electronics

MDPI

Article

Enhancing Regular Expression Processing through Field-Programmable Gate Array-Based Multi-Character Non-Deterministic Finite Automata

Chuang Zhang, Xuebin Tang and Yuanxi Peng *

Department of Intelligent Data Science, College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China; zhangchuang10@nudt.edu.cn (C.Z.); xbtang@nudt.edu.cn (X.T.)
* Correspondence: pyx@nudt.edu.cn

Abstract: This work investigates the advantages of FPGA-based Multi-Character Non-Deterministic Finite Automata (MC-NFA) for enhancing regular expression processing over traditional software-based methods. By integrating Field-Programmable Gate Arrays (FPGAs) within a data processing framework, our study showcases significant improvements in processing efficiency, accuracy, and resource utilization for complex pattern matching tasks. We present a novel approach that not only accelerates database and network security applications, but also contributes to the evolving landscape of computational efficiency and hardware acceleration. The findings illustrate that FPGA's coherent access to main memory and the efficient use of resources lead to considerable gains in processing times and throughput for handling regular expressions, unaffected by expression complexity and driven primarily by dataset size and match location. Our research further introduces a phase shift compensation technique that elevates match accuracy to optimal levels, highlighting FPGA's potential for real-time, accurate data processing. The study confirms that the benefits of using FPGA for these tasks do not linearly correlate with an increase in resource consumption, underscoring the technology's efficiency. This paper not only solidifies the case for adopting FPGA technology in complex data processing tasks, but also lays the groundwork for future explorations into optimizing hardware accelerators for broader applications.

Keywords: FPGA; heterogeneous computing system; regular expression filtering; multi-character non-deterministic finite automata

Citation: Zhang, C.; Tang, X.; Peng, Y. Enhancing Regular Expression Processing through Field-Programmable Gate Array-Based Multi-Character Non-Deterministic Finite Automata. *Electronics* **2024**, *13*, 1635. https://doi.org/10.3390/electronics13091635

Academic Editors: Pavel Lyakhov and Maxim Deryabin

Received: 15 March 2024
Revised: 15 April 2024
Accepted: 16 April 2024
Published: 24 April 2024

1. Introduction

The current state of the art (SOTA) for FPGA accelerators in relational databases focuses on optimizing the inherent performance and specialization within these databases' designs [1,2]. Relational databases, essential for large applications, utilize a query optimizer to translate user queries expressed in SQL into dynamic execution plans [3,4]. These plans involve selecting from various operator implementations at runtime, guided by complex heuristics and cost functions [5,6]. The emphasis on using prepared statements enhances query execution performance by pre-arranging the operator tree and dynamically filling in parameters during runtime.

In the context of string processing within these databases, existing relational engines employ mechanisms like LIKE, REGEXP_LIKE, and CONTAINS, each presenting trade-offs between generality and performance [7,8]. While CONTAINS offers swift searches through a pre-built inverted index, it introduces additional maintenance costs. Furthermore, indexing techniques, including suffix trees and n-grams, pose challenges such as substantial space requirements and periodic rebuilding for potentially stale data [5,9]. The SOTA for FPGA accelerators aims to address these challenges efficiently, offering specialized solutions that balance generality and performance. This emphasis particularly focuses on optimizing fixed operation types and dynamic parameterization for FPGA acceleration

within relational databases. Another critical domain facing challenges is streaming pattern-matching in network security, with imperative considerations, including processing data at the rapidly increasing pace of network bandwidth [10,11]. Adapting to dynamic cyber threats, maintaining an up-to-date pattern set, and striking the right balance between minimizing false positives and negatives while optimizing system resources remain pivotal concerns in this field.

Two key computational models, Deterministic Finite Automata (DFA) and Non-Deterministic Finite Automata (NFA), play fundamental roles in the implementation of regular expression matching. DFAs operate by processing each character in constant time, providing straightforward and deterministic pattern matching [2,12]. However, despite their simplicity and predictable processing times, DFAs face challenges, particularly the potential for state explosion, especially when dealing with complex patterns [13,14]. In scenarios involving regular expressions with numerous possibilities, the exponential growth in the number of states in DFAs becomes a drawback, impacting memory utilization and overall efficiency in applications.

On the other hand, NFAs offer distinct advantages in the realm of pattern matching, primarily attributed to their linear scalability (O(n)), ensuring a proportional increase in size corresponding to the complexity of the regular expression [15,16]. This scalability makes NFAs well-suited for deciphering intricate patterns, such as byte sequences indicative of specific patterns in relational databases. The synergistic integration of FPGAs with NFAs further elevates their potential. FPGAs, renowned for their parallel processing capabilities, enable the concurrent evaluation of multiple patterns, optimizing throughput and facilitating efficient real-time detection of pattern matching [17,18]. The inherent adaptability of FPGAs becomes evident as they accommodate customized and parallelized implementations of NFAs, fostering a blend of flexibility and speed in the pattern-matching process. Despite these promising advantages, challenges persist in the utilization of NFAs and FPGAs for regular expression matching, particularly in managing potential state explosions during the conversion of NFAs to DFAs. The intricate balance between scalability, memory utilization, and real-time processing efficiency requires careful consideration to harness the full potential of this powerful combination in addressing the evolving landscape of database matching.

This paper focuses on three key aspects. Firstly, a performance comparison between software and FPGA (MC-NFA) processing, showing FPGA's superior efficiency in handling regular expressions across various data set sizes. Secondly, the advantage of phase shift compensation in improving match accuracy, demonstrating perfect accuracy with this feature versus variable accuracy without it in FPGA projects. Lastly, A detailed analysis of FPGA resource utilization, indicating that increased processing performance does not linearly correlate with resource consumption, showcasing efficient use of FPGA resources. The subsequent sections of this paper are structured as follows: Section 2 provides the background information. The proposed approach is delineated in Section 3, and the performance evaluation is expounded upon in Section 4. Finally, Section 5 offers concluding remarks on this work.

2. Background

2.1. FPGA

Field Programmable Gate Arrays (FPGAs) are hardware chips known for their unique ability to be reprogrammed multiple times, exhibiting behavior similar to Application-Specific Integrated Circuits (ASICs) once programmed [19,20]. Traditionally, these chips are programmed using hardware description languages like Verilog or VHDL. However, recent advancements have introduced high-level languages and synthesis tools that facilitate the translation of C/C++ or OpenCL code into logic gates [21,22]. Due to the limited capacity of on-chip static random-access memory on FPGAs, commonly referred to as block RAMs or BRAMs, typically a few megabytes in size, standalone boards are frequently utilized as "bump-in-the-wire" accelerators for stream processing [23,24]. This strategy helps avoid

the need for extensive data or computational state storage. Nevertheless, contemporary FPGA boards often integrate DDR memory in the range of several gigabytes, albeit with higher access latency than the on-chip memory [25,26]. The hybrid system considered in this context offers an alternative to these established designs.

2.2. Database Operators

In the conceptual framework of a database table, each entry corresponds to a row containing multiple attributes. The database operates through query compilation by the query optimizer [27,28], resulting in a query plan structured as a tree of operators. Each operator within this tree executes a specific part of the query. The transformation of a database query into a tree involves the utilization of two operators: Filter and Min [29,30]. The Filter operator selects rows in the table that meet the condition specified by 'LIKE,' employing regular expression matching and forwarding the results to the Min operator. Subsequently, the Min operator identifies the row with the minimum value iteratively and extracts the associated Value attribute. This extracted value then serves as the ultimate output of the query.

Real-world queries tend to be more complex, and the challenge for the query optimizer is to strategically integrate available operators into an execution tree that optimizes the estimated execution time [31,32]. To facilitate viable operator combinations, these operators must operate within a consistent, well-defined data layout and share similar interfaces. In our conceptualized database, operators are implemented in both software and hardware, offering the query optimizer flexibility to choose optimally between the two. Hardware-implemented operators are required to adhere to the same data layout as their software counterparts. Databases employ diverse data layout strategies, ranging from row-oriented [27,33] to column-oriented structures [34,35]. Despite these variations, databases generally manage memory independently from the operating system, organizing records into pages.

