whose length is $\top$, we accumulate $\mathbf{S}$ to $\mathbf{R}$. Lastly, we update the first pattern $\mathbf{F}$ in the same way to $\mathbf{L}$ (Line 7). The entire iterations need $O(N)$ operations regardless of the pattern length range, thus the total complexity of this algorithm[2] is $O(N + \top)$. Finally, $\mathbf{R}$ includes all the hypervector representations of the desired lengths existing in the reference.

B. Pattern Matching

GenieHD performs the pattern matching by computing the similarity between $\mathbf{R}$ and $\mathbf{Q}$. Let us assume that $\mathbf{R}$ is the addition of P hypervectors (i.e., P distinct patterns), $\mathbf{H}_1 + \cdots + \mathbf{H}_P$. The dot product similarity is computed as follows:

$$\delta(\mathbf{R}, \mathbf{Q}) = \delta(\mathbf{H}_\lambda, \mathbf{Q}) + \underbrace{\sum_{i=1, i\neq\lambda}^{P} \delta(\mathbf{H}_i, \mathbf{Q})}_{\textbf{Noise}}.$$

If $\mathbf{H}_\lambda$ is equal to $\mathbf{Q}$, since the similarity for the two identical biopolar hypervectors are D, i.e., $\delta(\mathbf{H}_\lambda, \mathbf{Q}) = D$. The similarity between any two different patterns is nearly zero, i.e., $\delta(\mathbf{H}_i, \mathbf{Q}) \simeq 0$ of the noise term. Thus, the following criteria checks if $\mathbf{Q}$ exists in $\mathbf{R}$:

$$\frac{\delta(\mathbf{R}, \mathbf{Q})}{D} > T \tag{1}$$

where T is a threshold. We call $\frac{\delta(\mathbf{R},\mathbf{Q})}{D}$ as the *decision score*.

The accuracy of this decision process depends on (i) the amount of the noise and (ii) threshold value, T. To precisely identify patterns in GenieHD, we develop a concrete statistical method that estimates the worst-case accuracy. The similarity metric computes how many components of $\mathbf{Q}$ are the same to the corresponding components for each $\mathbf{H}_i$ in $\mathbf{R}$. There are $P \cdot D$ component pairs for $\mathbf{Q}$ and $\mathbf{H}_i$ ($0 \leq i < P$). The probability that each pair is the same is $\frac{1}{2}$ for all components if $\mathbf{Q}$ is a random hypervector. The similarity, $\delta(\mathbf{R}, \mathbf{Q})$, can be then viewed as a random variable, X, which follows a binomial distribution, $X \sim B(P \cdot D, \frac{1}{2})$. Since D is large enough, X can be approximated with the normal distribution:

$$X \sim N\left(\frac{P \cdot D}{2}, \frac{P \cdot D}{4}\right).$$

When x component pairs of $\mathbf{R}$ and $\mathbf{Q}$ have the same value, $(P \cdot D - x)$ pairs have different values, thus $\delta(\mathbf{R}, \mathbf{Q}) = 2x - P \cdot D$. Hence, the probability that satisfies Equation 1 is $Pr(X > \frac{(T+P)\cdot D}{2})$. We can convert X to the standard normal distribution, Z:

$$Pr\left(Z > T \cdot \sqrt{\frac{D}{P}}\right) = \frac{1}{\sqrt{2\pi}} \int_{T \cdot \sqrt{\frac{D}{P}}}^{\infty} e^{-t^2/2}\, dt \tag{2}$$

In other words, Equation 2 represents the probability that mistakenly determines that $\mathbf{Q}$ exists in $\mathbf{R}$, i.e., false positive.

Figure 3(a) and (b) visualizes the probability of the error for different D and P combinations. For example, when $D = 100,000$ and $T = 0.5$, we can identify $P = 10,000$ patterns with 5.7% error using a single similarity computation operation. The results also show that using larger D values can improve the accuracy. However, the larger dimensionality requires more hardware resources. Another option to improve the accuracy is using a larger similarity

[2]Due to the limited space, we omit the finalization step which combines the patterns for $t > N - \top$; it can be implemented in a straight-forward way by modifying the main loop so that it does not use $\mathbf{L}$.

