JCST Journal of Computer Science and Technology

Indexed in: SCIE, EI, Scopus, INSPEC, DBLP,CSCD, etc.
Sponspored by: ICT, CAS & CCF

JCST Papers

Only for Academic and Non-Commercial Use

Thanks for Reading!

[Survey](#)

[Computer Architecture and Systems](#)

[Artificial Intelligence and Pattern Recognition](#)

[Computer Graphics and Multimedia](#)

[Data Management and Data Mining](#)

[Software Systems](#)

[Computer Networks and Distributed Computing](#)

[Theory and Algorithms](#)

[Emerging Areas](#)

JCST WeChat
Subscription Account

JCST URL: https://jcst.ict.ac.cn

SPRINGER URL: https://www.springer.com/journal/11390

E-mail: jcst@ict.ac.cn

Online Submission: https://mc03.manuscriptcentral.com/jcst

Twitter: JCST_Journal

LinkedIn: Journal of Computer Science and Technology

Liu WQ, Li YW, Tan GM. iSCoder: Mitigating genomic sequencing data compression bottlenecks via in-SRAM computing. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY, 2026. DOI: 10.1007/s11390-025-5022-y, CSTR: 32374.14.s11390-025-5022-y

iSCoder: Mitigating Genomic Sequencing Data Compression Bottlenecks via In-SRAM Computing

Wan-Qi Liu[1, 2] (刘万奇), *Member, CCF*, Ye-Wen Li[3] (李叶文), *Member, CCF*
and Guang-Ming Tan[1, 2, *] (谭光明), *Senior Member, CCF, Member, ACM, IEEE*

[1] *State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*
[2] *School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100049, China*
[3] *The Hong Kong University of Science and Technology, Hong Kong 999077, China*

E-mail: liuwanqi18z@ict.ac.cn; yewenli@ust.hk; tgm@ict.ac.cn

Received November 18, 2024; accepted October 14, 2025.

Abstract With the rapid expansion of genomic sequencing data over the years, the costs associated with storage, transmission, and bandwidth are becoming the primary bottlenecks in genomic research and applications. Data compression is widely used to alleviate this burden, provided it achieves a sufficiently high compression ratio and fast compression speed. MPEG-G is a genome-specific compression standard that offers a higher compression ratio than general-purpose compression tools (4.3x), however, at the cost of performance reduction (5x). Following common strategies in compression acceleration, we design to the best of our knowledge, the first hardware accelerator for the MPEG-G genomic data compression pipeline utilizing in-SRAM (Static Random-Access Memory) computing, referred to as iSCoder. We identify and analyze MatchC (Match Coding) and LutC (Lut Coding) as two bottleneck algorithms within this pipeline, propose two optimized in-SRAM algorithms, and design a unified hardware architecture for these algorithms, considering the characteristics of genomic data. Compared with 72-core Intel processors operating at 3.0 GHz, experimental results demonstrate that iSCoder achieves an average speedup of 131x for MatchC and 191x for LutC.

Keywords genomic sequencing data, compression, in-SRAM computing, hardware accelerator

1 Introduction

Genomics is fundamentally reshaping medicine and deepening our understanding of life[1]. With the rapid advancement in sequencing technology, genomic sequencing data (e.g., Fastq file[2], can be up to 6 TB from a single sequencing[3]) has been expanding at a rate of doubling every seven months—far surpassing Moore's Law, outpacing the data volumes generated by platforms such as YouTube and Twitter[4]. This trend indicates that storage, transmission, and bandwidth costs will soon surpass the costs of sequencing itself and become the main bottleneck in genomics research and applications[5].

A common approach to addressing this bottleneck is data compression. As a genome-specific compression standard proposed by the Moving Picture Experts Group (MPEG), MPEG-G[6] describes a three-stage pipeline (Fig.1(a)) for sequencing data and is implemented by the open-source project called Genie[7], which achieves a higher compression ratio than general-purpose tools such as GZIP (GNU Zip, 19x higher for base sequence and 4.3x higher for the whole Fastq file). MPEG-G combines the advantages of other genome-specific algorithms[8, 9] and provides a more suitable choice due to its compatibility, standardization, and new functionalities, such as random access in the compressed domain.

However, the high compression ratio of MPEG-G comes at the cost of performance reduction (5x lower

Regular Paper
The work was supported by the National Natural Science Foundation of China under Grant Nos. 62032023 and T2125013.
*Corresponding Author
©Institute of Computing Technology, Chinese Academy of Sciences 2026

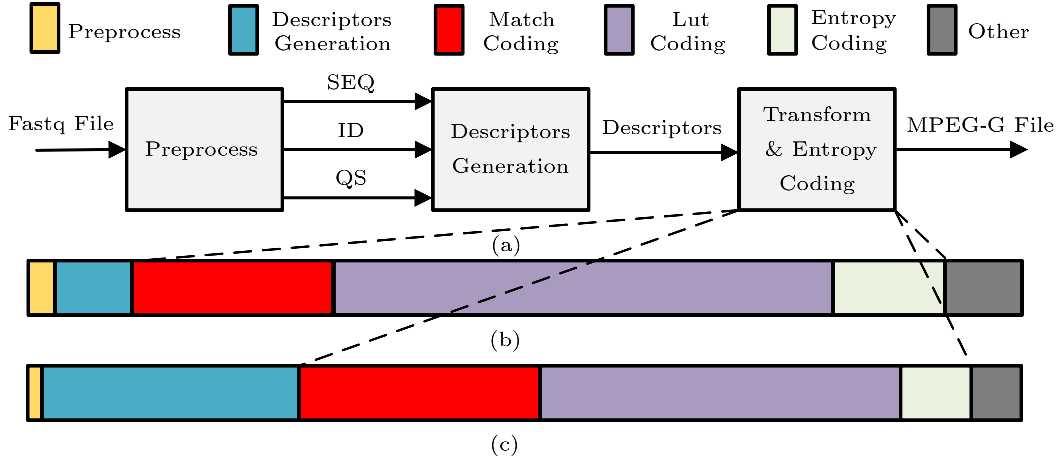

Fig.1. (a) Overview of the MPEG-G compression pipeline and execution time breakdown using (b) low-latency mode and (c) reorder mode.

than software GZIP) due to additional processing steps in the algorithm pipeline, such as data assembly and transformation. Given the large volume of data requiring compression and the frequent application of compression into real-time genomic analysis workflows, such as BWA (Burrows-Wheeler Aligner)[10], this extra latency cost is unacceptable. Moreover, Genie[7], the first multi-thread software implementation of the MPEG-G pipeline, has a complex workflow that involves processing for three fields in Fastq data, making it difficult to accelerate. To the best of our knowledge, we are the first to identify two bottleneck algorithms, Match Coding (MatchC) and Lut Coding (LutC), which can dominate the execution time (70%) of the MPEG-G pipeline.

This paper aims to design a hardware accelerator to reduce the execution latency, mitigating performance bottlenecks of this genomic data compression pipeline. The primary motivation behind this idea stems from the fact that many previous compression algorithms have employed hardware acceleration, including general-purpose compression accelerators[11, 12], as well as domain-specific compression accelerators[13, 14].

As an emerging branch of hardware acceleration technology, in-SRAM (Static Radom-Access Memory) computing has been widely adopted in prior work[15–17]. Compared with software systems, it inherently offers high parallelism and low latency, making it a promising solution to meet the performance demands of the MPEG-G genomic sequencing data compression pipeline. Moreover, unlike other hardware accelerator designs, such as the hardware hash search and the comparator-based CAM (content addressable memory), in-SRAM technology can accomplish the sequence-matching operations required by MatchC and LutC with higher parallelism and simpler logic.

In this paper, we propose iSCoder, a hardware accelerator for genomic sequencing data compression using the in-SRAM technology. Our contributions are as follows.

1) To the best of our knowledge, we are the first to profile and identify MatchC and LutC as two bottleneck algorithms in the MPEG-G pipeline, examine their performance limitations, and analyze opportunities and challenges for in-SRAM acceleration.

2) We propose two in-SRAM algorithms for MatchC and LutC respectively to leverage parallelism intra each SRAM array and schedule tasks inter SRAM arrays. Moreover, the algorithm for MatchC overcomes imbalance between computing and memory accessing, and the algorithm for LutC reduces the number of SRAM arrays and alleviates load imbalance.

3) We propose a unified-oriented hardware architecture according to the MatchC and LutC in-SRAM algorithms, which can be switched between two modes, achieving high hardware utilization.

4) The experimental results demonstrate that iSCoder achieves an average speedup of 131x for MatchC and 191x for LutC, compared with a 72-core Intel processor running at 3.0 GHz frequency.

2 Background and Motivation

2.1 MPEG-G Pipeline

MPEG-G is the first open ISO/IEC standard for genomic sequencing data compression, which specifies the decoding method, file format, and transmission format of compressed streams with good compatibili-

ty[6]. Fig.1(a) illustrates the three-stage MPEG-G compression pipeline for a Fastq file. In the pre-processing stage, the base sequence (SEQ), read identifiers (ID), and quality scores (QS) fields are divided into three streams. The descriptors generation stage employs specific data transformations for these steams based on their characteristics and generates more suitable descriptors for compression. In the last stage, the descriptors are further transformed, binarized, and encoded by an entropy coding algorithm. Although MPEG-G benefits from a higher compression ratio (4.3x higher) than the general-purpose compression tool such as GZIP, it comes at the expense of performance reduction (5x lower), leading to a burden for data storage and transmission for genomic analysis.

We perform performance analysis of an open-source implementation, i.e., Genie[7], on the traditional CPU-based platform. The profiling includes two modes, which differ in whether the descriptors generation stage includes a global assembly of SEQ (reorder mode) or not (low-latency mode). As shown in Fig.1(b) and Fig.1(c), the last stage is the bottleneck which can reach 89% and 75% of the whole execution latency in two modes, respectively. As MatchC and LutC dominate this stage (78%), we focus on them in this paper.

2.2 Algorithm Description

MatchC Algorithm. MatchC aims to identify the longest continuous matching symbols of an input sequence within a sliding window including 256 symbols, outputting the maximum length (*lengths*) and position (*pointers*) as the encoded result. As described in Algorithm 1, loop 3 performs the basic operation of this algorithm: comparisons between two 8-bit symbols. This loop terminates upon encountering a mismatch, records the match length, and then shifts the starting position within the sliding window for the next comparison in loop 2. Moreover, loop 1 refreshes the sliding window by shifting forward by the length of the longest match, and prepares for the next match.

LutC Algorithm. As shown in Algorithm 2, the goal of LutC is to encode an input symbol by indexing it into a lookup table based on the symbol's address and identifying its match position within the corresponding data row. Specifically, two addresses are used, corresponding to the two preceding symbols and the immediately preceding symbol of the input, which can be represented as the $tuple(addr1,\ addr2,\ sym)$. In typical genomic data compression, input symbols are less than 128, and data row lengths are 128 symbols, yielding a lookup table size of $128\times128\times128\times7$ bits.