2.3. Transitive Closure

The concept of transitive closure plays a pivotal role in the computational theory, especially within the domain of finite automata [36,37]. It essentially describes the reachability of nodes within a graph, facilitating the identification of all possible states that can be reached from any given state within a Non-Deterministic Finite Automaton (NFA). Given a directed graph $G = (V, E)$, where V represents the vertices or states of the automaton and E the transitions or edges between these states, the transitive closure of G, denoted as G^*, is a graph where a direct path from vertex i to vertex j exists if and only if there is a path from i to j in G. Formally, the transitive closure can be defined as:

$$G^* = (V, E^*), \quad \text{where } E^* = \{(i, j) \mid \text{there exists a path from } i \text{ to } j \text{ in } G\} \tag{1}$$

The computation of transitive closure is essential in the construction of Multi-Character NFAs, where the goal is to enhance the processing throughput by efficiently identifying all reachable states from any given state, considering multiple characters as input [38,39]. This capability is fundamental to optimizing state transitions in NFAs designed to process high volumes of data, thereby significantly reducing the computational overhead associated with single-character NFAs [38,40].

In the context of Multi-Character NFA construction, the transitive closure assists in merging multiple cycles of matching into a singular computational step. By doing so, it not only improves the matching speed, but also ensures a compact representation of state transitions, essential for high-throughput computing environments. This concept is illustrated further in the algorithm for constructing Multi-Character NFAs, where the transitive closure facilitates the identification of composite states resulting from the aggregation of individual transitions, thereby enabling the NFA to process multiple input characters simultaneously.

3. Principles of Constructing Multi-Character NFAs

3.1. Preliminaries

A NFA is defined as a five-tuple, $M = (Q, \Sigma, q_0, \delta, F)$, where:

- Q is a finite set of states;
- Σ is a finite set of input tokens;
- q_0 is the initial state;
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function;
- $F \subseteq Q$ is the set of accept states.

The transition function δ is defined such that $\delta(q, a) = \{p_1, p_2, \ldots, p_m\}$, indicating possible transitions from state q upon reading character a.

3.2. Multi-Character NFA Construction

The core principle behind MC-NFA construction is the exploitation of the transitive closure property in automata, enabling the processing of symbol sequences in bulk. The transitive closure, a well-known concept in mathematics and computer science, is used here to establish a connectivity matrix that represents all possible state transitions over sequences of input symbols. Consider an NFA $M = (Q, \Sigma, q_0, \delta, F)$, the enhanced transition function δ^* for a MC-NFA can be mathematically represented as:

$$\delta^*(q, w) = \bigcup_{p \in \delta(q,a)} \delta^*(p, w[1:]) \tag{2}$$

where w is a string of characters from Σ^*, and $w[1:]$ denotes the string w excluding its first character. This recursive formulation underscores the transition from processing a single character to handling strings of characters, significantly augmenting the automaton's processing throughput.

The capability to process sequences of characters is encapsulated by a recursive relationship, facilitating the iterative expansion of the NFA's processing scope. This relationship is formalized as:

$$\delta^n_{mc}(q, s) = \begin{cases} q & \text{if } n = 0 \\ \delta^1_{mc}(\delta^{n-1}_{mc}(q, s_{1:n-1}), s_n) & \text{otherwise} \end{cases} \tag{3}$$

where $s_{1:n-1}$ and s_n denote the substring from the first to the $(n-1)$th character and the nth character in s, respectively. In a case given $s =' ab'$, $n = 2$, we have:

$$\delta^2_{mc}(q,' ab') = \delta^1_{mc}(\delta^1_{mc}(q,' a'),' b') \tag{4}$$

In this special case, where $n = 2$ and $s =$ 'ab', the function first processes the substring 'a' ($s_{1:1}$), transitioning from the initial state q based on this input, and then processes the second character 'b' (s_2), further transitioning based on the result of the first transition. The specific case here demonstrates the method for a two-character input, enhancing processing throughput by reducing the number of transitions required for pattern matching over large datasets. The introduction of MC-NFAs fundamentally alters the computational landscape for automata by reducing the number of transitions required to process input strings. This reduction directly translates to increased efficiency, particularly in scenarios involving complex pattern matching over large datasets.

3.3. Algorithm

The construction of an MC-NFA involves a recursive traversal algorithm that systematically explores all possible paths through the NFA's state graph, identifying and merging paths that can be traversed by consuming multiple input characters. The algorithm operates as follows:

1. **Initialization:** start from the initial state q_0, marking the current node's layer as N and each connected child node's layer as $N + 1$.
2. **Traversal:** For each node, retrieve all connected child nodes. If a child node has been visited in an even layer previously, the recursion stops for that path to prevent the duplication of states.
3. **Layer Management:** nodes appearing in even layers are processed to simulate the consumption of 2 bytes at a time, aligning with the design of FPGA-based state machines.
4. **State Merging:** through recursive traversal, merge transitions that can be compactly represented by multi-character inputs, effectively reducing the graph's complexity.

The recursive nature of the algorithm ensures that all potential paths are explored, allowing for the efficient consolidation of state transitions based on the input characters' sequences. Upon transforming a regular expression into an MC-NFA, the automaton is characterized by a set of input tokens, states, and transition conditions tailored to process multiple characters.

We illustrated the algorithm in a simple example. Figure 1a is the Token table, which conveys the content of the smallest character groups that cause state transitions after multi-character transformation, represented using ASCII codes. For example, in a 2-NFA, a token in the table represents two characters, such as 'ab', and in a 4-NFA, a token represents four characters. The total number of tokens after transformation must be less than the maximum number of tokens supported by the FPGA, otherwise, it will indicate unsupported. This paper designs to support up to 16 tokens (totaling 64 characters) based on the application scenario of the database, which can meet the requirements. Figure 1b shows the state transition relationships triggered by tokens. For instance, in Figure 1b, "ab" can effectively trigger state S_3, which is marked as "1" in the table, and states that cannot be triggered are marked as "0". Figure 1c details the transitions between states. For example, under the effect of 'ab', state S_1 can transition to state S_3, which is marked as '1' in the corresponding place in the table, and other markings that do not result in state transitions are marked as "0". Based on the information from these three tables, the NFA's transition relationships can be obtained, thereby acquiring the matching rules for the regular expression.

Symbol	ASCII characters	Hex for FPGA
ab	97 98	0x61 0x62
ef	101 102	0x65 0x66
...	...	...

(a) Symbol encoding for 256 ASCII characters

State \ Symbol	ab	ef	cd	...
S1	0	0	0	...
S2	0	0	1	...
S3	1	0	0	...
...	...	...	...	...

(b) Transition table with symbol to States

State \ State	S1	S2	S3	...
S1	0	0	0	...
S2	1	0	0	...
S3	1	1	0	...
...	...	...	...	...

(c) Transition table with states to states

Figure 1. MC-NFA algorithm demonstration.

4. System Implementation

4.1. System Overview

The FPGA solution proposed in this paper relies on the conversion of regular expressions into NFA, leveraging the advantage of NFA allowing multiple states to be active simultaneously, and input Tokens can trigger many different state transitions in constant time.

We propose the implementation of multi-character regular expression matching in FPGAs, covering both its functionality and performance. The process begins with software extracting and optimizing the regular expressions, and then compiling them into NFAs (Non-deterministic Finite Automata) using a library (this part is not the contribution of this paper and relies on existing library functions). The main contribution of this paper starts from step 1 in Figure 2. Specifically, it first utilizes the conversion algorithm proposed in this paper to transform the NFA into a multi-character *NFA*. Then, it abstracts and parameterizes the multi-character NFA, forming pattern encoding information that can be recognized by the FPGA and transmitted to the FPGA board. Packet data is sent to the

FPGA's acceleration module through a 512-bit wide transmission channel, initiating the FPGA to perform accelerated computation of the regular expression. After completing the computation, the FPGA provides the Result information back to the user.

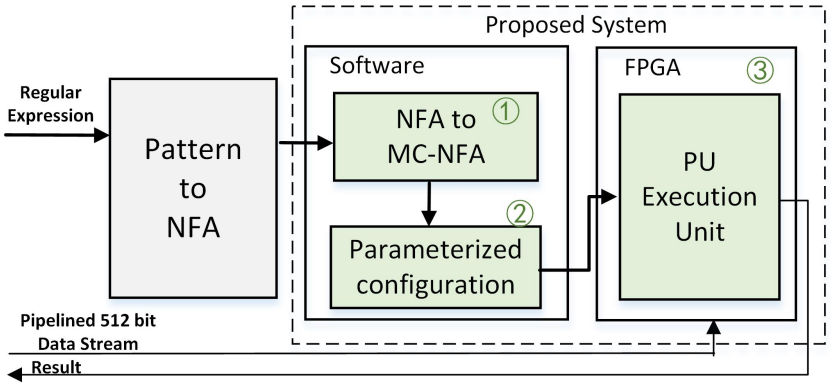

Figure 2. System overview.