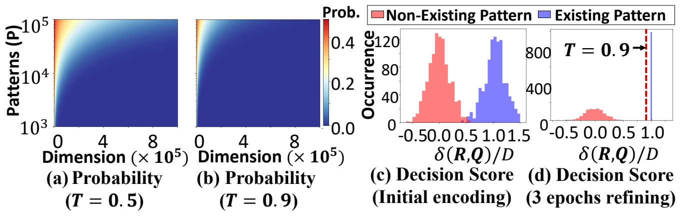


Fig. 3. Similarity Computation in Pattern Matching. (a) and (b) are computed using Equation 2. The histograms shown in (c) and (d) are obtained by testing 1,000 patterns for each of the existing and non-existing cases when $\mathbf{R}$ is encoded for a random DNA sequence using $D = 100,000$ and $P = 5,000$.

threshold, T, however it may increase true negatives. GenieHD uses the following two techniques to address this issue.

Hypervector refinement The first technique is to refine the reference hypervector. Let us recall Algorithm 1. In the refining step, GenieHD reruns the algorithm to update $\mathbf{R}$. Instead of accumulating $\mathbf{S}$ to $\mathbf{R}$ (Line 6), we add $\mathbf{S} \times (1 - \delta(\mathbf{S}, \mathbf{R})/D)$. The refinement is performed for multiple epochs. Figure 3(c) and (d) show how the distribution of the decision scores changes for the existing and non-existing cases by the refinement. The results show that the refinement makes the decision scores of the existing cases close to 1. Thus, we can use a larger T for higher accuracy. The successful convergence depends on i) the number of patterns included in $\mathbf{R}$ with D dimensions, i.e., D/P, and ii) the training epochs. In our evaluation, we observe that, when $\mathbf{R}$ includes $P = D/10$ patterns and use $T = 0.9$, we only need five epochs, and GenieHD can find all patterns with the error of less than 0.003%.

Multivector generation To precisely discover patterns of the reference sequence, we also use multiple hypervectors so that they cover every pattern existing in the reference without loss. During the initial encoding, whenever $\mathbf{R}$ reaches the maximum capacity, i.e., accumulating P distinct patterns, we store the current $\mathbf{R}$ and reset its components to 0s to start computing a new $\mathbf{R}$. GenieHD accordingly fetches the stored $\mathbf{R}$ during the refinement. Even though it needs to compute the similarity values for the multiple $\mathbf{R}$ hypervectors, GenieHD can still fully utilize the parallel computing units by setting D to a sufficiently large number.

V. Hardware Acceleration Design

A. Acceleration Architecture

Encoding Engine The encoding procedure runs i) the element-wise addition/multiplication and ii) permutation. The parallelized implementation of the element-wise operations is straight-forward, i.e., computing each dimension on different computing units. For example, if a computing platform can compute d dimensions (out of D) independently in parallel, the single operation can be calculated with $\lceil D/d \rceil$ stages. In contrast, the permutation is more challenging due to memory accesses. For example, a naive implementation may access all hypervector components from memory, but on-chip caches usually have no such capacity.

The proposed method significantly reduces the amount of memory accesses. Figure 4a illustrates our acceleration architecture for the initial reference encoding procedure as an example. The acceleration architecture represents typical parallel computing platforms which have many computing units and memory. As discussed in Section IV-A, the encoding procedures uses the permuted bipolar base hypervectors, $\mathbf{B}^{-1}, \mathbf{B}^{\perp}$ and $\mathbf{B}^{\top}$, as the inputs. Since there are four DNA alphabets, the inputs are 12 near-orthogonal hypervectors. It calculates the three intermediate hypervectors, $\mathbf{F}, \mathbf{L}$ and $\mathbf{S}$ while accumulating $\mathbf{S}$ into the output reference hypervector, $\mathbf{R}$.