Algorithm 1. MatchC Algorithm

Data: *symbols*: input sequence
Result: *pointers*, *lengths*: encoded results
1 **loop 1: for** $i = WindowSize$ to $|symbols| - 1$ **do**
2 $\quad pointer \leftarrow 0,\ length \leftarrow 0$
3 $\quad$ **loop 2: for** $w = i - WindowSize$ to $i - 1$ **do**
4 $\quad\quad offset \leftarrow i$
5 $\quad\quad$ **loop 3: while:** $offset < |symbols|$ AND $symbols[offset] = symbols[w + offset - i]$ AND $offset - i < WindowSize - 1$ **do**
6 $\quad\quad offset \leftarrow offset + 1$
7 $\quad\quad$ **end**
8 $\quad\quad$ **if:** $offset - i \geq length$ **then**
9 $\quad\quad\quad length \leftarrow offset - i,\ pointer \leftarrow w$
10 $\quad\quad$ **end**
11 $\quad$ **end**
12 $\quad$ Append $(i - pointer)$ to *pointers*
13 $\quad$ Append *length* to *lengths*
14 $\quad i \leftarrow i + length - 1$
15 **end**

Algorithm 2. LutC Algorithm

1 **Parameter:** $SIZE$: table size, NUM: symbol number
Data: $sym[]$: input symbol, $table[SIZE][SIZE][SIZE]$: lut
Result: pos: output positions based on lookup matches
2 **Initialization:**
3 $addr1 \leftarrow sym[0],\ addr2 \leftarrow sym[1]$
4 $pos[0] \leftarrow addr1,\ pos[1] \leftarrow addr2$
5 **loop 1: for** $i = 2$ to $NUM - 1$ **do**
6 $\quad row_t \leftarrow table[addr1][addr2]$
7 $\quad$ **loop 2: for** $j = 0$ to $SIZE - 1$ **do**
8 $\quad\quad$ **if:** $sym[i] == row_t[j]$ **then**
9 $\quad\quad\quad pos[i] \leftarrow j$
10 $\quad\quad$ **end**
11 $\quad$ **end**
12 $\quad$ Update addresses for the next iteration
13 $\quad addr1 \leftarrow sym[i-2],\ addr2 \leftarrow sym[i-1]$
14 **end**

Workload Analysis. We perform a cycles-per-instruction (CPI) stack analysis on MatchC and LutC using Sniper[18] shown in Fig.2. Both algorithms exhibit core-bound workload characteristics at 72% and 60%, respectively, because of their high memory locality. With the infrequent DRAM accesses, most pro-

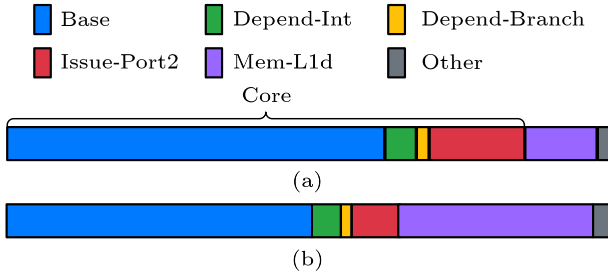

Fig.2. CPI stacks of (a) MatchC and (b) LutC.

cessing time is spent within the core instruction pipeline (Base). Furthermore, performance is impacted by lost cycles from limited execution units (Issue-port) and dependencies (Depend), further intensifying the core-bound workload. Additionally, SRAM access overheads are also required to be taken into account, particularly for LutC (33%) which accesses data from a larger table.

2.3 Limitation of Existing Solutions

Software Solutions. 1) *CPU.* The current multi-threaded implementation underperforms due to its inability to exploit the high theoretical parallelism in algorithms like MatchC and LutC. For example, MatchC can theoretically process 256 slide-window positions and 256 input symbols in parallel, while LutC can compare symbols across multiple data rows and input tuples. However, CPUs are limited by their insufficient core count and short vector units. Moreover, coding tasks for different data blocks can be parallelized as they are independent. 2) *GPU.* Despite being optimized for parallelism, GPUs are unsuitable for accelerating these algorithms due to the low-latency requirements of the MPEG-G pipeline. The relatively small workloads (e.g., 256×256 B in MatchC) per iteration do not justify the overhead of GPU synchronization and branching. Additionally, both CPU and GPU consume high power, which is inefficient for the simple computational patterns.

Hardware Solutions. Custom hardware accelerators are a common approach in designing compression algorithms. For instance, several studies[19–21] have developed hardware accelerators for GZIP, yielding substantial performance gains. The LZ77 (Lempel-Ziv'77) algorithm[22] in GZIP is similar to the MatchC algorithm in that both search for the longest match within a sliding window. However, these designs rely on hash searches due to the large window sizes (tens of KB or even several MB), while MatchC operates with a smaller window size of only 256 B, making intensive hash calculations unnecessary. In contrast, Abali *et al.*[11] utilized CAMs within a small window range instead of hash functions, enabling multiple string comparisons in a single cycle. However, this CAM is implemented by registers and comparators and incurs substantial hardware costs. Additionally, these hardware accelerators are designed for general-purpose compression algorithms, leaving a gap when applied to genomic sequencing data characteristics.

2.4 In-SRAM Computing

In recent years, in-SRAM computing has garnered significant attention since it not only exploits a high degree of parallelism but also mitigates the data movement overhead. As depicted in Fig.3, an SRAM-based content addressable memory (CAM) exemplifies this technology, functioning to search for all bit columns within the SRAM array based on the input provided on the left.

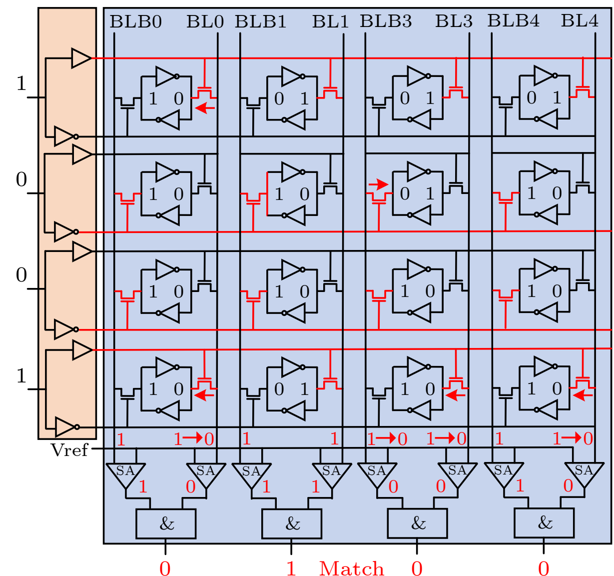

Fig.3. Hardware design of SRAM-based CAM[23].

When the input is "1001", the first row input being "1" activates the transistors on the right side of all SRAM cells, thereby connecting the corresponding data to the bit lines (BL). Conversely, an input of "0" results in the data being connected to the bit line bars (BLB). The data connected to either BL or BLB undergoes an AND logic operation along its respective lines, with the outcomes further aggregated through AND operations across the sense amplifiers (SAs). The output signal of "1" signifies that the column matches the input bit, and all columns can be performed simultaneously in one cycle.

2.5 Opportunities, Challenges, and Goals

Opportunities. 1) *Intra-SRAM Array Parallelism.* By designing an optimized memory layout and matching mechanism, we can leverage logical AND operations across both rows and columns, with parallelism scaling with the array size. 2) *Inter-SRAM Array Parallelism.* An efficient scheduling mechanism can improve the utilization rate of multiple arrays. 3) *Low Latency and Power Consumption.* They are due to reduced data movement to processors.

Challenges. 1) *Functional Gap Between MatchC/LutC and CAM.* Although in-SRAM design offers the above opportunities, the function of CAM in Fig.3 is limited to comparing two fixed-length strings and only yielding a binary match/miss-match result. However, directly applying this design is inadequate for MatchC because this algorithm requires identifying the longest matching string and its length, which is variable. 2) *Performance Dilemma Caused by the Basic In-SRAM Design.* Even if these two algorithms are mapped appropriately to the in-SRAM hardware, they also cannot yield significant performance improvements or could come at the cost of unacceptable resource overhead. For instance, there are imbalances between computation and memory access in MatchC (discussed in Subsection 3.2) and load imbalance issues in LutC (also discussed in Subsection 3.2). 3) *Resource Waste of Two Separate Accelerators for Two Algorithms.* Due to in-SRAM design, not only would the utilization of logic circuits be low, but the utilization of SRAM would also suffer.

Goals. As a result, this work aims to achieve the following three goals: 1) to establish the basic design of the two algorithms for functional mapping, including determining data layouts, atomic operation, and execution mechanisms, as well as the additional hardware modules, 2) to analyze performance limitations in the basic in-SRAM design and solve them by proposing optimization strategies according to algorithm flow characteristics, genomic data, and hardware architecture, and 3) to propose a unified-oriented design to reuse hardware resources for both algorithms.

3 In-SRAM MatchC Algorithm

3.1 Basic Design

Algorithm Description. As shown in Algorithm 3, the in-SRAM MatchC algorithm includes an initialization step, two computation loops, and data refreshing after each iteration. The Initialization step specifies the memory layout of the MatchC's slide window within each SRAM array (256×8 rows, 256 columns). In detail, columns 0–255 store symbol sequences of length 256, starting from positions 0 to 255 in the sliding window, with different colors representing different columns (Fig.4). Since the symbol width is 8 bits, each symbol occupies eight rows within a column. By adopting this memory layout, we can leverage column-level parallelism intra each SRAM array to parallelize sequence-level matching operations within the slide window and row-level parallelism to parallelize symbol-level operations in a sequence, thereby achieving a high degree of parallelism. As for computing steps, *OneCycleByteSearch* and *FindPos* are the atomic operations we define within the two loops based on the abovementioned memory layout. *OneCycleByteSearch* is for symbol comparisons across each column and input in one cycle, with results displayed at the bottom of columns: "1" denotes a match, while "0" indicates a miss match. The reason we limit each operation to an 8-bit comparison is twofold. 1) *Hardware Constraints.* Specifically, the prior work[23] showed that the SRAM array can only activate a