4.2. The Extraction and Preprocessing of Regular Expressions

4.2.1. Regular Expression Input

Regular expressions are divided into five metacharacter categories, each tied to a specific implementation method and corresponding to the creation of specialized regular expression HDL libraries. These categories encompass the Basic Matching Library, which handles single character and wildcard "." matching, serving as the foundation for complex expressions. The Relational Operations Library focuses on OR and AND operations within regular expressions, accommodating the "|" and "and" operators for pattern selection and logical conjunction.

The Quantity Limiting Library quantifies subexpression occurrences with support for "*", "+", "?", "{n}", "{n,}", and "{n,m}". Meanwhile, the Positional Matching Library anchors expressions to string positions using "^" and "$" metacharacters to match the start and end of strings. Finally, the Range Matching Library specializes in character set and range matching via "[ ]" and "[^]" for inclusive and negated sets, along with support for character ranges "[a–z]" and negated ranges "[^a–z]". Combining these metacharacters and single-byte matching modules enables the comprehensive implementation of all regular expression patterns, making it a versatile tool for various pattern matching tasks.

4.2.2. Regular Expression Preprocessing

In traditional theoretical models, using FPGA for matching is constrained by processing only one input character per cycle, limiting the matching speed of a single computational unit. To enhance matching efficiency, this solution introduces a single-cycle, multi-character (MC-NFA) construction method, allowing FPGAs to process multiple characters simultaneously within a single cycle. Therefore, to support multi-character matching, adjustments and adaptations to the algorithm are required during the software preprocessing phase, which is described in two distinct steps.

4.2.3. Software Compilation

Converting regular expressions into single-character NFAs can be achieved using the classical Thompson construction method. For example, consider the expression `'ab(cdb)*ef'`.The resulting NFA transformation is illustrated in Figure 3. In this figure, S_1 represents the initial state, S_6 denotes the final acceptance state, and ϵ signifies any character. The construction of NFAs is generated by a software program, with the input being the regular expression and the output being an NFA stack. As regular expressions are often lengthy and subject to frequent modifications, automating the generation of NFAs through a program enhances the efficiency of this construction approach.

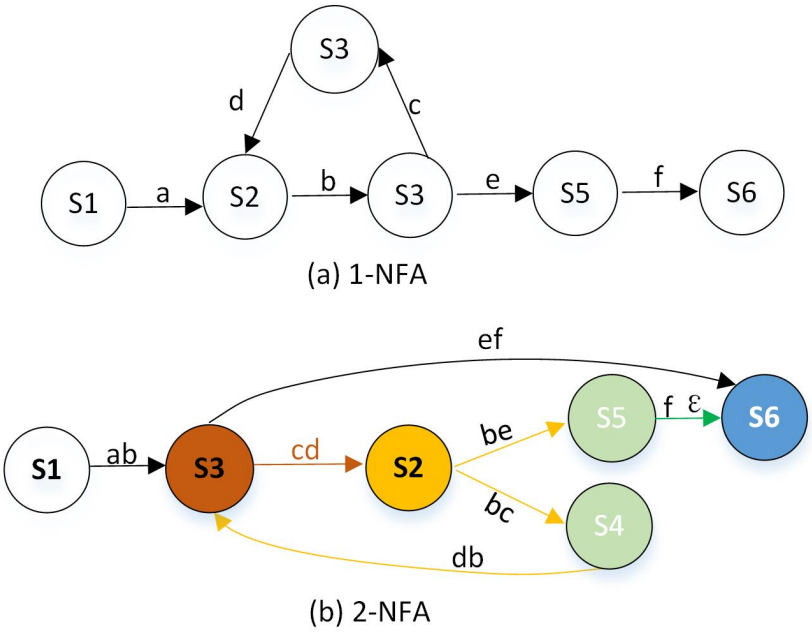

Figure 3. Non-Deterministic Finite Automaton (NFA) expression.

4.2.4. Phase Compensation

Upon a thorough examination of the principles underlying multi-character NFAs, particularly in the context of the two-step jump automaton transformation, a noteworthy observation emerges. In the case of double-byte matching, the default assumption is that the matching process commences with the first character, and subsequent jump conditions are predicated on pairs of characters. However, if the matching initiates from the second character of the input, adhering to the aforementioned jump conditions may lead to erroneous results, commonly known as "false positives".

To mitigate this issue and enhance the comprehensiveness of the multi-character matching algorithm, this paper introduces a phase compensation method. This technique is hardware-based and entails the creation of a sliding window that duplicates the input string and shifts it one byte to the left. In scenarios necessitating N-character matching within an NFA context, the input is replicated N times, with each copy shifting one byte to the left, resulting in a maximum shift of $N-1$ bytes.

The implementation process is exemplified using a 4-NFA as a reference. For instance, consider the regular expression `pattern=babana` and the input string `Str=єbanana` (Figure 4). When matching is executed, the first cycle "consumes" four characters, and it compares these with four tokens that have a phase offset, determining equality and outputting the match result. This result can be the final Match result, or it may serve as a prerequisite for whether a token related in the next cycle matches. Only after all connected matches are completed does it output whether the current input string matches. When matching is executed according to the previously outlined construction method, it yields a mismatch. However, following this "phase compensation" processing, successful matching is achieved. This hardware-driven solution not only mitigates false positives, but also streamlines the software-level algorithm. Depending on the availability of hardware logic resources, this approach offers a versatile trade-off between resource utilization and processing speed. This process only requires two cycles, whereas traditional single-character 1-NFA matching would need six cycles to complete. Therefore, this method not only resolves mismatch issues and improves accuracy, but also significantly enhances matching efficiency.

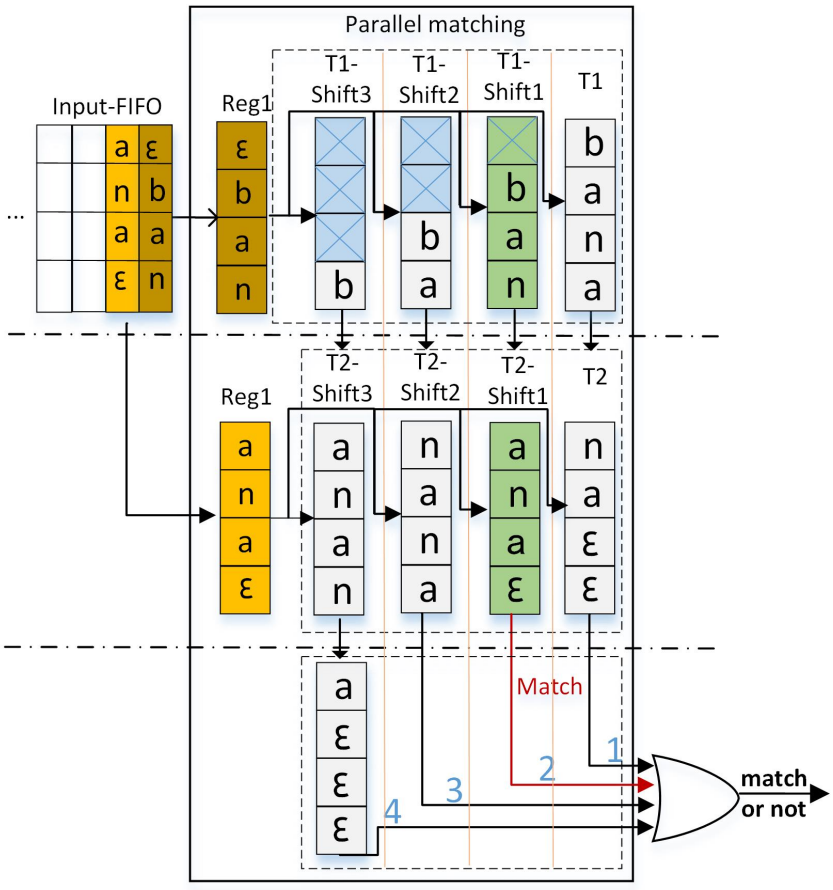

Figure 4. Phase compensation.

4.2.5. Parameterization

Runtime configurability means that an FPGA, once configured and in use, does not require a change in its design or recompilation when users modify the regular expression. The FPGA should support updates to the regular expression. Three tables with specific numerical values are filled into the configuration information in Figure 5 (the three tables generated by the software algorithm). This information is sent to the FPGA through the PCIe registers. The FPGA stores the relevant data in RAM and retrieves it when needed. Internally, the FPGA needs to translate the new configuration information into specific circuit connections to support the new regular expression. Taking the 2-NFA from Figure 1 as an example, the FPGA supports eight tokens and eight states, and the three tables are encoded according to the following protocol into a form that the FPGA can recognize and use.