Consider that the permuted base hypervectors and initial reference hypervector are pre-stored in the off-chip memory. To compute all components of $\mathbf{R}$, we run the main loop of the reference

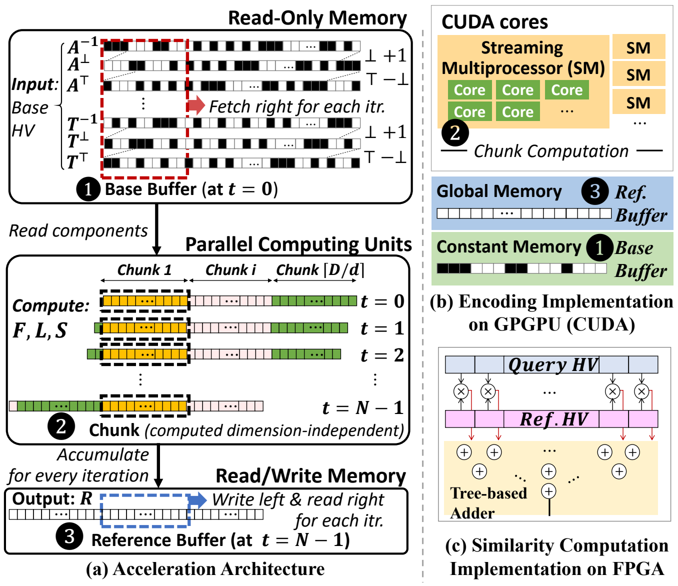


Fig. 4. Hardware Acceleration Design. The dotted boxes in (a) show the hypervector components required for the computation in the first stage of the reference encoding. Recall that t is the index of the iteration.

encoding $\lceil D/d \rceil$ times by dividing the dimensions into multiple groups, called *chunks*. In the first iteration, the *base buffer* stores the first d components of the 12 input hypervectors (❶). The same d dimensions of $\mathbf{F}, \mathbf{L}$ and $\mathbf{S}$ for the first chunk are stored in the local memory of each processing unit, e.g., registers of each GPU core (❷). For each iteration, the processing units compute the dimensions of the chunk in parallel, and accumulate to the *reference buffer* that stores the d components of $\mathbf{R}$ (❸). Then, the base buffer fetches the next elements for the 12 input hypervectors from the off-chip memory. Similarly, the reference buffer flushes its first element to the off-chip memory and reads the next element. When it needs to reset $\mathbf{R}$ for the multivector generation, the reference buffer is stored to the off-chip memory and filled with zeros. The key advantage of this method is that we do not need to know entire D components of $\mathbf{F}, \mathbf{L}$ and $\mathbf{S}$ for the permutation. Instead, we can regard that they are the d components starting from the τ-th dimension where $\tau = t \mod D$, and accumulate them in the reference buffer which already has the corresponding dimensions. Every iteration only needs to read a single element for each base and a single read/write for the reference, while fully utilizing the computing units for the HD operations. Once completing N iterations, we repeat the same procedure for the next chunk until covering all dimensions.

The similar method is generally applicable for the other procedures, the query encoding and refinement. For example, for the query encoding, we compute each chunk of $\mathbf{Q}$ by reading an element for each base hypervector and multiplying d components.

Similarity Computation The pattern discovery engine and refinement procedures use the similarity computation. The dot product is decomposed with the element-wise multiplication and the grand sum of the multiplied components. The element-wise multiplication can be parallelized on the different computing units, and then we can compute the grand sum by adding multiple pairs in parallel with $O(\log D)$ steps. The implementation depends on the parallel platforms. We explain the details in the following section.

B. Implementation on Parallel Computing Platforms

GenieHD-GPU We implement the encoding engine by utilizing the parallel cores and different memory resources in CUDA systems (refer to Figure 4b.) The base buffer is stored in the constant memory, which offers high bandwidth for read-only data. Each streaming core stores the intermediate hypervector components of the chunk in their registers; the reference buffer is located in the global memory (DRAM on GPU card). The data reading and writing to the constant and global memory are implemented with CUDA streams which concurrently copy data during computations. We implement the similarity computation using the parallel reduction technique [21]. Each stream core fetches and adds multiple components into the shared memory which provide high performance for inter-thread memory accesses. We then perform the tree-based reduction in the shared memory.

GenieHD-FPGA We implement the FPGA encoding engine by using Lookup Table (LUT) resources. We store the base hypervectors into block RAMs (BRAM), the on-chip FPGA memory. The base hypervectors are loaded to a distributed memory designed by the LUT resources. Depending on the reading sequence, GenieHD loads the corresponding base hypervector and combines them using LUT resources. In the pattern discovery, we use the DSP blocks of FPGA to perform the multiplications of the dot product and a tree-based adder to accumulate the multiplication results (refer to Figure 4c.) Since the query encoding and discovery use different FPGA resources, we implement the whole procedure in a pipeline structure to handle multiple queries. Depending on the FPGA available resources, it can process a different number of dimensions in parallel. For example, for Kintex-7 FPGA with 800 DSPs, we can parallelize the computation of 320 dimensions.