Algorithm 3. Basic In-SRAM MatchC Algorithm

Data: $symbols[]$: input sequence
$array[256 \times 8][256]$: unrolled window
$RV[256]$, $PRV[256]$: "1" for match, "0" for mismatch
Result: $pointers$, $lengths$: encoded results
1 Initialization: Refill the SRAM array by Col.
2 **for** $ColID = 0$ to 255 **do**
3 $\quad array[:][ColID] \leftarrow symbols[ColID : 255 + ColID]$
4 **end**
5 loop 1: **for** $i = 256$ to $|symbols| - 1$ **do**
6 $\quad pointer \leftarrow 0$, $length \leftarrow 0$
7 $\quad$ loop 2: **for** $R = 0$ to 255 **do**
8 $\quad\quad RV[:] \leftarrow OneCycleByteSearch(array[8R : 8(R+1)][:], symbols[i : 255 + i])$
9 $\quad\quad RV[:] \leftarrow PRV[:]$ AND $RV[:]$
10 $\quad\quad$ **if** All $RV[:] == 0$ **then**
11 $\quad\quad\quad$ Append $pointer$ to $pointers$
12 $\quad\quad\quad$ Append $length$ to $lengths$
13 $\quad\quad\quad$ break
14 $\quad\quad$ **end**
15 $\quad\quad pointer \leftarrow FindPos(RV[:])$
16 $\quad\quad length \leftarrow length + 1$
17 $\quad$ **end**
18 $\quad$ Refresh SRAM using $symbols[length : 255 + length]$
19 **end**

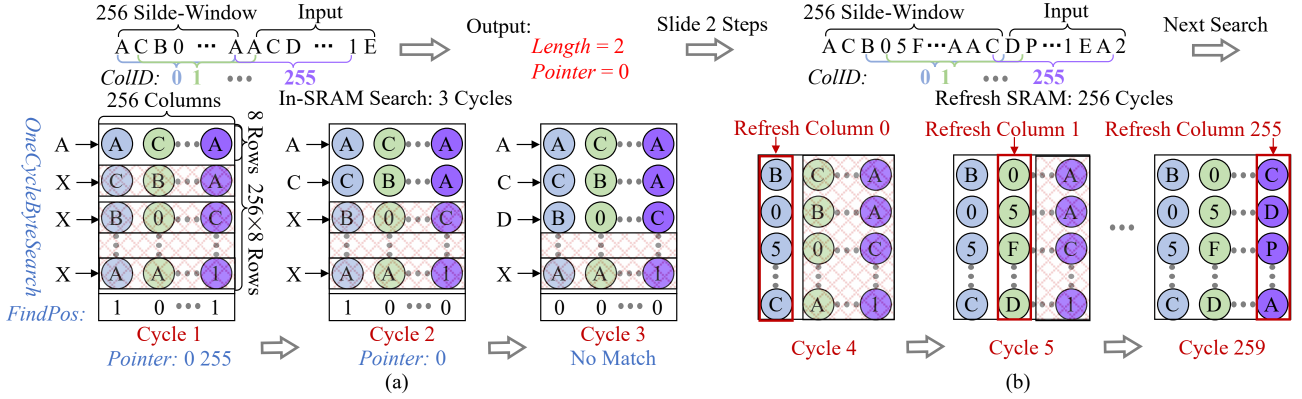

Fig.4. Example of MatchC basic design in one iteration. (a) In-SRAM searching flow. (b) Refreshing of slide-window in the SRAM array.

maximum of 64 rows simultaneously to avoid data corruption and erroneous results. 2) *Most of the Match Lengths are Short* (Fig.5). Comparing one 8-bit symbol per cycle will not bring too much cycle overhead (usually less than 10) while offering flexibility for special larger cases (such as 16-bit and 32-bit) using continuous AND operations between current and previous column results. Moreover, the *FindPos* operation identifies the column position of the rightmost "1" in the result vector of each *OneCycleByteSearch* in one cycle. If all columns return "0" (indicating no match), loop 2 terminates, with the previous *FindPos* result providing the position of the longest match. After each iteration that identifies the longest match, a data refresh is performed. This step shifts the start position of the sliding window based on the *length* and refills the SRAM array with the new windows to prepare for the next search.

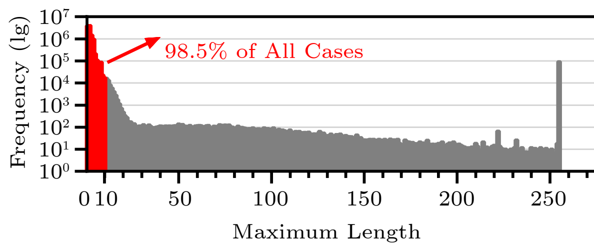

Fig.5. Distribution of length (0 to 255).

Execution Example. Fig.4(a) illustrates a searching flow. In cycle 1, each first symbol from all columns is compared with the input symbol "A", with "X" marking the rows that are not searched in this cycle, as the first and the last columns output "1", indicating that matched *ColID* is 0 and 255, respectively. In cycle 2, the input symbol "C" is performed, and the result is combined with the previous one by AND. This process continues until the results of all columns are "0", at which point the search terminates. The maximum match length (*length*) is equal to $cycle - 1$ (i.e., 2), and the position of the rightmost "1" in the previous result is selected to determine the *pointer* (i.e., 0). Finally, as shown in Fig.4(b), the sliding window shifts backwards by two symbols based on *length*, and then starts the next iteration.

3.2 Performance Analysis and Optimization

Search & Refresh Imbalance. Although the theoretical *length* can span the entire slide-window size of 256, in practice, due to local data redundancy not being as extreme, most loop 1 iterations yield a *length* concentrated within the range of [0, 10] in the genomic sequencing data compression pipeline, as shown in Fig.5. As mentioned, *length* corresponds to the number of cycles required for each iteration, meaning that the overhead of searching per iteration does not exceed ten cycles in most cases. However, Fig.4(b) illustrates that each iteration requires updating all columns in the SRAM array, resulting in 256 memory access cycles. Since each round of searching must wait for memory updates to complete, this introduces a significant imbalance. Such a disparity between computation and memory access will limit the performance.

Preload & Mask Strategy (*PMS*). A naive solution would be to preload the sliding window shifted by 0, 1, 2, ..., 255 steps across 256 SRAM arrays before the iteration begins. For each iteration, the next one would select one of these 256 arrays based on the *length* determined in the current iteration. This approach can eliminate the requirement for memory updates between iterations, reducing memory access overhead. However, it also introduces 256x additional area overhead of the SRAM array. To avoid this is-

sue, we inherit this preload idea but optimize it using a masking strategy according to our observation that adjacent columns differ by only two symbols at the beginning and end. As shown in Fig.6, we add 256 extra columns in an array to store the symbol sequences corresponding to future slide windows. In each iteration, only the results from 256 out of the 512 available columns are considered using a mask bit vector where "1" indicates a valid column and "0" indicates an invalid one. For instance, in cycle 3, if all valid column results are zero, *length* is 2, and the current iteration ends. However, we shift the mask vector two positions to the right instead of updating all columns for the next iteration as Fig.4(b). In cycle 4, the new valid 256-column data corresponds to the original sliding window shifted by two steps. This method can reduce memory refreshing cycles between iterations and achieves a better balance between computing of searching and memory access of refreshing while only doubling the SRAM array area overhead.

4 In-SRAM LutC Algorithm

4.1 Basic Design

Unified-Oriented Design. As mentioned in Subsection 2.5, our in-SRAM LutC algorithm design aims to remain unified with MatchC to improve resource utilization. Specifically, we reuse *OneCycleByteSearch* for symbol searching in the lookup table and *FindPos* for its match position, which have been defined by in-SRAM MatchC (Algorithm 3) as atomic operations. Moreover, in-SRAM LutC inherits the SRAM array dimensions constraints from MatchC optimized design, namely 256×8 rows and 256×2 columns. By adopting this unified-oriented design, we can switch between these two algorithms with only small adjustments, such as the number of arrays, data layout, and the termination condition for each iteration that can be defined without extra hardware changes.

Algorithm Description. As described in Algorithm 4, we redefine the input data of in-SRAM LutC to a tuple sequence that contains a 7-bit symbol (sym) and two symbols before it ($addr1$, $addr2$). The process of LutC uses $addr1$ and $addr2$ of input sym to index its data row ($array[addr1][addr2]$) and find the only position that matches (pos) using the above two atomic operations. A three-dimensional lookup table is required to be stored in SRAM. As a result, we adjust the number of SRAM arrays and their data layout like what Fig.7(a) illustrates: $addr1$ is used to index different SRAM arrays, while $addr2$ serves as the index for different data rows within each SRAM array. Furthermore, the bit planes of each symbol are stored row-wise, with each symbol in a data row placed column-wise in an SRAM array. As the symbol width is 7 bits, $addr1$ and $addr2$ range from 0 to 127, indicating the requirement of 128 SRAM arrays, 128 data rows in one SRAM array, and 128 symbols in one data row. We observe no data dependency between multiple input tuples, and SRAM arrays can also be performed in parallel. Therefore, multiple tuples can be

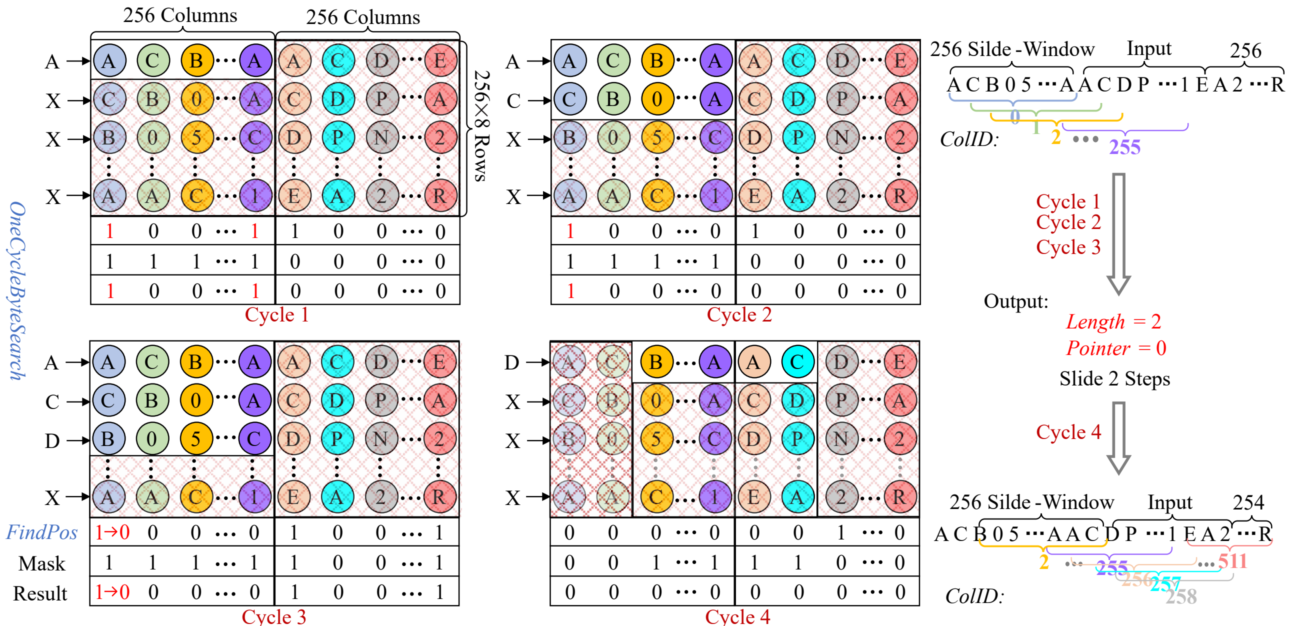

Fig.6. Optimization design of MatchC: sliding the window without refreshing SRAM before reaching the column edge.