From the aforementioned example, it becomes evident that, for a given regular expression, extracting information about the token set, the number of states, and the transition details between states is adequate for implementing regular expression matching on FPGA. This approach serves as the foundation for our design strategy. We organize the crucial information into three two-dimensional matrix storage structures, which are then written to the FPGA via online configuration. The FPGA interprets this information and carries out the matching process. When altering the pattern, you need only modify specific values within the two-dimensional matrix storage structures, without necessitating changes to the FPGA's underlying logic code. Within the FPGA's internal implementation, there are two main architectures: the transition mapping diagram between tokens and states, and the pre-linking between states (Figure 6). The input characters that satisfy the matching requirements are selected based on the configuration information.

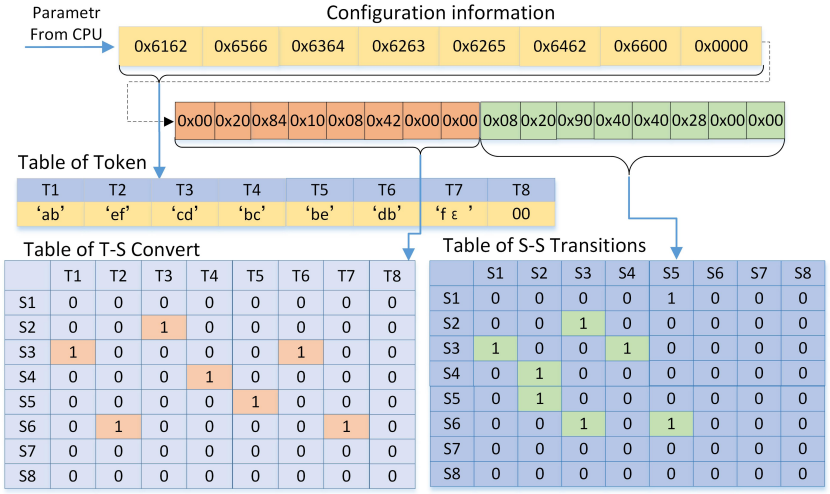

Figure 5. Configuration information to FPGA.

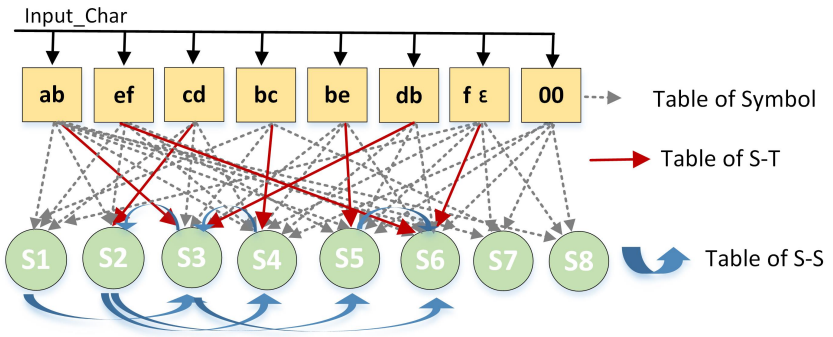

Figure 6. Configuration information and FPGA implementation.

4.2.6. FPGA Logic Design Solution

This section details the specific implementation method of 4-NFA as a functional module in FPGA. A processing module capable of handling multiple characters will be treated as a Process Unit (PU). The PU is designed to handle multi-byte matching operations between individual regular expressions and input strings. The use of multiple PUs enables parallel processing of input tuples, significantly enhancing the system's throughput.

Given the dynamic nature of regular expressions in response to various database application scenarios and customer needs, where different data tables may require different query demands, the flexibility of handling varying regular expressions is essential. This paper introduces a database-centric solution with several key features. It allows for online configuration of parameterized regular expressions, eliminating the need to re-download FPGA bitstreams when encountering new regular expressions in queries. Additionally, the logic overhead does not grow linearly, regardless of the complexity or length of the regular expressions used. Furthermore, the proposed multi-character NFA matching method is particularly effective for scenarios involving lengthy strings commonly found in databases, facilitating fast and efficient matching of extended strings. The internal implementation logic of FPGA is shown in Figure 7.

The PU mainly consists of a configuration information decoding module, a multi-character matching module, a phase compensation processing module, a Token and State trigger jump selection module, and a state-to-state jump selection module. In the specific logic design, the maximum number of characters and states supported are first determined. For illustration purposes, this article sets the number of characters to 16 and the number of states to 8. These two parameters are configurable. In the logic implementation, relevant modules can be modified by changing the parameter controlling the number of modules.

This article uses a quantity of 16 × 8 as an example. Figure 8 shows an NFA after four-character conversion for the regular expression $a(bc)^*(d|e)$. Subsequent modules will be detailed using this regular expression as an example.

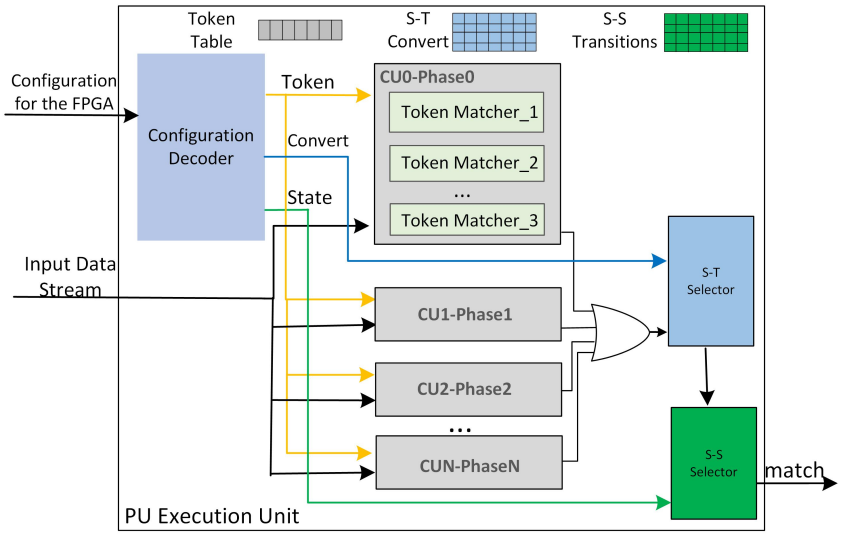

Figure 7. FPGA logic.

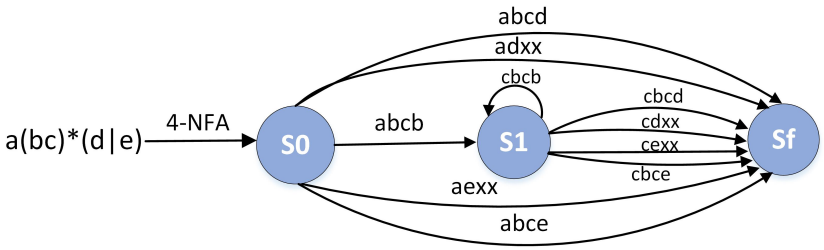

Figure 8. 4-character NFA demonstration.

4.2.7. Configuration Information Decoding Module

The function of this module is to decode the information configured from the CPU side, extract Tokens, Convert information, and States state transitions according to the protocol defined by both parties, and save them to the corresponding registers. It is important to note that Token is the character(s) to be matched in the regular expression. It can be a single character or multiple characters (or a word). In the single-character matching mode, Token is a character, while in the multi-character matching mode, the Token can be two or four characters, etc. Among them, from the 4-NFA configuration information in Figure 8, the following can be obtained: there are a total of 10 input characters, including abcd, adxx, abcb, cbcb, aexx, abce, cbcd, cdxx, cexx, and abce. Regarding States (State), there are three states: S0, S1, and Sf. The State Triggers and Transitions (Convert) consist of 10 transitions.

4.2.8. Token Matcher Implementation

In this solution, a multi-character matcher is used to match characters from the regular expression against the source data characters in the database. It performs basic character-to-character matching. Only when the input characters have a match is there a possibility of a regular expression match. If the characters do not match, the regular expression will definitely not match. Therefore, character matching is the first step in the logical implementation.

To support arbitrary expressions as designed in this solution, the FPGA internally only needs to configure a certain number of matchers without worrying about the specific characters to match. For example, in the detailed example described in Figure 8, the system supports four-character matching, with each comparator capable of simulta-

neously comparing 32 bits. For a system supporting a maximum of 16 character groups, the FPGA configures 16 modules for four-character matching. The specific number of configurations can be adjusted through parameters at the top level, and the maximum number of characters supported by the system needs to be evaluated based on the FPGA's logic resources.