GenieHD-ASIC The ASIC design has three major subcomponents: SRAM, interconnect, and computing block. We used the SRAM-based memory to keep all base hypervectors. The memory is connected to the computing block with the interconnect. To reduce the memory writes to SRAM, the interconnect implements n-bit shifts to fetch the hypervector components to the computing block with a single cycle. The computing units parallelize the element-wise operations. For the query discovery, it forwards the execution results to the tree-based adder structure located in the computing block in a similar way to the FPGA design. The efficiency depends on the number of parallel computing units. We design GenieHD-ASIC with the same size of the experimented GPU core, $471mm^2$. In this setting, our implementation parallelizes the computations for 8000 components.

VI. Evaluation

A. Experimental Setup

We evaluate GenieHD on parallel various computing platforms. We implement GenieHD-GPU on NVIDIA GTX 1080 Ti (3584 CUDA cores) and Intel i7-8700K CPU (12 multithreads) and measure power consumption using Hioki 3334 power meter. GenieHD-FPGA is synthesized on Kintex-7 FPGA KC705 using Xilinx Vivado Design Suite. We used Vivado XPower tool to estimate the device power. We design and simulate GenieHD-ASIC using RTL System-Verilog. For the synthesis, we use *Synopsys Design Compiler* with the TSMC 45 nm technology library and the general purpose process with high V_{TH} cells. We estimate the power consumption using *Synopsys PrimeTime* at (1V, 25 °C, TT) corner. The GenieHD family is evaluated using $D = 100,000$ and $P = 10,000$ with five refinement epochs.

Table I summarizes the evaluated DNA sequence datasets. We use E.coli DNA data (MG1655) and the human reference genome, chromosome 14 (CHR14) [20]. We also create a random synthetic DNA sequence (RND70) having a length of 70 million characters. The query sequence reads with the length in $[\perp, \top]$ are extracted using SRA toolkit from the FASTQ format. The total size of the generated hypervectors for each sequence (HV size) is linearly proportional to the length of the reference sequence. Note that state-of-the-art bioinformatics tools also have the peak memory footprint in up to two orders of gigabytes for the human genome [22].

TABLE I
Evaluated DNA Sequence Datasets

	Description	Length	$\perp, \top$	HV size
E.Coli (MG1655)	Escherichia coli	4.6M	199,201	53MB
Human (CHR14)	Human chromosome 14	107M	99,101	1.2GB
Synthetic (RND70)	Random sequence	70M	99,101	0.8GB

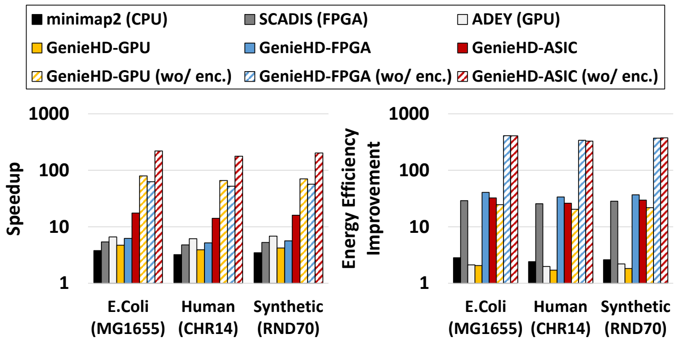


Fig. 5. Performance and Energy Comparison of GenieHD for State-of-the-art Methods. All results are compared and normalized to Bowtie2.

B. Efficiency Comparison

We compare the efficiency of GenieHD with state-of-the-art programs and accelerators, i) *Bowtie2* [23] running on Intel i7-8700K CPU and ii) *minimap2* [22], which runs on the same CPU, but tens of times faster than the previous mainstream such as BLASR and GMAP, iii) GPU-based design (*ADEY*) [3], and iv) FPGA-based design (*SCADIS*) [4] evaluated on the same chip to GenieHD-FPGA. Figure 5 presents that GenieHD outperforms the state-of-the-art methods. For example, even though including the overhead of the offline reference encoding, GenieHD-ASIC achieves up to 16× speedup and 40× higher energy efficiency as compared to Bowtie2. GenieHD can offer higher improvements if the references are encoded in advance. For example, when the encoded hypervectors are available, by eliminating the offline encoding costs, GenieHD-ASIC is 199.7× faster and 369.9× more energy efficient than Bowtie2. When comparing the same platforms, GenieHD-FPGA (GenieHD-GPU) achieves 11.1× (10.9×) speedup and 13.5× (10.6×) higher energy efficiency as compared to SCADIS running on FPGA (ADEY on the GPGPU).