Algorithm 4. Basic In-SRAM LutC Algorithm

Data: $tuple[](addr1,\ addr2,\ sym)$: input tuple sequence
$array[128][256\times 8][512]$: 128 tables (128KB-Array)
$busy[128]$: "1" for array busy, "0" for idle
$emit[N]$: "1" for emitting, "0" for waiting, $N=16$
$RV[256+256]$: "1" for match, "0" for mismatch
Result: $pos[]$: output position within row data

```
1 Initialization: Fill symbols to SRAM-Arrays by Row
2 loop 1: for i = 2 to |symbols| − 1, i = i + N do
3     All busy[:], emit[:] ← 0
4     loop 2: while: emit[:], is not all equal to 1 do
5         //Allocate N tuples to 128 arrays in parallel
6         Unroll: for n = i to i + N − 1 do
7             if: emit[n] AND busy[tuple[n].addr1] is 0
               then
8                 busy[tuple[n].addr1] ← 1
9                 RV[:] ← OneCycleByteSearch(
10                  array[tuple[n].addr1][tuple[n].addr2],
11                  tuple[n].sym)
12                pos[n] ← FindPos(RV[:])
13                emit[n] ← 1
14            end
15        end
16        Reset all busy[:] ← 0
17    end
18 end
```

encoded within a single iteration, i.e., multi-issue execution. For example, $tuple(0,1,1)$ and $tuple(2,1,1)$ can be searched in $array[0]$ and $array[2]$ in parallel. However, due to hardware constraints, each SRAM array can only process one tuple per iteration. If there are tuples sharing the same $addr1$, such as $tuple(3,\ 2,\ 1)$, $(3,\ 2,\ 2)$, and $(1,\ 1,\ 1)$, the first two tuples must wait until the array is not busy and cannot be performed in parallel, requiring two iterations in this case. For 128 SRAM arrays, the best-case scenario allows up to 128 tuples to be processed in one iteration, while the worst-case performs tuples one by one.

4.2 Performance Analysis and Optimization

Issue-1# : High SRAM Array Overhead. As the bit width of $addr1$ is 7-bit, 128 SRAM arrays are necessary for one LutC task. Worse still, due to the unified-oriented design, the SRAM arrays are set at 256×8 rows and 256×2 columns, but the lookup table for LutC only occupies a small portion of each SRAM array. Fig.7(a) shows that the utilization ratio of each array is only 12.5%.

Issue-2# : Load Imbalanced Inter-SRAM-Arrays. As shown in the left of Fig.7(a), we use different colors to represent the number of tuples each array needs to process over a given period. A deeper shade of red indicates a higher workload, while green represents idle periods. The load imbalance issue is serious because of the distribution of quality score symbols in genomic sequencing data (Fig.8).

Multi-Copies Strategy (MCS). As described in Fig.7(b), this approach involves duplicating the 16 "red" SRAM arrays based on the data distribution. While this strategy can alleviate issue-2 by spreading the tasks from 16 "red" arrays to 32, it worsens issue-1 due to the increase of arrays (from 128 to 144).

Array-Combined Strategy (ACS). As shown in the right of Fig.7(c), we combine the lookup table data originally distributed across 128 arrays into fewer arrays, e.g., 16. This strategy overcomes issue-1 by filling each array with valid data but worsens issue-2 due to loss of the inter SRAM arrays parallelism.

Hybrid Solution: Scheduled Array-Combined Strategy. 1) *Exploiting Row-Level Parallelism Intra SRAM Array.* For issue-1, we first inherit the combination approach from ACS. However, we exploit the row-level parallelism within each SRAM array to compensate for the parallelism lost by $addr1$ combination, as shown in the right of Fig.7(d). Specifically, when multiple input tuples share the same $addr1$, sym, and have $addr2$ values within the range of $[4N,\ 4(N{+}1){-}1]$, they can be processed in parallel, i.e., one-cycle search for four $addr2$ values in the same row. 2) *Rearranging Data Inter Different Arrays.* We then spread 16 hotspot $addr1$ data blocks in the range [28, 43] (the left of Fig.7(d)) to 16 SRAM arrays, following the old and new $addr1$ mapping rule described in (1) to alleviate the workload imbalance. 3) *Transposing Array.* To further enhance parallelism of intra-SRAM-Array, we transpose the array by swapping rows and columns to improve parallelism from 4 to 16.

$$newID=\begin{cases}96, & \text{if } oldID=33,\\ (oldID-28)\times M, & \text{if } 27<oldID\le 43\ \& \\ & oldID\ne 33,\\ \dfrac{oldID}{M}+28, & \text{if } oldID \bmod M=0,\\ oldID, & \text{otherwise,}\end{cases} \tag{1}$$

where M is the number of data blocks in each array ($M=8$).

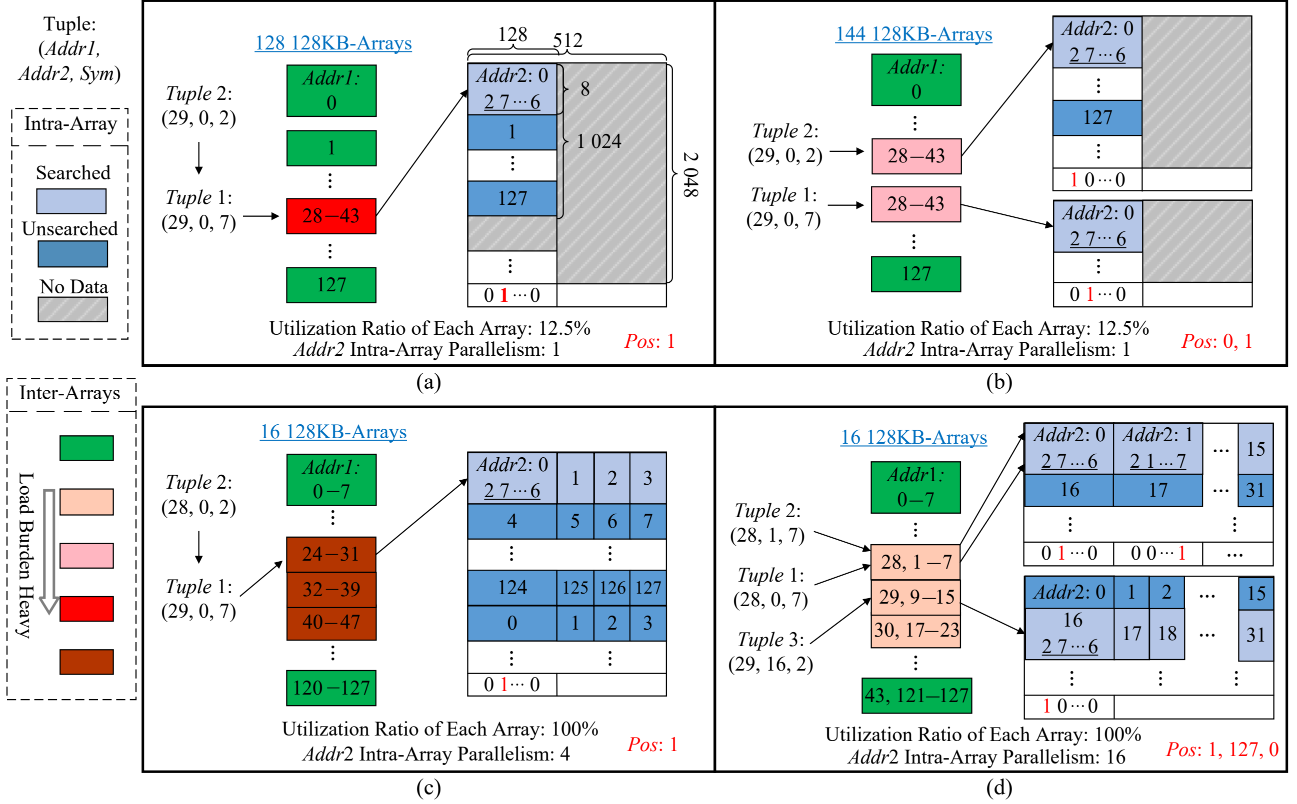

Fig.7. Four different in-SRAM LutC designs. (a) Basic design. (b) Multi-copies strategy (MCS). (c) Array-combined strategy (ACS). (d) Hybrid solution: scheduled array-combined strategy.

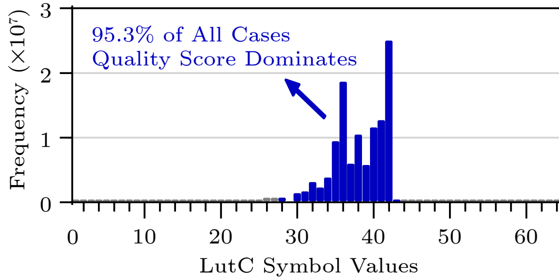

Fig.8. Distribution of quality score symbols.

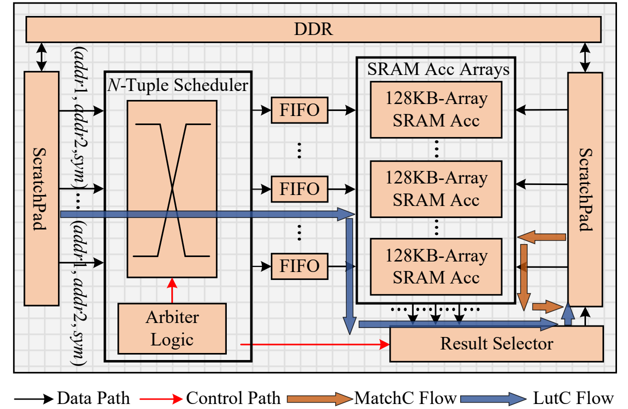

Fig.9. Overview of iSCoder hardware architecture.

5 iSCoder In-SRAM Architecture

5.1 Overview

The overview of our architecture is illustrated in Fig.9. The hallmark of the unified-oriented design is the two different execution flows sharing the same 128KB-Array SRAM Acc.

MatchC Flow. First, the input data is read from the DDR and entered into the scratchpad. Each 128KB-Array SRAM is then initialized using the basic memory layout and optimized preload design (PMS) described in Section 3 to store the sliding-window data. Next, an input sequence of 256 symbols is searched for its result of $(length, pointer)$ within the current sliding window. Furthermore, this result is then fetched by the result selector module, written back to the scratchpad, and transferred to the DDR.