To enhance matching performance and eliminate the need for backtracking in case of mismatches, we have designed a scheme that allows for the simultaneous matching of multiple characters. This design enables all 16 character matching modules to operate concurrently, providing synchronous feedback on whether the current input character matches the predefined configurations. The matching process is depicted in Figure 9. In previous examples, challenges arose during multi-character matching when the characters that matched did not align with multiples of four or two. For example, after some characters have been successfully matched, one or two characters might remain unmatched at the end of the sequence, with these being disregarded in terms of the broader matching context. In such instances, the symbol 'ϵ'—which represents any character—is used as a placeholder. When 'ϵ' is included in the Token configuration, it requires special handling to implement this matching functionality effectively. Alongside configuring the character information, a flag is associated with each character to indicate the presence of 'ϵ' characters. To accommodate these characters, a selector is employed to substitute the character information in the pattern with the actual characters from the string being matched. This strategy ensures that any input character is treated as a match, as illustrated in Figure 10.

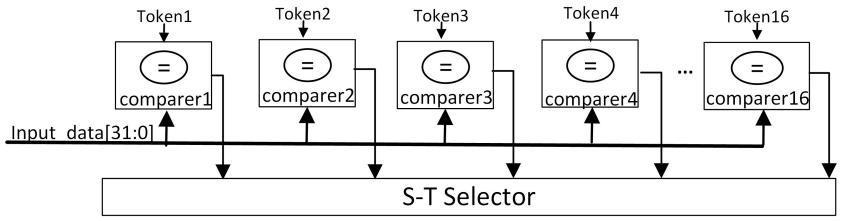

Figure 9. Mult-char matching implementation.

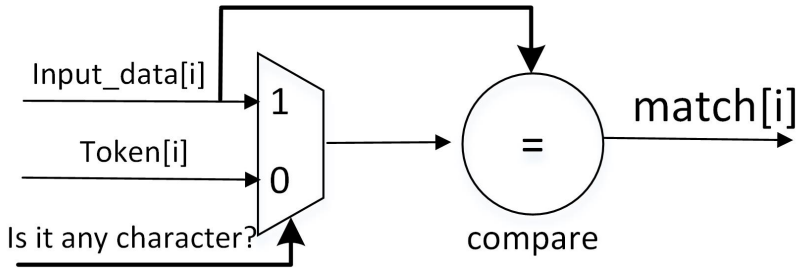

Figure 10. Handling of dynamic length character.

4.2.9. Mapping Diagram Between Token and State

As mentioned earlier, supporting any regular expression implies that any Token in the expression may trigger any one of the eight states supported by the logic, as it is related to NFA. In the hardware logic circuit design, this mapping relationship is virtually present, where each Token → state transition is present in the logic. In the specific configuration of the regular expression, the Convert configuration information in Figure 11 is used to select the states that are truly triggered, i.e., to choose the output results based on the configuration information. In the case of this solution's example, the mapping diagram between Token and State will be as shown in Figure 11.

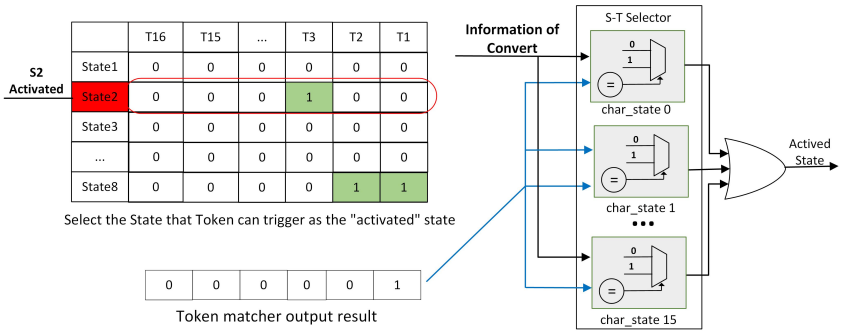

Figure 11. Principle and implementation logic of state mapping selection.

4.2.10. Pre-Linking Diagram between State

The pre-linking between states means that all states supported by the system are connected, and every regular expression is mapped to a state in the NFA. All states have the potential to transition to each other within the 16 states, and the final selection of connections that meet the expression requirements is based on the State Transitions configuration information in Figure 12. Each state is a node in the pre-linking diagram, and the transition conditions are the connections between nodes, marked as "1" in first table in Figure 12. The pre-linking state diagram for this solution's example is shown in Figure 12.

Based on the State Transitions configuration information, the "1" marks in the table indicate valid transitions between states, represented by solid line arrows in the diagram. With the described process, when a string to be matched undergoes regular expression filtering, the FPGA performs three comparisons: character matching, character and NFA state transition matching, and state-to-state transition matching, ultimately yielding the calculation result of whether it matches or not.

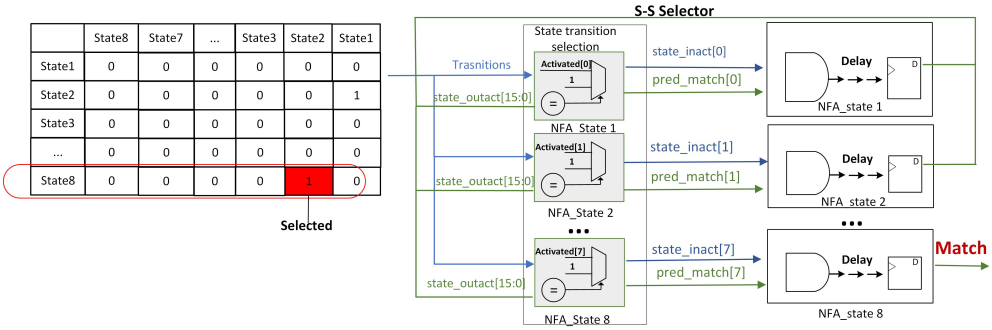

Figure 12. State pre-linking.

4.2.11. Result Processing Module

The result processing module in this system is responsible for handling the output generated by the parallel regular expression matching units. It processes the matching flags (match) and the positions of matched characters. The primary goal of this module is to refine the information related to matched rows and columns and provide feedback to the user.

5. Results

To evaluate the FPGA-based system for regular expression processing, a series of queries representing different complexity levels and use cases were utilized:

- **Q0**: searches for the word "engage" followed by "blithel" and any character from x to z, testing basic string matching.
- **Q1**: a simple query looking for occurrences of the word "Strasse", evaluating basic string search functionality.

- **Q2**: combines string search with repetition, looking for "Str" followed by any character repeated eight times, testing the system's ability to handle complex patterns with repetitions.
- **Q3**: aims at identifying patterns where a sequence of digits is followed by "USD", testing numeric processing and string concatenation capabilities.
- **Q4**: searches for patterns with three lowercase letters followed by a colon and four digits, evaluating the system's performance on mixed alphanumeric patterns with specific formats.

These queries help in assessing the system's efficiency across a range of regular expression functionalities, from simple text searches to more complex pattern matching involving repetitions and specific formats. The algorithm is implemented in Python 3.9, the system uses Ubuntu 18.04, the compilation environment is gcc7.5, with 32 GB of RAM, and the CPU is an Intel dual-core. The FPGA uses Xilinx's XCVU37P-L2FSVH28926LX565T chip model, and the simulation software is ISE14.7. The database uses PostgreSQL version 11.2.

5.1. Performance of MC-NFA

To validate the performance superiority of Multi-Character Non-Deterministic Finite Automata (MC-NFA) over software-based and single-character processing for regular expression retrieval, experiments were conducted across datasets of varying sizes using the regular expression Q0. The results indicated that the performance of processing with 1-NFA is approximately 6.5 times that of software-based processing. The performance of 2-NFA processing is about 1.7 times that of 1-NFA, and 4-NFA processing is approximately 3.5 times that of 1-NFA (Figure 13). Additionally, the performance of the Xilinx method is comparable to the 2-NFA method proposed in this paper. This similarity in performance is due to the Xilinx method's separation of data flow from control flow, its implementation of pre-fetch and post-store operations to improve memory access efficiency, and its resolution of read-and-write dependencies in on-chip RAMs by caching intermediate data in registers to reduce unnecessary accesses. Moreover, it executes a predictive (second) instruction in each iteration to accelerate the process under specific circumstances. The performance-optimized version executes two instructions every three cycles.