C. Pattern Matching for Multiple Queries

Figure 6(a) shows the breakdown of the GenieHD procedures. The results show that most execution costs come from the reference encoding procedure, e.g., more than 97.6% on average. It is because i) the query sequence is relatively very short and ii) the discovery procedure examines multiple patterns using a single similarity computation in a highly parallel manner. As discussed in Section III, GenieHD can *reuse* the same reference hypervectors for different queries newly sampled. Figure 6(b)-(d) shows the speedup of the accumulated execution time for multiple queries over the state-of-the-art counterparts. For fair comparison, we evaluate the performance of GenieHD based on the total execution costs including the reference/query encoding and query discovery engines. The results show that, by reusing the encoded reference hypervector, GenieHD achieves higher speedup as the number of queries increases. For example, when comparing the designs running on the same platform, we observe 43.9× and 44.4× speedup on average for 10^6 queries on (b) GPU and (c) FPGA,

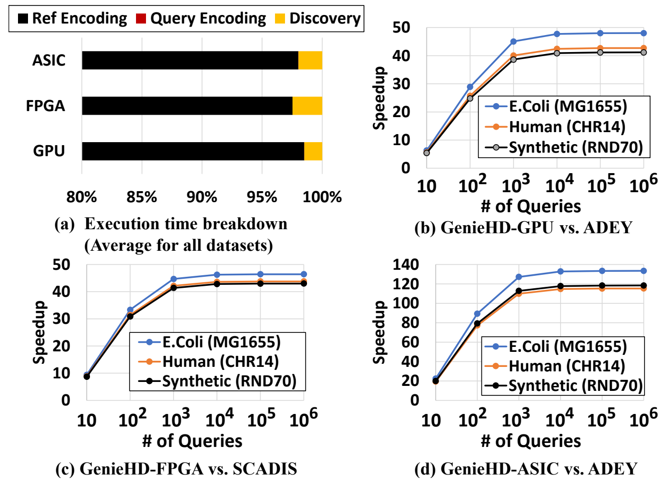


Fig. 6. Scalability of GenieHD. (a) shows the execution time breakdown to process the single query and reference. (b)-(d) shows how the speedup changes as increasing the number of queries for a reference.

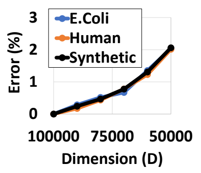


Fig. 7. Accuracy Loss over Dimension Size

TABLE II
GenieHD-ASIC Designs under Loss

		0%	1%		2%	
		Base	Gating	VoS	Gating	VoS
Power(W)	SRAM	3.4	2.5	3.4	1.8	3.4
	ITC	0.6	0.4	0.6	0.3	0.6
	CB	21.5	13.7	8.8	10.9	6.0
	Total	25.4	16.6	12.8	13.1	10.0
Throughput		640K / sec				

respectively. The energy-efficiency improvement for each case is 42.5× and 54.1×. As compared to ADEY, GenieHD-ASIC offers 122× speedup and 707× energy-efficiency improvements with the same area (d). It is because GenieHD consumes much less cost from the second run. The speedup converges at around 10^3 queries as the query discovery takes a more portion of the execution time for a larger number of queries.

D. Dimensionality Exploitation

In practice, the higher efficiency would be more desired than the perfect discovery, since DNA sequences are often error-prone [1]. The statistical nature of GenieHD facilitates such optimization. Figure 7 shows how much the additional error occurs from the baseline accuracy of 0.003% as decreasing the dimensionality. As anticipated with the estimation model shown in Section IV-B, the error increases with a less dimensionality. Note that it does not need to encode the hypervectors again; instead, we can use only a part of components in the similarity computation. The results suggest that we can significantly improve the efficiency with minimal accuracy loss. For example, we can achieve 2× speedup for all the GenieHD family with 2% loss as it only needs the computation for half dimensions. We can also exploit this characteristic for power optimization. Table II shows the power consumption for the hardware components of GenieHD-ASIC, SRAM, interconnect (ITC), and computing block (CB) along with the throughput. We evaluated two power optimization schemes, i) **Gating** which does not use half of the resources, and ii) voltage over scaling (**VoS**) which uses all resources at a lower frequency. The frequency is set to obtain the same throughput of 640K/sec (the number of similarity computations per second.) The results show that **VoS** is the more effective method since the frequency non-linearly influences the speed. GenieHD-ASIC with **VoS** saves 50% and 61% power with accuracy loss of 1% and 2%, respectively.