LutC Flow. This flow is more complex than that of MatchC. First, each 128KB-Array is initialized using the hybrid solution mentioned in Subsection 4.2 to store the lookup table data. As shown in Fig.9, several input tuples are initially read from the DDR into the scratchpad. Then, N input tuples are allocated in parallel to 16 128KB-Array SRAMs based on the N-tuple scheduler module. Finally, the positions

found in each array are filtered and obtained by the result selector module, written to the scratchpad and also subsequently transferred back to the DDR.

5.2 128KB-Array SRAM Module

This hardware module is the main execution unit of our in-SRAM design, requiring one array for one MatchC processing engine (PE) and 16 arrays for one LutC PE. Based on the in-SRAM CAM described in Fig.3, our 128KB-Array SRAM module adds several additional logic circuits to implement in-SRAM algorithms proposed in Section 3 and Section 4, including the row driver module and the peripheral design.

Row Driver. The function of this module is to convert the input symbol sequence and its address into the signals that SRAM array rows can identify. We define a mapping table between input and signals that CAM rows need, as shown in Fig.10. 1) Input bit "1" is mapped to "10" while input bit "0" is mapped to "01", of which the first bit 0 is connected to BL and the second bit "1" is connected to BLB. The Interpreter submodule is responsible for this mapping. 2) To set other rows to the don't-care mode when *OneCycleByteSearch* is performed, "X" is necessary and is mapped to "00", i.e., both BL and BLB lines are deactivated. This is implemented by the address decoder submodule and the mask AND circuit. The address decoder decodes the input address into a vector with "1"s at the required position of rows while all the other positions remain "0". The mask AND circuit then performs a bitwise AND operation between the decoded output and the already mapped input data. For example, if the addr is 0 and the input is "00001001", they will be mapped to "1111111111111110...00" and "01/01/01/01/10/01/01/10...00", respectively, and only the first eight rows are activated and participate in the CAM search,

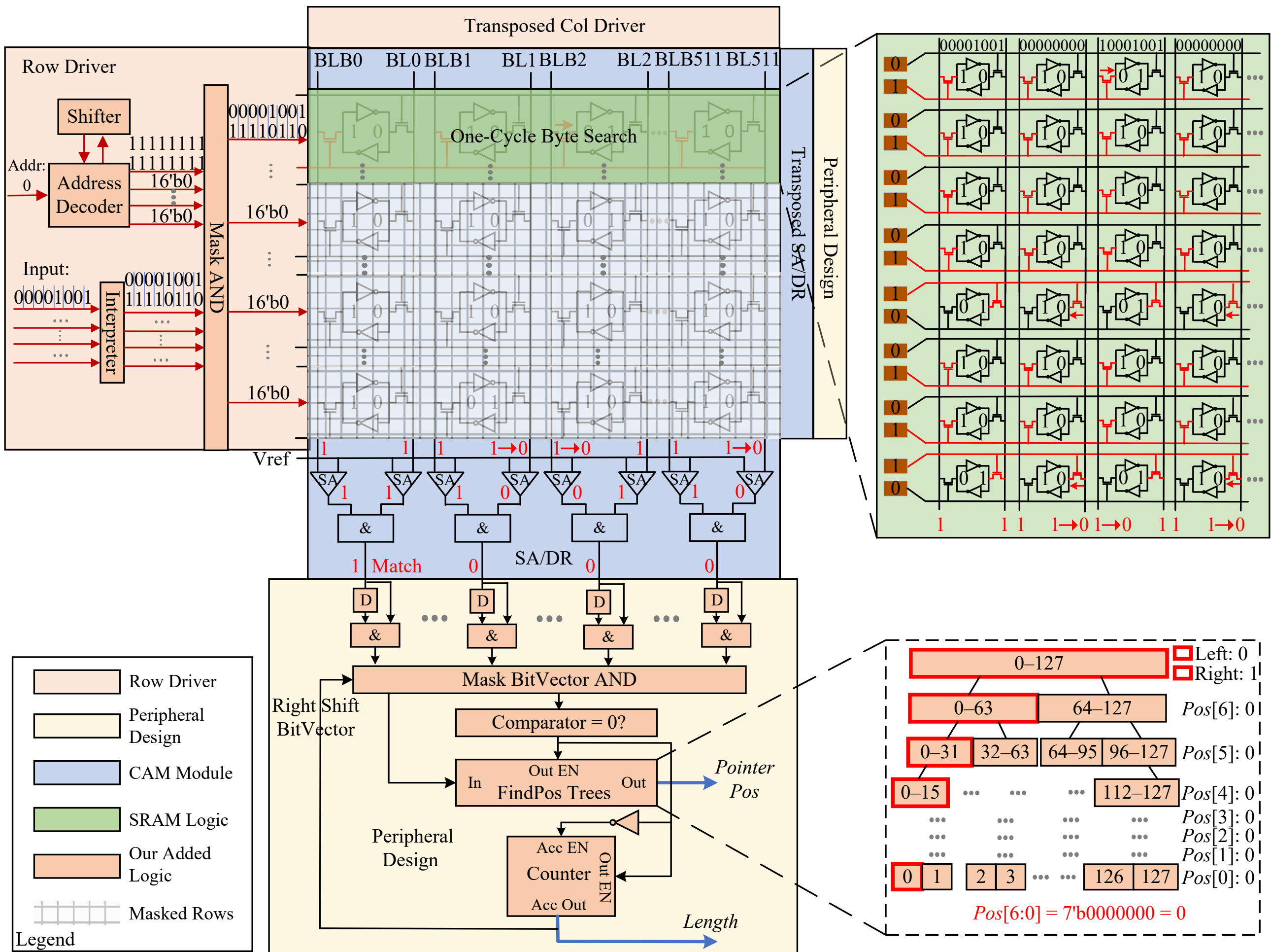

Fig.10. Micro-architecture of SRAM-Array Acc. The design primarily consists of three components: basic in-SRAM CAM, peripheral design, and row driver. Input 10 represents "1", 01 represents "0", 00 represents "X" (do not care), 11 represents "w/r" (write/read). This figure shows a search operation on rows 1 to 8 in the SRAM (first row symbol) for each column, and there is only the first column matching, i.e. $Pos[6:0]$ = 7'b0000000 = 0.

completing *OneCycleByteSearch* on the first eight rows. 3) "w/r" is mapped to "11" for writing or reading.

Peripheral Design. This module generates results of MatchC and LutC. The in-SRAM CAM design in Fig.3 includes a group of AND gates between the outputs of two SAs, where "1" represents a match and "0" represents a miss match. Based on this design, we introduce how our peripheral design works. 1) *D Flip-Flop and AND for Each Column.* This submodule is added to overcome the hardware constraint that an array can only activate a maximum of 64 rows simultaneously, as mentioned in Subsection 3.1 since D flip-flop can save the result of the previous cycle. 2) *FindPos Tree Circuit.* This submodule is responsible for the *FindPos* atomic operation for MatchC and LutC to find the position of "1" by a tree combinational logic. 3) *Comparator and Counter.* The two circuits are used to judge the termination conditions and record the cycle count of the current iteration as the result of the *length* for MatchC. 4) *Mask bitVector AND.* This submodule is designed to determine the valid columns for slide windows, implementing the preload & mask strategy.

5.3 N-Tuple Scheduler

This module allocates the input N tuples of LutC to 16 128KB-array SRAM accelerators (SRAM Accs) based on the principle of parallel execution priority. The circuit is divided into two parts: the Crossbar and the Arbiter Logic. The former handles the interconnections between the inputs and SRAM Accs, while the latter uses combinational logic to implement the condition-checking logic required by the LutC algorithm to determine the on/off states of the crossbar. For example, assuming N is 4, with the input tuples being (31, 35, 35), (35, 35, 35), (35, 35, 37), and (35, 37, 37), $tuple(31, 35, 35)$ will be assigned to the Acc-3, while the remaining three tuples will be assigned to the Acc-7. Furthermore, since $tuple(35, 35, 37)$ and $tuple(35, 37, 37)$ have the same sym and their $addr2$ values are in the same column (i.e., the values obtained by dividing $addr2$ by 16 and rounding down are both 2), considering the parallelism of the column (16, transposed by the row), they are scheduled to execute in parallel. In contrast, $tuple(35, 35, 35)$ must wait and be assigned to the next cycle. In our design, the value of N is set to 16. We will evaluate the influence of N in Section 6.

5.4 Result Selector

This module collects and selects the output results and writes them to the scratchPad. Specifically, the MatchC flow uses the completion signal of each iteration as a trigger to receive the *length* and *pointer* from the 128KB-array SRAM Accs, and then combines them and writes the result into the scratchPad. The LutC flow needs to receive the allocation information from the arbiter logic at each clock cycle to determine which SRAM Acc's output is valid and which tasks within the SRAM Acc are executed in parallel in the same SRAM array. It then extracts the required portion from the output set of positions and writes the valid positions to the location in the scratchPad.

6 Evaluation and Results

6.1 Methodology

Performance of iSCoder. We implement and verify the above modules using Verilog and SystemVerilog, which are then synthesized by the open-source EDA tool OpenROAD[24] performing place-and-route on a FreePDK 45 nm[25] process technology at the worst-case process-voltage-temperature (PVT) corner to obtain practical estimates of delay, power consumption, and area. We set the target frequency of iSCoder to 500 MHz. For SRAM, we simulate the access delay and energy that are obtained from CACTI 7.0[26], while area and power are estimated with OpenRAM[27]. Based on these results, we design a cycle-accurate simulator in C++ that takes into account the execution order, dependencies, and cycles for SRAM and auxiliary hardware modules, yielding the final simulated performance. The input data is preloaded into DDR and the output data is the same as that of the software therefore no variance in compression ratio.

Baselines. As iSCoder is the first work to accelerate the MPEG-G workflow to the best of our knowledge, we use the open-source software project Genie as the baseline for performance comparison. Genie runs on a 72-core Intel® Xeon® Gold 6354 Processor system with a 3.0 GHz clock speed. We adjust the number of threads to achieve optimal software performance based on the input data size. Additionally, we evaluate other hardware baselines: our basic in-SRAM designs (basic, multi-basic) and naive optimization strategies (MCS, ACS) for MatchC and LutC described in Section 3 and Section 4, respectively for comparison with iSCoder. As shown in Table 1, each

Table 1. Configuration of iSCoder in Two Modes

Parameter	Arrays per PE	Tuple Number (N)	Slide-Window Size	Symbol Size (bits)	Arrays per PE (Basic)
MatchC mode	1	/	256 byte	8	1
LutC mode	16	16	/	7	128

Note: The same configuration in both modes: row number: 256×8; column number: 256+256; array size: 128 KB; array number: 256; column number (basic): 256; target frequency: 500 MHz.

MatchC mode PE has one SRAM array, while each LutC mode PE has 16 SRAM arrays.