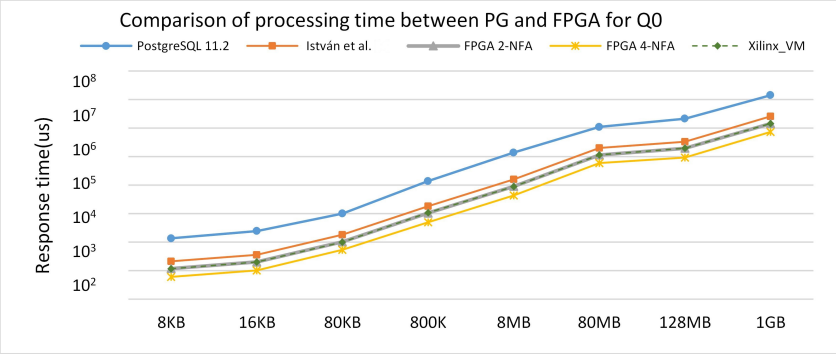

Figure 13. MC-NFA performance in processing time [1].

Consistently, Figure 14 presents a comparison of processing times for different regular expressions (Q2 to Q4) in PostgreSQL (PG) and FPGA (using 4-NFA), across datasets of varying sizes. The findings indicate that for FPGA, the processing time for various regular expressions is not influenced by the complexity of the expressions themselves but rather by the size of the dataset and the location of matches. Under the same dataset and match location conditions, regular expressions of the same length have consistent processing times on FPGA. Conversely, in PG software 11.2, the processing time varies with the complexity of the regular expressions, highlighting the efficiency and consistency advantages of FPGA (4-NFA) processing in handling regular expressions, regardless of their complexity.

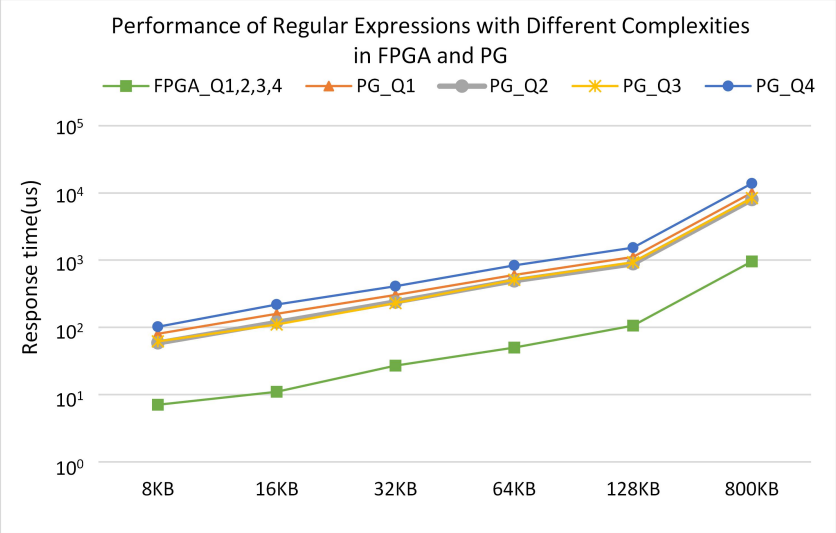

Figure 14. MC-NFA performance with complexities.

As mentioned in the previous section, the regular expressions Q1 to Q4 used in this experiment cover several common retrieval forms:

- Q1: a simple string search.
- Q2: involves the case of infinite repetition "*" occurrences.
- Q3: includes matching within the range of 0 to 9 and using "+" to represent one or more occurrences.
- Q4: contains matching for fixed character digit ranges and lowercase letter ranges.

Different levels of complexity in regular expressions, when executing retrieval functions in FPGA, have a response time that depends on the size of the dataset rather than the expression itself. For example, when detecting against an 8 K dataset size using FPGA, the retrieval time for Q1 to Q4 is consistently 7.08 microseconds, as shown in Figure 15. However, when executed in the PostgreSQL database software 11.2, the response times for Q1 to Q4 differ.

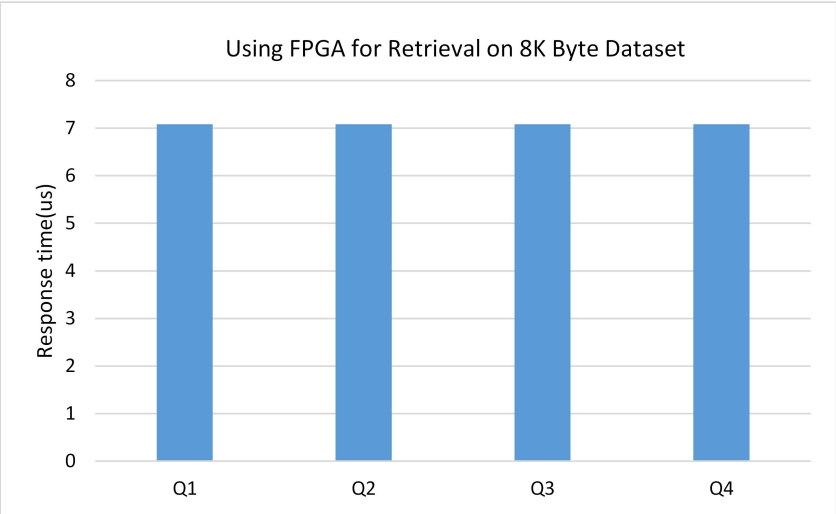

Figure 15. Retrieval on 8 k dataset.

In software, regular expressions of varying complexities require different algorithms for matching. More complex regular expressions may require more calculation and comparison steps, thus potentially taking longer to execute. The matching time is also related to the length of the matching string. Therefore, when executing regular expressions of

different complexities in software, it is necessary to consider the structure of the regular expressions and the efficiency of the matching algorithms, which will directly affect the length of execution time.

When using FPGA for regular expression matching, the complexity of the expression does not affect processing time, making it relatively more stable compared to software implementations. To verify the processing performance of multi-core CPUs, the following experiments were conducted. SQL statements were executed in the PostgreSQL database for data sizes of 128 MB and 8 KB:

```
select count(*) from stringq0 where l_stringq0 ~ 'engage blithel[x-z]';
```

The testing method is as follows: the Perf tool is used to collect the accumulated query process function counts via the CPU's Performance Monitoring Unit (PMU), generating periodic interrupts. This allows for obtaining the time consumption ratio of each function during the query process. The response times of the software were recorded with different numbers of cores enabled.

In Figure 16, the *x*-axis values 1, 2, 4, 8 represent the number of CPU cores enabled, while the y-axis represents the query time (in microseconds). From the figure, it can be observed that the time consumed for regular expression filtering decreases as the number of CPU cores increases, for both data sizes of 128 MB and 8 KB. When the number of CPU cores increases from 1 to 2, the time for regular expression filtering almost halves. As the CPU core count increases to 8, the decreasing trend in query time gradually slows down. This is because in a multi-core parallel situation, software execution requires the creation of multiple processes, which incurs a certain amount of time overhead. Therefore, as the number of created processes gradually increases, the reduction in time consumed for regular expression filtering slows down. When the CPU is configured with 8 cores, its processing performance is comparable to the 1-NFA processing performance shown in Figure 12, while the 4-NFA processing performance is 3.5 times that of the 1-NFA processing performance. Thus, the proposed FPGA-based acceleration of regular expression processing presents a significant advantage for offloading computational workload from the CPU.

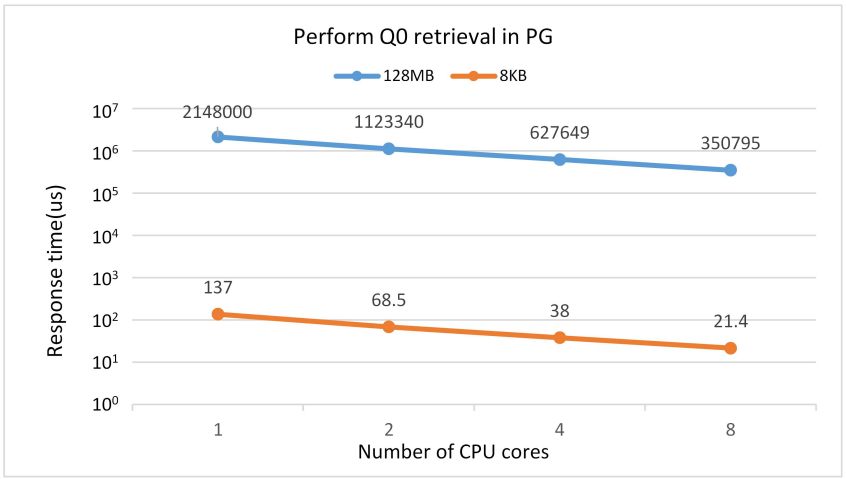

Figure 16. Retrieval in PG in terms of Q0.