VII. Conclusion

In this paper, we describe GenieHD which performs the DNA pattern matching algorithm using HD computing. The proposed technique maps DNA sequences to hypervectors, and accelerates the pattern matching procedure in a highly-parallelized way. We show an acceleration architecture to optimize the memory access patterns and perform pattern matching tasks with dimension-independent operations in parallel. The experimental results show that GenieHD significantly accelerates the pattern matching procedure, e.g., 44.4× speedup with 54.1× energy-efficiency improvements when comparing to the existing design on the same FPGA.

VIII. Acknowledgement

This work was supported in part by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, in part by SRC-Global Research Collaboration grant, and also NSF grants #1527034, #1730158, #1826967, and #1911095.

References

[1] Phillip Compeau et al. *Bioinformatics algorithms: an active learning approach*, volume 1. Active Learning Publishers La Jolla, 2015.
[2] Christiam Camacho, et al. Blast+: architecture and applications. *BMC bioinformatics*, 10(1):421, 2009.
[3] Snehal P Adey. Gpu accelerated pattern matching algorithm for dna sequences to detect cancer using cuda. 2013.
[4] Shiming Lei, et al. Scadis: A scalable accelerator for data-intensive string set matching on fpgas. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 1190–1197. IEEE, 2016.
[5] Pentti Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation*, 1(2):139–159, 2009.
[6] O. Rasanen et al. Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns. *IEEE Transactions on Neural Networks and Learning Systems*, PP(99):1–12, 2015.
[7] Abbas Rahimi, et al. High-dimensional computing as a nanoscalable paradigm. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(9):2508–2521, 2017.
[8] Yeseong Kim, et al. Efficient human activity recognition using hyperdimensional computing. In *Proceedings of the 8th International Conference on the Internet of Things*, page 38. ACM, 2018.
[9] Mohsen Imani, et al. A framework for collaborative learning in secure high-dimensional space. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 435–446. IEEE, 2019.
[10] Mohsen Imani, et al. Bric: Locality-based encoding for energy-efficient brain-inspired hyperdimensional computing. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.
[11] Mohsen Imani, et al. Hdna: Energy-efficient dna sequencing using hyperdimensional computing. In *2018 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI)*, pages 271–274. IEEE, 2018.
[12] Mohsen Imani, et al. Fach: Fpga-based acceleration of hyperdimensional computing by reducing computational complexity. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 493–498. ACM, 2019.
[13] Mohsen Imani, et al. Sparsehd: Algorithm-hardware co-optimization for efficient high-dimensional computing. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 190–198. IEEE, 2019.
[14] Sahand Salamat, et al. 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*, pages 53–62. ACM, 2019.
[15] Mohsen Imani, et al. A binary learning framework for hyperdimensional computing. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 126–131. IEEE, 2019.
[16] Mohsen Imani, et al. Exploring hyperdimensional associative memory. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 445–456. IEEE, 2017.
[17] Haitong Li, et al. Hyperdimensional computing with 3d vrram in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition. In *Electron Devices Meeting (IEDM), 2016 IEEE International*, pages 16–1. IEEE, 2016.
[18] Suejb Memeti et al. Analyzing large-scale dna sequences on multi-core architectures. In *2015 IEEE 18th International Conference on Computational Science and Engineering*, pages 208–215. IEEE, 2015.
[19] Edward B Fernandez, et al. Multithreaded fpga acceleration of dna sequence mapping. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–6. IEEE, 2012.
[20] National center for biotechnology information support center. https://www.ncbi.nlm.nih.gov.
[21] Mark Harris et al. Optimizing parallel reduction in cuda. *Nvidia developer technology*, 2(4):70, 2007.
[22] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.
[23] Ben Langmead et al. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357, 2012.