Datasets. As shown in Table 2, we use real Illumina sequencing data from eight species as test datasets, covering humans, animals, plants, and microorganisms, sourced from a common database[28]. We specially select Fastq data spanning different sizes, from 720 MB to 51 GB to evaluate the impact of data size especially the number of blocks on performance. Moreover, based on the human genome (human_g1k_v37_decoy.fasta), we use DWGSIM① to simulate eight Fastq files with varying error rates and another eight with different read lengths to analyze the sensitivity of iSCoder. We also test two citrus reticulata files sourced from different sequencing technologies including PacBio (SRR24601242, 57 GB) and ONT (SRR22063092, 12 GB).

Table 2. Eight Different Benchmark Datasets of Fastq

Database	Species	Size (GB)	Number of Blocks
DB0	Homo sapiens	51.00	203
DB1	Citrus reticulata	22.00	89
DB2	Pinus taeda	6.90	22
DB3	Camelus dromedarius	39.00	122
DB4	Triticum dicoccoides	6.90	27
DB5	Escherichia coli	0.72	2
DB6	Pseudotsuga menziesii	1.90	12
DB7	Venustaconcha ellipsiformis	50.00	191

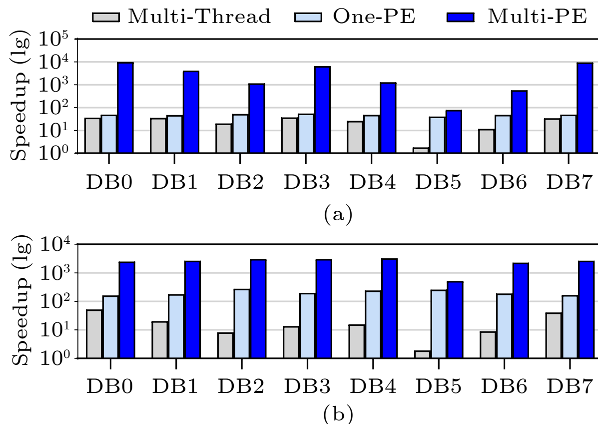

Fig.11. Overall speedup of our In-SRAM design. Results are normalized to that of a single-thread CPU. (a) Speedup of in-SRAM MatchC. (b) Speedup of in-SRAM LutC. In the MatchC mode, one-PE corresponds to a single SRAM array, while the number of SRAM arrays for the multi-PE depends on the number of Fastq data blocks. In the LutC mode, one-PE corresponds to a set of 16 SRAM arrays, with the number of SRAM arrays for the multi-PE being a multiple of 16. Each SRAM array is configured with 512 columns, including 256 additional columns, and the N in the N-Tuple scheduler is set to 16, as shown in Table 1.

6.2 Overall Results

Speedup Compared with Software. Figs.11(a) and 11(b) illustrate the speedup of MatchC and LutC, respectively. 1) For MatchC, iSCoder's multi-PE achieves an average speedup of 131x over the multi-thread software baseline across eight DBs. Even one-PE is also faster than multi-thread software (4.5x). 2) For LutC, the average speedup of multi-PE over multi-thread is 191x, with the one-PE achieving 29x. DB5 is a special case where both multi-thread and multi-PE perform significantly lower than other DBs. This is because each block is performed in parallel, but it has the fewest blocks (only 2, as shown in Table 2). When considering the latency of compressing a Fastq file, this parameter limits multi-PE's performance over multi-thread software, whereas the one-PE is unaffected. Moreover, the multi-PE performance of MatchC is noticeably lower than average on DB6, while LutC does not show this effect due to differences in utilization rates between MatchC and LutC multi-PEs on DB6. Specifically, each MatchC PE only requires one SRAM array, while LutC requires 16. Table 1 shows that the total number of arrays is 256, meaning that the maximum PE count for MatchC and LutC multi-PEs is 256 and 16, respectively. With 12 blocks in DB6, LutC can utilize 75% of its PEs, while MatchC can only utilize 4.7%. For other DBs, such as DB7 with 191 blocks, MatchC PE utilization rises from 4.7% to 74.6%, while LutC PE utilization only increases from 75% to 100%, resulting in different magnitudes of improvement. Finally, we estimate the speedup of complete sequencing compression

①https://github.com/nh13/DWGSIM, Nov. 2025.

workflows across eight different databases (DBs) under two configuration modes. In our setup, MatchC and LutC are processed using iSCoder, while entropy coding is implemented in hardware near SRAM, with the remaining workflows executed on the CPU. As illustrated in Fig.12, 1) the speedup ranges from 1.4x to 4.5x, depending on the workflows that are not accelerated such as descriptors generation (DG) and data movement, and 2) the speedup achieved in the reorder mode is lower than in the low-latency mode. This is primarily due to the inclusion of global assembly for SEQ in the reorder mode, which increases the execution overhead of the DG stage.

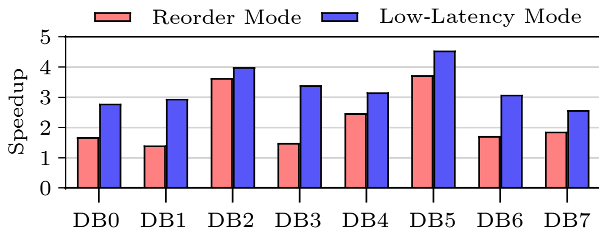

Fig.12. Speedup of complete compression workflows.

Compared with Hardware. We compare iSCoder with an FPGA-based hardware accelerator (Repaq[29]) designed for the genomic sequencing compression algorithm, and an ASIC-based hardware accelerator (BeeZip[21]) proposed for general-purpose compression algorithm. As shown in Table 3, iSCoder achieves the highest compression ratio (21.51x) while consuming significantly less power than Repaq. Although its higher compression ratio is at the cost of lower throughput compared with BeeZip, it can achieve real-time genomic sequencing data compression[29].

Power and Area. Table 4 shows iSCoder's estimated power and area, with total values of 13.82 W and 63.05 mm^2, respectively, and additional logic area accounting for nearly 15%. Moreover, the average power reduction for MatchC and LutC is 38x and 40x, respectively. The SRAM's power consumption is determined by static and dynamic power from activating rows and columns.

Table 3. Comparison with Other Compression Accelerators

Accelerator	Throughput (GB/s)	Power (W)	C. Ratio
Repaq[29]	0.105	225.00	10.71
BeeZip[21]	6.330	1.99	4.93
iSCoder	0.109	13.82	21.51

Note: Evaluated on Homo sapiens (NA12878). C.Ratio: compression ratio.

Table 4. Estimated Power and Area Result

Component	Power (W)	Area (mm^2)
Logic	6.50	9.77
SRAM	7.32	53.28
Total	13.82	63.05

6.3 Optimization Analysis

MatchC Mode. Fig.13 shows the impact of different hardware designs on speedup and memory access proportion relative to the multi-thread software baseline. We can derive two conclusions. 1) The imbalance between computation and memory access is widespread and significant in genomic MatchC algorithms. In the basic design, memory proportions exceed 90% for datasets DB0 through DB7 while computation proportions are less than 10%. And this imbalance results in the single-PE basic hardware design performing up to 10x slower than the multi-thread baseline. 2) Our optimization works. The PMS design yields over two orders of magnitude improvement in single-PE performance compared with the basic design, while the multi-PMS achieves a speedup of 45x to 280x over the multi-threaded baseline. Differences in speedup across datasets result from variations in the number of parallelizable data blocks. With greater data parallelism in our in-SRAM design (256 vs 72 on the CPU), speedup scales significantly when blocks exceed 72.

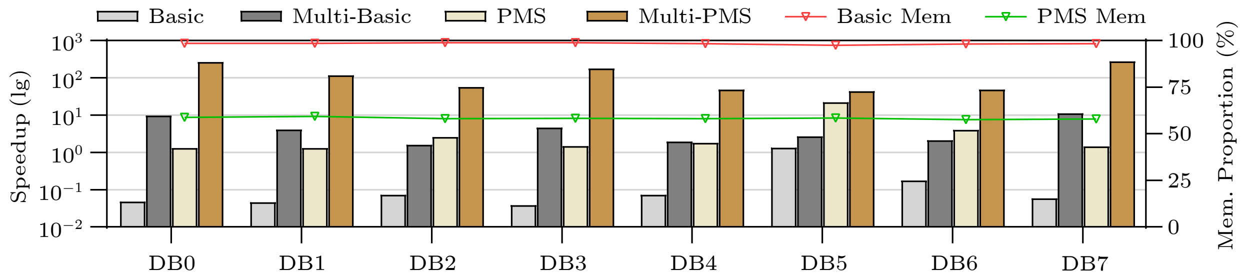

Fig.13. Speedup of the MatchC optimization and proportion of memory access. Basic: single PE of the basic design in the MatchC mode. Multi-Basic: 256 PEs of the basic design. PMS: single PE of the preload & mask strategy optimized design. Multi-PMS: 256 PEs of the PMS design. Basic Mem: the memory access proportion of the basic design. PMS Mem: the memory access proportion of the PMS optimized design.

LutC Mode. Fig.14 shows the speedup compared with the single-issue baseline and collision rate of six different LutC basic and optimized designs. We can draw three conclusions. 1) The load imbalance issue is prevalent and significant, as evidenced by the collision proportion ranging from 34% to 89% across all of the test datasets. 2) The performance of the multi-issue design is limited by the load imbalance issue. As the collision proportion increases, the speedup of the basic design decreases (e.g., from DB6 to DB5). 3) Our hybrid solution works, and each of the three strategies within it can improve performance. The performance improvement of DB2 is particularly significant because it has a large number of identical symbols (high collision rate). On the one hand, severe load imbalance can provide more room for the second strategy of the hybrid solution to improve the performance. On the other hand, similar input tuples can better benefit from the additional intra-array parallelism brought by the hybrid-1.

6.4 Design Space Exploration

In this subsection, we try to make tradeoff between speedup and area on several parameters respectively, including additional SRAM array column number, the scheduled tuples number in parallel and SRAM array number. The evaluated data is obtained on DB0 (Homo sapiens), as shown in Fig.15.