5.2. Impact of Selectivity

To investigate the impact of different hit rates on performance, experiments were conducted using datasets with varying hit rates, applying software, 1-NFA, 2-NFA, and 4-NFA for the same datasets and recording and comparing processing times. The results show that lower hit rates significantly enhance the performance improvement of multi-character processing (Figure 17). This is because in database searches, once a matching string is found in a row, the search for the remaining characters in that row is terminated. In this experiment, the matching strings were positioned at the beginning of each line. For the

FPGA projects, once the first few characters were successfully matched, the remaining characters were not searched, and the processing moved directly to the next line, enabling rapid completion of a data block's search. Therefore, the advantage of multi-character processing is less pronounced at higher hit rates. Consequently, in situations with lower hit rates (which are more common), the acceleration effect of 4-NFA becomes more apparent.

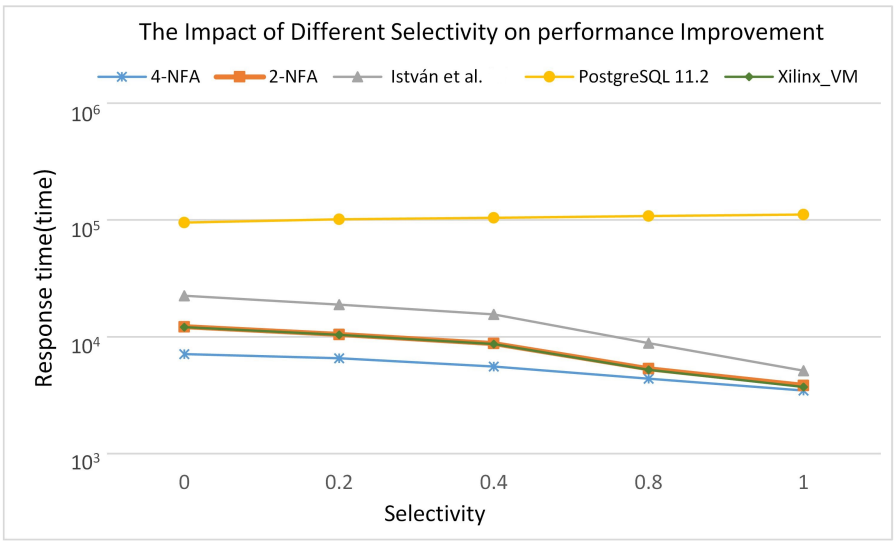

Figure 17. Impact of selectivity [1].

5.3. Resource Utility

IP projects with the same settings for Chars (supporting 16 characters with range matching) and State (eight transition states) are compared (Figure 18). It shows that even when performance doubles, resource utilization does not increase in exact proportion. For instance, while 4-NFA offers a 3.54-fold performance improvement over 1-NFA, the consumption of the most resource-intensive Logic resources increases by approximately 3.36 times. Only the CARRY8 resource usage reaches 4 times, but its overall impact on resource consumption is minimal. Additionally, the utilization of Registers and CLB resources remains below twice their original amount, indicating that the performance gains of 4-NFA do not require a proportional increase in resource consumption.

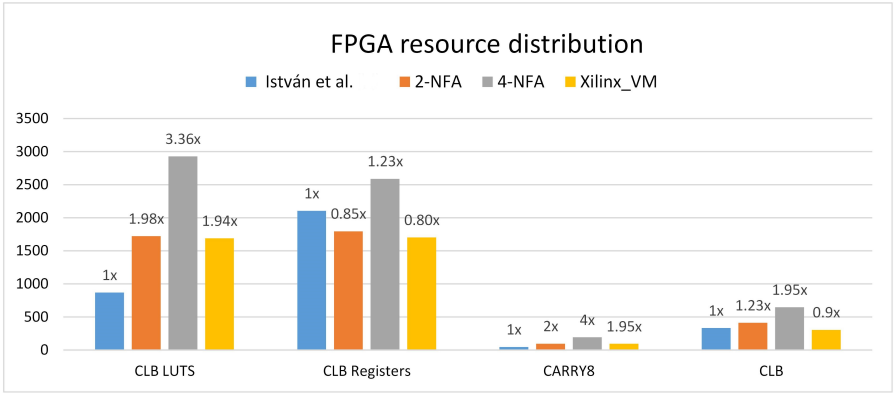

Figure 18. FPGA resource distribution [1].

6. Conclusions

This research has elucidated the advantages of employing FPGA-based Multi-Character Non-Deterministic Finite Automata (MC-NFA) for the processing of regular expressions, demonstrating notable superiority over traditional software-based approaches across various dataset sizes. Through comprehensive experimentation, we have illustrated that

FPGA processing markedly surpasses software in terms of efficiency, unaffected by the complexity of expressions, with performance improvements being primarily influenced by dataset size and the specific location of matches. Additionally, the incorporation of phase shift compensation has significantly enhanced match accuracy, highlighting the potential of FPGAs for precise, real-time data processing. Furthermore, our findings reveal that the efficiency gains achieved through FPGA processing do not necessitate proportional increases in resource consumption. This aspect underscores the inherent efficiency of FPGA technology in handling complex data processing tasks, offering a compelling alternative to software-based methods. The research presented here not only advocates for the broader adoption of FPGA technology in sophisticated data processing scenarios, but also lays the groundwork for further exploration into optimizing hardware accelerators for applications in databases and network security. In sum, the outcomes of this study contribute valuable insights into the potential of FPGA technology to revolutionize regular expression processing, both from performance and resource utilization perspectives. Future research directions may include the exploration of more complex pattern matching algorithms on FPGAs, the integration of FPGA-based processing units into existing database management systems, and the development of more advanced phase compensation techniques to further enhance matching accuracy.

Author Contributions: Conceptualization, C.Z.; Methodology, C.Z.; Software, C.Z.; Validation, C.Z.; Formal analysis, C.Z. and X.T.; Investigation, X.T.; Resources, C.Z. and Y.P.; Data curation, X.T.; Writing—review & editing, X.T. and Y.P.; Visualization, X.T.; Supervision, Y.P.; Project administration, Y.P.; Funding acquisition, C.Z. and Y.P. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by Shandong Smart Computing Program (NO. ZR2023LZH014).

Data Availability Statement: Data are contained within the article.

Acknowledgments: Thanks for the anonymous reviewers for their helpful comments.

Conflicts of Interest: We declare no conflicts of interest.

References

1. István, Z.; Sidler, D.; Alonso, G. Runtime parameterizable regular expression operators for databases. In Proceedings of the 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Washington, DC, USA, 1–3 May 2016; IEEE: New York, NY, USA, 2016; pp. 204–211.
2. Cicolini, L.; Carloni, F.; Santambrogio, M.D.; Conficconi, D. One Automaton to Rule Them All: Beyond Multiple Regular Expressions Execution. In Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Edinburgh, UK, 2–6 March 2024; IEEE: New York, NY, USA, 2024; pp. 193–206.
3. Sidler, D.; István, Z.; Owaida, M.; Alonso, G. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In Proceedings of the 2017 ACM International Conference on Management of Data, Chicago, IL, USA, 14–19 May 2017; pp. 403–415.
4. Maschi, F.; Korolija, D.; Alonso, G. Serverless FPGA: Work-In-Progress. In Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies, Rome, Italy, 8 May 2023; pp. 1–4.
5. Peltenburg, J.; Hadnagy, Á.; Brobbel, M.; Morrow, R.; Al-Ars, Z. Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators. In Proceedings of the 2021 International Conference on Field-Programmable Technology (ICFPT), Auckland, New Zealand, 6–10 December 2021; IEEE: New York, NY, USA, 2021; pp. 1–9.
6. Lee, D.; So, J.; Ahn, M.; Lee, J.G.; Kim, J.; Cho, J.; Oliver, R.; Thummala, V.C.; JV, R.s.; Upadhya, S.S.; et al. Improving in-memory database operations with acceleration DIMM (AxDIMM). In Proceedings of the 18th International Workshop on Data Management on New Hardware, Philadelphia, PA, USA, 18 June 2022; pp. 1–9.
7. Shani, S.; Majeed, M.; Alhassan, S.; Gideon, A. Internet of things (IoTs) in the hospitality sector: Challenges and opportunities. In *Advances in Information Communication Technology and Computing: Proceedings of AICTC 2022*; Springer: Singapore, 2023; pp. 67–81.
8. Lan, S.; Huang, J. Brief Analysis for Network Security Issues in Computing Power Network. In Proceedings of the International Conference on Emerging Networking Architecture and Technologies, Shenzhen, China, 15–17 October 2022; Springer: Singapore, 2022; pp. 298–311.
9. Matas, K. Runtime Management of Dynamic Dataflows with Partially Reconfigurable Pipelines on FPGAs. Ph.D. Thesis, University of Manchester, Manchester, UK, 2023.
10. Valizadeh, M.; Gorinski, P.J.; Iacobacci, I.; Berger, M. The Regular Expression Inference Challenge. *arXiv* **2023**, arXiv:2308.07899.

