**Deploying Memory Based NFA for Regular Expression matching on FPGA**

Kunal Buch, Michela Becchi

Networking and Parallel Systems Lab,

Electrical & Computer Engineering Department, North Carolina State University

**ABSTRACT**

**Regular Expression matching finds its applications in our day-to-day life for suggestions while performing web searches, the auto-correct on our phones or the simple search and replace operation on text editors. The applications like data mining, bio-informatics and networking also uses automata for regular expressions matching. In networking domain, Deep-packet inspection (DPI) is known as a technique that detects the patterns which might be malicious. In bio-informatics, the Reg-Ex matching can be used to match the DNA base sequences to derive important genomic observations.**

**Most string matching architectures are based on finite automata theory. The finite automaton starts at an arbitrary initial state and if a specific condition is applied, it transitions to a different state. In deterministic finite automata (DFA), the transition of the next state is uniquely determined by the current state and the new input condition. So there is only one transition to a next state. In nondeterministic finite automata (NFA), for each pair of state and input condition, there may be several possible next states. In our work, we target accelerating the regular expression matching using Non-Deterministic Finite Automata (NFA) on FPGA. We first implement the serial NFA traversal and then move on to the parallel implementation of the NFA traversal. Further, we also propose different techniques to increase the speed-up by increasing compute units on FPGA and efficiently accessing the memory.**

1. **INTRODUCTION**

A non-deterministic finite automaton accepts input characters as an input stream. For every given input character, all the active states transition to a next state until the stream of input is completed. Every state may see a transition to more than one states per input character or it may not see any transition at all. Some states in the NFA are declared as accepting states or terminating states. This means that a specific given regular expression has been matched if and only if the accepting state associated with that regular expression is reached.

In Figure 1.1, the basic structure of an NFA is represented which accepts the binary input stream and detects the series like “00” or “11” in the stream. Here A, B, C and D are NFA states with D being the accepting or terminating state. The branches from one state to another or onto itself are called the transitions stating that the given state accepts the annotated character.

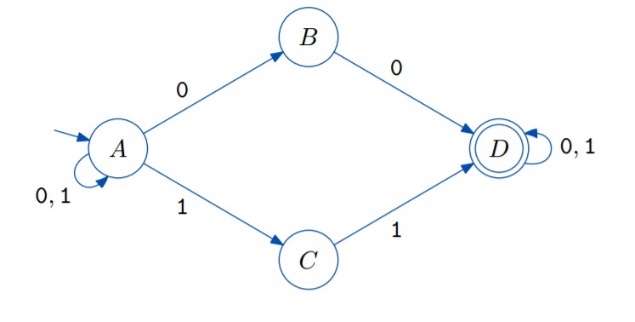


Figure 1.1: NFA structure for detecting pair of 0s and 1s.

Here, state A can make transition onto itself on input character ‘0’ or ‘1’, onto state B on input character ‘0’ and on state C on input character ‘1’ so it transitions to more than one state on a given input character.

1. **NFA TRAVERSAL**

The inherent characteristic of the NFA traversal is that of the breadth first search in that set of next states can be approached from its previous state(s) only. Breadth First Search or *bfs* is a famous algorithm that has been used to traverse or search structures like graphs or trees which are linked or dependent on their previous state. For example, in the Figure 1.1, states ‘B’ and ‘C’ can only be reached from state ‘A’, and state ‘D’ can be reached only either from ‘C’ or ‘B’.

Consider a stream of inputs i.e. “100011”. For first input ‘1’, states ‘A’ and ‘C’ will be activated. Now since the second character is not ‘1’, state ‘C’ will be declared dead as it does not have any transition on input ‘0’. For second input ‘0’, states ‘A’ and ‘B’ will be activated. For the third input of ‘0’, as ‘B’ has a transition to ‘D’ on input ‘0’, the accepting state of ‘D’ will be reached noting that a stream with two consecutive ‘0’s have been encountered. In the similar way, for the fourth input character ‘0’, the accepting state of D will be reached again, signifying the second pair of ‘0’s have been encountered. It is important to note that here we do not have different accepting states to differentiate between the pair of ‘0’s and ‘1’s being traced. For the regular expression patterns that we are trying to match, each of them have their own accepting state so as to differentiate and count the different regular expressions that we have matched.

Figure 2.1 shows the NFA traversal algorithm.