Number of Additional SRAM Columns. In Subsection 3.2, we proposed a column-shifting method to address the imbalance between computation and memory access in in-SRAM MatchC processing, achieved by increasing the SRAM array width (adding additional columns). Theoretically, this increase in column count has two effects. 1) The count of SRAM ar-

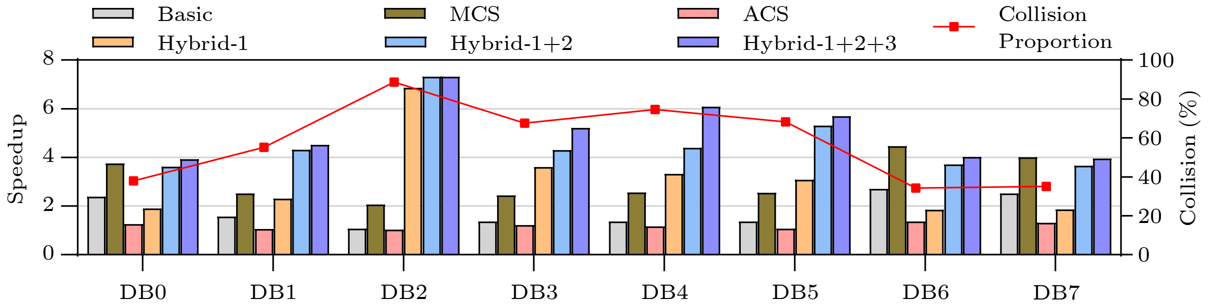

Fig.14. Speedup compared with the single-issue baseline and collision rate of different LutC optimizations. Basic: the basic multi-issue design in the LutC mode. MCS: the multi-copies strategy optimized design. ACS: the array-combined strategy optimized design. Hybrid-1: the optimized design using the first strategy of the hybrid solution described in Subsection 4.2. Hybrid-1+2: the optimized design using the first strategy and the second strategy of the hybrid solution. Hybrid-1+2+3: the optimized design using all of the three strategies of the hybrid solution. Collision proportion: the probability that adjacent input tuples share the same *addr*1, and higher collision proportion values indicate greater load imbalance for each array.

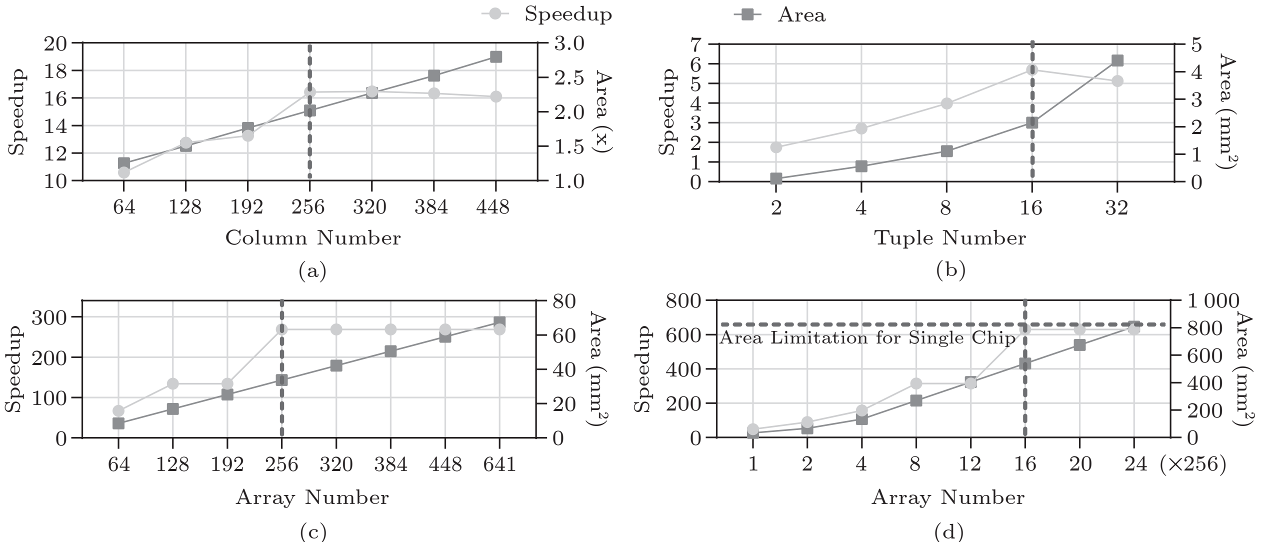

Fig.15. Tradeoff between speedup and area on (a) the number of additional SRAM array columns, (b) the number of scheduled tuples in parallel, and the number of SRAM arrays in (c) the MatchC mode and (d) the LutC mode.

ray refreshes decreases. For instance, with no additional columns, the array must be refreshed according to the match with the maximum length. However, with 64 extra columns, assuming each maximum match is of length 2, refreshes are needed only once every 32 iterations. 2) Execution time for each iteration increases. Since each array refresh involves writing new data column-by-column, a higher column count increases the count of cycles. These two factors have opposing effects on performance, and we conduct experiments to analyze their influence on performance and memory access proportion, as shown in Fig.15(a). The results indicate that the speedup peak occurs with 256 additional columns. While performance remains high at 320 columns, it incurs unnecessary area and power overhead due to increased SRAM size. Thus, we select 256 columns as the tradeoff parameter, yielding an SRAM array size of 128 KB.

Number of Parallel Tuples. This parameter, as described in Subsection 5.3 when discussing the N-tuple scheduler hardware module, represents the maximum number of tuples iSCoder can schedule at once in the LutC multi-issue mode. These tuples are then processed by the 128KB-SRAM array accelerator for encoding. Given the iSCoder's low latency, multi-level parallelism, and optimized solutions for the load imbalance issue, performance should increase as N grows, as shown in Fig.15(b). However, since the scheduler module relies on a crossbar, a larger N increases the size of the crossbar logic and the complexity of arbitration (mapping N tuples to 16 arrays), not only resulting in extra area requirements, but the delay of the scheduler also increases, where a value of $N = 32$ even reduces the speedup. As a result, we select 16 for N.

Number of SRAM Arrays. As this parameter increases, iSCoder scales to larger configurations, leading to higher area overhead. However, we observe that the speedup does not always grow linearly. Let us take the commonly used DB0 (203 blocks) as a specific example. 1) For MatchC, each PE consists of a single array. As shown in Fig.15(c), the area scales nearly linearly with the number of SRAM arrays. However, the maximum speedup is achieved around 256 arrays because each PE can process at most one atomic block ($203/256 < 1$) under parallel execution. 2) For LutC, each PE comprises 16 arrays, shifting the inflection point to 4 096 ($256\times16 = 4\,096$). Although further increasing the number of SRAM arrays can continue improving speedup for databases with more blocks, the area eventually reaches the limits of a single chip, as illustrated in Fig.15(d).

6.5 Sensitivity Analysis

Error Rate Sensitivity. Fig.16(a) shows the impact of error rates on the speedup. For MatchC, which processes read IDs, error rates have minimal effect (only 5% speedup variation) since errors rarely occur in this field. The slight variation stems from other little segments from SEQ. In contrast, LutC's performance varies significantly (up to 45%) because

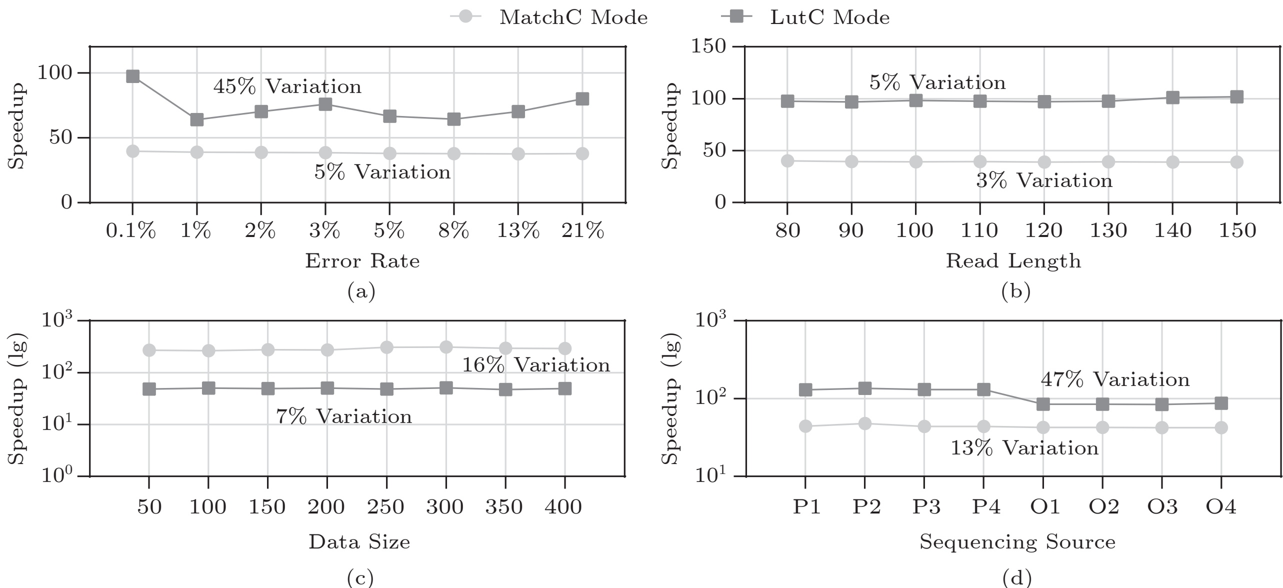

Fig.16. Speedup of different sequencing (a) error rate, (b) read length, (c) data size (GB), and (d) data from other sources (P: PacBio, O: ONT).

it works mainly on the quality score (QS) field. Lower error rates lead to higher QS values; for example, at a 0.1% error rate, QS values cluster around 40 (within [28, 43]), allowing optimal distribution of 16 lookup tables across 16 SRAM arrays.

Read Length Sensitivity. Read length is another parameter in Fastq data, and Fig.16(b) illustrate its impact on the speedup of MatchC and LutC. Similar to error rate sensitivity, the influence of read length on MatchC is minimal (3%) due to delta encoding applied to the ID field before MatchC processing. Additionally, since iSCoder is a segmented, parallel, streaming encoder, variations in QS length due to read length changes have a limited effect on speedup (5% between the maximum and the minimum compared with the average).

Data Size Scaling. We evaluate the scalability of iSCoder on larger datasets by processing eight Homo sapiens sequencing Fastq files ranging from 50 GB to 400 GB. The accelerator is configured according to Table 1. As shown in Fig.16(c), the speedup of iSCoder remains relatively stable in both MatchC and LutC modes. This is because these two algorithms are invoked per block in MPEG-G, and each block typically has a constant and relatively small size.

Other Sources. We further evaluate iSCoder on genomic sequencing data from two additional sources beyond Illumina, including four PacBio files (each has one block) sampled from citrus reticulata (SRR24601242, 57 GB) and four ONT files (also one block) sampled from Citrus reticulata (SRR22063092, 12 GB). As shown in Fig.16(d), the speedup in the MatchC mode remains stable across different sources. This is because MatchC operates on descriptors generated from the SEQ and ID fields, which are less dependent on the sequencing technology. However, the speedup varies for LutC due to its primary focus on QS, whose encoding schemes differ across sources.