11. Nam, J.; Na, S.H.; Shin, S.; Park, T. Reconfigurable regular expression matching architecture for real-time pattern update and payload inspection. *J. Netw. Comput. Appl.* **2022**, *208*, 103507. [CrossRef]
12. Le Glaunec, A.; Kong, L.; Mamouras, K. Regular expression matching using bit vector automata. *Proc. ACM Program. Lang.* **2023**, *7*, 492–521. [CrossRef]
13. Wang, H.; Pu, S.; Knezek, G.; Liu, J.C. Min-max: A counter-based algorithm for regular expression matching. *IEEE Trans. Parallel Distrib. Syst.* **2012**, *24*, 92–103. [CrossRef]
14. Xu, C.; Yu, K.; Xu, X.; Bao, X.; Wu, S.; Zhao, B. Offset-FA: A Uniform Method to Handle Both Unbounded and Bounded Repetitions in Regular Expression Matching. *Sensors* **2022**, *22*, 7781. [CrossRef] [PubMed]
15. Li, J.; Chen, B.; Gu, L.; Yu, Z. FPGA-based regular expression matching acceleration system design and implementation. In Proceedings of the 2nd International Conference on Artificial Intelligence, Automation, and High-Performance Computing (AIAHPC 2022), Zhuhai, China, 25–27 February 2022; SPIE: Bellingham, WA, USA, 2022; Volume 12348, pp. 1008–1015.
16. Zhong, J.; Chen, S.; Han, B. FPGA-CPU Architecture Accelerated Regular Expression Matching With Fast Preprocessing. *Comput. J.* **2023**, *66*, 2928–2947. [CrossRef]
17. Ivanova, A.; Kostadinov, N. An Approach to Introduce the Concept of Lexical Analysis through FPGA Based Finite State Machines. In Proceedings of the 24th International Conference on Computer Systems and Technologies, Ruse, Bulgaria, 16–17 June 2023; pp. 79–84.
18. Kaushik, B.; Saini, M. A Finite State Automaton is a Tool to Represent Formal Language. *Glob. J. Enterp. Inf. Syst.* **2023**, *15*, 93–101.
19. Teubner, J.; Woods, L. *Data Processing on FPGAs*; Morgan & Claypool Publishers: San Rafael, CA, USA, 2013.
20. Dann, J.; Ritter, D.; Fröning, H. Non-relational databases on FPGAs: Survey, design decisions, challenges. *ACM Comput. Surv.* **2023**, *55*, 1–37. [CrossRef]
21. Baguma, G. High Level Synthesis of FPGA-Based Digital Filters. Master's Thesis, Department of Information Technology, Uppsala University, Uppsala, Sweden, 2014.
22. Alfahham, A.; Berekovic, M. Energy efficient cooperative spectrum sensing in Cognitive Radio Sensor Network Using FPGA: A survey. In Proceedings of the 2017 21st Conference of Open Innovations Association (FRUCT), Helsinki, Finland, 6–10 November 2017; IEEE: New York, NY, USA, 2017; pp. 16–25.
23. Mueller, R.; Teubner, J.; Alonso, G. Streams on wires: A query compiler for FPGAs. *Proc. VLDB Endow.* **2009**, *2*, 229–240. [CrossRef]
24. Najafi, M.; Sadoghi, M.; Jacobsen, H.A. Configurable hardware-based streaming architecture using online programmable-blocks. In Proceedings of the 2015 IEEE 31st International Conference on Data Engineering, Seoul, Republic of Korea, 13–17 April 2015; IEEE: New York, NY, USA, 2015; pp. 819–830.
25. Moghaddamfar, M.; May, N.; Färber, C.; Lehner, W.; Kumar, A. A study of early aggregation in database query processing on FPGAs. In Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, USA, 12–14 February 2023; pp. 55–65.
26. Geethakumari, P.R.; Sourdis, I. Stream Aggregation with Compressed Sliding Windows. *ACM Trans. Reconfigurable Technol. Syst.* **2023**, *16*, 1–28. [CrossRef]
27. Hulsebos, M.; Demiralp, Ç.; Groth, P. Gittables: A large-scale corpus of relational tables. *Proc. ACM Manag. Data* **2023**, *1*, 1–17. [CrossRef]
28. Cutrona, V.; Chen, J.; Efthymiou, V.; Hassanzadeh, O.; Jiménez-Ruiz, E.; Sequeda, J.; Srinivas, K.; Abdelmageed, N.; Hulsebos, M.; Oliveira, D.; et al. Results of semtab 2021. In Proceedings of the Semantic Web Challenge on Tabular Data to Knowledge Graph Matching. 20th International Semantic Web Conference, Virtual, 24–28 October 2022; Volume 3103, pp. 1–12.
29. Akidau, T.; Barbier, P.; Cseri, I.; Hueske, F.; Jones, T.; Lionheart, S.; Mills, D.; Pauliukevich, D.; Probst, L.; Semmler, N.; et al. What's the Difference? Incremental Processing with Change Queries in Snowflake. *Proc. ACM Manag. Data* **2023**, *1*, 1–27. [CrossRef]
30. Kipf, A.; Pandey, V.; Böttcher, J.; Braun, L.; Neumann, T.; Kemper, A. Scalable analytics on fast data. *ACM Trans. Database Syst. (TODS)* **2019**, *44*, 1–35. [CrossRef]
31. Park, K.; Saur, K.; Banda, D.; Sen, R.; Interlandi, M.; Karanasos, K. End-to-end optimization of machine learning prediction queries. In Proceedings of the 2022 International Conference on Management of Data, Philadelphia, PA, USA, 12–17 June 2022; pp. 587–601.
32. Chen, T.; Gao, J.; Chen, H.; Tu, Y. LOGER: A learned optimizer towards generating efficient and robust query execution plans. *Proc. VLDB Endow.* **2023**, *16*, 1777–1789. [CrossRef]
33. Liu, J.; Chabot, Y.; Troncy, R.; Huynh, V.P.; Labbé, T.; Monnin, P. From tabular data to knowledge graphs: A survey of semantic table interpretation tasks and methods. *J. Web Semant.* **2023**, *76*, 100761. [CrossRef]
34. Suárez-Cabal, M.J.; Suárez-Otero, P.; de la Riva, C.; Tuya, J. MDICA: Maintenance of data integrity in column-oriented database applications. *Comput. Stand. Interfaces* **2023**, *83*, 103642. [CrossRef]
35. Petrov, B.; Bakenova, A.; Yensebayeva, S. Development of a Database of Digital Multicultural Content and Application in Journalism Lessons. *Sci. J. Astana IT Univ.* **2022**, *10*, 33–44. [CrossRef]
36. Yamakami, T. Power of counting by nonuniform families of polynomial-size finite automata. In Proceedings of the International Symposium on Fundamentals of Computation Theory, Trier, Germany, 18–21 September 2023; pp. 421–435.
37. Bell, P.C.; Hirvensalo, M.; Potapov, I. The membership problem for subsemigroups of GL2 (Z) is NP-complete. *Inf. Comput.* **2024**, *296*, 105132. [CrossRef]

38. Frumin, D.; Timany, A.; Birkedal, L. Modular Denotational Semantics for Effects with Guarded Interaction Trees. *Proc. ACM Program. Lang.* **2024**, *8*, 332–361. [CrossRef]
39. Bergsträßer, P.; Ganardi, M.; Lin, A.W.; Zetzsche, G. Ramsey Quantifiers in Linear Arithmetics. *Proc. ACM Program. Lang.* **2024**, *8*, 1–32. [CrossRef]
40. Cohen, L.; Jabarin, A.; Popescu, A.; Rowe, R.N. The Complex (ity) Landscape of Checking Infinite Descent. *Proc. ACM Program. Lang.* **2024**, *8*, 1352–1384. [CrossRef]