Figure 2.1: NFA traversal Algorithm

**while** input\_stream != NULL {

**for\_each** current active states {

**for\_each** transition of given active state {

**if** transition == input\_character

add target state to set of next active states;

}

}

* transfer all the next states to current states;

}

1. **METHODOLOGY**

In this section, we discuss the methods adopted to implement the NFA traversal on Field Programmable Gate Array (FPGA). First, we discuss the serial version of the design where in each cycle, at max one matching operation is done. Later, we move on to parallel implementation where multiple matches are done per cycle.

The NFA is stored in the Compressed Sparse Row (CSR) format in the Block Memory provided with FPGA board as an IP. This IP is configurable to read-write or read-only memory. For this implementation, we configure the given Block Memory as a read-only memory since we will just be looking up the stored NFA from the memory.

Figure 3.2: Serial NFA traversal Algorithm

CSR format is as represented below. Two array are stored contiguously in the Block Memory:

1. Address of starting points of transition pairs for the given state with index of that array.

Figure 3.1: Array indicating the address of their respective states’ transition pairs

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 |
| State | A | B | C | D | - |
| Address | 0 | 4 | 5 | 6 | 8 |

1. Transition pairs: Each entry stores a transition pair i.e. {transition character, target state}, a 32-bit value where bits 31:24 indicate the transition character and bits 23:0 indicate the target state.

For example, a 32-bit hexadecimal number 0100000F will mean that the given states will transition to state ‘15’ on input ‘1’.

**3.1 Serial Implementation on FPGA**

NFA\_state[0] = active;

get first input character;

**(@pose edge of clock)**

**if** FSM\_state == 1 and NFA\_state is active

Send rd\_address to block memory for finding the no. of tx (transition) pairs for NFA\_State;

FSM\_state = 2;

**else if** FSM\_state == 2

calculate the total number of tx pairs;

FSM\_state = 3;

**else if** FSM\_state == 3

**if** total no. of txs pairs != 0 {

get the txs pair;

total no. of tx pairs-- ;

**if** input character == tx

activate target state;

**else**

FSM\_state = 4;

**else if** FSM\_state == 4

**if** NFA\_state == last state

get a new input character;

**else**

go to next NFA\_state;

FSM\_state = 1;

The RTL level design for serial traversal is done keeping in mind the scope of pipelining the memory accesses to minimize the number of cycles taken to complete the matching. Also, this is done by keeping the number of cascaded logic blocks to minimum in each state of the pipeline. FSM has been created to match the behaviour of the NFA. The pseudo code for the serial NFA traversal is as shown in the Figure 3.2. The run is started with state ‘0’ as initially active state. The latency for the memory needs to be considered for deciding the FSM\_state transitions. Here, the Block Memory latency is of 1 cycle and hence we need to have and empty state where we wait for the memory to provide us the data. Note that here the currently active and next active states are stored in a bitmap where 1 denotes active while 0 denotes not active.

**3.2 Parallel Implementation on FPGA**

To accelerate the NFA traversal and to exploit the hardware area, we design an application specific parallel traversal engine that is done keeping in mind the scope pipelining the memory access as well as Data Level Parallelism (DLP) to maximize the throughput of the system. We design the traversal engine by replicating the traversal tracking logic so as to track more than one input streams at a time

To accelerate the NFA traversal and to exploit the hardware area, we design an application specific parallel traversal engine that is done keeping in mind the scope pipelining the memory access as well as Data Level Parallelism (DLP) to maximize the throughput of the system. We design the traversal engine by replicating the traversal tracking logic so as to track more than one input streams at a time.

This approach also utilizes the memory bandwidth available for the on-chip IPs in more efficient way than the one used for the serial implementation. To do more than one comparison operations per clock cycle, we need to have more data available in the registers. This is made possible by fetching data in the form of cache-line from the Block Memory. The initial implementation caches one line form the memory, utilizes all the elements that can be worked upon and fetches another one to continue working. Currently, the max vector length of the hardware is set to 16 to study the caching effects. This approach can easily be extended to the number of comparator units that can be synthesized on a given FPGA chip with required register memory support. Moreover, the more efficient the caching is, the better results can be obtained. For experiment purposes, we have used a 16-way fully associative cache kind of a structure to fetch the data into from the memory.

Figure 3.3 presents the algorithm designed for parallel execution with the vector length of 16 and the cache line size of 16 32-bit words with the localized cache containing one line. This kind of cache suffers from non-coalesced memory access patterns and will highly affect the performance of the system.