7 Related Work

Compression Accelerator. Designing the hardware accelerators for data compression is a common approach and can be divided into general-purpose[11, 19–21] and domain-specific[13, 14, 29] types. As an FPGA-based genomic sequencing data compression accelerator, Repaq[29] implements another sequencing data compression algorithm instead of MPEG-G using FPGA, achieves similar throughput but lower compression ratio than iSCoder. As general-purpose accelerators, Abali *et al.*[11] proposed a hardware accelerator integrated into its POWER9 and z15 processors for GZIP, which combines the near-history and the far-history CAM to handle character matching for LZ77[22]. This design achieves a 388x and a 13x speedup over a single core and the full processor chip, respectively. However, the far-history CAM relies on a hash-based pseudo-CAM with lower performance, and the high-performance near-history CAM uses a comparator array based on registers with high hardware overhead, which limits its size. In contrast, iSCoder employs an in-SRAM-based CAM, providing larger data storage capacity and higher parallelism.

In-SRAM Accelerator. In recent years, several accelerators based on in-SRAM computing have emerged, such as Compute Cache[15], Neural Cache[16], and Gencache[17]. Compute Cache repurposes existing cache arrays into active, large-scale vector computation units to perform various basic operations (ISA). Additionally, Neural Cache and Gencache integrate in-cache designs for operators in deep neural networks and sequence alignment algorithms, respectively. Similar to these designs, our work utilizes bit-line SRAM circuit technology and achieves acceleration. However, iSCoder implements and optimizes two new algorithms instead of simple operations using the in-SRAM technology and focuses not only on intra-SRAM-Array but also on scheduling inter-Arrays, distinguishing it from previous approaches.

8 Conclusions

This paper proposed a genomic sequencing data compression accelerator iSCoder to mitigate MPEG-G pipeline bottlenecks via in-SRAM computing. We identified and analyzed MatchC and LutC as two bottleneck algorithms in this pipeline, proposed two optimized in-SRAM algorithms, and designed a unified-oriented hardware architecture for these two algorithms, considering the characteristics of genomic data. Compared with 72-core Intel processors running at 3.0 GHz, experimental results show that iSCoder achieves an average speedup of 131x for MatchC and 191x for the LutC, respectively.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Turakhia Y, Bejerano G, Dally W J. Darwin: A genomics

co-processor provides up to 15,000X acceleration on long read assembly. *ACM SIGPLAN Notices*, 2018, 53(2): 199–213. DOI: 10.1145/3296957.3173193.

[2] Cock P J A, Fields C J, Goto N, Heuer M L, Rice P M. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, 2010, 38(6): 1767–1771. DOI: 10.1093/nar/gkp1137.

[3] Hernaez M, Pavlichin D, Weissman T, Ochoa I. Genomic data compression. *Annual Review of Biomedical Data Science*, 2019, 2(1): 19–37. DOI: 10.1146/annurev-biodatasci-072018-021229.

[4] Stephens Z D, Lee S Y, Faghri F, Campbell R H, Zhai C, Efron M J, Iyer R, Schatz M C, Sinha S, Robinson G E. Big data: Astronomical or genomical? *PLoS Biology*, 2015, 13(7): e1002195. DOI: 10.1371/journal.pbio.1002195.

[5] Numanagić I, Bonfield J K, Hach F, Voges J, Ostermann J, Alberti C, Mattavelli M, Sahinalp S C. Comparison of high-throughput sequencing data compression tools. *Nature Methods*, 2016, 13(12): 1005–1008. DOI: 10.1038/nmeth.4037.

[6] Voges J, Hernaez M, Mattavelli M, Ostermann J. An introduction to MPEG-G: The first open ISO/IEC standard for the compression and exchange of genomic sequencing data. *Proceedings of the IEEE*, 2021, 109(9): 1607–1622. DOI: 10.1109/JPROC.2021.3082027.

[7] Müntefering F, Adhisantoso Y G, Chandak S, Ostermann J, Hernaez M, Voges J. Genie: The first open-source ISO/IEC encoder for genomic data. *Communications Biology*, 2024, 7(1): Article No. 553. DOI: 10.1038/s42003-024-06249-8.

[8] Chandak S, Tatwawadi K, Ochoa I, Hernaez M, Weissman T. SPRING: A next-generation compressor for FASTQ data. *Bioinformatics*, 2019, 35(15): 2674–2676. DOI: 10.1093/bioinformatics/bty1015.

[9] Jones D C, Ruzzo W L, Peng X, Katze M G. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic Acids Research*, 2012, 40(22): e171. DOI: 10.1093/nar/gks754.

[10] Li H, Durbin R. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 2010, 26(5): 589–595. DOI: 10.1093/bioinformatics/btp698.

[11] Abali B, Blaner B, Reilly J, Klein M, Mishra A, Agricola C B, Sendir B, Buyuktosunoglu A, Jacobi C, Starke W J, Myneni H, Wang C. Data compression accelerator on IBM POWER9 and z15 processors: Industrial product. In *Proc. the 47th Annual International Symposium on Computer Architecture (ISCA)*, May 29-Jun. 3, 2020, pp.1–14. DOI: 10.1109/ISCA45697.2020.00012.

[12] Peng B, Ding D, Zhu X, Yu L. A hardware CABAC encoder for HEVC. In *Proc. the 2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2013, pp.1372–1375. DOI: 10.1109/ISCAS.2013.6572110.

[13] Snigdha F S, Sengupta D, Hu J, Sapatnekar S S. Optimal design of JPEG hardware under the approximate computing paradigm. In *Proc. the 53rd Annual Design Automation Conference*, Jun. 2016, Article No. 106. DOI: 10.1145/2897937.2898057.

[14] Huang Y W, Wang T C, Hsieh B Y, Chen L G. Hardware architecture design for variable block size motion estimation in MPEG-4 AVC/JVT/ITU-T H. 264. In *Proc. the 2003 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2003. DOI: 10.1109/ISCAS.2003.1206094.

[15] Aga S, Jeloka S, Subramaniyan A, Narayanasamy S, Blaauw D, Das R. Compute caches. In *Proc. the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2017, pp.481–492. DOI: 10.1109/HPCA.2017.21.

[16] Eckert C, Wang X, Wang J, Subramaniyan A, Iyer R, Sylvester D, Blaaauw D, Das R. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proc. the 45th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2018, pp.383–396. DOI: 10.1109/ISCA.2018.00040.

[17] Nag A, Ramachandra C N, Balasubramonian R, Stutsman R, Giacomin E, Kambalasubramanyam H, Gaillardon P E. GenCache: Leveraging in-cache operators for efficient sequence alignment. In *Proc. the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2019, pp.334–346. DOI: 10.1145/3352460.3358308.

[18] Heirman W, Carlson T, Eeckhout L. Sniper: Scalable and accurate parallel multi-core simulation. In *Proc. the 8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*, Jul. 2012, pp.91–94. DOI: 10.1145/2063384.2063454.

[19] Qiao W, Du J, Fang Z, Lo M, Chang M C F, Cong J. High-throughput lossless compression on tightly coupled CPU-FPGA platforms. In *Proc. the 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 29–May 1, 2018, pp.37–44. DOI: 10.1109/FCCM.2018.00015.

[20] Choi S, Kim Y, Song Y H. False history filtering for reducing hardware overhead of FPGA-based LZ77 compressor. *Journal of Systems Architecture*, 2018, 88: 110–119. DOI: 10.1016/j.sysarc.2018.06.001.

[21] Gao R, Li Z, Tan G, Li X. BeeZip: Towards an organized and scalable architecture for data compression. In *Proc. the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, Apr. 27–May 1, 2024, pp.133–148. DOI: 10.1145/3620666.3651323.

[22] Ziv J, Lempel A. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 1977, 23(3): 337–343. DOI: 10.1109/TIT.1977.1055714.

[23] Jeloka S, Akesh N B, Sylvester D, Blaauw D. A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6T bit cell enabling logic-in-memory. *IEEE*

Journal of Solid-State Circuits, 2016, 51(4): 1009–1021. DOI: 10.1109/JSSC.2016.2515510.

[24] Ajayi T, Chhabria V A, Fogaça M, Hashemi S, Hosny A, Kahng A B, Kim M, Lee J, Mallappa U, Neseem M, Pradipta G, Reda S, Saligane M, Sapatnekar S S, Sechen C, Shalan M, Swartz W, Wang L, Wang Z, Woo M, Xu B. Toward an open-source digital flow: First learnings from the OpenROAD project. In *Proc. the 56th Annual Design Automation Conference 2019*, Jun. 2019, Article No. 76. DOI: 10.1145/3316781.3326334.

[25] Stine J E, Castellanos I, Wood M, Henson J, Love F, Davis W R, Franzon P D, Bucher M, Basavarajaiah S, Oh J, Jenkal R. FreePDK: An open-source variation-aware design kit. In *Proc. the 2007 IEEE International Conference on Microelectronic Systems Education*, Jun. 2007, pp.173–174. DOI: 10.1109/MSE.2007.44.

[26] Balasubramonian R, Kahng A B, Muralimanohar N, Shafiee A, Srinivas V. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Trans. Architecture and Code Optimization (TACO)*, 2017, 14(2): Article No. 14. DOI: 10.1145/3085572.

[27] Guthaus M R, Stine J E, Ataei S, Chen B, Wu B, Sarwar M. OpenRAM: An open-source memory compiler. In *Proc. the 35th International Conference on Computer-Aided Design*, Nov. 2016, Article No. 93. DOI: 10.1145/2966986.2980098.

[28] Sayers E W, Beck J, Bolton E E, Bourexis D, Brister J R, Canese K, Comeau D C, Funk K, Kim S, Klimke W, Marchler-Bauer A, Landrum M, Lathrop S, Lu Z, Madden T L, O'Leary N, Phan L, Rangwala S H, Schneider V A, Skripchenko Y, Wang J, Ye J, Trawick B W, Pruitt K D, Sherry S T. Database resources of the national center for biotechnology information. *Nucleic Acids Research*, 2021, 49(D1): D10–D17. DOI: 10.1093/nar/gkaa892.

[29] Chen S, Chen Y, Wang Z, Qin W, Zhang J, Nand H, Zhang J, Li J, Zhang X, Liang X, Xu M. Efficient sequencing data compression and FPGA acceleration based on a two-step framework. *Frontiers in Genetics*, 2023, 14: 1260531. DOI: 10.3389/fgene.2023.1260531.

Wan-Qi Liu received his B.S. degree in electronic engineering from Minzu University of China, Beijing, in 2018. He is currently a Ph.D. candidate at the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. His research interests include domain-specific architecture and data compression.

Ye-Wen Li received his B.S. degree in computer science from Minzu University of China, Beijing, in 2019, and his Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2024. He is a post-doctoral fellow at The Hong Kong University of Science and Technology, Hong Kong. His research interests encompass computer architecture and computational biology.

Guang-Ming Tan received his B.S. degree in mathematics from Xiangtan University, Xiangtan, in 2002, and his Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2008. He is a professor at ICT, CAS, Beijing. His research interests include parallel computing, domain-specific architecture, and big data.