Some of the other approaches that can be studied to improve memory accesses are as below:

* Doing the comparison for multiple elements in parallel but fetching the transition pairs from the memory in series. This will result in low performance despite providing extra hardware for acceleration. Moreover, this kind of design will not use the memory bandwidth available. What we may be able to get rid of in this kind of implementation is to reduce the cache look-up logic and cache pollution.

Figure 3.3: Parallel NFA traversal Algorithm

* Pad the memory and make the memory aligned to improve the access patterns. This in-turn will ensure that the memory access is coalesced. But this kind of design implementation would suffer from memory fragmentation and essentially wasting the amount of Block Memory, i.e. the fast memory provided as an IP along with the FPGA module.

NFA\_state[0] = active;

get first input character;

**(@pose edge of clock)**

**if** (FSM\_state == 1 and NFA\_state is active for stream ‘1’ or stream ‘2’ routine)

Send rd\_address to block memory for finding the no. of tx (transition) pairs for NFA\_state;

FSM\_state = 2;

**else if** FSM\_state == 2

**if** (block offset == 15 for the range calculation of the state that sent rd\_address request in FSM\_state)

fetch the next block from the memory;

FSM\_state = 3;

**else if** FSM\_state == 3

**if**(block offset !=15 )

calculate the range and fetch the corresponding line

**else**

wait one cycle and then calculate the range

**else if** FSM\_state == 4

**if** total no. of txs pairs != 0 {

get the cache line which contains the first tx for the current NFA\_state;

**if** total txs > total tx in cached block

total txs = total txs – total tx in caches block;

**else**

total txs pairs = 0;

**if** (input character == txs of NFA\_state in cached line)

activate target states of NFA\_state in the cached line;

}

**else** FSM\_state = 4;

**else if** FSM\_state == 4

**if** NFA\_state == last state

get a new input character;

**else**

go to next NFA\_state;

FSM\_state = 1;

1. **RELATED WORK**

Regular Expression matching using NFA has been the topic of interest for a long time now. From an implementation perspective, existing regular expression matching engines can be classified into two categories: memory-based and logic-based. In the former, the FA is stored in memory; in the latter, it is stored in (combinatorial and sequential) logic. Memory-based implementations can be (and have been) deployed on various parallel platforms: general purpose multi-core processors, network processors, ASICs, FPGAs, and GPUs; logic-based implementations typically target FPGAs. For the logic-based approaches, updates in the pattern-set require the underlying platform to be reprogrammed. In a memory-based implementation, the design goals are the minimization of the memory size needed to store the automaton and of the memory bandwidth needed to operate it. Similarly, in a logic-based implementation the design should aim at minimizing the logic utilization while allowing fast operation (that is, a high clock frequency). One of the drawback of Logic-based implementation in FPGA is that the size of the NFA that it can process is limited by the resources available and so it may not be able to accommodate real-world data sets.

There is a body of literature advocating FPGA-based pattern matching [9]–[y]. FPGA can provide not only a fast matching cycle but also parallel matching operations. The normal regex matching for smaller data sets has been tested on FPGA by creating a reconfigurable logic and designing a supporting compiler has been studied in [9]. However, the data sets tested are for file search or similar applications and the approach is not automata based. In [10], they describe the mechanism by which SNORT IDS utilizes the PCRE compiler for translating the regular expression based rules from the SNORT database and matching them on the payload using the PCRE engine. In

Work has been done to reduce the pre-processing time for the NFA construction as presented in [1], [2] and [11].

Moreover, exploring different underlying hardware to optimize the regex matching has also been the topic of interest with the advent of new hardware like GPU, FPGA and Automata Processors as presented in [3] and [1].

In our work, we aim at creating a design that is able to support real-world size and number of regular expressions by specifically targeting 2 applications i.e. bio-informatics and Deep Packet Inspection. The maximum size of the NFA tested is over 100k states. This size is limited by the size of Block Memory IP present in FPGA. Using external memory we could increase the size of NFA beyond this while not requiring to change the design. Moreover, we present a style of storing data that is generic to different applications and can be optimized as per design choices. We choose the underlying platform as FPGA because of its reconfigurable nature and also less ‘time-to-market’ feature. Due to its reconfigurable nature, we aim to study and optimize different logic designs. We also aim to exploit the parallelism that FPGA can offer and improve its results by creating on-chip cache. The cache design is naïve and its look-up logic supports the non-coalesced memory accesses.

1. **IMPLEMENTATION**

The above presented algorithms for NFA traversal are have been implemented using Verilog 2001 Hardware Description Language and the block memory stated is the one provided by the Xilinx Vivado 2016.2 IP library. The FPGA device used for simulation is Xq7100rf1156-1l by Xilinx as it provides maximum Block Memory size which facilitates using large data sets for testing.

The data sets from different application backgrounds have been considered for testing the feasibility of the design for all those applications. These include data sets from bio-informatics and network intrusion detection system. The data sets who’s NFAs fit precisely in the Block Memory provided by the FPGA device are considered.

Figure 5: Datasets with their respective sizes of NFA

|  |  |
| --- | --- |
| **Data Set** | **No. of NFA states** |
| Snort534 | 9514 |
| 10ups\_r500\_k8\_d2 | 33432 |
| 100ups\_r500\_k8\_d2 | 137967 |

The data sets with larger no. of NFA states can also be considered with more off-chip memory support.

1. **RESULTS**

The simulations were done for the above stated data sets and the speedup in terms of throughput was studied i.e. number of cycles taken to process a specific length of input streams.

The results in Figure 5.1 shows that we have the improvement in throughput but not as significant. This is because of the reason as discussed earlier that as the memory access pattern, a single line caching will not be as affective due to non-coalesced memory accesses.

Figure 5.1 shows the speedup for the input stream with 50 probability of the regex occurrence and Figure 5.2 shows the speedup for the input stream with 90 probability of the occurrence of the regular expression. These graphs are normalized to their total number of cycles taken to complete the provided input stream on serial implementation.

1. **CONCLUSION AND FUTURE SCOPE**

From the above experiments, we can see that FPGA is good choice for accelerating the regular expression matching and it can be scaled with off-chip memory in use. Moreover, using FPGA also provides more hardware for computation (comparison) which in our case is not that costly in terms of hardware.

With aggressive caching techniques like CPUs i.e. increasing the size and making it n-way associative, and the multicore architecture that the FPGA can provide, the design can be greatly improved. Moreover, for scalability, the on-chip register files can also be considered before going off-chip as that will greatly improve the access time for the operations. The trade-off here would be of the reduction in on-chip hardware available for computation and for much larger sets which cannot fit on-chip, we will have to store the NFA in the off-chip memory. Moreover, by creating different caches for calculating the range and for fetching the transitions, we can reduce the number of memory accesses for the states which have more than 16 transitions. This will result in cache hit for 15 states for each memory access to calculate the range of each state.

For our future work, we suggest the other approach to storing the NFA is using transitions as the indices and its source and destination states as the values. In this way, we can check on all the states which have a transition on given input and then keep a check for the ones which are currently active. This storage structure is suitable for GPU which offers high memory bandwidth and high level of data parallelism. With this approach, at each input character, all the states with given input character will be traversed and the ones which are active will be able to activate their respective target states.

The other optimization that can be added on top the suggested ones are to pipeline the input characters so that we can process multiple input characters at the same time. This will require us to replicate the traversal routine hardware and modify the NFA storage format in order to keep track of back-transitions. This will be necessary to prevent the younger input characters from going ahead and editing traversal details before older input characters complete writing.

1. **REFRENCES**

[1] - Yan Sun, Victor C. Valgenti, and Min Sik Kim: Hierarchical NFA-Based Pattern Matching for Deep Packet Inspection

[2] - Michela Becchi, Patrick Crowly: Efficient Regular Expression Evaluation – Theory to Practice.

[3] - Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng and Michela Becchi: Demystifying Automata Processing: Comparing GPUs, FPGAs and Micron’s Automata Processor on Large-Scale Datasets

[4] - Andrew Todd1, Huan Truong2, Justin Deters3, John Long4, Gavin Conant2, Michela Becchi1,2: Parallel Gene Upstream Comparison via Multi-Level Hash Tables on GPU.

[5] - Snort User Manual 2.8.6, The Snort Project, Apr. 2010,

<http://www.snort.org/assets/140/snort_manual_2_8_6.pdf>

[6] - M. Becchi and P. Crowley, “An improved algorithm to accelerate regular expression evaluation”

[7] - Vlastimil Koˇsaˇr, Martin Z´adn´ık, Jan Koˇrenek: NFA Reduction for Regular Expressions Matching Using FPGA

[8] – R. Sindhu, V. Prasanna: Fast Regular Expression matching using FPGAs

[9] - An adaptable FPGA-based System for Regular Expression Matching

[10] - Compiling PCRE to FPGA for Accelerating SNORT IDS